# Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Monero FCMP++

Veridise Inc.
June 3, 2025

► **Prepared For:**

MAGIC Grants
https://magicgrants.org

► **Prepared By:**

Alp Bassa
Benjamin Sepanski
Petr Susil

► **Contact Us:**

contact@veridise.com

► **Version History:**

| | |
|---|---|
| Jun. 03, 2025 | V3 - Incorporate fixes |
| May 20, 2025 | V2 - Update client matter |
| May 8, 2025 | V1 |
| May 7, 2025 | Initial Draft |

**© 2025 Veridise Inc. All Rights Reserved.**

# Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 4

3 Security Assessment Goals and Scope 5
3.1 Security Assessment Goals . . . 5
3.2 Security Assessment Methodology & Scope . . . 5
3.3 Classification of Vulnerabilities . . . 7

4 Trust Model 9

5 Vulnerability Report 10
5.1 Detailed Description of Issues . . . 11
5.1.1 V-FCMP-VUL-001: Duplicate program constructs . . . 11
5.1.2 V-FCMP-VUL-002: Typos and incorrect comments . . . 12

6 Formal verification 13
6.1 Background . . . 13
6.2 Methodology . . . 13
6.3 Technical Comments on the Translation . . . 14
6.4 Results . . . 14
6.4.1 V-FCMP-SPEC-001: Equality . . . 15
6.4.2 V-FCMP-SPEC-002: Inverse . . . 15
6.4.3 V-FCMP-SPEC-003: Inequality . . . 16
6.4.4 V-FCMP-SPEC-004: Member of List . . . 17
6.4.5 V-FCMP-SPEC-005: On Curve . . . 21
6.4.6 V-FCMP-SPEC-006: Incomplete Curve Addition . . . 22
6.4.7 V-FCMP-SPEC-007: Tuple Member of List . . . 27

7 Soundness Proofs 30
7.1 Interactive Gadgets. . . . 30
7.1.1 Discrete Logarithm Gadget . . . 30
7.1.2 Tuple Member of List Gadget . . . 30
7.1.3 Protocol for relation $\mathcal{R}_{TML}$ . . . 32

Glossary 33

 © 2025 Veridise Inc.

# 1 Executive Summary

From Apr. 7, 2025 to May 2, 2025, MAGIC Grants engaged Veridise to conduct a security assessment of their Monero FCMP++ project. The security assessment covered the Monero full-chain membership proof algorithm, arithmetization, and implementation. This circuit proves a blinded Pedersen committment is contained within a publicly known set without revealing any other information. Veridise analysts analyzed the sub-circuits which compose the Monero FCMP++ circuit in prior reviews [1–3]. This review focused on their use and implementation in the full Monero FCMP++ codebase. Veridise conducted the assessment over 12 person-weeks, with 3 security analysts reviewing the project over 4 weeks from commits `78754718-6dc3332b`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

**Project Summary.** The security assessment covered the Monero FCMP++ membership proof circuit [4, Section 4]. The Monero blockchain uses the UTXO model. For privacy, the current proof system allows spenders to prove their spent UTXO is one of a large subset of the possible valid UTXOs. Monero FCMP++ [4, 5] improves on this by allowing spenders to prove their UTXO is a member of the *entire* valid UTXO set without revealing anything else about the UTXO. There are two primary components to this scheme: (1) representing the UTXO-set as a *curve tree* [6] and (2) using a new, highly efficient method for proving knowledge of discrete logarithms [1–3, 7].

Curve trees [6] are very similar to Merkle Trees. The UTXO-set is stored as the set of leaves of the tree. Each internal node is labeled with a Pedersen committment to its children's coordinates. To prove set membership of an element without revealing its index, a prover re-randomizes each Pedersen commitment on the path from the leaf to the root. Then, an outer proof system like Bulletproofs [8] (or in this case, Generalized Bulletproofs [9]) is used to prove that the re-randomization was performed correctly and that the root matches the expected value. The key issue, of course, is that a Pedersen commitment's coordinates lie in the *Base field* of the curve, while the committed coordinates lie on the *Scalar field*. For efficiency, two curves forming a Curve cycle must be used. This means that two bulletproofs must be performed: one for each curve. In particular, Monero FCMP++ uses a curve cycle which towers over Wei25519: the Helios and Selene curves [10]. This means that each leaf is in Wei25519, each "odd" layer above the leaves is labeled with a Pedersen commitment which lies on the Selene curve, and each "even" layer above the leaves is labeled with a Pedersen commitment which lies on the Helios curve.

Curve tree proofs involve a large number of discrete logarithm proofs-of-knowledge. The leaf data, along with each node on the path from leaf to root, must be provably re-randomized using the expected generator. This can become prohibitively computationally expensive. To optimize these costs, a new technique [7] is used which leverages the formal definition of the elliptic curve group law. In short, a set of elliptic curve points sums to zero if and only if there exists a polynomial over the base field with zeros of the appropriate multiplicities at exactly that set of points (and poles only at the point at infinity). This new technique leverages this fact, along with a Schwartz-Zippel -style argument, to show that a prover-provided polynomial has

 © 2025 Veridise Inc.

zeros at the expected set of points. Finally, the Fiat-Shamir transformation is applied to turn the argument non-interactive. See Section 7.1.1 for a more complete description.

Putting this all together, the Monero FCMP++ consists of two bulletproofs, one for each curve of the cycle of curves used in the curve tree. Each proof proves that the re-randomization was performed correctly (using the novel discrete-log proof) and that the de-randomized point is in the pre-image of the next Pedersen commitment on the path. The underlying Generalized Bulletproofs proof-system proves knowledge of the Pedersen pre-images, allowing each bulletproof to safely use the others' public input as the previous, proven value on the path from leaf to root.

The code consists of a prover and verifier built on top of a previously audited Generalized Bulletproofs implementation. Working through the entire workflow, a prover takes their desired leaf and the path to the root, provably blinds each Pedersen commitment, and then stores these blinded commitments as public input. They hash the public input into the Fiat-Shamir transcript, then use those pseudo-random challenges to generate two Generalized Bulletproof circuits, one for each curve. The prover creates the two bulletproofs, then sends them (along with the public input) to the verifier.

The verifier recomputes the Fiat-Shamir challenges. It then uses the public input and Fiat-Shamir challenges to build the two Generalized Bulletproofs circuits. Finally, it verifies the proofs against the circuits.

**Code Assessment.** The Monero FCMP++ developers provided the source code of the Monero FCMP++ for the code review. The project appears to be mostly original code written by the Monero FCMP++ developers. It contains thorough documentation comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the Monero FCMP++ developers provided a thorough writeup of the system design, each sub-circuit used, the intended arithmetizations, and all referenced papers/techniques.

The source code contained a test suite, which the Veridise security analysts noted included randomized property-based testing as well as negative tests to check for under-constrained issues. Several files in the source code also indicate that the developers use linters to ensure high code quality.

The Veridise analysts note that the developers made very effective use of Rust's typing system to ensure a high degree of security. Each important property is clearly documented, and associated to a type which can only be constructed by a routine which guarantees the desired property holds. Additionally, the developers stick very closely to the algorithm described in the specification. There are almost no deviations, making the theoretical analysis apply almost directly to the implementation.

**Summary of Issues Detected.** The security assessment uncovered no issues of low severity or higher. The Veridise analysts identified 1 warning, and 1 informational finding. These issues may help to improve the maintainability of the project. The developers provided fixes for both issues, which were validated by the Veridise analysts.

**Recommendations.** After conducting the assessment of the protocol, the security analysts had a few suggestions to improve on and safely use the Monero FCMP++ proofs.

*Generator Validation.* All of the techniques rely explicitly on the pairwise relative discrete logarithms of all generators being unknown. The selection of generators should be publicly disclosed and heavily scrutinized by several parties.

*Automated Tape Serialization.* Currently, variable indexing within the Generalized Bulletproofs system is manual, and therefore error-prone. While it is performed carefully in the codebase, and no related errors were found, an automated system may remove this vector as a possibility.

*Additional Documentation.* The documentation provided by the Monero FCMP++ is extremely extensive. However, given the highly technical nature of each aspect of the design, it may be difficult for new practitioners/analysts to tackle individual components of the design. Adding diagrams and small examples may greatly benefit future developers or analysts in their understanding of the project.

**Developer Comments on Recommendations.** The developers wish to note that, in practice, their generators are deterministically sampled from a very simple transcript. Additionally, they wish to note that the current system implements a somewhat-automated tape serialization via a read/write flow, akin to reading/writing from an IO stream. It is not directly embedded into the framework, but out of scope for what is a bespoke framework currently solely needing to be so expansive as to promote the safe and secure implementation of this project (not to be the next framework anybody is using for ZK circuits).

**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.

© 2025 Veridise Inc. 

# 2 Project Dashboard

**Table 2.1:** Application Summary.

| Name | Version | Type | Platform |
| --- | --- | --- | --- |
| Monero FCMP++ | `78754718-6dc3332b` | Rust | Monero |

**Table 2.2:** Engagement Summary.

| Dates | Method | Consultants Engaged | Level of Effort |
| --- | --- | --- | --- |
| Apr. 7–May 2, 2025 | Manual & Tools | 3 | 12 person-weeks |

**Table 2.3:** Vulnerability Summary.

| Name | Number | Acknowledged | Fixed |
| --- | --- | --- | --- |
| Critical-Severity Issues | 0 | 0 | 0 |
| High-Severity Issues | 0 | 0 | 0 |
| Medium-Severity Issues | 0 | 0 | 0 |
| Low-Severity Issues | 0 | 0 | 0 |
| Warning-Severity Issues | 1 | 0 | 1 |
| Informational-Severity Issues | 1 | 0 | 1 |
| TOTAL | 2 | 0 | 2 |

**Table 2.4:** Category Breakdown.

| Name | Number |
| --- | --- |
| Maintainability | 2 |


© 2025 Veridise Inc.

# 3 Security Assessment Goals and Scope

## 3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Monero FCMP++'s new Full chain membership proof implementation. During the assessment, the security analysts aimed to answer questions such as:

- Does the rust implementation match the description in the Monero FCMP++ paper [4]?
- Are all gadgets [4, Section 4.3] deterministic?
- Can a malicious prover create a proof for an element which is not in the curve tree?
- Is the Fiat-Shamir transformation implemented securely?
- Does the implementation make any undocumented assumptions?
- Is any unintended information about the execution or input data leaked to the verifier?
- Several gadgets are intentionally incomplete. For example, there is no representation for $\mathcal{O}$, the addition circuit constraints that $P \neq \pm Q$, and discrete-log proofs require existence of a divisor with non-zero $x$-coefficient. Can any incompleteness be leveraged to cause a denial of service to benign users?
- Can the padding values used in the Pedersen commitments be used to the prover's advantage, or successfully set to non-zero values?
- Can paths shorter or longer than the expected curve tree height be used to generate false proofs?
- Is there any input a prover could pass a verifier to force them to generate an incorrect circuit?
- Can any inconsistency in the expected length of the proof or other input sizes lead a verifier to accept a false statement?
- Are any relations expected to hold across gadgets, but not enforced (for example, that the randomness committed to by $R$ matches the randomness used to blind $I$)?
- Is there any way the prover can use the randomness to alter the linking tag, or change the committed randomness?

## 3.2 Security Assessment Methodology & Scope

**Security Assessment Methodology.** To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools.

*Methodology*. Veridise security analysts reviewed the reports of previous audits for Monero FCMP++, inspected the provided tests, and read the Monero FCMP++ documentation. They then wrote a translator to Picus, and worked to formally verify determinism of each non-interactive gadget. Next, they began a manual review of the code assisted by automated testing, and formal verification.

For full confidence in the soundness of the interactive gadgets, the analysts developed soundness proofs of several components of the protocol which can be found in Chapter 7. Finally, the

 © 2025 Veridise Inc.

analysts developed automated tests to find potential deviations from intended implementation. This effort focused on the Fiat-Shamir transformation, denial-of-service, and privacy violations. These are described in more detail below.

During the security assessment, the Veridise security analysts regularly met with the Monero FCMP++ developers to ask questions about the code.

*Formal verification.* The Veridise team leveraged the custom formal verification tool Picus to verify properties of the zero-knowledge circuits and various functions (see Chapter 6 for discussion and results). Picus is designed to prove or find violations of determinism, which is an important safety property for zero-knowledge circuits. The analysts developed a translator of critical subroutines written in rust to a formal verification tool Picus.

*Automated testing.* As part of the security assessment, Veridise security analysts developed automated tests to evaluate the implementation of Monero FCMP++ programmatically.

- *Fiat-Shamir transformation.* The Fiat-Shamir transformation is a cryptographic technique that converts interactive proofs of knowledge, which involve multiple rounds of challenge-response between a prover and a verifier, into non-interactive proofs. The Fiat-Shamir transformation replaces the verifier's random challenge with a deterministic challenge generated by applying a cryptographic hash function to the prover's commitment and other public data. Known attacks against the Fiat-Shamir transform primarily arise from incorrect or non-standard implementations rather than flaws in the transform itself when modeled as a random oracle [11, 12]. If a challenge is computed independently of certain proof elements, a malicious prover can forge proofs of false statements.
  As part of the assessment, Veridise analysts tested that (1) every variable corresponding to "input" from the prover was stored in a `CG` variable, i.e. the ones which would be committed to as part of the Generalized Bulletproofs scheme, and (2) any modification to prover input leads to a different Fiat-Shamir challenge. This test provides additional assurance that all prover-chosen values must be set before the Fiat-Shamir challenge is known.
- *Proof size.* This test validated that the number of rows for both bulletproofs matches the expected value. It was run for up to 8 layers of curve tree and up to 256 inputs. Since the verifier is constructing a different circuit for each claim, this increases assurance that no misconfiguration of the verifier will lead to completeness issues.
- *Blinded branches.* This test targets the `FCMP::prove` function. Since some blinds must be stored and saved to de-randomize commitments inside the proof, it is possible that indexing errors could lead to reuse of blinds. This would leak critical information. Each run of this test validates that the current implementation of the prover does not reuse any blinds.

**Scope** The scope of this security assessment is limited to folders `circuit-abstraction`, `ec-gadgets` and `src` of the source code provided by the Monero FCMP++ developers, which contains the implementation of the Monero FCMP++.

- `src/`
  - `circuit.rs`
  - `lib.rs`
  - `params.rs`

  - tape.rs
  - tests.rs
  - tree.rs
  - gadgets/
    - interactive.rs
  - prover/
    - blind.rs
    - mod.rs
- circuit-abstraction/src/
  - gadgets.rs
  - lib.rs
- ec-gadgets/src/
  - dlog.rs
  - lib.rs

## 3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

**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 |

The likelihood of a vulnerability is evaluated according to the 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)<br>- OR -<br>Requires a small set of users to perform an action |
| Very Likely | Can be easily performed by almost anyone |

The impact of a vulnerability is evaluated according to the 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<br>- OR -<br>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<br>- OR -<br>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 Trust Model

Monero FCMP++ uses Pedersen commitments, Generalized Bulletproofs, and Curve trees using curves Wei25519, Helios, and Selene. The discrete logarithm problem is believed to be hard for each of these curves (in the standard computation model). Curve25519 [13] (which is birationally equivalent to Wei25519), and the cycle of curves Helios and Selene [10] were chosen based on publicly known requirements and therefore, it is widely believed there are no backdoors in the choice of parameters for these elliptic curves. Generalized Bulletproofs does not require a trusted setup. The security and privacy claims are based only on standard security assumptions, the correctness of the implementation, and choosing appropriate generators with no known relations.

This review targeted only the membership proof portion of the the Monero FCMP++ upgrade. In this setting, as is common in zero-knowledge applications, there are two roles: the prover and the verifier. Provers are responsible for using cryptographic random number generators and unique blinds to ensure privacy. Verifiers are responsible for using a correct curve tree root, and performing the actual proof verification. Additionally, this circuit does not provide replay protection. Verifiers must use the provided commitment $R$, which commits to the randomness applied to the linking tag, to ensure that no attacker may use a member of the public set more than once.

During this review, analysts considered any means by which a potentially dishonest prover could use unexpected or invalid values to produce a valid proof to be a soundness risk to the protocol. On the other side of things, Veridise analysts considered any information which could be abstracted from an honest prover by the verifier to be a privacy risk. The analysts assumed that the prover was executing in a secure environment, and that attackers would not have access to information leaked through side channels. Despite this, the MAGIC Grants team did take steps to ensure memory was zeroed out and constant-time operations were used wherever possible.

 © 2025 Veridise Inc.

# 5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

**Table 5.1:** Summary of Discovered Vulnerabilities.

| ID | Description | Severity | Status |
|---|---|---|---|
| V-FCMP-VUL-001 | Duplicate program constructs | Warning | Fixed |
| V-FCMP-VUL-002 | Typos and incorrect comments | Info | Fixed |

 © 2025 Veridise Inc.

## 5.1 Detailed Description of Issues

### 5.1.1 V-FCMP-VUL-001: Duplicate program constructs

| Severity | Warning | Commit | 6dc3332 |
|---|---|---|---|
| Type | Maintainability | Status | Fixed |
| File(s) | See issue description | | |
| Location(s) | See issue description | | |
| Confirmed Fix At | 0898182c | | |

**Description** The following program constructs duplicate already-implemented logic:

1. `crypto/fcmps/ec-gadgets/src/dlog.rs`:
   a) `<Circuit as EcDlogGadgets>::discrete_log_challenge()`: Two random points are generated: `(c0_x, c0_y)` and `(c1_x, c1_y)`. The two points are generated using identical logic, but have separate implementations.
2. `crypto/fcmps/src/params.rs`:
   a) `FcmpParams::new()`: This function repeatedly uses the expression `<<C as FcmpCurves>::OC as Ciphersuite>::G::to_xy(*).unwrap();`.

**Impact** These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

**Developer Response** We defined a single function to sample points on the embedded elliptic curve.

We do not replace the usage of `to_xy().unwap()` as that is two function calls, making the only more minimal form a single function call. A single function would omit the declaration of the panic, requiring a more verbose name, such as `to_xy_panicking`, removing benefit of a shorter form.

### 5.1.2 V-FCMP-VUL-002: Typos and incorrect comments

| Severity | Info | Commit | 6dc3332 |
|---|---|---|---|
| Type | Maintainability | Status | Fixed |
| File(s) | See issue description | | |
| Location(s) | See issue description | | |
| Confirmed Fix At | cebfb4bd | | |

**Description** In the following locations, the auditors identified minor typos and potentially misleading comments:

1. `crypto/fcmps/ec-gadgets/src/dlog.rs`:
   a) `divisor_challenge_eval()`: The below comment indicates that the loop does not raise values by `j + 1` as required. However, the `x`-values *are* raised to `j + 1`, just stored in index `j` rather than index `j + 1`. Distinguishing between indexing by `j + 1` and raising by a power of `j + 1` may help readers better understand the intent.

```
1 // This does not raise by ‘j + 1‘ as x_p_0_n_0 omits x**0
2 p_0_n_1 = p_0_n_1.term(challenge.x_p_0_n_0[j], *var);
```

**Snippet 5.1:** Snippet from `divisor_challenge_eval()`

2. `crypto/fcmps/src/prover/blinds.rs`:
   a) `OutputBlinds::new()`: The documentation indicates that `new()` returns `None` if any the points in the resulting `Input` would be identity. However, this is not the case.

3. `crypto/fcmps/src/lib.rs`:
   a) `prove()`: This function contains commented out `dbg!()` statements.
   b) `verify()`: This function contains commented out `dbg!()` statements.

**Impact** These minor errors may lead to future developer confusion.

**Developer Response** The developers fixed the above issues.

# 6 Formal verification

## 6.1 Background

Formal verification of zero-knowledge circuits addresses critical security challenges in cryptographic systems by mathematically proving important security properties under all defined conditions. Errors in arithmetization can compromise security, leaving critical values under the control of a possibly malicious prover. By employing formal methods, developers can systematically eliminate these vulnerabilities.

**Picus** is a formal verification tool which combines lightweight static analysis with SMT solvers to detect under-constrained conditions in zero-knowledge circuits, or formally verify that no such conditions exist. Picus V2 defines its own constraint language (Picus Constraint Language) and verifies the determinism of circuits written in that language. As such, V2 can verify determinism of any circuit which has been translated into its language.

## 6.2 Methodology

To formally verify that each of the provided gadgets is deterministic, Veridise analysts wrote a small translator which takes a `Circuit` * and outputs a program in Picus† V2's DSL. The analysts then ran the Picus tool on each circuit, instantiated with both the Selene and Helios scalar fields, to determine if the gadget is deterministic. Proving a circuit is deterministic means that, for any fixed setting of the input variables, there is at most one solution for all other variables defined in the circuit.

For example, the elliptic-curve addition circuit takes in four field values—`b.x`, `b.y`, `c.x`, and `c.y`—then constrains `a + b == c` for some fixed `a`. In this case, `b.x` and `b.y` are classified as inputs. To be deterministic, `c.x`, `c.y`, and all other variables used in the circuit must be uniquely determined by (`b.x`, `b.y`). This concept is defined more formally in the Picus docs ‡. Readers should note an important distinction in the above example. All of `b.x`, `b.y`, `c.x`, `c.y` are uninterpreted values provided to the gadgets — i.e. they are treated as arbitrary field values. However, from the perspective of Picus, only `b.x` and `b.y` are classified as inputs. This classification must be performed manually.

Each gadget's `Circuit` is defined as follows:

1. Determine the uninterpreted field variables provided to the gadget. In the running example of the elliptic-curve addition circuit, `b.x`, `b.y`, `c.x`, and `c.y` are the uninterpreted field variables provided to the circuit.
2. Pack the uninterpreted field variables into the `Circuit`. In the elliptic curve addition example, one might encode `aL_0 = b.x`, `aL_1 = b.y`, `aR_0 = c.x`, `aR_1 = c.y`.

* https://github.com/kayabaNerve/fcmp-plus-plus/blob/163fb9c484c66c26635742fa1d5cba71bbf45006/crypto/fcmps/circuit-abstraction/src/lib.rs#L67

† https://docs.veridise.com/picus-v2

‡ https://docs.veridise.com/picus-v2/picus_constraint_language/#2-basics/

 © 2025 Veridise Inc.

3. Apply the gadget using the predefined method from the Monero FCMP++[§] codebase. In the elliptic curve addition example, this means calling `incomplete_add_fixed()`[¶].
4. Manually annotate which variables are inputs (for Picus).
5. Run the translator on the `Circuit` and annotated input variables.

## 6.3 Technical Comments on the Translation

It is important to ensure the values provided to the gadgets are in fact uninterpreted, i.e. have no assumed relations. To this end, Veridise also supplies a value `num_unconstrained_rows` to the `Circuit`-to-Picus translator. The first `num_unconstrained_rows`-many rows are considered to be uninterpreted fields provided to the gadgets. This is done by two means:

1. For rows `0, 1, 2, ..., num_unconstrained_rows-1`, the constraints `aL_i * aR_i == aO_i` are removed. 2. For rows `0, 1, 2, ..., num_unconstrained_rows-1`, each variable `aL_i`, `aR_i`, and `aO_i` is removed **unless** any of the following is true:

1. The variable is referenced in a constraint.
2. The variable is listed as input.

By including all inputs and all variables in the constraint, the only possible bug introduced by these "unconstrained rows" is if a circuit leaves an uninterpreted, non-input field totally unconstrained. The Veridise team has manually reviewed the translations to ensure this is not the case.

## 6.4 Results

The results of this effort are summarized below in Table 6.1. Full details of each circuit verified to be deterministic are provided below. This additionally includes annotated translations (over the Selene field) of each gadget to Picus V2. These annotations are for the benefit of the reader, and ignored by Picus as comments. For several circuits, many distinct instantiations were verified. These variants are detailed for each individual circuit.

Additional Picus translations for the Helios field, other list-lengths, or the curve addition circuit without assumptions are not shown, but may be available upon request.

**Table 6.1:** Summary of Circuits Formally Verified to be Deterministic.

| ID | Gadget | Status |
|---|---|---|
| V-FCMP-SPEC-001 | Equality | Determinism Verified |
| V-FCMP-SPEC-002 | Inverse | Determinism Verified |
| V-FCMP-SPEC-003 | Inequality | Determinism Verified |
| V-FCMP-SPEC-004 | Member of List | Determinism Verified |
| V-FCMP-SPEC-005 | On-Curve | Determinism Verified |
| V-FCMP-SPEC-006 | Incomplete Curve Addition | Determinism Verified |
| V-FCMP-SPEC-007 | Tuple Member of List | Determinism Verified |

§ https://github.com/kayabaNerve/fcmp-plus-plus/tree/develop
¶ https://github.com/kayabaNerve/fcmp-plus-plus/blob/78754718faa21f0a5751fbd30c9495d7f7f5c2b1/crypto/fcmps/ec-gadgets/src/lib.rs#L89

© 2025 Veridise Inc. 

### 6.4.1 V-FCMP-SPEC-001: Equality

| Gadget | Equality | Status | Verified by Picus |
|---|---|---|---|
| Configuration | Solver: multi-solver, Solver timeout: 60s | | |
| Variants | Selene, Helios | | |

This section describes formal verification of Equality gadget [4, Section 4.3.1].

```
let mut equality_circuit: Circuit<C> = Circuit::<C>::empty(1);
let l0 = Variable::aL(0);
let r0 = Variable::aR(0);

equality_circuit.equality(l0.into(), &r0.into());

// Only input is left lincomb
PicusInputs {
  assume_circuits: vec![],
  assert_circuits: vec![equality_circuit],
  num_unconstrained_rows: 1,
  input_vars: vec![l0],
}
```

**Snippet 6.1:** Generate Equality verification circuit

#### Picus (Annotated)

```
(prime-number
    57896044618658097711785492504343953926634992332820282019728792003956564819949)
(begin-module equality)
  (input aL_0)
  (output aR_0)
  ;
  ; 1 a + -1 b = 0
  (assert (= (+ (* 1 aL_0) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_0)) 0))
(end-module)
```

**Snippet 6.2:** Equality circuit

### 6.4.2 V-FCMP-SPEC-002: Inverse

| Gadget | Inverse | Status | Verified by Picus |
|---|---|---|---|
| Configuration | Solver: multi-solver, Solver timeout: 60s | | |
| Variants | Selene, Helios | | |

This section describes formal verification of Inverse gadget [4, Section 4.3.2].

```
let mut inverse_circuit: Circuit<C> = Circuit::<C>::empty(1);
let l0 = Variable::aL(0);

```

```
4   let _inv_l0 = inverse_circuit.inverse(Some(l0.into()), None);
5
6   // Input is the inverting variable
7   PicusInputs {
8     assume_circuits: vec![],
9     assert_circuits: vec![inverse_circuit],
10    num_unconstrained_rows: 1,
11    input_vars: vec![l0],
12  }
```

**Snippet 6.3:** Generate Inverse verification circuit

#### Picus (Annotated)

**Constraints.** Below is an annotated version of the constraints passed to Picus. None of the annotations are used by Picus, they are only so that the user may additionally validate the translation performed on the circuit code.

```
1  (prime-number
       57896044618658097711785492504343953926634992332820282019728792003956564819949)
2  (begin-module inverse)
3    (input aL_0)
4    (output aL_1)
5    (output aO_1)
6    (output aR_1)
7    ;
8    ; Equality(al:push(a); a)  ---->     1 a + -1 b = 0
9    (assert (= (+ (* 1 aL_0) (*
       57896044618658097711785492504343953926549254372227246365156541811699034343326
       aL_1)) 0))
10   ;
11   ; 1ao:last() + -1 = 0
12   (assert (= (* 1 aO_1) 1))
13   ;
14   ; quadratic constraints
15   (assert (= (* aL_1 aR_1) aO_1))
16 (end-module)
```

**Snippet 6.4:** Inverse verification circuit

### 6.4.3 V-FCMP-SPEC-003: Inequality

| Gadget | Inequality | Status | Verified by Picus |
|---|---|---|---|
| Configuration | Solver: multi-solver, Solver timeout: 60s | | |
| Variants | Selene, Helios | | |

This section describes formal verification of Inequality gadget [4, Section 4.3.3].

```
1   let mut inequality_circuit: Circuit<C> = Circuit::<C>::empty(1);
2   let l0 = Variable::aL(0);
3   let r0 = Variable::aR(0);
4
```

```
    inequality_circuit.inequality(l0.into(), &r0.into(), None);

    // Return inequality, ensuring all other variables
    PicusInputs {
      assume_circuits: vec![],
      assert_circuits: vec![inequality_circuit],
      num_unconstrained_rows: 1,
      input_vars: vec![Variable::aL(0), Variable::aR(0)],
    }
```

**Snippet 6.5:** Generate Inequality verification circuit

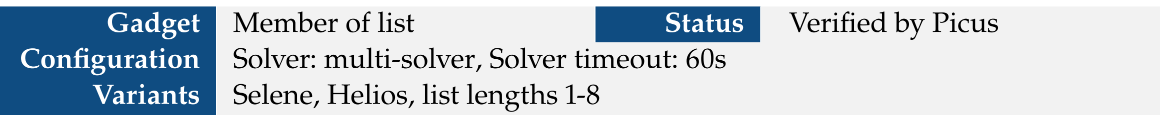
### Picus (Annotated)

**Constraints.** Below is an annotated version of the constraints passed to Picus. None of the annotations are used by Picus, they are only so that the user may additionally validate the translation performed on the circuit code.

```
(prime-number
    57896044618658097711785492504343953926634992332820282019728792003956564819949)
(begin-module inequality)
  (input aL_0)
  (input aR_0)
  (output aL_1)
  (output aO_1)
  (output aR_1)
  ;
  ; ( aL_0 - aR_0 ) - aL_1
  (assert (= (+ (+ (* 1 aL_0) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_0)) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aL_1)) 0))
  ;
  ; aR_1 is inverse of aL_1
  (assert (= (* 1 aO_1) 1))
  ;
  ; If a = b, c will be 0, and ao:last() must be 0 (since any number multiplied by 0
    is 0).
  (assert (= (* aL_1 aR_1) aO_1))
(end-module)
```

**Snippet 6.6:** Inequality verification circuit

### 6.4.4 V-FCMP-SPEC-004: Member of List



| Gadget | Member of list | Status | Verified by Picus |
|---|---|---|---|
| Configuration | Solver: multi-solver, Solver timeout: 60s | | |
| Variants | Selene, Helios, list lengths 1-8 | | |

This section describes formal verification of Member of List gadget [4, Section 4.3.4]. The Veridise analysts verified instantiations of this circuit with list lengths one through eight for both the Selene and Helios curves.

```
// Build variables
let list = (0..list_length)
  .map(|i| if i % 2 == 0 { Variable::aL(i / 2) } else { Variable::aR(i / 2) })
  .collect::<Vec<_>>();
let maybe_member_var_index = list_length / 2;
let maybe_member_var = if list_length % 2 == 0 {
  Variable::aL(maybe_member_var_index)
} else {
  Variable::aR(maybe_member_var_index)
};
let all_vars = list.iter().cloned().chain(Some(maybe_member_var)).collect::<Vec<_>>();

// Add membership check
let list: Vec<LinComb<C::F>> = list.into_iter().map(|var| var.into()).collect::<Vec<_>>();
let num_predefined_vars = (all_vars.len() + 1) / 2;
let member_of_list_circuit: Circuit<C> = Circuit::<C>::empty(num_predefined_vars);
let member_of_list_circuit =
  constrain_member_of_list(member_of_list_circuit, maybe_member_var.into(), list);

// Return the circuit along with input variable (the point)
PicusInputs {
  assume_circuits: vec![],
  assert_circuits: vec![member_of_list_circuit],
  num_unconstrained_rows: num_predefined_vars,
  input_vars: all_vars,
}
```

**Snippet 6.7:** Generate Member of list verification circuit

### Picus (Annotated)

**Constraints.** Below is an annotated version of the constraints passed to Picus. None of the annotations are used by Picus, they are only so that the user may additionally validate the translation performed on the circuit code.

Only the length-7 version is shown below.

```
(prime-number 57896044618658097711785492504343953926634992332820282019728792003956564819949)
(begin-module member_of_list7)
  ; list
  (input aL_0)
  (input aL_1)
  (input aL_2)
  (input aL_3)
  (input aR_0)
  (input aR_1)
```

```
10  (input aR_2)
11
12  ; member
13  (input aR_3)
14
15  (output aL_4)
16  (output aL_5)
17  (output aL_6)
18  (output aL_7)
19  (output aL_8)
20  (output aL_9)
21  (output aO_4)
22  (output aO_5)
23  (output aO_6)
24  (output aO_7)
25  (output aO_8)
26  (output aO_9)
27  (output aR_4)
28  (output aR_5)
29  (output aR_6)
30  (output aR_7)
31  (output aR_8)
32  (output aR_9)
33
34  ; let (_l, _r, constrainable_carry) = self.mul(Some(carry), Some(next), witness)
      --> (aL_4, aR_4, aO_4)
35  ; constrainable_carry <== aO_4
36  (assert (= (+ (+ (* 1 aL_0) (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aR_3)) (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aL_4)) 0))
37  (assert (= (+ (+ (* 1 aR_0) (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aR_3)) (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aR_4)) 0))
38
39  ; let (_l, _r, constrainable_carry) = self.mul(Some(carry), Some(next), witness)
      --> (aL_5, aR_5, aO_5)
40  ; constrainable_carry <== aO_5
41  (assert (= (+ (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aL_5) (* 1 aO_4)) 0))
42  (assert (= (+ (+ (* 1 aL_1) (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aR_3)) (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aR_5)) 0))
43
44
45  ; member                                                     <==        aR_3
46  ; - member                                                   <==          (*
```

```
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_3)
47  ; next = list_member - member         <==          (* 1 aR_1) + (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_3)
48  ; current carry                                      ==>  0_5
49  ; constrainable_carry                 <==  a0_6
50  ; let (_l, _r, constrainable_carry) = self.mul(Some(carry), Some(next), witness)
    --> (aL_6, aR_6, a0_6)
51  (assert (= (+ (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aL_6) (* 1 a0_5)) 0))
52  (assert (= (+ (+ (* 1 aR_1) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_3)) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_6)) 0))
53
54  ; let (_l, _r, constrainable_carry) = self.mul(Some(carry), Some(next), witness)
    --> (aL_7, aR_7, a0_7)
55  (assert (= (+ (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aL_7) (* 1 a0_6)) 0))
56  (assert (= (+ (+ (* 1 aL_2) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_3)) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_7)) 0))
57
58  ; let (_l, _r, constrainable_carry) = self.mul(Some(carry), Some(next), witness)
    --> (aL_8, aR_8, a0_8)
59  (assert (= (+ (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aL_8) (* 1 a0_7)) 0))
60  (assert (= (+ (+ (* 1 aR_2) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_3)) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_8)) 0))
61
62  ; let (_l, _r, constrainable_carry) = self.mul(Some(carry), Some(next), witness)
    --> (aL_9, aR_9, a0_9)
63  (assert (= (+ (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aL_9) (* 1 a0_8)) 0))
64  (assert (= (+ (+ (* 1 aL_3) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_3)) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_9)) 0))
65
66  (assert (= (* 1 a0_9) 0))
67  (assert (= (* aL_4 aR_4) a0_4))
```

```
68   (assert (= (* aL_5 aR_5) aO_5))
69   (assert (= (* aL_6 aR_6) aO_6))
70   (assert (= (* aL_7 aR_7) aO_7))
71   (assert (= (* aL_8 aR_8) aO_8))
72   (assert (= (* aL_9 aR_9) aO_9))
73 (end-module)
```

**Snippet 6.8:** Member of list (length 7) verification circuit

### 6.4.5 V-FCMP-SPEC-005: On Curve

| | | | |
|---|---|---|---|
| **Gadget** | On Curve | **Status** | Verified by Picus |
| **Configuration** | Solver: multi-solver, Solver timeout: 60s | | |
| **Variants** | Selene, Helios | | |

This section describes formal verification of On Curve gadget [4, Section 4.3.5].

```
1   // Build variables
2   let pt = (Variable::aL(0), Variable::aR(0));
3   let num_predefined_vars = 1;
4
5   // Add on-curve sertions
6   let curve = CurveSpec { a: BaseCurve::a(), b: BaseCurve::b() };
7   let mut on_curve_circuit: Circuit<C> = Circuit::<C>::empty(num_predefined_vars);
8   let _ = on_curve_circuit.on_curve(&curve, pt);
9
10  // Return the circuit along with input variable (the point)
11  PicusInputs {
12    assume_circuits: vec![],
13    assert_circuits: vec![on_curve_circuit],
14    num_unconstrained_rows: 1,
15    input_vars: vec![pt.0, pt.1],
16  }
```

**Snippet 6.9:** Generate On curve circuit

#### Picus (Annotated)

```
1  (prime-number
       57896044618658097711785492504343953926549254372227246365156541811699034343327)
2  (begin-module ec_on_curve)
3    (input aL_0)
4    (input aR_0)
5    (output aL_1)
6    (output aL_2)
7    (output aL_3)
8    (output aO_1)
9    (output aO_2)
10   (output aO_3)
11   (output aR_1)
12   (output aR_2)
13   (output aR_3)
```

```
  (assert (= (+ (* 1 aL_0) (*
    57896044618658097711785492504343953926549254372227246365156541811699034343326
    aL_1)) 0))
  (assert (= (+ (* 1 aL_0) (*
    57896044618658097711785492504343953926549254372227246365156541811699034343326
    aR_1)) 0))
  (assert (= (+ (*
    57896044618658097711785492504343953926549254372227246365156541811699034343326
    aL_2) (* 1 aO_1)) 0))
  (assert (= (+ (* 1 aL_0) (*
    57896044618658097711785492504343953926549254372227246365156541811699034343326
    aR_2)) 0))
  (assert (= (+ (* 1 aR_0) (*
    57896044618658097711785492504343953926549254372227246365156541811699034343326
    aL_3)) 0))
  (assert (= (+ (* 1 aR_0) (*
    57896044618658097711785492504343953926549254372227246365156541811699034343326
    aR_3)) 0))
  (assert (= (+ (+ (* 3 aL_0) (*
    57896044618658097711785492504343953926549254372227246365156541811699034343326
    aO_2)) (* 1 aO_3))
    50691664119640283727448954162351551669994268339720539671652090628799494505816))
  (assert (= (* aL_1 aR_1) aO_1))
  (assert (= (* aL_2 aR_2) aO_2))
  (assert (= (* aL_3 aR_3) aO_3))
(end-module)
```

### 6.4.6 V-FCMP-SPEC-006: Incomplete Curve Addition

| | | | |
|---|---|---|---|
| **Gadget** | Incomplete Addition | **Status** | Verified by Picus |
| **Configuration** | Solver: multi-solver, Solver timeout: 60s | | |
| **Variants** | Selene, Helios, with and without on-curve assumptions. | | |

This section describes formal verification of Incomplete Addition gadget [4, Section 4.3.6]. In the codebase and paper, it is assumed that the input points $(b_x, b_y)$ and $(c_x, c_y)$ are constrained to be on the curve. In the annotated Picus code below, this is assumed explicitly by using `assume` statements, generated from the same on-curve circuit used for V-FCMP-SPEC-005. However, the Veridise analysts additionally verified the circuit to be deterministic the `assume` statements removed. The generator of each curve was used as the fixed point $a$.

```
// Build variables
let a = BaseCurve::generator();
let a = BaseCurve::to_xy(a).expect("Generator is on curve");
let b = (Variable::aL(0), Variable::aL(1));
let c = (Variable::aR(0), Variable::aR(1));
let num_predefined_vars = 2;

// Add on-curve assumptions
let curve = CurveSpec { a: BaseCurve::a(), b: BaseCurve::b() };
let mut on_curve_circuit: Circuit<C> = Circuit::<C>::empty(num_predefined_vars);
let b = on_curve_circuit.on_curve(&curve, b);
```

```
let c = on_curve_circuit.on_curve(&curve, c);

// Constrain addition
let mut addition_circuit: Circuit<C> = Circuit::<C>::empty(on_curve_circuit.muls());
addition_circuit.incomplete_add_fixed(a, b, c);

// Return the circuit along with input variables (a is fixed, b is input, and c is
    output).
PicusInputs {
  assume_circuits: vec![on_curve_circuit],
  assert_circuits: vec![addition_circuit],
  input_vars: vec![b.x(), b.y()],
}
```

**Snippet 6.10:** Generate Incomplete addition verification circuit

### Picus (Annotated)

**Constraints.** Below is an annotated version of the constraints passed to Picus. None of the annotations are used by Picus, they are only so that the user may additionally validate the translation performed on the circuit code.

```
(prime-number
    57896044618658097711785492504343953926634992332820282019728792003956564819949)
(begin-module ec_incomplete_add_fixed)
    ; x1
  (input aL_0)
  ; y1
  (input aL_1)
  (output aL_10)
  (output aL_11)
  (output aL_2)
  (output aL_3)
  (output aL_4)
  (output aL_5)
  (output aL_6)
  (output aL_7)
  (output aL_8)
  (output aL_9)
  (output aO_10)
  (output aO_11)
  (output aO_2)
  (output aO_3)
  (output aO_4)
  (output aO_5)
  (output aO_6)
  (output aO_7)
  (output aO_8)
  (output aO_9)
  ; y1
  (output aR_0)
  ; y2
  (output aR_1)
```

```
31 (output aR_10)
32 (output aR_11)
33 (output aR_2)
34 (output aR_3)
35 (output aR_4)
36 (output aR_5)
37 (output aR_6)
38 (output aR_7)
39 (output aR_8)
40 (output aR_9)
41 ; Helios: y^2 = x^3 - 3*x +
     15789920373731020205926570676277057129217619222203920395806844808978996083412
     over GF(p)
42 ;         y^2 = x^3 + a*x + b
43 ;
44 ;
45 ; On-curve assumptions for (L0, L1) = (x1, y1) with fresh variables 2-4
46 ;
47 ; L2 := L0 = x
48 (assume (= (+ (* 1 aL_0) (*
     57896044618658097711785492504343953926634992332820282019728792003956564819948
     aL_2)) 0))
49 ;
50 ; R2 := L0 = x
51 (assume (= (+ (* 1 aL_0) (*
     57896044618658097711785492504343953926634992332820282019728792003956564819948
     aR_2)) 0))
52 ;
53 ; L3 := O2 = R2 * L2 = x * x
54 (assume (= (+ (*
     57896044618658097711785492504343953926634992332820282019728792003956564819948
     aL_3) (* 1 aO_2)) 0))
55 ;
56 ; R3 := L0 = x
57 ; so
58 ; O3 = L3 * R3 = x^2 * x = x^3
59 (assume (= (+ (* 1 aL_0) (*
     57896044618658097711785492504343953926634992332820282019728792003956564819948
     aR_3)) 0))
60 ;
61 ; L4 := L1 = y
62 (assume (= (+ (* 1 aL_1) (*
     57896044618658097711785492504343953926634992332820282019728792003956564819948
     aL_4)) 0))
63 ;
64 ; R4 := L1 = y
65 ; so O4 = L4 * R4 = y^2
66 (assume (= (+ (* 1 aL_1) (*
     57896044618658097711785492504343953926634992332820282019728792003956564819948
     aR_4)) 0))
67  ;
68 ; 3 * L0 - O3 + O4 = b
69 ; -a * x - x^3 + y^2 = b
```

```
70  (assume (= (+ (+ (* 3 aL_0) (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aO_3)) (* 1 aO_4))
      15789920373731020205926570676277057129217619222203920395806844808978996083412))
71  ;
72  ; On-curve assumptions for (R0, R1) = (x2, y2), same as above, but with fresh
      variables 5-7
73  ;
74  (assume (= (+ (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aL_5) (* 1 aR_0)) 0))
75  (assume (= (+ (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aR_5) (* 1 aR_0)) 0))
76  (assume (= (+ (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aL_6) (* 1 aO_5)) 0))
77  (assume (= (+ (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aR_6) (* 1 aR_0)) 0))
78  (assume (= (+ (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aL_7) (* 1 aR_1)) 0))
79  (assume (= (+ (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aR_7) (* 1 aR_1)) 0))
80  (assume (= (+ (+ (*
      57896044618658097711785492504343953926634992332820282019728792003956564819948
      aO_6) (* 1 aO_7)) (* 3 aR_0))
      15789920373731020205926570676277057129217619222203920395806844808978996083412))
81  (assume (= (* aL_2 aR_2) aO_2))
82  (assume (= (* aL_3 aR_3) aO_3))
83  (assume (= (* aL_4 aR_4) aO_4))
84  (assume (= (* aL_5 aR_5) aO_5))
85  (assume (= (* aL_6 aR_6) aO_6))
86  (assume (= (* aL_7 aR_7) aO_7))
87  ;
88  ; Begin curve addition using fresh variables 8-11
89  ;
90  ; Constants:
91  ;
92  ; -1 =
      57896044618658097711785492504343953926634992332820282019728792003956564819948
93  ;
94  ; Helios generator for https://gist.github.com/tevador/4524
      c2092178df08996487d4e272b096
95  ; G = [Gx, Gy] := [3,
      37760095087190773158272406437720879471285821656958791565335581949097084993268]
96  ;                = [3,
      -20135949531467324553513086066623074455349170675861490454393210054859479826681]
97  ;
98  ; aL_8 := aL_0 - 3
99  ;
```

```
100  (assert (= (+ (* 1 aL_0) (*
       57896044618658097711785492504343953926634992332820282019728792003956564819948
       aL_8)) 3))
101  ;
102  ; aO_8 := 1
103  ; This finishes the inequality constraint for (aL_0 != 3)
104  (assert (= (* 1 aO_8) 1))
105  ;
106  ; R_9 := x1 - x0 = aL_0 - 3
107  (assert (= (+ (* 1 aL_0) (*
       57896044618658097711785492504343953926634992332820282019728792003956564819948
       aR_9)) 3))
108   ;
109  ; O_9 - L_1 == -Gy
110  ; i.e. O_9 = L_1 - Gy = y1 - y0
111  ;
112  ; We have then the
113  ; L9 * R9 == O9
114  ; corresponds to
115  ; L9 * (x1 - x0) == y1 - y0
116  ; i.e. L9 is the slope
117  ;
118  (assert (= (+ (*
       57896044618658097711785492504343953926634992332820282019728792003956564819948
       aL_1) (* 1 aO_9))
       20135949531467324553513086066623074455349170675861490454393210054859479826681))
119  ;
120  ; L10 := L9 = slope
121  (assert (= (+ (* 1 aL_9) (*
       57896044618658097711785492504343953926634992332820282019728792003956564819948
       aL_10)) 0))
122  ;
123  ; R10 := R0 - 3 = x2 - x0
124  (assert (= (+ (*
       57896044618658097711785492504343953926634992332820282019728792003956564819948
       aR_10) (* 1 aR_0)) 3))
125  ;
126  ; O10  = -Gy - R1 = -y2 - y0
127  ; so slope * (x2 - x) == -y2 - y0
128  (assert (= (+ (* 1 aO_10) (* 1 aR_1))
       20135949531467324553513086066623074455349170675861490454393210054859479826681))
129  ;
130  ; L11 := L9 = slope
131  (assert (= (+ (* 1 aL_9) (*
       57896044618658097711785492504343953926634992332820282019728792003956564819948
       aL_11)) 0))
132  ;
133  ; R11 := L9 = slope
134  (assert (= (+ (* 1 aL_9) (*
       57896044618658097711785492504343953926634992332820282019728792003956564819948
       aR_11)) 0))
135  ;
136  ; O11 = L0 + R0 + 3 = x1 + x2 + x0
```

```
137   ; So slope * slope == x0 + x1 + x2
138   (assert (= (+ (+ (*
        57896044618658097711785492504343953926634992332820282019728792003956564819948
        aL_0) (* 1 aO_11)) (*
        57896044618658097711785492504343953926634992332820282019728792003956564819948
        aR_0)) 3))
139   ;
140   ; quadratic constraints
141   (assert (= (* aL_2 aR_2) aO_2))
142   (assert (= (* aL_3 aR_3) aO_3))
143   (assert (= (* aL_4 aR_4) aO_4))
144   (assert (= (* aL_5 aR_5) aO_5))
145   (assert (= (* aL_6 aR_6) aO_6))
146   (assert (= (* aL_7 aR_7) aO_7))
147   (assert (= (* aL_8 aR_8) aO_8))
148   (assert (= (* aL_9 aR_9) aO_9))
149   (assert (= (* aL_10 aR_10) aO_10))
150   (assert (= (* aL_11 aR_11) aO_11))
151 (end-module)
```

### 6.4.7 V-FCMP-SPEC-007: Tuple Member of List

| Gadget | Inverse | Status | Verified by Picus |
| --- | --- | --- | --- |
| Configuration | Solver: multi-solver, Solver timeout: 60s | | |
| Variants | Selene, Helios, list lengths 1-8, tuple lengths 1-8 | | |

This section describes formal verification of Tuple Member of List gadget [4, Section 4.4.1]. The Veridise analysts verified instantiations of this circuit with all possible list and tuple lengths ranging from one through eight for both the Selene and Helios curves. The list of length 2, tuple of length 2 case is shown below.

A fixed random challenge was used. In particular, this does not prove anything about the interactive nature of the protocol, only that no intermediate computations are under-constrained. In particular, for this particular fixed challenge, all variables are well-defined. See Section 7.1.2 for a full proof of the security of this gadget.

```
1  assert!(list_length > 0);
2  assert!(tuple_length > 0);
3
4  // Build variables
5  let list = (0..list_length)
6    .map(|i| {
7      (0..tuple_length).map(|j| Variable::CG { commitment: i, index: j }).collect::<Vec
       <_>>()
8    })
9    .collect::<Vec<Vec<_>>>();
10 let maybe_member_var = (0..tuple_length)
11   .map(|j| Variable::CG { commitment: list_length, index: j })
12   .collect::<Vec<_>>();
13 let all_vars =
```

```
  list.iter().cloned().chain(Some(maybe_member_var.clone())).flatten().collect::<Vec<
    _>>();

// Add membership check
let num_predefined_vars = 0;
let dummy_proof = [];
let transcript = dummy_transript(&dummy_proof);
let tuple_member_of_list_circuit: Circuit<C> = Circuit::<C>::empty(
    num_predefined_vars);
let tuple_member_of_list_circuit = constrain_tuple_member_of_list::<C, _>(
  transcript,
  tuple_member_of_list_circuit,
  maybe_member_var.into(),
  list,
);

// Return the circuit along with input variable (the point)
PicusInputs {
  assume_circuits: vec![],
  assert_circuits: vec![tuple_member_of_list_circuit],
  num_unconstrained_rows: num_predefined_vars,
  input_vars: all_vars,
}
```

**Snippet 6.11:** Generate tuple member of list verification circuit

```
(prime-number
    57896044618658097711785492504343953926634992332820282019728792003956564819949)
(begin-module tuple_member_of_list_2_2_Selene)
  (input aCG_0_0)
  (input aCG_0_1)
  (input aCG_1_0)
  (input aCG_1_1)
  (input aCG_2_0)
  (input aCG_2_1)
  (output aL_0)
  (output aO_0)
  (output aR_0)
  (assert (= (+ (+ (+ (+ (*
    48987924366988163166499717813511560407789668924147703997325607327024239626881
    aCG_0_0) (*
    54907593356499508694473682620962583587923461388726878955500069801262130156242
    aCG_0_1)) (*
    8908120251669934545285774690832393518845323408672578022403184676932325193068
    aCG_2_0)) (*
    2988451262158589017311809883381370338711530944093403064228722202694434663707
    aCG_2_1)) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aL_0)) 0))
  (assert (= (+ (+ (+ (+ (*
    48987924366988163166499717813511560407789668924147703997325607327024239626881
    aCG_1_0) (*
    54907593356499508694473682620962583587923461388726878955500069801262130156242
    aCG_1_1)) (*
```

```
    8908120251669934545285774690832393518845323408672578022403184676932325193068
    aCG_2_0)) (*
    2988451262158589017311809883381370338711530944093403064228722202694434663707
    aCG_2_1)) (*
    57896044618658097711785492504343953926634992332820282019728792003956564819948
    aR_0)) 0))
14  (assert (= (* 1 aO_0) 0))
15  (assert (= (* aL_0 aR_0) aO_0))
16 (end-module)
```

**Snippet 6.12:** Tuple member of list (length 2, tuple-length 2) verification circuit

# 7 Soundness Proofs

## 7.1 Interactive Gadgets.

In addition to the non-interactive gadgets Monero FCMP++ includes two interactive gadgets, `Tuple Member of List` and `Discrete Log Proof`.

### 7.1.1 Discrete Logarithm Gadget

The discrete logarithm gadget is based on ideas introduced by Eagen [7]. Points (with multiplicities) on an elliptic curve can be shown to add up to zero (the identity point of the elliptic curve group) precisely when there is a function on the curve vanishing exactly at the given points with the same multiplicities and having poles only at zero. Such a function is called a divisor witness and has the form $a(x) - y \cdot b(x)$, for $a(x)$ and $b(x)$ polynomial expressions in $x$. The vanishing of the divisor witness at the points with the given multiplicities can be turned into a question about univariate polynomials by projecting everything onto a random line chosen by the verifier. The corresponding proof for univariate polynomials can then be handled by the use of the Schwartz–Zippel Lemma.The choice of the random line and the point of evaluation for the Schwartz–Zippel Lemma can be combined into a single step in a way tackles questions about rationality as well. This yields a slightly different probability distribution for the challenge space. A detailed description of the protocol and a soundness proof have previously been provided by Veridise [2].

The protocol can be made even more efficient by the use of logarithmic derivatives. A proof for this and a corresponding security analysis has been provided by Veridise [1].

Discrete logarithm relations form a special case of proofs of zero sums for points on elliptic curves. The discrete logarithm gadget described in [4, Section 4.4.2] and utilised in Monero FCMP++ precisely corresponds to this case. The soundness of the resulting interactive protocol for proof of knowledge of discrete logarithm was established by Veridise in [3].

For further details we refer to [1–3].

### 7.1.2 Tuple Member of List Gadget

The interactive Tuple Member of List Gadget described in [4, Section 4.4.1] allows to prove membership of a tuple in a given list. Below we provide a soundness proof for the interactive protocol. For definitions and details on $k$-special soundness, see [14].


© 2025 Veridise Inc.

Prover $\mathrm{P}(\mathscr{C}_{TML}, M)$ | Verifier $\mathrm{V}(\mathscr{C}_{TML})$

$\xrightarrow{M}$

$R = (R_1, \ldots, R_n) \leftarrow_\$ \mathbb{F}_q^n$

$\xleftarrow{R}$

Compute:
$m = \sum_{i=1}^{n} R_i \cdot M_i$
$u_j = \sum_{i=1}^{n} R_i \cdot L_{j,i}$ for $j = 1, \ldots T$
$f_j = \prod_{i=1}^{j} (m - u_j)$ for $j = 1, \ldots T$.

$\xrightarrow{(f_1,\ldots,f_n)}$

Compute:
$m = \sum_{i=1}^{n} R_i \cdot M_i$
$u_j = \sum_{i=1}^{n} R_i \cdot L_{j,i}$ for $j = 1, \ldots T$
Check:
$f_1 = (m - u_1)$
$f_j = f_{j-1} \cdot (m - u_j)$ for $j = 1, \ldots T$
$f_T = 0$

### Configuration

The configuration $\mathscr{C}_{TML}$ consists of the following information and is agreed upon by the prover and the verifier before the protocol: A finite field $\mathbb{F}_q$, the length $n \in \mathbb{N}$ of each of the tuples in the list, the list of tuples $L \subset \mathbb{F}_q^n$, and the number of tuples in the list $T = |L|$. We have $L = \{L_1, \ldots, L_T\}$ and $L_i = (L_{i,1}, \ldots, L_{i,n})$ for $i = 1, \ldots T$.

### Public Input

There is no public input to the protocol.

### Witness

The witness is the element $M = (M_1, \ldots, M_n) \in \mathbb{F}_q^n$ claimed to be in the list. The witness is only known to the verifier.

### Relation

The protocol will provide a proof for the relation

$$\mathscr{R}_{TML} = \{(\cdot; M) | M \in L\}.$$

### 7.1.3 Protocol for relation $\mathcal{R}_{TML}$

The following is a protocol for the relation $\mathcal{R}_{TML}$. We give a proof of special soundness of the protocol. As is classically known, this implies knowledge soundness of the protocol.

It is a 3 move public coin protocol, with one challenge from the verifier drawn randomly from a set of size $q^n$. The checks performed by the verifier have degree at most 2.

**Theorem 7.1.1** *The protocol $\Pi$ for the relation $\mathcal{R}_{TML}$ is $k$-out-of-$N$-special-sound, where $k = q^{n-1} \cdot T + 1$ and $N = q^n$.*

*Proof.* Consider the extractor returning $M = (M_1, \ldots, M_n)$. We have to prove that

$$M \in L.$$

We will show that $k = q^{n-1} \cdot T + 1$ accepting transcripts with common first message $M$ and distinct challenges $R^{(j)}$, $j = 1, \ldots, k$ imply $M \in L$.

The check $f_T = 0$ amounts to checking

$$0 = \prod_{i=1}^{T} f_i = \prod_{i=1}^{T} (m - u_i) = \prod_{i=1}^{T} \big(\langle R, M\rangle - \langle R, L_i\rangle\big) = \prod_{i=1}^{T} \langle R, M - L_i\rangle. \tag{7.1}$$

Let $M - L_i = (c_1^{(i)}, \ldots, c_n^{(i)})$ and define the multivariate polynomial

$$p(X_1, \ldots, X_n) = \prod_{i=1}^{T} \big(c_1^{(i)} X_1 + \cdots + c_n^{(i)} X_n\big).$$

Equation (7.1) hence corresponds to

$$p(R^{(j)}) = 0 \quad \text{for } j = 1, \ldots, k. \tag{7.2}$$

The zero locus of the polynomial $p(X_1, \ldots, X_n)$ is a union of $T$ hyperplanes through the origin and hence has less than $k - 1 = T \cdot q^{n-1}$ points over $\mathbb{F}_q$. So Equation (7.2) implies that one of the linear factors $(c_1^{(i)} X_1 + \cdots + c_n^{(i)} X_n)$ must be identically zero. Hence $(c_1^{(i)}, \ldots, c_n^{(i)}) = M - L_i = (0, \ldots, 0)$, i.e., $M = L_i$ and hence $M \in L$, as was to be shown. □

As a $k$-out-of-$N$ special sound 3-move sigma protocol is knowledge sound with knowledge error

$$\kappa \leq \frac{k-1}{N},$$

we obtain the following result:

**Corollary 7.1.2** *The protocol $\Pi$ is knowledge sound with knowledge error*

$$\kappa \leq \frac{T}{q} = \frac{|L|}{q}$$

**Base field** $\mathbb{F}_q$ of an elliptic curve is the field over which the curve is defined [15]. 1

**Curve cycle** A (2-)cycle of curves is a pair of elliptic curves such that the base field of each curve is the scalar field of the other. 1

**Fiat-Shamir** A well-known method for converting interactive proofs to non-interactive ones [16]. See `https://en.wikipedia.org/wiki/Fiat-Shamir_heuristic` to learn more. 2

**Generalized Bulletproofs** A variant of the Bulletproofs arithmetic circuit statement to support Pedersen vector commitments, see [17]. 2

**Merkle Trees** A cryptographic commitment to a list of values which can be opened at individual entries in the list. See `https://en.wikipedia.org/wiki/Merkle_tree` to learn more. 1

**Monero** Monero is a privacy-focused, decentralized cryptocurrency. Monero's blockchain is designed to be opaque: transaction details-including sender, receiver, and amount-are hidden by default. Monero uses a proof-of-work consensus mechanism and is designed to be resistant to ASIC mining. 1

**Pedersen committment** An (information-theoretically) hiding and computationally binding commitment scheme based on the difficulty of the discrete logarithm. 1

**Satisfiability Modulo Theories** The problem of determining whether a certain mathematical statement has any solutions. SMT solvers attempt to do this automatically. See `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories` to learn more. 33

**Scalar field** $\mathbb{F}_r$ is the field whose cardinality equals the size of the (prime-order) elliptic curve. See [15] for further explanation. 1

**Schwartz-Zippel** The Schwartz-Zippel lemma bounds the probability that a polynomial of degree at most $d$ evaluates to zero at a point chosen randomly from set $S$ [18, 19]. 1

**SMT** Satisfiability Modulo Theories. 13

**Unspent Transaction Output** An unspent transaction output is the accounting model used by Bitcoin. See `https://en.wikipedia.org/wiki/Unspent_transaction_output` to learn more. 33

**UTXO** Unspent Transaction Output. 1

**Wei25519** Short Weierstrass model for the Ed25519 curve. For details see `http://ed25519.cr.yp.to`. 1


© 2025 Veridise Inc.

# Bibliography


[1] Alp Bassa. *On the Use of Logarithmic Derivatives in Eagen's Proof of Sums of Points*. URL: https://repo.getmonero.org/-/project/54/uploads/bfe9f49326a843ef1c9466e30a5d42c8/VAR_Monero_Logarithmic_Derivatives_Final.pdf (cited on pages 1, 30).

[2] Alp Bassa. *Soundness Proof for Eagen's Proof of Sums of Points*. Technical report available on Monero Research Lab. 2024. URL: https://moneroresearch.info/index.php?action=attachments_ATTACHMENTS_CORE&method=downloadAttachment&id=222&resourceId=229&filename=4caad9da11edbb69960fc02be5718c2837d8e57f (cited on pages 1, 30).

[3] Alp Bassa. *Soundness Proof for an Interactive Protocol for the Discrete Logarithm Relation*. Technical report available on Monero Research Lab. 2025. URL: https://moneroresearch.info/index.php?action=attachments_ATTACHMENTS_CORE&method=downloadAttachment&id=252&resourceId=259&filename=8783379c16a55920d0ea0f102ba99a02fae7b077 (cited on pages 1, 30).

[4] Luke 'Kayaba' Parker. *FCMP++ Paper*. https://github.com/kayabaNerve/fcmp-plus-plus-paper/blob/develop/fcmp++.pdf. Accessed: 2025-05-02. 2024 (cited on pages 1, 5, 15, 16, 18, 21, 22, 27, 30).

[5] Monero Project. *Full-Chain Membership Proofs Development*. Accessed: 2025-05-02. Apr. 2024. URL: https://www.getmonero.org/2024/04/27/fcmps.html (cited on page 1).

[6] Matteo Campanelli, Mathias Hall-Andersen, and Simon Holmgaard Kamp. *Curve Trees: Practical and Transparent Zero-Knowledge Accumulators*. Cryptology ePrint Archive, Paper 2022/756. 2022. URL: https://eprint.iacr.org/2022/756 (cited on page 1).

[7] Liam Eagen. *Zero Knowledge Proofs of Elliptic Curve Inner Products from Principal Divisors and Weil Reciprocity*. Cryptology ePrint Archive, Paper 2022/596. 2022. URL: https://eprint.iacr.org/2022/596 (cited on pages 1, 30).

[8] Benedikt Bünz et al. 'Bulletproofs: Short Proofs for Confidential Transactions and More'. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020 (cited on page 1).

[9] Simon Kamp. *Generalized Bulletproofs*. https://github.com/simonkamp/curve-trees/blob/main/bulletproofs/generalized-bulletproofs.md. Accessed: 2025-05-02. 2025 (cited on page 1).

[10] tevador. *Elliptic curve tower-cycle for Curve25519*. https://gist.github.com/tevador/4524c2092178df08996487d4e272b096. Accessed: 2025-05-02. 2024 (cited on pages 1, 9).

[11] Quang Dao et al. *Weak Fiat-Shamir Attacks on Modern Proof Systems*. Cryptology ePrint Archive, Paper 2023/691. 2023. URL: https://eprint.iacr.org/2023/691 (cited on page 6).

[12] Dmitry Khovratovich, Ron D. Rothblum, and Lev Soukhanov. *How to Prove False Statements: Practical Attacks on Fiat-Shamir*. Cryptology ePrint Archive, Paper 2025/118. 2025. URL: https://eprint.iacr.org/2025/118 (cited on page 6).


 © 2025 Veridise Inc.

[13] Daniel J. Bernstein. ‘Curve25519: New Diffie-Hellman Speed Records’. In: *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*. Ed. by Moti Yung et al. Vol. 3958. Lecture Notes in Computer Science. Springer, 2006, pp. 207–228. DOI: 10.1007/11745853_14 (cited on page 9).

[14] Thomas Attema, Ronald Cramer, and Lisa Kohl. ‘A Compressed \Sigma-Protocol Theory for Lattices’. In: *Advances in Cryptology – CRYPTO 2021*. Ed. by Tal Malkin and Chris Peikert. Cham: Springer International Publishing, 2021, pp. 549–579 (cited on page 30).

[15] Timofey. *Elliptic Curves: Cheat Sheet*. https://hackmd.io/@timofey/rJ8HP8Yaj. Accessed: 2025-05-07. 2023 (cited on page 33).

[16] Amos Fiat and Adi Shamir. ‘How To Prove Yourself: Practical Solutions to Identification and Signature Problems’. In: *Advances in Cryptology — CRYPTO’ 86*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194 (cited on page 33).

[17] Cypher Stack. *Generalized Bulletproofs Code*. https://github.com/cypherstack/generalized-bulletproofs-code. Accessed: 2025-05-02. 2024 (cited on page 33).

[18] J. T. Schwartz. ‘Fast Probabilistic Algorithms for Verification of Polynomial Identities’. In: *Journal of the ACM* 27.4 (1980), pp. 701–717. DOI: 10.1145/322217.322225 (cited on page 33).

[19] Richard Zippel. ‘Probabilistic algorithms for sparse polynomials’. In: *Symbolic and Algebraic Computation*. Ed. by Edward W. Ng. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 216–226 (cited on page 33).

© 2025 Veridise Inc. 