Google Longfellow

Security Assessment

August 25, 2025

Prepared for:
abhi shelat and Lingzi Xue
Google

Prepared by: **Joe Doyle and Marc Ilunga**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Kimberly Espinoza, Project Manager
kimberly.espinoza@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 27, 2025	Pre-project kickoff call
July 14, 2025	Status update meeting #1
July 21, 2025	Report readout meeting
August 13, 2025	Completion of fix review
August 25, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Google engaged Trail of Bits to review the security of its Longfellow library, which enables the building of privacy-preserving, zero-knowledge protocols for legacy identity verification standards.

A team of two consultants conducted the review from July 7 to July 18, 2025, for a total of four engineer-weeks of effort. Our review and testing efforts focused on the components involved in the anonymous-credential implementation, especially the mdoc circuit, the sumcheck and Ligero protocols, and the composed "ZK" protocol. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

Overall, the Longfellow library is a high-quality implementation of a sumcheck-based zero-knowledge proof system. However, the circuits and other functionality outside this core appear to be less well tested and less mature by comparison. This is evident in issues we identified, including a high-severity finding that allows arbitrary forgery of mdoc attribute claims (TOB-LIBZK-10) and an underconstrained lookup index in the ECDSA verification circuit (TOB-LIBZK-7).

We also found several issues that can lead to unexpected or incorrect behavior if parts of the library are used in unintended ways, including misidentification of the circuit being proven (TOB-LIBZK-1), accepting of improper Merkle proofs (TOB-LIBZK-4, TOB-LIBZK-9), unexpected rejection of valid proofs (TOB-LIBZK-5), and incorrect multi-circuit composition (TOB-LIBZK-12).

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Google take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.

- **Develop additional testing strategies and tools for circuit implementations.** The most severe finding (TOB-LIBZK-10) is a good example of an underconstrained witness bug. Whenever any part of a circuit's witness can be modified without causing it to be rejected, that should be carefully analyzed to determine whether it is exploitable. Although the final analysis will be manual, it should be possible to discover "malleable" witness positions with automatic testing, which would help detect and prevent issues like this in the future.

- **Define and document the library's public API.** Several findings relate to undocumented assumptions in functions that may not be intended as part of the public API. A clear public API with explicit usage requirements would mitigate these issues.

- **Perform an additional security review.** Due to the limited time available for this engagement, some components of the library were not comprehensively reviewed. We recommend performing a full review of the remaining components to minimize the risk of remaining vulnerabilities.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	0
Low	2
Informational	8
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Configuration	1
Cryptography	9
Data Validation	3

Project Goals

The engagement was scoped to provide a security assessment of Google's Longfellow library. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is the zero-knowledge proof system verifier implemented soundly?
- Are the parameters of the proof system set appropriately to achieve the desired security level?
- Do the circuits correctly implement the desired functions?
    - In particular, are all inputs properly constrained to prevent a malicious prover from bypassing desired checks?
- What security properties does the zero-knowledge proof system guarantee?

Project Targets

The engagement involved reviewing and testing the following target.

longfellow-zk

Repository	https://github.com/google/longfellow-zk
Version	`981a349fad7eee38db94734e99718be052ad20ed`
Type	C++
Platform	Multiple

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Calculation and review of test coverage results
- Semgrep code linting
- Manual code review, focused on the following:
    - Sumcheck, Ligero, and combined zero-knowledge verification
    - The ECDSA, MAC, and SHA-256 verification circuits
    - The mdoc attribute parsing circuit, primarily evaluating its soundness
    - Bit-logic circuit gadgets
    - Core cryptographic primitives, including the Fiat-Shamir transcript and Merkle tree implementations

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- Non-mdoc credential circuits in `jwt/` and `anoncred/`
- The `mdoc_1f` circuit
- The CBOR parsing circuits
- The circuit compiler
- The SHA-3 circuit
- Arithmetic primitive implementations (i.e., the finite field, elliptic curve, and FFT implementations, and other functionality in the `algebra/` folder)

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
lcov	The LLVM code coverage analyzer

Areas of Focus

Our automated testing and verification work focused on the following:

- Are there any problematic patterns in the codebase?
- Are there any untested or weakly tested code paths that are vulnerable?

Test Results

The results of this focused testing are detailed below.

Test Coverage

The overall coverage of the library is quite high, but some of the uncovered paths are potential useful targets for additional tests, especially testing cases that should be rejected. The high-level coverage summary is shown below.

LCOV - code coverage report

			Coverage	Total	Hit
Current view:	top level				
Test:	coverage.info	**Lines:**	97.9 %	9099	8909
Test Date:	2025-07-21 07:41:11	**Functions:**	90.2 %	5017	4526

Directory	Line Coverage Rate		Line Coverage Total	Line Coverage Hit	Function Coverage Rate	Function Coverage Total	Function Coverage Hit
algebra		99.3 %	1318	1309	91.2 %	1559	1422
arrays		100.0 %	216	216	88.5 %	165	146
cbor		85.1 %	141	120	91.7 %	12	11
circuits/anoncred		100.0 %	110	110	100.0 %	14	14
circuits/base64		100.0 %	342	342	100.0 %	12	12
circuits/cbor_parser		95.1 %	509	484	100.0 %	128	128
circuits/compiler		98.4 %	627	617	84.5 %	446	377
circuits/ecdsa		99.7 %	307	306	100.0 %	18	18
circuits/jwt		97.9 %	189	185	100.0 %	20	20
circuits/logic		99.6 %	1168	1163	92.2 %	1301	1199
circuits/mac		100.0 %	121	121	100.0 %	22	22
circuits/mdoc		94.1 %	937	882	98.8 %	85	84
circuits/sha		100.0 %	284	284	87.9 %	182	160
circuits/sha3		97.7 %	130	127	100.0 %	12	12
ec		100.0 %	338	338	87.8 %	49	43
gf2k		99.7 %	379	378	99.4 %	156	155
ligero		96.4 %	411	396	78.2 %	147	115
merkle		100.0 %	133	133	100.0 %	21	21
proto		96.0 %	173	166	100.0 %	51	51
random		100.0 %	138	138	81.4 %	59	48
sumcheck		96.0 %	451	433	82.1 %	319	262
util		97.7 %	171	167	100.0 %	39	39
zk		97.6 %	506	494	83.5 %	200	167

Generated by: LCOV version 2.0-1

Figure 1: Test coverage report from lcov

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Circuit ID is not checked during circuit deserialization	Data Validation	High
2	Collision of transcript separation tags	Cryptography	Informational
3	FSPRF does not limit the size of the output stream	Cryptography	Informational
4	MerkleTreeVerifier::verify_proof is vulnerable to path extension	Cryptography	Informational
5	COSE1 length values are incorrectly serialized	Data Validation	Low
6	Ligero parameter search can be improved	Configuration	Informational
7	ECDSA circuit allows off-curve intermediate points	Cryptography	Undetermined
8	The specification describes an incorrect quadratic test	Cryptography	Informational
9	MerkleCommitmentVerifier::verify_compressed_proof assumes nonrepeating indices	Cryptography	Informational
10	mdoc attribute check can be bypassed	Data Validation	High
11	ECDSA witness-building timing may leak hidden witness values	Cryptography	Low
12	MAC scheme is vulnerable to existential forgery on input zero and may break zero-knowledge in other uses of the library	Cryptography	Informational

13	Ligero matrix construction deviates from the specification	Cryptography	**Informational**

Detailed Findings

1. Circuit ID is not checked during circuit deserialization	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LIBZK-1
Target: `lib/proto/circuit.h`	

Description

The combined zero-knowledge protocol combines a sumcheck-based protocol for layered circuits and the Ligero MPC-in-the-head protocol for proving the satisfiability of arithmetic circuits. In the combined protocol, the messages of the sumcheck protocol are used to compute the circuit proven with Ligero. This protocol is then made noninteractive through the Fiat-Shamir heuristic, where random challenges are generated by hashing all previous messages. In particular, the circuit and the public inputs, which together represent the statement to be proven, must be included in the transcript to ensure soundness.

In the verifier, shown in figure 1.1, the transcript is initialized in `initialize_sumcheck_fiat_shamir`, then a sumcheck verifier is run via the `verifier_constraints` function, and finally the Ligero verifier is run against the resulting set of constraints in A and b.

```
// Verifies the proof.
bool verify(const ZkProof<Field>& zk, const Dense<Field>& pub,
            Transcript& tv) const {
  log(INFO, "verifier: verify");

  ZkCommon<Field>::initialize_sumcheck_fiat_shamir(tv, circ_, pub, f_);

  // Derive constraints on the witness.
  using Llc = LigeroLinearConstraint<Field>;
  std::vector<Llc> A;
  std::vector<Elt> b;
  const LigeroHash hash_of_A{0xde, 0xad, 0xbe, 0xef};
  size_t cn = ZkCommon<Field>::verifier_constraints(circ_, pub, zk.proof,
                                                    /*aux=*/nullptr, A, b, tv,
                                                    n_witness_, f_);

  const char* why = "";
  bool ok = LigeroVerifier<Field, RSFactory>::verify(
      &why, param_, zk.com, zk.com_proof, tv, cn, A.size(), &A[0], hash_of_A,
      &b[0], &lqc_[0], rsf_, f_);
```

```
  log(INFO, "verify done: %s", why);
  return ok;
}
```

Figure 1.1: The combined verifier (longfellow-zk/lib/zk/zk_verifier.h#61–84)

The layered circuit is added to the transcript via its “ID,” which is intended to be a hash of the circuit’s description, as shown in figure 1.2.

```
// append public parameters to the FS transcript
static void initialize_sumcheck_fiat_shamir(Transcript& ts,
                                            const Circuit<Field>& circuit,
                                            const Dense<Field>& pub,
                                            const Field& F) {
  ts.write(circuit.id, sizeof(circuit.id));

  // Public inputs:
  for (size_t i = 0; i < circuit.npub_in; ++i) {
    ts.write(pub.at(i), F);
  }

  // Outputs pro-forma:
  ts.write(F.zero(), F);

  // Enough zeroes for correlation intractability, one byte
  // per term.
  ts.write0(circuit.nterms());
}
```

Figure 1.2: The circuit is represented by its ID field in the transcript.
(lib/zk/zk_common.h#162–180)

If a circuit is constructed via the `mkcircuit` function, shown in figure 1.3, its ID field will be populated by the `circuit_id` function, which computes a SHA-256-based hash of the layered circuit.

```
std::unique_ptr<Circuit<Field>> mkcircuit(size_t nc) {
  size_t depth_ub = compute_depth_ub();
  fixup_last_layer_assertions(depth_ub);
  compute_needed(depth_ub);

  Scheduler<Field> sched(nodes_, f_);
  std::unique_ptr<Circuit<Field>> c =
      sched.mkcircuit(constants_, depth_ub, nc);

  // re-export the scheduler telemetry
  nwires_ = sched.nwires_;
  nquad_terms_ = sched.nquad_terms_;
  nwires_overhead_ = sched.nwires_overhead_;
```

```
  c->ninputs = ninput();
  c->npub_in = npub_input_;
  c->subfield_boundary = subfield_boundary_;

  circuit_id(c->id, *c, f_);
  return c;
}
```

Figure 1.3: Circuit ID calculation in `mkcircuit`
(`lib/circuits/compiler/compiler.h#245–265`)

However, if the circuit is deserialized, this ID is read from the file without additional checks, as shown in figure 1.4.

```
// Read the circuit name from the serialization.
if (!buf.have(32)) {
  return nullptr;
}
buf.next(32, c->id);
return c;
```

Figure 1.4: Circuit ID parsing in `CircuitRep::from_bytes(ReadBuffer&)`
(`lib/proto/circuit.h#228–233`)

Since the circuit itself is separate from the proof, this can be exploited only if the attacker can manipulate the verifier's stored representation of the circuit. If the validator only checks the circuit ID, an attacker can replace the circuit with a trivial one, while keeping the circuit ID unchanged.

Exploit Scenario

Alice deploys an anonymous credentials validator on a system with signed code but potentially insecure storage. To avoid having to trust the storage, the credentials validator was designed so that the circuit's ID is checked whenever it is loaded. However, Bob compromises the storage and replaces the circuit file with a trivially satisfied circuit, while keeping the circuit ID the same. Bob then bypasses the credential check.

Recommendations

Short term, update the implementation so that it computes the circuit ID when the circuit is deserialized instead of trusting the provided ID.

Long term, ensure that all data used during proof validation is either constant or included in the transcript via some commitment.

2. Collision of transcript separation tags

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-LIBZK-2
Target: `lib/random/transcript.h`, `lib/algebra/nat.h`	

Description

The zero-knowledge protocol is made noninteractive through the Fiat-Shamir transform, wherein the verifier's random coins are instead generated by hashing the proof transcript with a suitable hash function. The transcript consists of byte strings, field elements, and an array of field elements. The transcript encoding rules describe how a transcript object builds the transcript using the prover's output. The encoding of various elements in a transcript uses domain-separation tags for different types to prevent ambiguous transcript encoding, which can ultimately result in the acceptance of fake proofs. However, field elements and field arrays use the same separation tag, likely due to a typo.

Figure 2.1 shows how a field element and an array of field elements are written into the transcript.

```
// one field element
template <class Field>
void write(const typename Field::Elt& e, const Field& F) {
  tag(TAG_FIELD_ELEM);

  write_untyped(e, F);
}

// array of field elements
template <class Field>
void write(const typename Field::Elt e[/*n*/], size_t ince, size_t n,
           const Field& F) {
  tag(TAG_ARRAY);
  length(n);

  for (size_t i = 0; i < n; ++i) {
    write_untyped(e[i * ince], F);
  }
}
```

Figure 2.1: Writing of a field element and an array of field elements into the transcript (`longfellow-zk/lib/random/transcript.h#L128–L146`)

Figure 2.1 shows that both the `TAG_FIELD_ELEM` and `TAG_ARRAY` transcript separation tags are set to 1. This collision allows for potentially ambiguous encoding of inputs in the transcript.

```
class Transcript : public RandomEngine {
  enum { TAG_BSTR = 0, TAG_FIELD_ELEM = 1, TAG_ARRAY = 1 };
```

Figure 2.2: Transcript separation tags `TAG_FIELD_ELEM` and `TAG_FIELD_ELEM` have the same value. (`longfellow-zk/lib/random/transcript.h#L65–L66`)

Since the library explicitly differentiates between a single field element and an array with a single element, a malicious prover may attempt to exploit the tag value collision to prove a statement about an array of a single element while performing the proof with a single element. This attempt and other attempts to exploit the collision will fail due to the deserialization requirements that field elements be encoded as fixed-byte values, as figure 2.3 shows. Furthermore, as figure 2.1 shows, the length of an array of field elements is also encoded as a fixed 8-byte value.

```
// Interpret A[] as a little-endian nat
static T of_bytes(const uint8_t a[/* kBytes */]) {
  T r;
  for (size_t i = 0; i < kLimbs; ++i) {
    a = Super::of_bytes(&r.limb_[i], a);
  }
  return r;
}
```

Figure 2.3: Deserialization of a field element (`longfellow-zk/lib/algebra/nat.h#L100–L107`)

Recommendations

Short term, use different values for `TAG_FIELD_ELEM` and `TAG_FIELD_ELEM`.

Long term, implement additional negative tests to catch violations of invariants.

3. FSPRF does not limit the size of the output stream

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-LIBZK-3
Target: `lib/random/transcript.h`	

Description

The Fiat-Shamir transform is used to generate the verifier's random coin by hashing the proof transcript with a suitable hash function. The libZK specification augments the basic transform with a Fiat-Shamir PRF (FSPRF) object that produces an "infinite" output stream. However, the PRF is instantiated with AES in counter mode, and, therefore, the quality of the generated randomness decreases quadratically with the number of calls to AES.

In the library, the randomness API is implemented through the `bytes` function, which takes as input a buffer and a byte count and generates the requested number of bytes. The counter-based PRF is implemented through the call to the `refill` function.

```
void bytes(uint8_t buf[/*n*/], size_t n) {
  while (n-- > 0) {
    if (rdptr_ == kPRFOutputSize) {
      refill();
    }
    *buf++ = saved_[rdptr_++];
  }
}
```

Figure 3.1: Obtaining an arbitrary stream of random bytes for the FSPRF (`longfellow-zk/lib/random/transcript.h#L42–L49`)

After 2^{64} calls, the FSPRF output has a significant distance for a uniform random sequence of the same size. This is not merely an artifact of the switching lemma; the interpolation probabilities directly show the nonnegligible distinguishing advantage. However, the libZK implementation never needs such a long output stream and is therefore operating in a secure regime for the FSPRF.

Recommendations

Short term, limit the output size of the FSPRF to a value that retains indistinguishability from a random bit stream.

Long term, ensure all cryptographic primitives cannot be used outside of their secure regime.

4. MerkleTreeVerifier::verify_proof is vulnerable to path extension	
Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-LIBZK-4
Target: `lib/merkle/merkle_tree.h`	

Description

Merkle tree inclusion proofs act as evidence that a particular position in a committed vector has a particular value. However, if leaf and branch nodes are not domain-separated from each other, it is possible to construct a leaf node that can serve as a branch in a Merkle path. This type of leaf would then allow an attacker to generate proofs for leaves in the tree that have never been inserted, or in some cases to generate two different openings for the same index in the tree.

The Merkle tree implementation treats leaves purely as SHA-256 hashes, so an attacker only needs to craft a tree where a leaf contains the hash of a further subtree in order to generate a Merkle path that passes through that leaf. If the leaf is at position `i`, the `MerkleTreeVerifier::verify_proof` function, shown in figure 4.1, will accept a proof for that leaf at index `i` and will accept proofs that use that leaf as an internal branch at the indices `i` + `k*n_`.

```
bool verify_proof(const Digest* proof, size_t pos) const {
  Digest t = *proof++;
  for (pos += n_; pos > 1; pos >>= 1) {
    t = (pos & 1) ? Digest::hash2(*proof++, t) : Digest::hash2(t, *proof++);
  }
  return t == root_;
}
```

Figure 4.1: `verify_proof` does not check that `pos` is in range, and internal branch hashes are not domain-separated from leaf hashes. (`lib/merkle/merkle_tree.h#158–164`)

Since those indices are out of range for normal use, and the Ligero verifier uses the `verify_compressed_proof` function instead, this issue is not currently exploitable. However, it is good practice to use domain separation between branch and leaf nodes to prevent this class of attack.

Recommendations

Short term, add a check to `verify_proof` to ensure that `pos` is below `n_`, and add a domain-separation tag to leaf and branch hashes.

Long term, ensure that data types that are represented by their hashes are domain-separated so that cross-type collisions are prevented.

5. COSE1 length values are incorrectly serialized	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LIBZK-5
Target: `lib/circuits/mdoc/mdoc_witness.h`	

Description

The mdoc verification circuit checks that the witness includes a valid ECDSA signature for a "liveness transcript," which is used to ensure that proofs cannot be reused across requests. When computing the hash of the liveness transcript, length fields in the underlying data are serialized using the helper functions `append_bytes_len` and `append_text_len`, shown in figure 5.1.

These functions do not properly handle edge-case length values: `append_bytes_len` encodes the length value 256 instead as 0, and `append_text_len` does not add anything to the transcript if the length is exactly 255.

```
static inline void append_bytes_len(std::vector<uint8_t>& buf, size_t len) {
  if (len > 256) {
    uint8_t ll[] = {0x59, (uint8_t)((len >> 8) & 0xff), (uint8_t)(len & 0xff)};
    buf.insert(buf.end(), ll, ll + 3);
  } else {
    uint8_t ll[] = {0x58, static_cast<uint8_t>(len & 0xff)};
    buf.insert(buf.end(), ll, ll + 2);
  }
}

static inline void append_text_len(std::vector<uint8_t>& buf, size_t len) {
  check(len < 256, "Text length too large");
  if (len < 24) {
    buf.push_back(0x60 + len);
  } else if (len < 255) {
    buf.push_back(0x78);
    buf.push_back(len);
  }
}
```

Figure 5.1: Length-serializing functions that improperly handle the values 256 and 255, respectively (`lib/circuits/mdoc/mdoc_witness.h#321–339`)

Since these functions are used only when computing the liveness transcript, this issue would result only in completeness problems. For example, these functions are used in the `compute_transcript_hash` function, which computes the public input used in the `run_mdoc_verifier` function. If a server uses the `run_mdoc_verifier` function, it will not accept proofs corresponding to transcripts where an incorrect length field would be serialized.

Exploit Scenario

Alice sets up a server that calls `run_mdoc_verifier`. Bob sends an identity claim to the server, and coincidentally the value passed to `append_bytes_len` is 256. This causes the server to generate the wrong transcript when verifying proofs and spuriously reject Bob's valid identity claim.

Recommendations

Short term, fix the `append_bytes_len` and `append_text_len` functions' handling of the edge-case length values.

Long term, ensure that boundary conditions of serialization functions are thoroughly tested, ideally with round-trip tests to ensure that serialized data will be parsed to the same value.

6. Ligero parameter search can be improved

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Configuration	Finding ID: TOB-LIBZK-6
Target: `lib/ligero/ligero_param.h`	

Description

The soundness of Ligero is primarily determined by the encoding rate and the number of column queries, which the Longfellow library generally sets to $\frac{1}{4}$ and 128, respectively, to target a statistical security level of 86 bits. The remaining parameter for Ligero is the size of a row, which determines the size of the proof. The components of the proof size are the Merkle proof length, the size of each column, and the size of the prover's polynomials. The `LigeroParam` constructor has a search process, shown in figure 6.1, that estimates the total proof size for powers of 2 up to 2^{28} and chooses the one with the smallest estimated size.

```
LigeroParam(size_t nw, size_t nq, size_t rateinv, size_t nreq)
    : nw(nw), nq(nq), rateinv(rateinv), nreq(nreq) {
  r = nreq;

  size_t min_proof_size = SIZE_MAX;
  size_t best_block_enc = 1;
  for (size_t e = 1; e <= (1 << 28); e *= 2) {
    size_t proof_size = layout(e);
    if (proof_size < min_proof_size) {
      min_proof_size = proof_size;
      best_block_enc = e;
    }
  }

  // recompute parameters
  layout(best_block_enc);
  proofs::check(block_enc > block, "block_enc > block");

  ildt = 0;
  idot = 1;
  iquad = 2;
  iw = 3;
  iq = iw + nwrow;
  proofs::check(nrow == iq + 3 * nqtriples, "nrow == iq + 3 * nqtriples");
}
```

Figure 6.1: The `LigeroParam` constructor searches for the best value of `block_enc`.
(`longfellow-zk/lib/ligero/ligero_param.h#148–172`)

However, this search process may miss the best available row length. Experimentally, an exhaustive search leads to an approximately 14 KB, or roughly 5%, reduction in proof size for the one-claim mdoc circuit, where the hash proof's size is roughly unchanged but the signature circuit's proof size is reduced from approximately 204 KB to approximately 190 KB.

Doing an exhaustive search is not as expensive as it appears, since the number of allowable row sizes is proportional to the total witness count, and it can be done only once, when initially building the circuit.

Recommendations

Short term, update the Ligero implementation so that it computes optimal shape parameters for each circuit when building it and stores these parameters with the circuit.

Long term, evaluate Ligero parameter choices to optimize the proof size and prover complexity.

7. ECDSA circuit allows off-curve intermediate points

Severity: **Undetermined**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-LIBZK-7
Target: `lib/circuits/ecdsa/verify_circuit.h`	

Description

The ECDSA verification circuit implements the multiexponentiation version of the ECDSA verification check $(-s) \cdot R + r \cdot pk + e \cdot g = O$. Exponentiation is computed by iterating $w_{i+1} = 2 \cdot w_i + T(b_i)$, where each $b_i \in [0, 7]$ encodes the ith bits of $(-s, r, e)$, and T is a table-lookup function based on evaluating three interpolated polynomials.

The circuit attempts to guarantee that the b_i values are in the correct range by creating an additional lookup table vv, which is 1 for the set [0,7], and then asserting `vv(bi[i])` == 1. The vv table declaration and lookups are shown in figure 7.1.

```
void verify_signature3(EltW pk_x, EltW pk_y, EltW e, const Witness& w) const {
…
  EltW arr_v[] = {one, one, one, one, one, one, one, one};
…
  EltMuxer<LogicCircuit, 3> vv(lc_, arr_v);

  Bitvec r_bits, s_bits;

  // Traverses the bits of the scalar from high-order to low-order.
  for (size_t i = 0; i < kBits; ++i) {
    // Use the arr{X..V} arrays and the muxer to pick the correct point
    // slice based on the bits of advice in the witness.
    EltW tx = xx.mux(w.bi[i]);
    EltW ty = yy.mux(w.bi[i]);
    EltW tz = zz.mux(w.bi[i]);

    // Update the exponent.
    EltW e_bi = ee.mux(w.bi[i]);
    EltW r_bi = rr.mux(w.bi[i]);
    EltW s_bi = ss.mux(w.bi[i]);
    auto k2 = lc_.konst(k2_);
    est = lc_.add(&e_bi, lc_.mul(&k2, est));
    rst = lc_.add(&r_bi, lc_.mul(&k2, rst));
    sst = lc_.add(&s_bi, lc_.mul(&k2, sst));
    r_bits[kBits - i - 1] = BitW(r_bi, ec_.f_);
    s_bits[kBits - i - 1] = BitW(s_bi, ec_.f_);

    // Verify that the advice bit is in [0,7].
```

```
EltW range = vv.mux(w.bi[i]);
lc_.assert_eq(&range, one);
```

Figure 7.1: The values in the `bi` array are not correctly restricted by the highlighted assertion.
(`longfellow-zk/lib/circuits/ecdsa/verify_circuit.h#103–186`)

Since these tables are based on Lagrange interpolation, the polynomial representing the `vv` table is the lowest-degree polynomial matching the points $\{(i, 1): i \in \{0,\ldots, 7\}\}$, i.e., $p(x) = 1$. Thus, this check is a no-op and the b_i values are not directly constrained at all.

Due to the other checks in this circuit, we have not been able to construct an attack on this ECDSA verification, but we also have not been able to rule out the possibility that an attacker could generate a proof for an invalid signature. In particular, any attacker must navigate several difficult but not clearly impossible constraints:

- Purported values of `e`, `r`, and `s` are directly recalculated via evaluations in the `ee`, `rr`, and `ss` lookup tables.
- The purported values of `e` and `r` are directly checked to match the signature.
- The computed values of `r` and `s` must be nonzero in the base field, and the evaluations from `rr` and `ss` must pass a bit-logic comparison to the curve order.
- The coordinates from which the `xx` and `yy` lookup tables are constructed are on-curve points based on R and `pk`, so it is difficult to gain an advantage by manipulating the choice of R.
- Most points computed from the table will be off-curve, since any on-curve point is a solution to the degree-21 polynomial identity $y(v)^2 z(v) = x(v)^3 + ax(v)z(v)^2 + bz(v)^3$.

It is possible that these constraints suffice to prevent a full forgery, but it is clearly possible to create an off-curve point as an intermediate value, which violates an internal invariant.

Exploit Scenario

Alice discovers some choice of R and `bi` such that `rst == R.x` and `est == sha256(m)` for some maliciously constructed mdoc `m` with Bob's identity information. She then constructs fraudulent proofs based on `m` and impersonates Bob.

Recommendations

Short term, replace the table-based `bi` range check with a direct check, either by bit-decomposition or by asserting that the result of evaluating the polynomial $p(x) = \Pi_{i=0}^{7}(x - i)$ is zero.

Long term, modify the `EltMuxer` API to include a correct range check helper function. Consider introducing a wrapper type for values that have been range-checked, and modify the `mux()` method to take the wrapper type instead of a raw `EltW`.

8. The specification describes an incorrect quadratic test	
Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-LIBZK-8
Target: libZK specification	

Description

In Ligero, quadratic and linear constraints are enforced by having the prover send additional polynomials, which are interpolated and checked point-wise against the opened columns. In both cases, these checks involve multiplications, so the degree of the polynomial will necessarily be higher than k. In the case of the linear check, this higher degree will be $k + l - 1$ (or in the libZK version, $2k - 1$), while the quadratic check polynomial will have degree $2k - 1$. The Longfellow implementation correctly performs these checks, but the libZK draft specification does not include a quadratic-check polynomial in the proof and suggests an alternative check that is insufficient and will not result in a correct implementation if it is followed:

```
def quadratic_check(proof) {
  reqx = IQ * NREQ
  reqy = reqx + (NQT * NREQ)
  reqz = reqy + (NQT * NREQ)

  FOR 0 <= i < NQT * NREQ DO
    IF proof.columns[reqz + i] !=
       proof.columns[reqx + i] * proof.columns[reqy + i] {
      return false

  return true
}
```

Figure 8.1: An incorrect quadratic check from the libZK specification (https://www.ietf.org/id/draft-google-cfrg-libzk-00.html#section-6.5)

Recommendations

Short term, update the libZK specification to use the correct quadratic check implemented in Longfellow.

Long term, develop a security proof for the full libZK/Longfellow proof system.

9. MerkleCommitmentVerifier::verify_compressed_proof assumes nonrepeating indices	
Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-LIBZK-9
Target: `lib/merkle/merkle_tree.h`	

Description

The `verify` function of `MerkleCommitmentVerifier` uses the function `verify_compressed_proof` to compute the Merkle root from a list of leaf hashes and indices. If there is a repeated index, only the second leaf hash at that index will be used to compute the Merkle root, as shown in figure 9.1.

```
// set LAYERS at all leaves in POS
for (size_t ip = 0; ip < np; ++ip) {
  size_t l = pos[ip] + n_;
  layers[l] = leaves[ip];
  defined[l] = true;
}

// Recompute as many inner nodes as we can
for (size_t i = n_; i-- > 1;) {
  if (defined[2 * i] && defined[2 * i + 1]) {
    layers[i] = Digest::hash2(layers[2 * i], layers[2 * i + 1]);
    defined[i] = true;
  }
}
```

Figure 9.1: Computation of the Merkle root from the provided leaf array (`longfellow-zk/lib/merkle/merkle_tree.h#197–210`)

This function is currently used only in the `merkle_check` function of the Ligero verifier, as shown in figure 9.2. This usage is safe, since the indices passed to `merkle_check` come from the `gen_idx` function and are guaranteed to be distinct.

```
LigeroTranscript<Field>::gen_idx(&idx[0], p, ts, F);

if (!merkle_check(p, commitment, proof, &idx[0], F)) {
  *why = "merkle_check failed";
  return false;
}
```

Figure 9.2: The indices passed to `merkle_check` are guaranteed to be distinct since they are generated by the `gen_idx` function.
(`longfellow-zk/lib/ligero/ligero_verifier.h#84–89`)

Recommendations
Short term, document this assumption, and consider adding an additional check to ensure that no indices are repeated.

Long term, ensure that assumptions about function parameters are either documented or enforced.

10. mdoc attribute check can be bypassed

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LIBZK-10
Target: `lib/circuits/mdoc/`	

Description

To assert that an mdoc has a given attribute, the mdoc circuit checks the following:

- There is a 32-byte value h at some position in the MSO prefixed by the 2-byte sequence [0x58,0x20].
- There is a preimage p with a length of 128 bytes such that $h = unpadded\_sha256(p)$.
- p matches the value in the corresponding `OpenAttribute` struct.

However, the final comparison, highlighted in figure 10.1, is performed by calling `assert_attribute` with the prover-controlled secret witness value `vw.attr_ei_[ai].len`. The `assert_attribute` function, shown in figure 10.2, performs a comparison only on the first `len` positions. A malicious prover can set this witness value to 0 and successfully generate a proof for any attribute at all.

```
// Attributes parsing
// valueDigests, ignore byte 13 \in {A1,A2} representing map size.
r_.shift(vw.value_digests_.k, kValueDigestsLen, cmp_buf.data(), kMaxMsoLen,
         vw.in_ + 5 + 2, zz, /*unroll=*/3);
assert_bytes_at(13, &cmp_buf[0], kValueDigestsCheck);
assert_bytes_at(18, &cmp_buf[14], &kValueDigestsCheck[14]);

// Attributes: Equality of hash with MSO value
for (size_t ai = 0; ai < vw.num_attr_; ++ai) {
  v8 B[96];
  // Check the hash matches the value in the signed MSO.
  r_.shift(vw.attr_mso_[ai].k, 2 + 32, &cmp_buf[0], kMaxMsoLen,
           vw.in_ + 5 + 2, zz, /*unroll=*/3);

  // Basic CBOR check of the Tag
  assert_bytes_at(2, &cmp_buf[0], kTag32);

  v256 mm;
  // The loop below accounts for endian and v256 vs v8 types.
  for (size_t j = 0; j < 256; ++j) {
    mm[j] = cmp_buf[2 + (255 - j) / 8][(j % 8)];
  }
```

```
  auto two = lc_.template vbit<8>(2);
  sha_.assert_message_hash(2, two, vw.attrb_[ai].data(), mm,
                           vw.attr_sha_[ai].data());

  // Check that the attribute_id and value occur in the hashed text.
  r_.shift(vw.attr_ei_[ai].offset, 96, B, 128, vw.attrb_[ai].data(), zz, 3);
  assert_attribute(96, vw.attr_ei_[ai].len, B, oa[ai]);
}
```

Figure 10.1: Attribute checking in the mdoc circuit, with the prover-controlled length highlighted (`longfellow-zk/lib/circuits/mdoc/mdoc_hash.h#184–214`)

```
// Checks that an attribute id or attribute value is as expected.
// The len parameter holds the byte length of the expected id or value.
void assert_attribute(size_t max, const vind& len, const v8 got[/*max*/],
                      const OpenedAttribute& oa) const {
  // Copy the attribute id and value into a single array.
  v8 want[96];
  for (size_t j = 0; j < 32; ++j) {
    want[j] = oa.attr[j];
  }
  for (size_t j = 0; j < 64; ++j) {
    want[32 + j] = oa.v1[j];
  }

  // Perform an equality check on the first len bytes.
  for (size_t j = 0; j < max; ++j) {
    auto ll = lc_.vlt(j, len);
    auto same = lc_.eq(8, got[j].data(), want[j].data());
    lc_.assert_implies(&ll, same);
  }
}
```

Figure 10.2: The `assert_attribute` function, which does not apply any restrictions to positions beyond the prover-controlled index `len` (`longfellow-zk/lib/circuits/mdoc/mdoc_hash.h#226–245`)

This issue also appears in the `mdoc_1f` circuit, as shown in figure 10.3. However, it does not appear in the `Small` and `PtrCred` circuits.

```
// Attributes parsing
PathEntry ak[2] = {{vw.value_digests_, kValueDigestsLen, kValueDigestsID},
                   {vw.org_, kOrgLen, kOrgID}};
assert_path(2, ak, vw, dsC, psC);

// Attributes: Equality of hash with MSO value
for (size_t ai = 0; ai < vw.num_attr_; ++ai) {
  auto two = lc_.template vbit<8>(2);
  v8 B[96];
  sha_.assert_message(2, two, vw.attrb_[ai].data(),
                      vw.attr_sha_[ai].data());
```

```
  EltW h = repack32(vw.attr_sha_[ai][1].h1);
  // Check the hash matches the value in the signed MSO.
  cbor_.assert_map_entry(kMdoc1MaxMsoLen, vw.org_.v, 2, vw.attr_mso_[ai].k,
                         vw.attr_mso_[ai].v, vw.attr_mso_[ai].ndx,
                         dsC.data(), psC.data());
  cbor_.assert_elt_as_be_bytes_at(kMdoc1MaxMsoLen, vw.attr_mso_[ai].v, 32,
                                  h, dsC.data());

  // Check that the attribute_id and value occur in the hashed text.
  r_.shift(vw.attr_ei_[ai].offset, 96, B, 128, vw.attrb_[ai].data(), zz, 3);
  assert_attribute(96, vw.attr_ei_[ai].len, B, oa[ai].attr);
}
```

Figure 10.3: Attribute checking in the `mdoc_1f` *circuit, with the prover-controlled length highlighted (*`longfellow-zk/lib/circuits/mdoc/mdoc_1f.h#254–277`*)*

The diff shown in figure 10.4 demonstrates this vulnerability. In this modified implementation, instead of actually searching for each attribute, witness generation instead takes the very first attribute unconditionally and sets the length to 0. With this modification, the only test in the test suite that fails is `MdocZKTest.wrong_witness`, which attempts to build proofs with incorrect attribute values and expects the prover to fail.

```
diff --git a/lib/circuits/mdoc/mdoc_witness.h b/lib/circuits/mdoc/mdoc_witness.h
index 4e3aac0..2fc573a 100644
--- a/lib/circuits/mdoc/mdoc_witness.h
+++ b/lib/circuits/mdoc/mdoc_witness.h
@@ -688,22 +688,24 @@ class MdocHashWitness {
       atw_[i].resize(2);
       bool found = false;
       for (auto fa : pm_.attributes_) {
-        if (fa == attrs[i]) {
           FlatSHA256Witness::transform_and_witness_message(
               fa.tag_len, &fa.doc[fa.tag_ind], 2, attr_n_[i],
               &attr_bytes_[i][0], &atw_[i][0]);
           attr_mso_[i] = fa.mso;
           attr_ei_[i].offset = fa.id_ind - fa.tag_ind;
-          attr_ei_[i].len = fa.id_len;
+          attr_ei_[i].len = 0;
           if (version > 2) {
-            attr_ei_[i].len = fa.witness_length(attrs[i]);
+            attr_ei_[i].len = 0;
           }
           attr_ev_[i].offset = fa.val_ind - fa.tag_ind;
           attr_ev_[i].len = fa.val_len;
           found = true;
+          log(ERROR, "found attribute '%.*s' == '%.*s'", attrs[i].id_len,
+            attrs[i].id, fa.id_len, &fa.doc[fa.id_ind]);
           break;
         }
```

```
-       }
      if (!found) {
        log(ERROR, "Could not find attribute %.*s", attrs[i].id_len,
            attrs[i].id);
```

Figure 10.4: A diff showing modifications to witness generation to exploit this issue to generate incorrect proofs

This attack also appears to be possible for the JWT circuit, with the relevant checks shown in figure 10.5. The additional checks on the positions of delimiting strings partially mitigates the attack. In particular, by setting `attr_id_len_` to 0 and setting `attr_ind_` to the position of some substring of the form <`”:”`> s <`”`> (e.g., `“:”a”`), an attacker should be able to prove that any attribute has any value with s as a prefix. A similar attack may also be possible by manipulating the value of `payload_len_`, which could allow invalid Base64 to be "decoded."

```
std::vector<v8> dec_buf(64 * kMaxJWTSHABlocks);
Base64Decoder<LogicCircuit> b64(lc_);
b64.base64_rawurl_decode_len(shift_buf.data(), dec_buf.data(),
                             64 * (kMaxJWTSHABlocks - 2), vw.payload_len_);

// For each attribute, shift the decoded payload so that the
// attribute is at the beginning of B. Verify the attribute id, the
// json separator, the attribute value, and the end quote.
for (size_t i = 0; i < vw.attr_ind_.size(); ++i) {
  v8 B[32 + 3 + 64 + 1];

  // Check that values of the attribute_id.
  r_.shift(vw.attr_ind_[i], 100, B, dec_buf.size(), dec_buf.data(), zz, 3);
  assert_string_eq(32, vw.attr_id_len_[i], B, oa[i].attr);

  r_.shift(vw.attr_id_len_[i], 100, B, 100, B, zz, 3);
  uint8_t sep[3] = {'"', ':', '"'};
  for (size_t j = 0; j < 3; ++j) {
    auto want_j = lc_.template vbit<8>(sep[j]);
    lc_.vassert_eq(&B[j], want_j);
  }

  auto three = lc_.template vbit<2>(3);
  r_.shift(three, 100, B, 100, B, zz, 3);

  assert_string_eq(64, vw.attr_value_len_[i], B, oa[i].v1);

  r_.shift(vw.attr_value_len_[i], 100, B, 100, B, zz, 3);

  auto end_quote = lc_.template vbit<8>('"');
  lc_.vassert_eq(&B[0], end_quote);
}
```

Figure 10.5: JWT attribute parsing, with two potentially exploitable prover-controlled lengths (longfellow-zk/lib/circuits/jwt/jwt.h#136–167)

We have not attempted to build a proof-of-concept exploit targeting the JWT circuit, but the Google Longfellow team should thoroughly investigate whether such an exploit is possible and, if so, implement mitigations.

Exploit Scenario

Alice wishes to impersonate Bob online. She signs up to a website that uses the Google Longfellow library for identity verification and submits a zero-knowledge proof purporting to show Bob's name and birth date, using the witness-building modification shown above. The proof succeeds, and Alice is able to create a verified account impersonating Bob.

Recommendations

Short term, move the length field used in the `assert_attribute` comparison to the public input of the circuit.

Long term, ensure that private witness values cannot be used to skip checks on public inputs. Consider adding defensive-programming mitigations that require any unconstrained part of the public input to have a fixed value; for example, if `assert_attribute` were modified to enforce that every byte in `oa` after the length is 0, it would not be possible to arbitrarily exploit this issue.

11. ECDSA witness-building timing may leak hidden witness values	
Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LIBZK-11
Target: `lib/circuits/ecdsa/verify_witness.h`	

Description

When the ECDSA witness is built, values associated with the signature and the public key are passed to variable-time functions such as `Field::invertf` and `EC::scalar_multf`, as shown in figure 11.1.

```
bool compute_witness(const Elt pkX, const Elt pkY, const Nat e, const Nat r,
                     const Nat s) {
  const Field& F = ec_.f_;
  const Scalar _s = fn_.invertf(fn_.to_montgomery(s));
  const Scalar tms = fn_.negf(fn_.to_montgomery(s));

…
  Point bases[] = {ec_.generator(), Point(pkX, pkY, F.one())};
  Nat scalars[] = {nes, nrs};
  auto pr = ec_.scalar_multf(2, bases, scalars);
  ec_.normalize(pr);

  rx_ = F.to_montgomery(r);
  ry_ = pr.y;

  // In the case of a malicious input with rx=0 or s=0, the proof will fail.
  if (rx_ != F.zero()) {
    rx_inv_ = F.invertf(rx_);
    check(F.mulf(rx_, rx_inv_) == F.one(), "bad inv");
  }

  s_inv_ = F.to_montgomery(fn_.from_montgomery(tms));
  if (s_inv_ != F.zero()) {
    F.invert(s_inv_);
  }

  if (pkX != F.zero()) {
    pk_inv_ = F.invertf(pkX);
  }
```

Figure 11.1: Some of the variable-time calls in the ECDSA witness-building function (`longfellow-zk/lib/circuits/ecdsa/verify_witness.h#75–108`)

Since the zero-knowledge proofs generated by the Longfellow library are intended to be created on demand, a server may be able to use this timing to extract information about the client's witness values. For example, certain public keys may have significantly faster or slower witness generation than others, allowing a server to fingerprint users whose documents are signed by a particular entity.

Alternatively, it may be possible to use timing information from several different zero-knowledge proof requests to learn partial information about several (r, s) pairs and perform a variant of ECDSA key recovery to reveal a specific public key. We are not aware of good estimates of how difficult it is to extract a public key from partial information about several signatures; since public keys and signatures are generally public, it may not be a well-studied problem.

Exploit Scenario

Alice sets up a service that receives identity attestations. By analyzing the time Bob's device takes to build several attestation proofs, she is able to learn information about Bob's device public key, potentially revealing his specific identity.

Recommendations

Short term, implement mitigations to prevent a malicious server from learning precise secret-dependent timing data—for example, by inserting delays to make the total time taken to respond to a request near-constant.

Long term, define an explicit attacker model and ensure that malicious servers cannot bypass the intended guarantees of the zero-knowledge proof system.

12. MAC scheme is vulnerable to existential forgery on input zero and may break zero-knowledge in other uses of the library

Severity: **Informational**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-LIBZK-12
Target: `lib/circuits/mac/mac_circuit.h`	

Description

To improve performance, specific circuits may be verified by splitting verification operations so that different properties are verified by specific circuits defined over the appropriate field, where the best performance is achieved. For example, ECDSA verification is split between verification of the ECDSA equation on the message hash over a prime order field and verification of the hash of the message over a binary field. A MAC consistency check ensures that the witness is consistent across circuits. However, the MAC scheme in the implementation uses undocumented optimizations and does not follow the specification; as a consequence, it is not unconditionally secure as expected, and it may leak information about the witness in other applications that may want to use the MAC circuit.

The specification defines the MAC by $mac_{a,b}(x) = ax + b$, where the key (a, b) is a pair of randomly sampled field elements. When using the MAC consistency check, the zero-knowledge protocol is modified so that the prover also sends the MAC to the verifier. For security, the MAC key is jointly sampled by the prover and the verifier. However, the implemented MAC scheme is instead $mac_a(x) = ax$. Figure 14.1 shows the verification algorithm corresponding to this MAC scheme.

```
// Verify a mac on the 256-bit message msg.
void verify_mac(const EltW mac[/*2*/], const EltW& av, const v256& msg,
                const Witness& vw) const {
  // Check that mac[i] = (a_p + a_v)*mm[i] for i=0..1.
  for (size_t i = 0; i < 2; ++i) {
    EltW mm = pack(&msg[i * 128]);
    EltW key = lc_.add(&av, vw.aa_[i]);
    EltW got = lc_.mul(&key, mm);
    lc_.assert_eq(&mac[i], got);
  }
}
```

Figure 12.1: MAC verification over a prime order field
(`longfellow-zk/lib/circuits/mac/mac_circuit.h#L166–L17`)

```
// Checks mac[i] = (a_p + a_v)*xi[i] for i=0..1.
void assert_mac(const v128 mac[/*2*/], const v128& av, const v128 xi[/*2*/],
                const Witness& vw) const {
  v128 mv;
  for (size_t i = 0; i < 2; ++i) {
    v128 ap = bp_.template unpack<v128, packed_v128>(vw.aa_[i]);
    v128 key = lc_.vxor(&av, ap);
    lc_.gf2_128_mul(mv, key, xi[i]);
    lc_.vassert_eq(&mac[i], mv);
  }
}
```

Figure 12.2: MAC verification over GF (2^{128})
(longfellow-zk/lib/circuits/mac/mac_circuit.h#L100–L110)

As a stand-alone MAC on field elements, the MAC is not unconditionally secure since $x = 0$ always admits a forgery regardless of the key. Furthermore, Theorem 3.2 says that the composition of circuits with the MAC should retain zero-knowledge security, assuming the MAC is statistically secure. However, the MAC scheme also leaks whether the witness is nonzero.

The issues above do not constitute a threat to the use case of anonymous credentials, the primary use case of the library. It is a reasonable optimization that reduces the number of witness elements required in the prime-field circuit. In the anonymous credential use case, the MAC inputs are the upper or lower 128 bits of `e`, `dpkx`, and `dpky`. The value of `e` is a hash output, and the chance of finding a hash with 128 zeros in the high or low bits is roughly 2^{-127}. The values `dpkx` and `dpky` are the coordinates of the device public key (that is, an elliptic curve point generated by a hardware secure element). While finding elliptic curve points with the required pattern of zeros is not hard, honestly generated public keys will have a uniform distribution over all curve points, and a random point's coordinates have a very small chance of containing 128 zeros in the high or low bits. We have not precisely estimated this probability, but we believe that it is low enough that unforgeability and zero-knowledge should still hold as claimed.

However, the library can also be used for general-purpose computation. In that case, users must carefully examine whether the information leakage through the MAC is detrimental to the security of their application. For instance, an application where a random binary value is used in each session cannot expect privacy for the witness since all bits will leak through the MAC with overwhelming probability.

Recommendations

Short term, document the MAC optimization and add a warning for users who might consider using the MAC consistency check for their application.

13. Ligero matrix construction deviates from the specification

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-LIBZK-13
Target: `lib/ligero/ligero_prover.h`, `lib/ligero/ligero_param.h`	

Description

In the libZK specification, the second blinding row (`IDOT`) is a Reed–Solomon encoding of a `BLOCK`-sized word generated at random with the constraint that the sum of field elements in the word is 0. However, the implementation of `IDOT` constrains only the witness portion of the row to sum to 0. This is sufficient because the corresponding verification check has also been changed to cover only the witness section of the row.

```
random_row(p_.idot, p_.dblock, rng, F);

// Then constrain to sum(W) = 0
Elt sum = Blas<Field>::dot1(p_.w, &tableau_at(p_.idot, p_.r), 1, F);
F.sub(tableau_at(p_.idot, p_.r), sum);

interp->interpolate(&tableau_at(p_.idot, 0));

// quadratic-test blinding row constrained to W = 0.  First
// randomize the entire dblock:
random_row(p_.iquad, p_.dblock, rng, F);

// Then constrain to W = 0
Blas<Field>::clear(p_.w, &tableau_at(p_.iquad, p_.r), 1, F);

interp->interpolate(&tableau_at(p_.iquad, 0));
```

Figure 13.1: `longfellow-zk/lib/ligero/ligero_prover.h#L186–L201`

```
 // check the putative value of the inner product
  Elt want_dot = Blas<Field>::dot(nl, b, 1, &alphal[0], 1, F);
  Elt proof_dot = Blas<Field>::dot1(p.w, &proof.y_dot[p.r], 1, F);
  if (want_dot != proof_dot) {
    *why = "wrong dot product";
    return false;
  }
}
```

Figure 13.2: `longfellow-zk/lib/ligero/ligero_verifier.h#108–115`

Recommendations

Short term, update the specification to match the protocol changes.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be a single rule, a directory of rules, or the name of a ruleset hosted on the Semgrep registry. Trail of Bits' public ruleset can be used by running `semgrep --config "p/trailofbits"`.

lcov

`lcov` can be installed either from a distribution's package repository or according to the source repository's instructions. Once the test suite has been run with coverage enabled, a `coverage.info` file can be generated with the command `lcov --capture --directory <test-dir> --output-file coverage.info`, and an HTML report can be generated with the command `genhtml --exclude '*test*' coverage.info --output-directory <report-dir>`.

C. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From August 11 to August 13, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Google team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 13 issues described in this report, Google has resolved 11 issues and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Circuit ID is not checked during circuit deserialization	High	Resolved
2	Collision of transcript separation tags	Informational	Resolved
3	FSPRF does not limit the size of the output stream	Informational	Resolved
4	MerkleTreeVerifier::verify_proof is vulnerable to path extension	Informational	Resolved
5	COSE1 length values are incorrectly serialized	Low	Resolved
6	Ligero parameter search can be improved	Informational	Unresolved
7	ECDSA circuit allows off-curve intermediate points	Undetermined	Resolved
8	The specification describes an incorrect quadratic test	Informational	Resolved
9	MerkleCommitmentVerifier::verify_compressed_proof assumes nonrepeating indices	Informational	Resolved
10	mdoc attribute check can be bypassed	High	Resolved

11	ECDSA witness-building timing may leak hidden witness values	Low	Unresolved
12	MAC scheme is vulnerable to existential forgery on input zero and may break zero-knowledge in other uses of the library	Informational	Resolved
13	Ligero matrix construction deviates from the specification	Informational	Resolved

Detailed Fix Review Results

TOB-LIBZK-1: Circuit ID is not checked during deserialization
Resolved in commit `2cb614684020cd5fa8753cfd1cafab1e61377aa9`. The Google team introduced the configuration constants `enforce_circuit_id_in_verifier` and `enforce_circuit_id_in_prover`, which configure whether the circuit ID is checked during deserialization. The comments on these constants describe what an application using the library should do to ensure that circuit IDs are consistent:

> *The larger application is expected to contain a hardcoded list of supported circuit IDs. After downloading the circuit ((or compiling it locally) the application is expected to check the ID once, and then store the checked circuit in trusted local storage.*

TOB-LIBZK-2: Collision in transcript separation tags
Resolved in commit `60c7180b78da34af9ba47e0f0a08a4eaca148349`. The `TAG_ARRAY` constant has been changed to equal 2.

TOB-LIBZK-3: FSPRF does not limit the size of the output stream
Resolved in commit `60c7180b78da34af9ba47e0f0a08a4eaca148349`. A check has been added to limit the transcript's random number generation to 2^{40} blocks.

TOB-LIBZK-4: MerkleTreeVerifier::verify_proof is vulnerable to path extension
Resolved in commit `60c7180b78da34af9ba47e0f0a08a4eaca148349`. The unneeded `verify_proof` method has been removed.

TOB-LIBZK-5: COSE1 length values are incorrectly serialized
Resolved in commit `60c7180b78da34af9ba47e0f0a08a4eaca148349`. The edge-case handling has been correctly updated. Note that the description of `append_bytes_len` has a typo: it states, "This method handles bytestrings that are up to 255 bytes long," but the function itself handles strings up to 65535 bytes.

TOB-LIBZK-6: Ligero parameter search can be improved
Unresolved. The Google team provided the following context for this finding's fix status:

> *We appreciate the suggestion and will consider introducing a more fine-grained search, perhaps offline.*

TOB-LIBZK-7: ECDSA circuit allows off-curve intermediate points
Resolved in commit `60c7180b78da34af9ba47e0f0a08a4eaca148349`. The range check polynomial is now interpolated through [(0,0),...,(7,0),(8,1)], and the result is now checked to be 0. This check is sufficient to ensure that the bit slice is in the range [0,7].

To see why, recall that Lagrange interpolation outputs the lowest-degree polynomial matching the given points, and the polynomial passing through those points has at least 8 zeros. Let $f(X) = \prod_{i=0}^{7}(X - i)$. Since $f(8) \neq 0$, then $g(X) = \frac{f(X)}{f(8)}$ is a degree-7 polynomial that interpolates those points, and Lagrange interpolation will return $g(X)$. If $g(z) = 0$, we can conclude that $z \in [0, 7]$. Note that the polynomial generated by `EltMuxer` may use a different encoding of the x-coordinates depending on the field, but the reasoning still applies.

TOB-LIBZK-8: The specification describes an incorrect quadratic test
Resolved in commit `487b3a585a695dc7992305b77bd9a797ac77d196`. The specification now reflects the correct quadratic check implementation.

TOB-LIBZK-9: MerkleCommitmentVerifier::verify_compressed_proof assumes non-repeating indices
Resolved in commit `2cb614684020cd5fa8753cfd1cafab1e61377aa9`. The comments in `merkle_tree.h` now state this assumption:

> *The list of leaves must be a set, i.e., with no duplicates. All usage within this library satisfies this requirement because the FS methods that produce the challenge set of indices includes no duplicates.*

TOB-LIBZK-10: mdoc attribute check can be bypassed
Resolved in commit `60c7180b78da34af9ba47e0f0a08a4eaca148349`. The mdoc circuits now include a length value along with each attribute. The potential issue in the JWT circuit has not been addressed, so the Google team should ensure that it will not be used until it has either mitigated the issue or confirmed that an attack is not possible.

TOB-LIBZK-11: ECDSA witness-building timing may leak hidden witness values
Unresolved. The Google team provided the following context for this finding's fix status:

> *The current library does not guard against server timing attacks. In the current application, there is a substantial human timing factor involved in producing a proof on the device—namely, the time for the user to perform a biometric authentication on the device and choose to send the proof. This delay perhaps swamps the timing variations introduced by the optimizations in the proof library. While the issue pointed out in the report can be addressed, there are several other aspects of the system that may have timing channels. We therefore defer mitigations of all timing channels until after assessing the security model and the scope of variability.*

TOB-LIBZK-12: MAC scheme has existential forgery and may break zero-knowledge in other uses of the library
Resolved in commit `2cb614684020cd5fa8753cfd1cafab1e61377aa9`. The comments above the `Mac` class now document the MAC scheme as implemented and state that "the caller must ensure that the MACed values are non-zero with very high probability."

TOB-LIBZK-13: Ligero matrix construction deviates from the specification
Resolved in commit `487b3a585a695dc7992305b77bd9a797ac77d196`. The specification has been updated to match the implementation.

D. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up to date with our latest news and announcements, please follow @trailofbits on X or LinkedIn, and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.