Network Working Group
Internet-Draft
Intended status: Informational
Expires: 4 September 2025

M. Frigo
A. Shelat
Google

3 March 2025

libZK: a zero-knowledge proof library
draft-google-cfrg-libzk-00

Abstract

This document defines a technique for generating a succinct non-interactive zero-knowledge argument that for a given input x and a circuit C, there exists a witness w, such that C(x,w) evaluates to 0. The technique here combines the MPC-in-the-head approach for constructing ZK arguments described in Ligero [ligero] with a verifiable computation protocol based on sumcheck for proving that C(x,w)=0.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 4 September 2025.

Copyright Notice

Table of Contents

1. Introduction

A zero-knowledge (ZK) protocol allows a Prover who holds an arithmetic circuit C defined over a finite field F and two inputs (x,w) to convince a Verifier who holds only (C,x) that the Prover knows w such that C(x,w) = 0 without revealing any extra information to the Verifier.

The concept of a zero-knowledge proof was introduced by Goldwasser, Micali, and Rackoff [GMR], and has since been rigourously explored and optimized in the academic literature.

There are several models and efficiency goals that different ZK systems aim to achieve, such as reducing prover time, reducing verifier time, or reducing proof size. Some ZK protocols also impose other requirements to achieve their efficienc goals. This document considers the scenario in which there are no common reference strings, or trusted parameter setups that are available to the

parties. This immediately rules out several succinct ZK proof systems from the literature. In addition, this document also focuses on proof systems that can be instantiated from a collision-resistant hash function and require no other complexity theoretic assumption. Again, this rules out several schemes in the literature. All of the ZK schemes from the literature that remain can be defined in the Interactive Oracle Proof (IOP) model, and this document specifies a particular one that enjoys both efficiency and simplicity.

1.1. The libzk system

This document specifies the efficient ZK proof system that is described by Frigo and shelat [libzk]. This proof system consists of two major components: the outer proof is a Ligero ZK proof that checks a property on a committed transcript; the commited transcript corresponds to a proof for a bespoke verifiable-computation scheme that asserts C(x,w)=0. This document first specifies the verifiable computation protocol which is based on the well-known sumcheck protocol. It then specifies the Ligero ZK proof system, and finally specifies how the systems are combined and how the proof is structured.

2. Basic Operations and Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Additionally, the key words "*MIGHT*", "*COULD*", "*MAY WISH TO*", "*WOULD PROBABLY*", "*SHOULD CONSIDER*", and "*MUST (BUT WE KNOW YOU WON'T)*" in this document are to interpreted as described in RFC 6919 [RFC6919].

Except if said otherwise, random choices in this specification refer to drawing with uniform distribution from a given set (i.e., "random" is short for "uniformly random"). Random choices can be replaced with fresh outputs from a cryptographically strong pseudorandom generator, according to the requirements in [RFC4086], or pseudorandom function.

2.1. Array primitives

The notation A[0..N] refers to the array of size N that contains A[0],A[1],...,A[N-1], i.e., the right-boundary in the notation X..Y is an exclusive index bound. The following functions are used throughout the document:

* copy(n, Dst, Src): copies n elements from Src to Dst with different strides
* axpy(n, Y, A, X): sets Y[i] += A*X[i] for 0 <= i < n.
* sum(n, A): computes the sum of the first n elements in array A
* dot(n, A, Y): computes the dot product of length n between arrays A and Y.
* add(n, A, Y): returns the array [A[0]+Y[0], A[1]+Y[1], ..., A[n-1]+Y[n-1]].
* prod(n, A, Y): returns the array [A[0]*Y[0], A[1]*Y[1], ..., A[n-1]*Y[n-1]].
* equal(n, A, Y): true if A[i]==Y[i] for 0 <= i < n and false otherwise.
* gather(n, A, I): returns the array [A[I[0]], A[I[1]], ..., A[I[n-1]].
* A[n][m] = [0]: initializes the 2-dimensional n x m array A to all zeroes.
* A[0..NREQ] = X : array assignment, this operation copies the first NREQ elements of X into the corresponding indicies of the A array.

2.2. Polynomial operations

This section describes operations on and associated with polynomials that are used in the main protocol.

2.2.1. Extend method in Field F_p

The extend(f, n, m) method interprets the array f[0..n] as the evaluations of a polynomial P of degree less than n at the points 0,...,n-1, and returns the evaluations of the same P at the points 0,...,m-1. For sufficiently large fields |F_p| = p >= n, polynomial P is uniquely determined by the input, and thus extend is well defined.

As there are several algorithms for efficiently performing the extend operation, the implementor can choose a suitable one. In some cases, the brute force method of using Lagrange interpolation formulas to compute each output point independently may suffice. One can employ a convolution to implement the extend operation, and in some cases, either the Number Theoretic Transform or Nussbaumer’s algorithm can be used to efficiently compute a convolution.

2.2.2. Extend method in Field GF 2^k

The previous section described an extend method that applies to odd prime-order finite fields which contain the elements 0,1,2...,m. In the special case of GF(2^k), the extend operator is defined in an opinionated way inspired by the Additive FFT algorithm by Lin et al [additivefft]. Lin et al. define a novel polynomial basis for polynomials as an alternative to the usual monomial basis x^i, and give an algorithm for evaluating a degree-(d-1) polynomial at all d points in a subspace, for d=2^ell, and for polynomials expressed in the novel basis.

Specifically, we implement GF(2^128) as GF{2}[x] / (Q(x)) where

```
Q(x) = x^{128} + x^{7} + x^{2} + x + 1
```

With this choice of Q(x), x is a generator of the multiplicative group of the field. Next, we choose GF(2^16) as the subfield of GF(2^128) with g=x^{(2^{128}-1) / (2^{16}-1)} as its generator, and beta_i=g^i^ for 0 <= i < 16 as the basis of the subfield. For relevant problem sizes, this allows us to encode elements in our commitment scheme with 16-bits instead of 128.

Writing j_i for the i-th bit of the binary representation of j, that is,

```
j = sum_{0 <= i < k} j_i 2^i     j_i \in {0,1}
```

we inject integer j into a field element inj(j) by interpreting the bits of j as coordinates in terms of the basis:

```
inj(j) = sum_{0 <= i < k} j_i beta_i
```

We define the extend operator to interpret the array f[0..n] to consist of the evaluations of a polynomial p(x) of degree at most n-1 at the n points x \in { inj(i) : 0 <= i < n } and to return the set { p(inj(i)) : 0 <= i < m} which consist of the evaluations of the same polynomial p(x) at the injected points 0,...,m-1.

This convention allows this operation to be completed efficiently using various forms of the additive FFT as described in [libzk] [additivefft].

3. Fiat-Shamir primitives

A ZK protocol must in general be interactive whereby the Prover and Verifier engage in multiple rounds of communication. However, in practice, it is often more convenient to deploy so-called ‘‘non-interactive" protocols that only require a single message from Prover to Verifier. It is possible to apply the Fiat-Shamir heuristic to transform a special class of interactive protocols into single-message protocols from Prover to Verifier.

The Fiat-Shamir transform is a method for generating a verifier’s public coin challenges by processing the concatenation of all of the Prover’s messages. The transform can be proven to be sound when applied to an interactive protocol that is round-by-round sound and when the oracle is implemented with a hash function that satisfies a correlation-intractability property with respect to the state function implied by the round-by-round soundness. See Theorem 5.8 of [rbr] for details.

In practice, whether an implementation of the random oracle satisfies this correlation-intractability property becomes an implicit assumption. Towards that, this document adapts best practices in selecting the oracle implementation. First, the random oracle should have higher circuit depth and require more gates to compute than the circuit C that the protocol is applied to. Furthermore, the size of the messages which are used as input to the oracle to generate the Verifier’s challenges should be larger than C. These choices are easy to implement and add very little processing time to the protocol. On the other hand, they seemingly avoid attacks against correlation-intractability in which the random oracle is computed within the ZK protocol thereby allowing the output of the circuit to be related to the verifier’s challenge.

As an additional property, each query to the random oracle should be able to be uniquely mapped into a protocol transcript. To facilitate this property, the type and length of each message is incorporated into the query string.

3.1. Implementation

Let H be a collision-resistant hash function. A protocol consists of multiple rounds in which a Prover sends a message, and a verifier responds with a public-coin or random challenge. The Fiat-Shamir transform for such a protocol is implemented by maintaining a transcript object.

3.1.1. Initialization

At the beginning of the protocol, the transcript object must be initialized.

* transcript.init(session_id): The initialization begins by selecting an oracle, which concretely consists of selecting a fresh session identifier. This process is handled by the encapsulating protocol---for example, the transcript that is used for key exchange for a session can be used as the session identifier as it is guaranteed to be unique.

3.1.2. Writing to the transcript

The transcript object supports a write method that is used to record the Prover's messages. To produce the verifier's challenge message, the transcript object internally maintains a Fiat-Shamir Pseudo-random Function (FSPRF) object that generates a stream of pseudo-random bytes. Each invocation of write creates a new FSPRF object, which we denote by fs.

* transcript.write(msg): appends the Prover's next message to the transcript.

There are three types of messages that can be appended to the transcript: a field element, an array of bytes, or an array of field elements.

* To append a field element, first the byte designator 0x1 is appended, and then the canonical byte serialization of the field element is appended.

* To append an array of bytes, first the byte designator 0x2 is appended, an 8-byte little-endian encoding of the number of bytes in the array is appended, and then the bytes of the array are appended.

* To append an array of field elements, the byte designator 0x3 is added, an 8-byte little-endian encoding of the number of field elements is appended, and finally, all of the field elements in array order are serialized and appended.

3.1.3. Special rules for the first message

The write method for the first prover message incorporates additional steps that enhance the correlation-intractability property of the oracle. To process the Prover's first message (which is usually a commitment):

1. The Prover message is appended to the transcript. Specifically, the length of the message, as per the above convention, is appended, and then the bytes of the message are appended.
2. Next, an encoding of the statement to be proven, which consists of the circuit identifier, and a serialization of the input and output of the statement is appended. Each of these three message are added as byte sequences, with their length appended as per convention.
3. Finally, the transcript is augmented by the byte-array 0^(|C|), which consists of |C| bytes of zeroes.

One might at first think of performing steps 2 and 3 first so as to simplify the description of the protocol, and moreover step 3 may appear to be unnecessary. Performing the steps in the indicated order protects against the attack described in [krs], under the assumption that it is infeasible for a circuit C that contains |C| arithmetic gates to compute the hash of a string of length |C|.

Subsequent calls to the write method are used to record the Prover's response messages msg. In this case, the message is appended following the conventions described above.

3.2. The FSPRF object

Each write internally creates an FSPRF object fs that is seeded with the hash digest of the transcript at the end of the write operation.

The FSPRF object is defined to produce an infinite stream of bytes that can be used to sample all of the verifier's challenges in this round. The stream is organized in blocks of 16 bytes each, numbered consecutively starting at 0. Block i contains

```
AES256(KEY, ID(i))
```

where KEY is the seed of the FSPRF object, and ID(i) is the 16-byte little-endian representation of integer i.

The FSPRF object supports a bytes method:

* b = fs.bytes(n) returns the next n bytes in the stream.

Thus, fs implicitly maintains an index into the next position in the stream. Calls to bytes without an intervening write read pseudo-random bytes from the same stream.

3.3. Generating challenges

Whenever the prover is done sending messages in the interactive protocol, it can make a sequence of calls to transcript.generate_{nat,field_element,challenge} to obtain the Verifier's random challenges.

The bytes method of the FSPRF is used by the transcript object to sample pseudo-random field elements and pseudo-random integers via rejection sampling as follows:

* transcript.generate_nat(m) generates a random natural between 0 and m-1 inclusive, as follows.

Let l be minimal such that 2^l >= m. Let nbytes = ceil(l / 8). Let b = fs.bytes(nbytes). Interpret bytes b as a little-endian integer k. Let r = k mod 2^l, i.e., mask off the high 8 * nbytes - l bits of k. If r < m return r, otherwise start over.

* transcript.generate_field_element(F) generates a field element.

If the field F is Z / (p), return generate_nat(fs, p) interpreted as a field element.

If the field is GF(2)[X] / (X^128 + X^7 + X^2 + X + 1) obtain b = fs.bytes(16) and interpret the 128 bits of b as a little-endian polynomial. This document does not specify the generation of a field element for other binary fields, but extensions SHOULD follow a similar pattern.

* a = transcript.generate_challenge(F, n) generates an array of n field elements in the straightforward way: for 0 <= i < n in ascending order, set a[i] = transcript.generate_field_element(F).

4. Overview of the ZK protocol

The full ZK protocol is a variant of the sumcheck protocol, modified to support zero knowledge.

Informally, the non-ZK sumcheck prover takes the description of a circuit and the concrete values of all the wires in the circuit, and produces a proof that all wires have been computed correctly. The proof itself is a sequence of field elements. In the ZK variant used in this document, besides the circuit and the wires, the sumcheck prover takes a random one-time pad and it outputs a "padded" proof such that each element in the padded proof is the difference of the element in the non-padded proof and of the element in the pad. (The choice of "difference" instead of "sum" is purely a matter of convention.)

In this ZK sumcheck variant, the verifier cannot check the proof directly, because it cannot access the pad. Instead of running the sumcheck verifier directly, a commitment scheme is used to hide the pad, and the sumcheck verifier is translated into a sequence of linear and quadratic constraints on the inputs and the pad. The commitment scheme then produces a proof that the constraints are satisfied.

Some of the wires of the circuit are _inputs_, i.e., set outside the circuit and not computed by the circuit itself. Some of the inputs are _public_, i.e., known to both parties, and some are _private_, i.e., known only to the prover. Sumcheck does not use the distinction between public and private inputs, but it needs to distinguish inputs from the pad. On the contrary, the commitment scheme does not use public inputs at all, but it does treat private inputs and the pad equally. These constraints motivate the following terminology.

* _public inputs_: inputs to the circuit known to both parties.
* _private inputs_: inputs to the circuit known to the prover but not to the verifier.
* _inputs_: both public and private inputs. When forming an array of all inputs, the public inputs come first, followed by the private inputs.
* _witnesses_: the private inputs and the pad. When forming an array of all witnesses, the private inputs come first, followed by the pad.

Thus, at a high level, the sequence of operations in the ZK protocol is the following:

1. The prover commits to all witnesses.

2. The prover takes all inputs and the pad, runs the padded sumcheck prover producing a padded proof, and sends the padded proof to the verifier.

3. Both the prover and the verifier take the public inputs and the padded proof and produce a sequence of constraints.

4. Using the commitment scheme and the witnesses, the prover generates a proof that the constraints from step 3 are satisfied.

5. The verifier uses the proof from step 4 and the constraints from step 3 to check the constraints.

Steps 2 and 3 are referred to as "sumcheck", and the rest as "commitment scheme". While the classification of step 3 as "sumcheck" is somewhat arbitrary, there are situations where one might want to use a commitment scheme other than the Ligero protocol specified in this document. In this case, the "commitment scheme" can change while the "sumcheck" remains unaffected.

5. Sumcheck

5.1. Special conventions for sumcheck arrays

The square brackets A[j] denote generic array indexing.

For the arrays of field elements used in the sumcheck protocol, however, it is convenient to use the conventions that follow.

The sumcheck array A[i] is implicitly assumed to be defined for all nonnegative integers i, padding with zeroes as necessary. Here, "zero" is well defined because A[] is an array of field elements.

Arrays can be multi-dimensional, as in the three-dimensional array Q[g, l, r]. It is understood that the array is padded with infinitely many zeroes in each dimension.

Given array A[] and field element x, the function bind(A, x) returns the array B such that

```
B[i] = (1 - x) * A[2 * i] + x * A[2 * i + 1]
```

In case of multiple dimensions such as Q[g, l, r], always bind across the first dimension. For example,

```
bind(Q, x)[g, l, r] =
   (1 - x) * Q[2 * g, l, r] + x * Q[2 * g + 1, l, r]
```

This bind can be generalized to an array of field elements as follows:

```
bindv(A, X) =
     A                                  if X is empty
     bindv(bind(A, X[0]), X[1..])       otherwise
```

Two-dimentional arrays can be transposed in the usual way:

```
transpose(Q)[l, r] = Q[r, l] .
```

5.2. The EQ[] array

EQ_{n}[i, j] is a special 2D array defined as

```
EQ_{n}[i, j] = 1   if i = j and i < n
               0   otherwise
```

The sumcheck literature usually assumes that n is a power of 2, but this document allows n to be an arbitrary integer. When n is clear from context or unimportant, the subscript is omitted like EQ[i, j].

EQ[] is important because the general expansion

```
V[i] = SUM_{j} EQ[i, j] V[j]
```

commutes with binding, yielding

```
bindv(V, X) = SUM_{j} bindv(EQ, X)[j] V[j] .
```

That is, one way to compute bindv(V, X) is via dot product of V with bindv(EQ, X). This strategy may or may not be advantageous in practice, but it becomes mandatory when bindv(V, X) must be computed via a commitment scheme that supports linear constraints but not binding.

This document only uses bindings of EQ and never EQ itself, and therefore the whole array never needs to be stored explicitly. For n = 2^l and X of size l, bindv(EQ_{n}, X) can be computed recursively in linear time as bindv(EQ_{n}, X) = bindeq(l, X) where

```
bindeq(l, X) =
   LET n = 2^l
   allocate B[n]
   IF l = 0 THEN
      B[0] = 1
   ELSE
      LET A = bindeq(l - 1, X[1..])
      FOR 0 <= 2 * i < n DO
         B[2 * i]     = (1 - X[0]) * A[i]
         B[2 * i + 1] = X[0] * A[i]
      ENDFOR
   ENDIF
   return B
```

For m <= n, bindv(EQ_{n}, X)[i] and bindv(EQ_{m}, X)[i] agree for 0 <= i < m, and thus bindv(EQ_{m}, X)[i] can be computed by padding m to the next power of 2 and ignoring the extra elements. With some care, it is possible to compute bindeq() in-place on a single array of arbitrary size m and eliminate the recursion completely.

5.2.1. Remark

Let m <= n, A = bindv(EQ_{m}, X) and B = bindv(EQ_{n}, X). It is true that A[i] = B[i] for i < m. However, it is also true that A[i] = 0 for i >= m, whereas B[i] is in general nonzero. Thus, care must be taken when computing a further binding bindv(A, Y), which is in general not the same as bindv(B, Y). A second binding is not needed in this document, but certain closed-form expressions for the binding found in the literature agree with these definitions only when m is a power of 2.

5.3. Circuits

5.3.1. Layered circuits

A circuit consists of NL _layers_. By convention, layer j computes wires V[j] given wires V[j + 1], where each V[j] is an array of field elements. A _wire_ is an element V[j][w] for some j and w. Thus, V[0] denotes the output wires of the entire circuit, and V[NL] denotes the input wires.

A circuit is intended to check that some property of the input holds, and by convention, the check is considered successful if all output wires are 0, that is, if V[0][w] = 0 for all w.

5.3.2. Quad representation

The computation of circuit is defined by a set of _quads_ Q[j], one per layer. Given the output of layer j + 1, the output of of layer j is given by the following equation:

```
V[j][g] = SUM_{l, r} Q[j][g, l, r] V[j + 1][l] V[j + 1][r] .
```

The quad Q[j][] is thus a three-dimensional array in the indices g, l, and r where 0 <= g < NW[j] and 0 <= l, r < NW[j + 1]. In practice, Q[j][] is sparse.

The specification of the circuit contains an auxiliary vector of quantities LV[j] with the property that V[j][w] = 0 for all w >= 2^{LV[j]}. Informally, LV[j] is the number of bits needed to name a wire at layer j, but LV[j] may be larger than the minimum required value.

5.3.3. In-circuit assertions

In the libzk system, a theorem is represented by a circuit such that the theorem is true if and only if all outputs of the circuit are zero. It happens in practice that many output wires are computed early in the circuit (i.e., in a layer closer to the input), but because of layering, they need to be copied all the way to output layer in order to be compared against zero. This copy seems to introduce large overheads in practice.

A special convention can mitigate this problem. Abstractly, a layer is represented by _two_ quads Q and Z, and the operation of the layer is described by the two equations

```
V[j][g] = SUM_{l, r} Q[j][g, l, r] V[j + 1][l] V[j + 1][r]
     0  = SUM_{l, r} Z[j][g, l, r] V[j + 1][l] V[j + 1][r]
```

Thus, the Z quad asserts that, for given layer j and output wire g, a certain quadratic combination of the input wires is zero.

The actual protocol verifies a random linear combination of those two equations, effectively operating on a combined quad QZ = Q + beta * Z for some random beta.

To allow for a compact representation of the two quads without losing any real generality, the following conditions are imposed:

* The two quads Q and Z are disjoint: for all layers j and output wire g, if any Q[j][g, ., .] are nonzero, then all Z[j][g, ., .] are zero, and vice versa.

* Z is binary: Z[j][g, l, r] \in {0, 1}

With these choices, the two quads allow a compact sparse representation as a single list of 4-tuples (g, l, r, v) with the following conventions:

* If v = 0, the 4-tuple represents an element of Z, and Z[j][g, l, r] = 1.
* If v != 0, the 4-tuple represents an element of Q, and Q[j][g, l, r] = v.
* All other elements of Q and Z not specified by the list are zero.

Moreover, this compact representation can be transformed into a representation of QZ = Q + beta * Z by replacing all v = 0 with v = beta.

5.4. Representation of polynomials

In a generic sumcheck protocol, the prover sends to the verifier polynomials of a degree specified in advance. In the present document, the polynomials are always of degree 2, and are represented by their evaluations at three points P0 = 0, P1 = 1, and P2, where 0 and 1 are the additive and multiplicative identities in the field. The choice of P2 depends upon the field. For fields of characteristic greater than 2, set P2 = 2 (= 1 + 1 in the field). For GF(2^128) expressed as GF(2)[X] / (X^128 + X^7 + X^2 + X + 1), and set P2 = X. This document does not prescribe a choice of P2 for binary fields other than GF(2^128), but other binary fields represented as GF(2)[X] / (Q(X)) SHOULD choose P2 = X for consistency.

5.5. Transform circuit and wires into a padded proof

```
   sumcheck_circuit(circuit, wires, pad, transcript) {
     G[0] = G[1] = transcript.gen_challenge(circuit.lv)
     FOR 0 <= j < circuit.nl DO
        // Let V[j] be the output wires of layer j.
        // The body of the loop reduces the verification of the
        // two "claims" bind(V[j], G[0]) and bind(V[j], G[1])
        // to the verification of the two claims
        // bind(V[j + 1], G'[0]) and bind(V[j + 1], G'[1]),
        // where the new bindings G' are chosen in sumcheck_layer()

        alpha = transcript.gen_challenge(1)

        // Form the combined quad QZ = Q + beta Z
        // to handle in-circuit assertions
        beta = transcript.gen_challenge(1)
        QZ = circuit.layer[j].quad + beta * circuit.layer[j].Z;

        // QZ is three-dimensional QZ[g, l, r]
        QUAD = bindv(QZ, G[0]) + alpha * bindv(QZ, G[1])
        // having bound g, QUAD is two-dimensional QUAD[l, r]

        (proof[j], G) =
            sumcheck_layer(QUAD, wires[j], circuit.layer[j].lv,
                           pad[j], transcript)
     ENDFOR
     return proof
   }
```

```
sumcheck_layer(QUAD, wires, lv, layer_pad, transcript) {
   (VL, VR) = wires
   FOR 0 <= round < lv DO
      FOR 0 <= hand < 2 DO
        Let p(x) =
           SUM_{l, r} bind(QUAD, x)[l, r] * bind(VL, x)[l] * VR[r]
        evals.p0 = p(P0) - layer_pad.evals[round][hand].p0
        // p(P1) is implied and not needed
        evals.p2 = p(P2) - layer_pad.evals[round][hand].p2
        layer_proof.evals[round][hand] = evals
        transcript.write(evals);
        challenge = transcript.gen_challenge(1)
        G[round][hand] = challenge

        // bind the L variable to CHALLENGE
        VL = bind(VL, challenge)
        QUAD = bind(QUAD, challenge)

        // swap L and R
        (VL, VR) = (VR, VL)
        QUAD = transpose(QUAD)
      ENDFOR
   ENDFOR
   layer_proof.vl = VL[0] - layer_pad.vl
   layer_proof.vr = VR[0] - layer_pad.vr
   transcript.write(layer_proof.vl)
   transcript.write(layer_proof.vr)
   return (layer_proof, G)
}
```

5.6. Generate constraints from the public inputs and the padded proof

This section defines a procedure constraints_circuit for transforming the proof returned by sumcheck_circuit into constraints for the commitment scheme. Specifically, each layer produces one linear constraint and one quadratic constraint.

The main difficulty in describing the algorithm is that it operates not on concrete witnesses, but on expressions in which the witnesses are symbolic quantities. Symbolic manipulation is necessary because the verifier does not have access to the witnesses. To avoid overspecifying the exact representation of such symbolic expressions, the convention is that the prefix sym_ indicates not a concrete value, but a symbolic representation of the value. Thus, w[3] is the fourth concrete witness in the w array, and sym_w[3] is a symbolic representation of the fourth element in the w array. The algorithm does not need arbitrarily complex symbolic expressions. It suffices to keep track of affine symbolic expressions of the form k + SUM_{i} a[i] sym_w[i] for some (concrete, nonsymbolic) field elements k and a[].

```
constraints_circuit(circuit, public_inputs, sym_private_inputs,
                    sym_pad, transcript, proof) {
  G[0] = G[1] = transcript.gen_challenge(circuit.lv)
  claims = [0, 0]
  FOR 0 <= j < circuit.nl DO
     alpha = transcript.gen_challenge(1)
     beta = transcript.gen_challenge(1)
     QZ = circuit.layer[j].quad + beta * circuit.layer[j].Z;
     QUAD = bindv(QZ, G[0]) + alpha * bindv(QZ, G[1])
     (claims, G) = constraints_layer(
               QUAD, circuit.layer[j].lv, sym_pad[j], transcript,
               proof[j], claims, alpha)
  ENDFOR

  // now add constraints that the two final claims
  // equal the binding of sym_inputs at G[0], G[1]

  gamma = transcript.gen_challenge(1)
  LET eq2 = bindv(EQ, G[0]) + gamma * bindv(EQ, G[1])
  LET sym_layer_pad = sym_pad[circuit.nl - 1]
  LET npub = number of elements in public_inputs

  Output the linear constraint
      SUM_{i} (eq2[i + npub] * sym_private_inputs[i])
      - sym_layer_pad.vl
      - gamma * sym_layer_pad.vr
    =
      - SUM_{i} (eq2[i] * public_inputs[i])
      + claims[0]
      + gamma * claims[1]
}
```

```
   constraints_layer(QUAD, wires, lv, sym_layer_pad, transcript,
                     layer_proof, claims, alpha) {
      // Initial symbolic claim, which happens to be
      // a known constant but which will be updated to contain
      // symbolic linear terms later.
      LET sym_claim = claims[0] + alpha * claims[1]

      FOR 0 <= round < lv DO
         FOR 0 <= hand < 2 DO
           LET hp = layer_proof.evals[round][hand]
           LET sym_hpad = sym_layer_pad.evals[round][hand]

           transcript.write(hp);
           challenge = transcript.gen_challenge(1)
           G[round][hand] = challenge

           // Now the unpadded polynomial evaluations are expected
           // to be
           //   p(P0) = hp.p0 + sym_hpad.p0
           //   p(P2) = hp.p2 + sym_hpad.p2
           LET sym_p0 = hp.p0 + sym_hpad.p0
           LET sym_p2 = hp.p2 + sym_hpad.p2

           // Compute the implied p(P1) = claim - p(P0) in symbolic form
           LET sym_p1 = sym_claim - sym_p0

           LET lag_i(x) =
                  the quadratic polynomial such that
                         lag_i(P_k) = 1  if i = k
                                      0  otherwise
                  for 0 <= k < 3

           // given p(P0), p(P1), and p(P2), interpolate the
           // new claim symbolically
           sym_claim =   lag_0(challenge) * sym_p0
                       + lag_1(challenge) * sym_p1
                       + lag_2(challenge) * sym_p2

           // bind L
           QUAD = bind(QUAD, challenge);

           // swap left and right
           QUAD = transpose(QUAD)
         ENDFOR
      ENDFOR

      // now the bound QUAD is a scalar (a 1x1 array)
      LET Q = QUAD[0,0]
```

```
      // now verify that
      //
      //   SYM_CLAIM = Q * VL * VR
      //
      // where VL = layer_proof.vl + layer_pad.vl
      //       VR = layer_proof.vr + layer_pad.vr

      // decompose SYM_CLAIM into the known constant
      // and the symbolic part
      LET known + symbolic = sym_claim

      Output the linear constraint
         symbolic
         - (Q * layer_proof.vr) * sym_layer_pad.vl
         - (Q * layer_proof.vl) * sym_layer_pad.vr
         - Q * sym_layer_pad.vl_vr
        =
         Q * layer_proof.vl * layer_proof.vl - known

      Output the quadratic constraint

         sym_layer_pad.vl * sym_layer_pad.vr = sym_layer_pad.vl_vr

      transcript.write(layer_proof.vl)
      transcript.write(layer_proof.vr)

      return (G, [layer_proof.vl, layer_proof.vr])
   }
```

6. Ligero ZK Proof

This section specifies the construction and verification method for a Ligero zero-knowledge proof. The Ligero system is described by Ames, Hazay, Ishai, and Venkitasubramaniam [ligero], and this specification closely follows the academic paper.

6.1. Merkle trees

This section describes how to construct a Merkle tree from a sequence of n strings, and how to verify that a given string x was placed at leaf i in a Merkle tree. These methods do not assume that n is a power of two. This construction is parameterized by a cryptographic hash function such as SHA-256 [RFC6234]. In this application, a leaf in a tree is a message digest instead of an arbitrary string; for example, if the hash function is SHA-256, then the leaf is a 32-byte string.

A tree that contains n leaves is represented by an array of 2 * n message digests in which the input digests are written at indicies n..(2*n - 1). The tree is constructed by iteratively hashing indicies 2*j with 2*j+1, starting at j=n-1, and continuing until j=1. The root is at index 1.

6.1.1. Constructing a Merkle tree from n digests

```
struct {
   Digest a[2 * n]
} MerkleTree

def set_leaf(M, pos, leaf) {
  assert(pos < M.n)
  M.a[pos + n] = leaf
}

def build_tree(M) {
  FOR M.n < i <= 1 DO
    M.a[i] = hash(M.a[2 * i] || M.a[2 * i + 1])
  return M.a[1]
}
```

6.1.2. Constructing a proof of inclusion

This application of the Merkle tree requires verifying that a batch of k input digests at indicies i[0],...,i[k-1] belong to the tree. The simplest way to generate such a proof is to produce independent proofs for each of the k leaves. However, this turns out to be wasteful in that internal nodes could be included multiple times along different paths, and some nodes may not need to be included at all because they are implied by nodes that have already been included.

To address these inefficiencies, this section explains how to produce a batch proof of inclusion for k leaves. The main idea is to start from the requested set of leaves and build all of the implied internal nodes given the leaves. For example, if sibling leaves are included, then their parent is implied, and the parent need not be included in the compressed proof. Then it suffices to revisit the same tree and include the necessary siblings along all of the Merkle paths. It is assumed that the verifier already has the leaf digests that are at the indicies, and thus the proof only contains the necessary internal nodes of the Merkle tree that are used to verify the claim.

It is important in this formulation to treat the input digests as a sequence, i.e. with a given order. Both the prover and verifier of this batch proof needs to use the same order of the requested_leaves array.

```
def compressed_proof(M, requested_leaves[], n) {
  marked = mark_tree(requested_leaves, n)
  FOR n < i <= 1 DO
    IF (marked[i]) {
      child = 2 * i
      IF (marked[child]) {
        child += 1
      }
      IF (!marked[child]) {
        proof.append(M.a[child])
      }
    }
  return proof
}

def mark_tree(requested_leaves[], n) {
  bool marked[2 * n]   // initialized to false

  for(index i : requested_leaves)
    marked[i + n] = true

  FOR n < i <= 1 DO
    // mark parent if child is marked
    marked[i] = marked[2 * i] || marked[2 * i + 1];

  return marked
}
```

6.1.3. Verifying a proof of inclusion

This section describes how to verify a compressed Merkle proof. The claim to verify is that "the commitment root defines an n-leaf Merkle tree that contains k digests s[0],..s[k-1] at corresponding indicies i[0],...i[k-1]." The strategy of this verification procedure is to deduce which nodes are needed along the k verification paths from index to root, then read these values from the purported proof, and then recompute the Merkle tree and the consistency of the root digest. As an optimization, the defined[] array avoids recomputing internal portions of the Merkle tree that are not relevant to the verification. By convention, a proof for the degenerate case of k=0 digests is defined to fail.

```
def verify_merkle(root, n, k,  s[], indicies[], proof[]) {
  tmp = []
  defined = []

  proof_index = 0
  marked = mark_tree(indicies, n)
  FOR n < i <= 1 DO
    if (marked[i]) {
      child = 2 * i
      if (marked[child]) {
        child += 1
      }
      if (!marked[child]) {
        if proof_index > |proof| {
          return false
        }
        tmp[child] = proof[proof_index++]
        defined[child] = true
      }
    }

  FOR 0 <= i < k DO
    tmp[indicies[i] + n] = s[i]
    defined[indicies[i] + n] = true

  FOR n < j <= 1 DO
    if defined[2 * i] && defined[2 * i + 1] {
      tmp[i] = hash(tmp[2 * i] || tmp[2 * i + 1])
      defined[i] = true
    }

  return defined[1] && tmp[1] = root
}
```

6.2. Common circuit parameters

The Prover and Verifier must agree on a circuit C, and this circuit implicitly defines the following parameters. Section describes how these parameters are serialized with C. It is assumed that both Prover and Verifier have analyzed C before-hand, and trust its implementation as well as the choice of these parameters.

* NREQ: The number of columns of the commitment matrix that the Verifier requests to be revealed by the Prover.
* rate: The inverse rate of the error correcting code. This parameter, along with NREQ and Field size, determines the soundness of the scheme.
* BLOCK: the size of each row, in terms of number of field elements

* WR: the number of witness values included in each row
* QR: the number of quadratic constraints written in each row
* IW: Row index at which the witness values start, usually IW = 2.
* IQ: Row index at which the quadratic constraints begin, it is the first row after all of the witnesses have been encoded.
* NL: Number of linear constraints.
* NQ: Number of quadratic constraints.
* NWROW: Number of rows used to encode witnesses.
* NQT: Number of row triples needed to encode the quadratic constraints.
* NQW: NWROW + NQT, rows needed to encode witnesses and quadratic constraints.
* NROW: Total number of rows in the witness matrix, NQW + 2
* NCOL: Total number of columns in the tableau matrix, this value is equal to (rate + 1) * BLOCK.

6.2.1. Constraints on parameters

* BLOCK < |F_q| The block size must be smaller than the field size.
* BLOCK > NREQ The block size must be larger than the number of columns requested.
* BLOCK >= 2 * (NREQ + QR) + (NREQ + WR) - 2
* WR >= QR.
* BLOCK >= 2 * (NREQ + WR) - 1.
* QR >= NREQ (and thus WR >= NREQ) to avoid wasting too much space.

6.3. Ligero commitment

The first step of the proof procedure requires the Prover to commit to a witness vector w for the circuit C.

The commitment is the root of a Merkle tree. The leaves of the Merkle tree are a sequence of columns of the tableau matrix T[][].

This tableau matrix is constructed row-by-row by applying the extend procedure to arrays that are formed from random field elements and elements copied from the witness vector. Matrix T[][] has size NROW x NCOL and has the following structure:

```
row ILDT = 0                          : RANDOM
row IDOT = 1                          : RANDOM
row i for IW = IDOT + 1 <= i < IQ     : witness rows
row i for IQ <= i < NROW              : quadratic rows
```

1) The first ILDT row is defined as

   extend(RANDOM[BLOCK], BLOCK, NCOL)

by selecting BLOCK random field elements and applying extend.

2) The second IDOT row is formed in the same way with as the first row with the extra constraint that the first BLOCK elements of the row sum to zero. This constraint can be satisfied by selecting BLOCK-1 random field elements, and then setting the last one to be the additive inverse of the sum of the previous elements.

3) The next rows from IW=2,...,IQ are _padded witness_ rows that contain random elements and portions of the witness vector. Specifically, row i is formed by applying extend to an array that consists of R random elements and then WR elements from the vector W:

```
extend([RANDOM[NREQ], W[(i-2) * WR .. (i-1) * WR]], BLOCK, NCOL)
```

4) The final portion of the witness matrix consists of _padded quadratic_ rows that consists of NREQ random elements and QR witness elements:

```
extend([RANDOM[NREQ], QQ[QR]], NREQ + QR, NCOL)
```

The specific elements in the QQ array are determined by the quadratic constraints on the witness values that are verified by the proof.

The protocol is such that a quadratic constraint induces three entries in separate quadratic blocks. Thus, for NQ total quadratic constraints and Q entries per block, there are a total of 3 * (NQ / Q) quadratic blocks. The code stores NQT = (NQ / Q) instead of the number 3 * NQTRIPLES of blocks.

The second step of the procedure is to compute a Merkle tree on columns of the tableau matrix. Specifically, the i-th leaf of the tree is defined to be columns BLOCK...NCOL of the i-th row of the tableau T.

Input:

* The witness vector w.
* Array of quadratic constraints lqc[], which consists of triples (x,y,z) that represent the constraint that w[x] * w[y] = w[z].

Output:

* A digest; root of a Merkle tree formed from columns of the tableau.

```
   def commit(w[], lqc[]) {
       T[NROW][NCOL] = [0];   // 2d array initialized with 0

       layout_zk_rows(T);
       layout_witness_rows(T, w);
       layout_quadratic_rows(T, W, lqc);

       MerkleTree M;
       FOR BLOCK <= j < NCOL DO
         M.set_leaf(j - BLOCK,
             hash( T[0][j] || T[1][j] || .. || T[NROW][j]) );

       return M.build_tree();
   }

   def layout_zk_rows(T) {
       T[0][0..NCOL] = extend(random_row(BLOCK), BLOCK, NCOL);
       dot = random_row(BLOCK - 1);
       dot[BLOCK-1] = - sum(dot[0...BLOCK-2]);
       T[1][0..NCOL] = extend(dot, BLOCK, NCOL);
   }

   def layout_witness_rows(T, w) {

     FOR IW <= i <= IQ DO
       bool subfield = false;

       IF w[i * WR .. (i+1) * WR] are all in the subfield {
         subfield = true;
       }

       row[0...NREQ-1] = random_row(NREQ, subfield)
       row[NREQ..BLOCK] = w[i * WR .. (i+1) * WR]

       T[i + IW][0..NCOL] = extend(row, BLOCK, NCOL)
   }
```

```
def layout_quadratic_rows(T, w, lqc[]) {
    FOR 0 <= i < NQT DO
      rowx[0..NREQ] = random_row(NREQ)
      rowy[0..NREQ] = random_row(NREQ)

      FOR 0 <= j < NQT  DO
        IF (j + i * Q < NQ)
          assert( w[ lqc[j].x ] * w[ lqc[j].x ] == w[ lqc[j].z ] )
          rowx[NREQ + j] = w[ lqc[j].x ]
          rowy[NREQ + j] = w[ lqc[j].y ]

      T[i + IQ][0..NCOL]       = extend(rowx, BLOCK, NCOL)
      T[i + IQ + NQT][0..NCOL] = extend(rowy, BLOCK, NCOL)

      // Compute the z row point-wise to same time.
      FOR 0 <= j < NCOL DO
        T[i + IQ + 2 * NQT][j] = T[i + IQ][j] * T[i + IQ + NQT][j]
}
```

6.4. Ligero Prove

This section specifies how a Ligero proof for a given sequence of linear constraints and quadratic constraints on the committed witness vector w is constructed. The proof consists of a low-degree test on the tableau, a linearity test, and a quadratic constraint test.

6.4.1. Low-degree test

In the low-degree test, the verifier sends a challenge vector consisting of NROW field elements, u[0..NROW]. This challenge is generated via the Fiat-Shamir transform. The prover computes the sum of u[i]*T[i] where T[i] is the i-th row of the tableau, and returns the first BLOCK elements of the result. The verifier applies the extend method to this response, and then verifies that the extended row is consistent with the positions of the Merkle tree that the verifier will later request from the Prover.

The Prover’s task is therefore to compute a summation. For efficiency, set u[0]=1 because this first row corresponds to a random row meant to ‘‘pad" the witnesses for zero-knowledge.

6.4.2. Linear and Quadratic constraints

The linear test is represented by a matrix A, and a vector b, and aims to verify that A*w = b. The constraint matrix A is given as input in a sparse form: it is an array of triples (c,w,k) in which c indicates the constraint number or row of A, w represents the index of the witness or column of A, and k represents the constant factor. For example, if the first constraint (at index 0) was w[2] + 2w[3] = 3, then the linear constraints array contains the triples (0,2,1), (0,3,2) and the b vector has b[0]=3.

The quadratic constraints are given as input as an array names lqc[] that contains triples (x,y,z); one such triple represents the constraint that w[x] * w[y] = w[z]. To process quadratic constraints, the Tableau is augmented with 3 extra rows, called Ax, Ay, and Az which hold _copied_ witnesses and their products. If the i-th quadratic constraint is (x,y,z), then the prover sets Ax[i] = w[x], Ay[i]=w[y] and Az[i]= w[x] * w[y]. Next, the prover adds a linear constraint that Ax[i] - w[x] = 0, Ay[i] - w[y] = 0 and Az[i] - w[z] = 0 to ensure that the copied witness is consistent.

In this sense, the quadratic constraints are reduced to linear constraints, and the additional requirement for the verifier to check that each index of the Az row is the product of its counterpart in the Ax and Ay row.

6.4.3. Ligero Prover procedure

```
   def prove(transcript, digest, linear[], lqc[])  {

       u = transcript.generate_challenge([BLOCK]);
       transcript.write(digest)

       ldt[0..BLOCK] = T[0][0..BLOCK]

       for(i=1; i < NROW; ++i) {
         ldt[0..BLOCK] += u[i] * T[i][0..BLOCK]
       }

       alpha_l = transcript.generate_challenge([NL]);
       alpha_q = transcript.generate_challenge([NQ,3]);

       A = inner_product_vector(linear, alpha_l, lqc, alpha_q);

       dot = dot_proof(A);

       transcript.write(ldt);
       transcript.write(dot);

       challenge_indicies = transcript.generate_challenge([NREQ]);

       columns = requested_columns(challenge_indicies);

       mt_proof = M.compressed_proof(challenge_indicies);

       return (ldt, dot, columns, mt_proof)
     }
```

```
   Input:
   - linear: array of (w,c,k) triples specifying the linear constraints
   - alpha_l: array of challenges for the linear constraints
   - lqc: array of (x,y,z) triples specifying the quadratic constraints
   - alpha_q: array of challenges for the quadratic constraints

   Output:
   - A: a vector of size WR x NROW that contains the combined
        witness constraints.
        The first NW * W positions correspond to coefficients
        for the linear constraints on witnesses.
        The next 3*NQ positions correspond to coefficients
        for the quadratic constraints.
```

```
   def inner_product_vector(A, linear, alpha_l, lqc, alpha_q) {
     A = [0]

     // random linear combinations of the linear constraints
```

```
    FOR 0 <= i < NL DO
      assert(linear[i].w < NW)
      assert(linear[i].c < NL)
      A[ linear[i].w ] += alpha_l[ linear[i].c ] * linear[i].k

    // pointers to terms for quadratic constraints
    a_x = NW * W
    a_y = NW * W + (NQ * W)
    a_z = NW * W + 2 * (NQ * W)

    FOR 0 <= i < NQT DO
      FOR 0 <= j < QR DO
        IF (j + i * QR < NQ)
          ilqc = j + i * QR  // index into lqc
          ia   = j + i * WR  // index into Ax,Ay,Az sub-arrays
          (x,y,z) = lqc[ilqc]

          // add constraints that the copies are correct
          A[a_x + ia] += alpha_q[ilqc][0]
          A[x]        -= alpha_q[ilqc][0]

          A[a_y + ia] += alpha_q[ilqc][1]
          A[y]        -= alpha_q[ilqc][1]

          A[a_z + ia] += alphaq[ilqc][2]
          A[z]        -= alphaq[ilqc][2]

    return A
  }

  def dot_proof(A) {
    y = T[IDOT][0..BLOCK]

    Aext[0..BLOCK] = [0]
    FOR 0 <= i < NQW DO
      Aext[0..R]  = [0]
      Aext[R..R + WR] = A[i * WR..(i+1) * WR]
      Af = extend(Aext, R+WR, BLOCK)

      add(BLOCK, y,
                 times(BLOCK, Af[0..BLOCK],
                              T[i + IW][0...BLOCK]))

    return y
  }

  def requested_columns(challenge_indicies) {
    cols = []   // array of columns of T
```

```
    FOR (index i : challenge_indicies) {
      cols.append( [ T[0..NROW][i] ] )
    }
    return cols
  }
```

6.5. Ligero verification procedure

This section specifies how to verify a Ligero proof with respect to a common set of linear and quadratic constraints.

Input:
- commitment: the first Prover message that commits to the witness
- proof: Prover’s proof
- transcript: Fiat-Shamir
- linear: array of (w,c,k) triples specifying the linear constraints
- b: the vector b in the constraint equation A*w = b.
- lqc: array of (x,y,z) triples specifying the quadratic constraints

Output:
- a boolean

```
def verify(commitment, proof, transcript,
           linear[], digest, b[], lqc[]) {

  u = transcript.generate_challenge([BLOCK]);
  transcript.write(digest)
  alpha_l = transcript.generate_challenge([NL]);
  alpha_q = transcript.generate_challenge([NQ,3]);
  transcript.write(proof.ldt);
  transcript.write(proof.dot);
  challenge_indicies = transcript.generate_challenge([NREQ]);

  A = inner_product_vector(linear, alpha_l, lqc, alpha_q);

  // check the putative value of the inner product
  want_dot  = dot(NL, b, alpha_l);
  proof_dot = sum(proof.dot);

  return
    verify_merkle(commitment.root, BLOCK*RATE, NREQ,
          proof.columns, challenge_indicies, mt_proof.mt)
    AND quadratic_check(proof)
    AND low_degree_check(proof, challenge_indicies, u)
    AND dot_check(proof, challenge_indicies, A)
    AND want_dot == proof_dot
}
```

```
   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
   }

   def low_degree_check(proof, u, challenge_indicies) {

     got = proof.columns[ILDT][0..NREQ]

     FOR 1 <= i < NROW DO
       axpy(NREQ, got, u[i], proof.columns[i][...])
     }

     row = extend(proof.ldt, BLOCK, NCOL)
     want = gather(NREQ, row, challenge_indicies)

     return equal(NREQ, got, want)
   }

   def dot_check(proof, challenge_indicies, A) {
     yc = proof.columns[IDOT][0..NREQ]

     Aext[0..BLOCK] = [0]
     FOR 0 <= i < NQW DO
       Aext[0..R]  = [0]
       Aext[R..R + WR] = A[i * WR..(i+1) * WR]
       Af = extend(Aext, R + WR, BLOCK)

       Areq = gather(NREQ, Af, challenge_indicies);

       // Accumulate z += A[j] \otimes W[j].
       sum( yc, prod(NCOL, Areq[0..NREQ],
                           proof.columns[i][0..NREQ]))

     row = extend(proof.dot, BLOCK, NCOL)
     yp  = gather(NREQ, row, challenge_indicies)

     return equal(NREQ, yp, yc)
   }
```

7. Serializing objects

This section explains how a proof consists of smaller, related objects, and how to serialize each such component. First, the standard methods for serializing integers and arrays are used:

* write_size(n): serializes an integer in [0, 2^{24} - 1] that represents the size of an array or an index into an array. The integer is serialized in little endian order.
* write_array(arr): A variable-sized array is represented as type array[] and serialized by first writing its length as a size element, and then serializing each element of the array in order.
* write_fixed_array(arr): When the length of the array is explicitly known to be n, it is specified as type array[n] and in this case, the array length is not written first.

7.1. Serializing structs

When a section includes just a struct definition, it is serialized in the natural way, starting from the top-most component and proceeding to the last one, each component is serialized in order.

7.2. Serializing Field elements

This section describes a method to serialize field elements, particularly when the field structure allows efficient encoding for elements of subfields.

Before a field element can be serialized, the context must specify the finite field. In most cases, the Circuit structure will specify the finite field, and all other aspects of the protocol will be defined by this field.

A finite field or FieldID is specified using a variable-length encoding. Common finite fields have been assigned special 1-byte codes. An arbitrary prime-order finite field can be specified using the special 0xF_ byte followed by a variable number of bytes to specify the prime in little-endian order. For example, the 3 byte sequence f11001 specifies F_257. Similarly, a quadratic extension using the polynomial x^2 + 1 can be specified using the 0xE_ designators.

Finite field	FieldID
p256	0x01
p384	0x02
p521	0x03
GF(2^128)	0x04
GF(2^16)	0x05
2^128 - 2^108 + 1	0x06
2^64 - 59	0x07
2^64 - 2^32 + 1	0x08
F_{2^64 - 59}^2	0x09
secp256	0x0a
F_{2^({0--15})-byte prime}^2	0xe{0--f}
F_{2^({0--15})-byte prime}	0xf{0--f}

Table 1: Finite field identifiers.

The GF(2^128) field uses the irreducible polynomial x^128 + x^7 + x^2 + x + 1. The p256 prime is equal to 115792089210356248762697446949407573530086143415290314195533631308867097853951, which is the base field used by the NIST P256 elliptic curve. The p384 prime is equal to 394020061963944792122790401001436138050797392704654466679482934042457217714968703290472660882589380018616069731123 19 which is the base field used by the NIST P384 curve. The p512 prime is equal to 2^521 - 1. The F_p64^2 field is the quadratic field extension of the base field defined by prime 18446744073709551557 using polynomial x^2 + 1, i.e. by injecting a square root of -1 to the field.

7.2.1. Serializing a single field element

Unless specified otherwise, a field element, referred to as an Elt, is serialized to bytes in little-endian order. For example, a 256-bit element of the finite field F_p256 is serialized into 32-bytes starting with the least-significant byte.

* write_elt(e, F): produces a byte encoding of a field element e in field F.

7.2.2. Serializing an element of a subfield

In some cases, when both Prover and Verifier can explicitly conclude that a field element belongs to a smaller subfield, then both parties can use a more efficient sub-field serialization method. This optimization can be used when the larger field F is a field extension of a smaller field, and both parties can conclude that the serialized element belongs to the smaller subfield.

* write_subfield(Elt e, F2, F1): produce a byte encoding of a field element e that belongs to a subfield F2 of field F1.

7.3. Serializing a Sumcheck Transcript

```
struct {
        PaddedTranscriptLayer layers[];  // NL layers
} PaddedTranscript;

struct {
        Elt wires[];  // array of 2 * log_w Elts that store the
                // evaluations of deg-2 polynomial at 0, 2
        Elt wc0;
        Elt wc1;
} PaddedTranscriptLayer;
```

The padded transcript incorporates the optimization in which the eval at 1 is omitted and reconstructed from the expected value of the previous challenge.

7.4. Serializing a Ligero Proof

```
def serialize_ligero_proof(C, ldt, dot, columns, mt_proof) {
  write_array(ldt, C.BLOCK)
  write_array(dot, C.BLOCK)
  write_runs(columns, C.NREQ * C.NROW, C.subFieldID, C.FieldID)
  write_merkle(mt_proof)
}
```

The concept of a run allows saving space when a long run of field elements belong to a subfield of the Finite field. Runs consist of a 4-byte size element, and then size Elt elements that are either in the field or the subfield. Runs alternate, beginning with full field elements. In this way, rows that consist of subfield elements can save space. The maximum run length is set to 2^25.

```
 def write_runs(columns, N, F2, F) {
     bool subfield_run = false
     FOR 0 <= ci < N DO
       size_t runlen = 0
       while (ci + runlen < N &&
              runlen < kMaxRunLen &&
              columns[ci + runlen].is_in_subfield(F2) == subfield_run) {
         ++runlen;
       }
       write_size(runlen, buf);
       for (size_t i = ci; i < ci + runlen; ++i) {
         if (subfield_run) {
           write_subfield(columns[i], F2, F);
         } else {
           write_elt(columns[i], F);
         }
       }
       ci += runlen;
       subfield_run = !subfield_run;
 }

 def write_merkle(mt_proof) {
   FOR (digest in mt_proof) DO
      write_fixed_array(digest, HASH_LEN)
 }
```

7.5. Serializing a Sequence of proofs

For the multi-field optimization, the proof string consists of a sequence of two proofs. This is handled by using the circuit identifier to specify the sequence of proofs to parse.

```
struct {
   Public pub;  // Public arguments to all circuits
   Proof proofs[]; // array of Proof
} Proofs;

struct {
  uint8 oracle[32]; // nonce used to define the random oracle,
  Digest com;       // commitment to the witness
  PaddedTranscript sumcheck_transcript;
  LigeroProof lp;
} Proof;

struct {
  char* arguments[];   // array of strings representing
                       // public arguments to the circuit
} Public;
```

7.6. Serializing a Circuit

A circuit structure consists of size metadata, a table of constants, and an array of structures that represent the layers of the circuit as follows.

```
struct {
  Version version;     // 1-byte identifier, 0x1.
  FieldID field;       // identifies the field
  FieldID subfield;    // identifies the subfield
  size nv;             // number of outputs
        size pub_in;         // number of public inputs
        size ninputs;        // number of inputs, including witnesses
        size nl;             // number of layers
        Elt const_table[];   // array of constants used by the quads
        CircuitLayer layers[];  // array of layers of size nl
} Circuit;
```

The const_table structure contains an array of Elt constants that can be referred by any of the CircuitLayer structures. This feature saves space because a typical circuit uses only a handful of constants, which can be referred by a small index value into this table.

```
struct {
  size logw;     // log of number of wires
  size nw;       // number of wires
  Quads quads[];  // array of nw Quads
} CircuitLayer;
```

The quads array stores the main portion of the circuit. Each Quad structure contains a g, h0, h1 and a constant v which is represented as an index into the const_table array in the Circuit. Each g,h0, and h1 is stored as a difference from the corresponding item in the _previous_ quad. In other words, these three values are delta-encoded in order to improve the compressibility of the circuit representation. The Delta spec uses LSB as a sign bit to indicate negative numbers.

```
struct {
  Delta g;     // delta-encoded gate number
  Delta h0;    // delta-encoded left wire index
  Delta h1;    // delta-encoded right wire index
  size v;      // index into the const_table to specify const v
} Quad;

typedef Delta uint;
```

8. Security Considerations

The libzk system satisfies the standard properties of a zero-knowledge argument system: completeness, soundness, and zero-knowledge.

Frigo and shelat [libzk] provide an analysis of the soundness of the system, as it derives from the Soundness of the Ligero proof system and the sumcheck protocol. Similarly, the zero-knowledge property derives almost entirely from the analysis of Ligero [ligero]. It is a goal to provide a mechanically verifiable proof for a high-level statement of the soundness.

9. IANA Considerations

This document does not make any requests of IANA.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC6919] Barnes, R., Kent, S., and E. Rescorla, "Further Key Words for Use in RFCs to Indicate Requirement Levels", RFC 6919, DOI 10.17487/RFC6919, April 2013, <https://www.rfc-editor.org/info/rfc6919>.

10.2. Informative References

[GMR] Goldwasser, S., Micali, S., and C. Rackoff, "THE KNOWLEDGE COMPLEXITY OF INTERACTIVE PROOF SYSTEMS", 1989.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[additivefft]
Lin, S., Chung, W., and Y. Han, "Novel polynomial basis and its application to Reed-Solomon erasure codes", 2014, <https://arxiv.org/abs/1404.3458>.

[krs] Khovratovich, D., Rothblum, R. D., and L. Soukhanov, "How to Prove False Statements: Practical Attacks on Fiat-Shamir", 2025, <https://eprint.iacr.org/2025/118>.

[libzk] Frigo, M. and a. shelat, "Anonymous credentials from ECDSA", 2024, <https://eprint.iacr.org/2024/2010>.

[ligero] Ames, S., Hazay, C., Ishai, Y., and M. Venkitasubramaniam, "Ligero: Lightweight Sublinear Arguments Without a Trusted Setup", 2022, <https://eprint.iacr.org/2022/1608>.

[rbr] Canetti, R., Chen, Y., Holmgren, J., Lombardi, A., Rothblum, G., and R. Rothblum, "Fiat-Shamir From Simpler Assumptions", 2018, <https://eprint.iacr.org/2018/1004>.

Appendix A. Acknowledgements

Appendix B. Test Vectors

This section contains test vectors. Each test vector in specifies the configuration information and inputs. All values are encoded in hexadecimal strings.

B.1. Test Vectors for Merkle Tree

B.1.1. Vector 1

* Leaves:
  4bf5122f344554c53bde2ebb8cd2b7e3d1600ad631c385a5d7cce23c7785459a
  dbc1b4c900ffe48d575b5da5c638040125f65db0fe3e24494b76ea986457d986
  084fed08b978af4d7d196a7446a86b58009e636b611db16211b65a9aadff29c5
  e52d9c508c502347344d8c07ad91cbd6068afc75ff6292f062a09ca381c89e71
  e77b9a9ae9e30b0dbdb6f510a264ef9de781501d7b6b92ae89eb059c5ab743db
* Root:
  f22f4501ffd3bdffcecc9e4cd6828a4479aeedd6aa484eb7c1f808ccf71c6e76
* Proof for leaves (0,1):
  084fed08b978af4d7d196a7446a86b58009e636b611db16211b65a9aadff29c5
  f03808f5b8088c61286d505e8e93aa378991d9889ae2d874433ca06acabcd493
* Proof for leaves (1,3):
  e77b9a9ae9e30b0dbdb6f510a264ef9de781501d7b6b92ae89eb059c5ab743db
  084fed08b978af4d7d196a7446a86b58009e636b611db16211b65a9aadff29c5
  4bf5122f344554c53bde2ebb8cd2b7e3d1600ad631c385a5d7cce23c7785459a

B.2. Test Vectors for Circuit

B.2.1. Vector 1

* Description: Circuit C(n, m, s) = 0 if and only if n is the m-th s-gonal number in F_p128. This circuit verifies that 2n = (s-2)m^2 - (s - 4)*m.
* Field: 2^128 - 2^108 + 1 (Field ID 6)
* Depth: 3 Quads: 11 Terms: 11
* Serialization: 0106000001000001000002000004000002000004000fffffff
ffffffffffffffffffefffff000000000000000000000000f0ffff010000000
000000000000000000000fdffffffffffffffffffffffefffff03000006000
003000000000020000000000000000000000080000040000010000000000003000
002000002000002000004000008000000000000000000000020000060000000000
000000000000040000000000000000300000900000200000000002000002000
002000000000002000002000002000000000020000040000000000000000002000
003000003000004000020000

B.3. Test Vectors for Sumcheck

B.3.1. Vector 1

* Description: Circuit C(n, m, s) = 0 if and only if n is the m-th s-gonal number in F_p128. This circuit verifies that 2n = (s-2)m^2 - (s - 4)*m.
* Field: 2^128 - 2^108 + 1 (Field id 6)
* Fiat-Shamir initialized with
* Serialization: 90e734c42b5f14ee432a0ed95ba2ada05c3f9ecc9b026ded61f
00bf57434f93c6f70e9c8b6e3de005ba8b4da93b5fa35fc3efae1e6068399c7f7d
009ab5a2711084c97cd5a6e28dd30c598907b328d81915e487c34dbf80aa5da14f
0621011a33d838a7b0d9a03533c63c6606f5360f88cf97c728630afdcb9755894a
6f5c9068e1fc29f97efc125ba580de64089c6e72433de2a3267b90daeaf418ac8a
3df3bbddc6cb141c764c8262346baac2e28033778b1a71f153ba571e80ab29951f
9440ba93fede225a35accf6e0114d5240ae92df02d2870e5258ebba416f3d815e1
554b05627998fc9d3bf354b89394b27b39f69c6538dbc968a779369e47f214252e
0955624e9f4d6dc2a95cf41c57703b8749b959315458d4076f0daf5fdbde23e16c
10394ac884ab9cad0782e8f472cb4edb69682d17465363691aafc31b83cd764fb9
09b50e2fe907fd2137566ddb8c47cc13974957e7f76180860571035f7a4d2658a8
2e1be8fe155353bc10feae9541365926f0646b4a5351907cbd5d9dbb4

B.4. Test Vectors for Ligero

B.4.1. Vector 1

* Description: Circuit C(n, m, s) = 0 if and only if n is the m-th s-gonal number in F_p128. This circuit verifies that 2n = (s-2)m^2 - (s - 4)*m.
* Field: 2^128 - 2^108 + 1 (Field id 6)

* Witness vector: [1, 45, 5, 6]
* Pad elements: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4]
* Parameters:
  - NREQ: 6
  - RATE: 4
  - WR: 20
  - QR: 2
  - NROW: 7
  - NQ: 1
  - BLOCK: 51
* Commitment: 738d2ffb3a8bf24e7aedb94be59041fb2dc13da30fe6b05ebe5126ef8fc36ec2
* Proof size: 3180 bytes
* Proof: fa8d88a73b3a0f9c067658c45bb394a6020000000000000000000000000000000fa8d8...2cd5f61cd2b2eb84c79e1707cbad0048fcd820c716584f31991cf1628fb041

B.5. Test Vectors for libzk

Authors' Addresses

Matteo Frigo
Google
Email: matteof@google.com

abhi shelat
Google
Email: shelat@google.com