Longfellow ZK: Comprehensive Library Documentation

Table of Contents

Overview

Longfellow ZK is a high-performance C++ library for constructing zero-knowledge proofs (ZKPs) targeting legacy identity verification standards such as ISO MDOC, JWT, and W3C Verifiable Credentials. The library implements a modern, modular ZKP system combining efficient arithmetization with the Ligero proof system.

Key Features

- **Efficient Arithmetization**: Novel quad gate representation for optimized sumcheck protocols
- **Modular Architecture**: Clean separation between frontend (circuit compilation) and backend (proof generation)
- **Application-Specific Circuits**: Pre-built circuits for SHA-256/3, ECDSA, JWT, MDOC, and more
- **Zero-Knowledge**: Uses Ligero protocol with Merkle commitments for witness hiding
- **High Performance**: Optimized field arithmetic, FFT operations, and parallel processing

Academic References

- Anonymous credentials from ECDSA
- libzk: A C++ Library for Zero-Knowledge Proofs

Architecture

High-Level Architecture

```
  Application          Frontend            Backend
  Circuits           Arithmetization     Proof System

• SHA-256/3          • QuadCircuit         • Sumcheck
• ECDSA              • Compiler            • Ligero
• JWT                • Scheduler           • Merkle
• MDOC               • Optimization        • Transcript
• Base64
```

Directory Structure

```
lib/
  algebra/          # Field arithmetic, FFT, polynomials, Reed-Solomon
  arrays/           # Dense arrays, affine transformations, equality checks
  circuits/         # Application circuits and compilation framework
     compiler/     # Circuit compilation and optimization
     sha/          # SHA-256 circuits
     sha3/         # SHA-3 circuits
     ecdsa/        # ECDSA verification circuits
     jwt/          # JWT parsing and verification
     mdoc/         # ISO MDOC circuits
     base64/       # Base64 decoding circuits
     mac/          # MAC verification circuits
     logic/        # Bit manipulation and Plücker encoding
  sumcheck/         # Sumcheck protocol implementation
  ligero/           # Ligero proof system
  zk/               # Zero-knowledge protocol glue
  merkle/           # Merkle tree commitments
  random/           # Randomness and transcript management
  util/             # Utilities (logging, crypto, error handling)
```

Core Components

1. Field Arithmetic (`lib/algebra/`)

The library supports multiple field types: - **Prime Fields**: `FpGeneric`, `Fp`, `Fp2` for extension fields - **Binary Fields**: `GF2_128` for efficient binary operations - **Specialized Fields**: Optimized implementations for specific prime sizes

Key classes: - `Field`: Abstract field interface - `Blas<Field>`: Basic Linear Algebra Subprograms for field operations - `FFT<Field>`: Fast Fourier Transform for polynomial operations - `ReedSolomon<Field>`: Reed-Solomon codes for Ligero

2. Array Operations (`lib/arrays/`)

Efficient array operations supporting the proof system: - `Dense<Field>`: Multi-dimensional dense arrays with binding operations - `Affine<Field>`: Affine transformations and interpolations - `Eq<Field>`: Equality check polynomials - `Eqs<Field>`: Multiple equality check evaluations

3. Circuit Representation (`lib/sumcheck/`)

Core circuit and proof structures:

```
template <class Field>
struct Circuit {
  corner_t nv;            // number of outputs
  size_t logv;            // log of output dimension
  corner_t nc;            // number of copies
  size_t logc;            // log of copy dimension
  size_t nl;              // number of layers
  size_t ninputs;         // total inputs
  size_t npub_in;         // public inputs
  std::vector<Layer<Field>> l;  // circuit layers
  uint8_t id[32];         // unique circuit identifier
};

template <class Field>
struct Layer {
  corner_t nw;            // number of wires
  size_t logw;            // log of wire dimension
  std::unique_ptr<const Quad<Field>> quad;  // quadratic gates
};
```

4. Quad Gates (`lib/sumcheck/quad.h`)

Novel gate representation for efficient sumcheck:

```
template <class Field>
class Quad {
  struct corner {
    quad_corner_t g;      // gate variable
    quad_corner_t h[2];   // hand variables (left, right)
    Elt v;                // coefficient
  };
  std::vector<corner> c_;  // quad terms
};
```

Quad gates represent sums of quadratic terms: `v_i * w_l_i * w_r_i`

Frontend: Arithmetization

QuadCircuit Compiler (`lib/circuits/compiler/`)

The frontend converts high-level predicates into optimized arithmetic circuits:

```
template <class Field>
class QuadCircuit {
public:
  // Basic operations
  size_t input();                             // Create input wire
  size_t add(size_t op0, size_t op1);        // Addition gate
  size_t mul(size_t op0, size_t op1);        // Multiplication gate
  size_t assert0(size_t op);                 // Assert wire equals zero
  size_t konst(const Elt& k);                // Constant wire

  // Optimization and compilation
  std::unique_ptr<Circuit<Field>> mkcircuit(size_t nc);

private:
  // Optimization passes
  void compute_needed(size_t depth_ub);              // Dead code elimination
  node merge(size_t op0, size_t op1);                // Common subexpression elimination
  size_t push_node(node n);                         // CSE and node creation
};
```

Compilation Pipeline

1. **Circuit Construction**: Build DAG using arithmetic operations
2. **Constant Propagation**: Fold constant expressions
3. **Common Subexpression Elimination**: Remove duplicate computations
4. **Dead Code Elimination**: Remove unused wires
5. **Layer Scheduling**: Organize into sumcheck layers
6. **Quad Gate Generation**: Group terms into quadratic forms

Example Circuit Construction

```
// Create a circuit verifying x^2 + y^2 = z
QuadCircuit<Field> circuit(field);

// Public inputs
size_t z = circuit.input();
circuit.private_input();

// Private inputs
size_t x = circuit.input();
size_t y = circuit.input();
```

```cpp
// Computation
size_t x2 = circuit.mul(x, x);
size_t y2 = circuit.mul(y, y);
size_t sum = circuit.add(x2, y2);
size_t constraint = circuit.sub(sum, z);
circuit.assert0(constraint);

// Compile to sumcheck circuit
auto compiled = circuit.mkcircuit(1);
```

Backend: Proof System

Sumcheck Protocol (`lib/sumcheck/`)

Implements the sumcheck protocol for circuit satisfiability:

```cpp
template <class Field>
class ProverLayers {
public:
  // Evaluate circuit and generate witness
  std::unique_ptr<Dense<Field>> eval_circuit(
    inputs* in, const Circuit<Field>* circ,
    std::unique_ptr<Dense<Field>> W0, const Field& F);

  // Generate sumcheck proof
  void prove(Proof<Field>* pr, const Proof<Field>* pad,
             const Circuit<Field>* circ, const inputs& in,
             ProofAux<Field>* aux, bindings& bnd,
             TranscriptSumcheck<Field>& ts, const Field& F);
};
```

Zero-Knowledge Layer (`lib/zk/`)

Adds zero-knowledge to sumcheck using Ligero commitments:

```cpp
template <class Field, class ReedSolomonFactory>
class ZkProver : public ProverLayers<Field> {
public:
  // Commit to witness with random padding
  void commit(ZkProof<Field>& zkp, const Dense<Field>& W,
              Transcript& tp, RandomEngine& rng);

  // Generate zero-knowledge proof
  bool prove(ZkProof<Field>& zkp, const Dense<Field>& W, Transcript& tsp);

private:
  // Generate random padding for zero-knowledge
  void fill_pad(RandomEngine& rng);
};
```

Ligero Protocol (`lib/ligero/`)

Implements the Ligero SNARK for proving committed witness satisfies constraints:

```
template <class Field, class InterpolatorFactory>
class LigeroProver {
public:
  // Commit to witness using Merkle tree
  void commit(LigeroCommitment<Field>& commitment, Transcript& ts,
              const Elt W[], size_t subfield_boundary,
              const LigeroQuadraticConstraint lqc[],
              const InterpolatorFactory& interpolator,
              RandomEngine& rng, const Field& F);

  // Generate Ligero proof
  void prove(LigeroProof<Field>& proof, Transcript& ts,
             size_t nl, size_t nllterm,
             const LigeroLinearConstraint<Field> llterm[],
             const LigeroHash& hash_of_llterm,
             const LigeroQuadraticConstraint lqc[],
             const InterpolatorFactory& interpolator, const Field& F);
};
```

Application Circuits

SHA-256 Circuit (`lib/circuits/sha/`)

```
template <class Field>
class FlatSHA256Circuit {
  // Create circuit for SHA-256 hash verification
  static std::unique_ptr<Circuit<Field>> make_circuit(
    size_t message_len, const Field& F);

  // Generate witness for specific message
  static std::unique_ptr<Dense<Field>> make_witness(
    const uint8_t* message, size_t len, const Field& F);
};
```

ECDSA Verification (`lib/circuits/ecdsa/`)

```
template <class Field>
class ECDSAVerifyCircuit {
  // Create circuit for ECDSA signature verification
  static std::unique_ptr<Circuit<Field>> make_circuit(const Field& F);

  // Generate witness for signature verification
  static std::unique_ptr<Dense<Field>> make_witness(
    const ECDSASignature& sig, const ECPublicKey& pubkey,
    const uint8_t* message, size_t len, const Field& F);
};
```

JWT Parsing (`lib/circuits/jwt/`)

```
template <class Field>
class JWT {
  // Parse and verify JWT structure
  static std::unique_ptr<Circuit<Field>> make_parser_circuit(const Field& F);

  // Generate witness for JWT claims
  static std::unique_ptr<Dense<Field>> make_witness(
    const std::string& jwt_token, const JWTClaims& claims, const Field& F);
};
```

Detailed Workflows

Complete Proof Generation Workflow

1. **Circuit Definition**

   ```
   QuadCircuit<Field> qc(field);
   // Define predicate using qc.input(), qc.add(), qc.mul(), etc.
   auto circuit = qc.mkcircuit(num_copies);
   ```

2. **Witness Assignment**

   ```
   Dense<Field> witness(num_copies, circuit->ninputs);
   // Assign public and private input values
   ```

3. **ZK Proof Generation**

   ```
   ZkProver<Field, ReedSolomonFactory> prover(*circuit, field, rs_factory);
   ZkProof<Field> proof(ligero_params);

   // Commit phase
   prover.commit(proof, witness, transcript, rng);

   // Prove phase
   bool success = prover.prove(proof, witness, transcript);
   ```

4. **Verification**

```
    ZkVerifier<Field, ReedSolomonFactory> verifier;
    const char* error;
    bool valid = verifier.verify(&error, *circuit, proof,
                                 public_inputs, transcript, field);
```

Sumcheck Protocol Details

For each layer of the circuit:

1. **Bind Copy Variables**: Reduce over copy dimension using random challenges
2. **Bind Wire Variables**: For each sumcheck round:
    - Compute polynomial over remaining variables
    - Send polynomial to verifier (or transcript)
    - Receive random challenge
    - Bind one variable using challenge
3. **Final Claims**: Output claims on input wires for next layer

Ligero Protocol Details

1. **Tableau Construction**: Organize witness into Reed-Solomon codewords
2. **Merkle Commitment**: Commit to tableau columns using Merkle tree
3. **Low-Degree Test**: Prove committed polynomials have correct degree
4. **Linear Test**: Prove linear constraints on committed values
5. **Quadratic Test**: Prove quadratic constraints on committed values
6. **Column Opening**: Open random columns for verification

API Reference

Core Types

```
// Field element type
using Elt = typename Field::Elt;

// Circuit corner type (wire/gate identifier)
using corner_t = uint32_t;
using quad_corner_t = uint32_t;

// Proof structures
template <class Field> struct Proof;
template <class Field> struct ZkProof;
template <class Field> struct LigeroProof;
```

Key Interfaces

```
// Field arithmetic interface
template <class Field>
class FieldInterface {
  Elt zero() const;
  Elt one() const;
  Elt add(const Elt& a, const Elt& b) const;
  Elt mul(const Elt& a, const Elt& b) const;
  Elt inv(const Elt& a) const;
};

// Random number generation
class RandomEngine {
  virtual Elt elt(const Field& F) = 0;
  virtual void bytes(uint8_t* buf, size_t len) = 0;
};

// Transcript for Fiat-Shamir
class Transcript {
  void write(const void* data, size_t len);
  template <class Field> void write(const Elt& x, const Field& F);
  template <class Field> Elt elt(const Field& F);
};
```

Usage Examples

Basic Circuit Example

```
#include "circuits/compiler/compiler.h"
#include "zk/zk_prover.h"

// Define field
using Field = FpGeneric<64>;
Field field;

// Create circuit: prove knowledge of square root
QuadCircuit<Field> qc(field);
size_t y = qc.input();  // public: y
qc.private_input();
size_t x = qc.input();  // private: x
size_t x2 = qc.mul(x, x);
size_t constraint = qc.sub(x2, y);
qc.assert0(constraint);  // assert x^2 = y

auto circuit = qc.mkcircuit(1);

// Create witness
Dense<Field> witness(1, circuit->ninputs);
witness.v_[0] = field.from_int(25);  // y = 25
witness.v_[1] = field.from_int(5);   // x = 5
```

```
// Generate proof
LigeroParam<Field> params(circuit->ninputs, 0, 2, 10);
ZkProof<Field> proof(params);
ReedSolomonFactory rs_factory;
ZkProver<Field, ReedSolomonFactory> prover(*circuit, field, rs_factory);

Transcript transcript;
SecureRandomEngine rng;

prover.commit(proof, witness, transcript, rng);
bool success = prover.prove(proof, witness, transcript);
```

SHA-256 Circuit Example

```
#include "circuits/sha/flatsha256_circuit.h"

// Message to hash
std::string message = "Hello, World!";
const uint8_t* msg_bytes = reinterpret_cast<const uint8_t*>(message.c_str());

// Create SHA-256 circuit
auto circuit = FlatSHA256Circuit<Field>::make_circuit(message.length(), field);

// Create witness
auto witness = FlatSHA256Circuit<Field>::make_witness(
  msg_bytes, message.length(), field);

// Expected hash (public input)
uint8_t expected_hash[32];
// ... compute expected hash ...

// Set public inputs (hash output)
for (size_t i = 0; i < 32; ++i) {
  witness->v_[i] = field.from_int(expected_hash[i]);
}

// Generate proof that message hashes to expected value
// ... use ZkProver as above ...
```

ECDSA Verification Example

```
#include "circuits/ecdsa/verify_circuit.h"

// ECDSA signature components
ECDSASignature signature;
```

```
ECPublicKey public_key;
std::vector<uint8_t> message;

// Create ECDSA verification circuit
auto circuit = ECDSAVerifyCircuit<Field>::make_circuit(field);

// Create witness
auto witness = ECDSAVerifyCircuit<Field>::make_witness(
  signature, public_key, message.data(), message.size(), field);

// Generate proof of valid ECDSA signature
// ... use ZkProver as above ...
```

Performance Considerations

Field Choice

- **Prime Fields**: Best for general arithmetic, good for hash functions
- **Binary Fields**: Efficient for bit operations, good for symmetric crypto
- **Extension Fields**: Required for elliptic curve operations

Circuit Optimization

- **Minimize Depth**: Reduces sumcheck rounds and proof size
- **Maximize Parallelism**: Use multiple copies for batch verification
- **Optimize Constants**: Constant propagation reduces circuit size
- **Reuse Subexpressions**: CSE eliminates redundant computations

Proof Size vs. Time Tradeoffs

- **Ligero Parameters**:
  - Larger `block_enc` → smaller proofs, longer proving time
  - More `nreq` → higher security, larger proofs
  - Higher `rateinv` → smaller proofs, longer proving time

Memory Usage

- **Circuit Size**: O(number of gates)
- **Witness Size**: O(number of wires × number of copies)
- **Proof Size**: O(√(circuit size)) for Ligero
- **Verification Time**: O(proof size + public input size)

Recommended Settings

```
// For small circuits (< 10K gates)
LigeroParam<Field> params(num_witnesses, num_quadratic_constraints,
                          /*rateinv=*/4, /*nreq=*/10);
```

```
// For medium circuits (10K - 1M gates)
LigeroParam<Field> params(num_witnesses, num_quadratic_constraints,
                         /*rateinv=*/8, /*nreq=*/20);

// For large circuits (> 1M gates)
LigeroParam<Field> params(num_witnesses, num_quadratic_constraints,
                         /*rateinv=*/16, /*nreq=*/40);
```

Security Considerations

Cryptographic Assumptions

- **Discrete Log**: Security of Merkle commitments
- **Random Oracle**: Fiat-Shamir transformation security
- **Reed-Solomon**: Low-degree testing security

Implementation Security

- **Constant Time**: Field operations should be constant-time
- **Memory Safety**: All array accesses are bounds-checked
- **Randomness**: Use cryptographically secure random number generation

Protocol Security

- **Soundness**: Probability of accepting false proof is 2^(-)
- **Zero-Knowledge**: Simulator can generate proofs without witness
- **Knowledge Extraction**: Can extract witness from successful prover

---

This documentation covers the core functionality of Longfellow ZK. For the latest updates and detailed API documentation, refer to the source code and academic papers.