# Placeholder

Yeah, that is the actual proof system name.

Alisa Cherniaeva
=nil; Foundation
a.cherniaeva@nil.foundation

Ilia Shirobokov
=nil; Foundation
i.shirobokov@nil.foundation

Mikhail Komarov
=nil; Foundation
nemo@nil.foundation

November 15, 2022

**Abstract**

*Proof systems evolution brought a wide variety of ways to provide a short description of a computation performed. Each of them brings different trade-offs between proving time, proof size, verification complexity, absence or presence of a zero-knowledge properties, recursion friendliness in the various forms of trade-offs, putting all the same components together in a different way.*

*This paper introduces a proof system called Placeholder focused on providing the most compact circuit representation along with the cheapest verification within different environments (register-based executors, certain stack-based executors). This is achieved by adjusting arithmetization methods, commitment schemes, verification process commitments handling way and switching in-recursion layers proof sub-systems components themselves.*



## 1 Introduction

## 2 Preliminaries

### 2.1 Groups

A binary operation $*$ on a set $G$ is a mapping from $G \times G$ to $G$, which associates to elements $x$ and $y$ of $G$ a third element $x * y$ of $G$.

**Definition 1** (Group)**.** A group $(G, *)$ consists of a set $G$ together with a binary operation $*$ for which the following properties are satisfied:

- Associativity: $(x * y) * z = x * (y * z),\ \forall x, y, z \in G$
- Neutral element: $\exists!\ e \in G,\ e * x = x = x * e,\ \forall x \in G$
- Inverse element: $\forall x \in G,\ \exists!\ x' \in G,\ x * x' = e = x' * x$ where $e$ is the neutral element of $G$.

A group $G$ is Abelian (or commutative) if: $x * y = y * x, \forall x, y \in G$

**Definition 2** (Cyclic Group)**.** A group $G$ is said to be cyclic, with generator $x$, if every element of $G$ is of the form $x^n$ for some integer $n$.

### 2.2 Fields

A field $F$ is a set with two binary operations $+$ and $*$ that satisfies the following field axioms:

- Closure under addition: $\forall x, y \in F,\ x + y \in F$
- Closure under multiplication: $\forall x, y \in F,\ x * y \in F$
- Additive inverses: $\forall x \in F, y \in F$ such that $x + y = 0$
- Multiplication inverses: $\forall x \in F$ such that $x \neq 0,\ \exists\ y \in F$ such that $x * y = 1$, $y$ is called the multiplicative inverse of $x$ and is denoted $x^{-1}$ or $\frac{1}{x}$
- The distributive law: $\forall x, y, z \in F,\ x * (y + z) = x * y + x * z$

**Definition 3** (Finite Field)**.** A finite field is a field $F_q$ with a finite number of elements. The order of a finite field

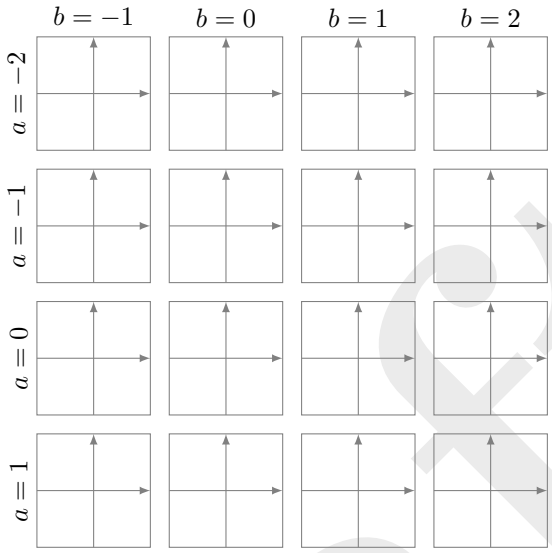
$$|F_q| = q = p^k$$

for some integer $k \geq 1$ and $p$ prime equals the number of elements in the field.

## 2.3 Elliptic Curves

For Placeholder pairing friendly elliptic curves defined over very large finite fields $|F_p| = 2^{256}$ are used. In the following we will explain what is an elliptic curve, pairings and elliptic curves over finite fields.

**Definition 4** (Elliptic Curve)**.** An elliptic curve $E$ is a mathematical object defined over a field $F$ and generally expressed in the following Weierstrass form: $y^2 = x^3 + ax + b$ for some $a, b \in F_q$ where $(x, y)$ are called "affine coordinates"




$$y^2 = x^3 + a \cdot x + b$$

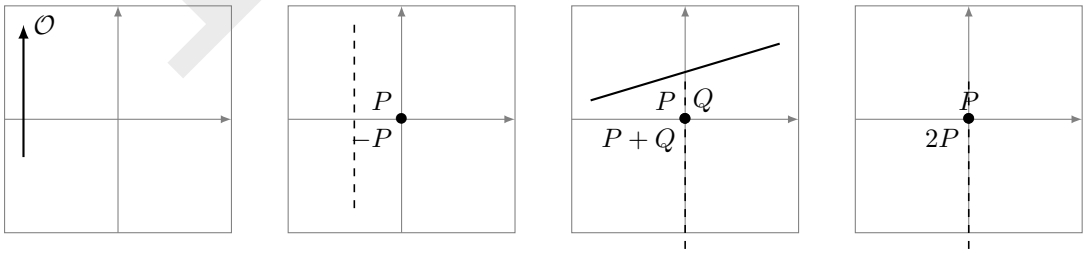
### 2.3.1 Elliptic Curves Over Finite Fields

We will mainly focus on elliptic curves over finite fields since they are the ones used for cryptographic applications. An elliptic curve over a finite field $F_q$ is an abelian group $G$ with a finite number of points $n$ such that $n = |G|$ (order of the group $G$).

The group on which the elliptic curve is defined is the one used for many cryptographic protocols. An elliptic curve over a finite field is represented in the projective plane, such a curve will have an additional point at infinity $O$ which serves as the identity of the group. There are several ways to define the curve equation, but for the purpose of doing the group law arithmetics, let $y^2 = x^3 + b$ for some constant $b \in F_q$.

**Definition 5** (Group Law)**.** We can define an abelian group $G$ over elliptic curves as follows:

- The elements of the group are the points of an elliptic curve
- The identity element is the point $O$ defined as $(0, 1, 0)$
- The inverse of a point $P = (x, y, z)$ is the point $-P = (x, -y, z)$
- Commutativity: $P + Q = Q + P$
- Associativity: $P + (Q + R) = (P + Q) + R$




Neutral element $\mathcal{O}$ | Inverse element $-P$ | Addition $P + Q$ "Chord rule" | Doubling $P + P$ "Tangent rule"

We can identify four cases in the group law: point addition $P + Q = -R$ which basically draws the line that intersects both points and finds a third point $R$, the addition is simply $-R$ (negate the y-coordinate). The second case $P + Q + Q = O$ is basically when the addition of both points $P$ and $Q$ doesn't result in a

third point $R$ cause in this case the line passing through $P$ and $Q$ is tangent to the curve. Let us assume that $Q$ is the tangency point, then $P + Q = -Q$. The third case $P + Q = O$ is when a point $P$ is being added to its negation $Q = -P$, their addition equals point at infinity $O$. The final case $P + P = O$ is when a point is being added to itself (called point doubling).

**Point addition** We will first compute $P + Q$ in the affine form and then projective form.

1. We have two points $P = (x_p, y_p)$ and $Q = (x_q, y_q)$ where $P \neq Q$. If $P = O$ then $P$ is the identity point which means $P+Q = Q$. Likewise if $Q = O$, then $P+Q = P$ Else, $P+Q = S$ for $S = (x_s, y_s)$ where $S = -R$ (the point represented in the graph) such that:

$$x_s = \lambda^2 - x_p - x_q$$

and

$$y_s = \lambda(x_p - x_s) - y_p \ \forall \lambda = \frac{y_q - y_p}{x_q - x_p}$$

2. In the projective form, each elliptic curve point has 3 coordinates instead of 2 $(x, y, z)$ for $z \neq 0$ (for all points except the point at infinity). Using the projective form allows us to give coordinates to the point at infinity. It also speeds up some of the most used arithmetic operations in the curve. The forward mapping is given by $(x, y) \rightarrow (xz, yz, z)$ and reverse mapping is giving by $(x, y, z)(\frac{x}{z}, \frac{y}{z})$. Let $P = (x_p, y_p, z_p)$ and $Q = (x_q, y_q, z_q)$ and $\frac{x_p}{z_p} \neq \frac{x_q}{z_q}$.

By expanding the previous arithmetics, we have:

$$\lambda = \frac{\frac{y_q}{z_q} - \frac{y_p}{z_p}}{\frac{x_q}{z_q} - \frac{x_p}{z_p}} = \frac{\frac{y_q}{z_q} - \frac{y_p}{z_p}}{\frac{x_q}{z_q} - \frac{x_p}{z_p}} * \frac{z_p z_q}{z_p z_q}$$

$$\lambda = \frac{y_q z_p - y_p z_q}{x_q z_p - x_p z_q}$$

$$\mathrm{x}_s = \lambda^2 - \frac{x_p}{z_p} - \frac{x_q}{z_q}$$

$$\mathrm{y}_s = \lambda(\frac{x_p}{z_p} - x_s) - \frac{y_p}{z_p}$$

**Point doubling** We will compute $P + P = 2P$ in the affine form as follows:

1. If $P = O$ then $2P = O$
2. Else $P = (x, y)$:
   - 2.1 If $y = 0$ then $2P = O$
   - 2.2 Else $2P = (x', y')$

such that: the derivative $\lambda = \frac{dy}{dx} = \frac{3x^2}{2y}$, $x' = \lambda^2 - 2x$ and $y' = \lambda(x - x') - y = \lambda^3 + 3\lambda x - y$

**The discrete logarithm problem** The security of many cryptographic techniques depends on the intractability of the discrete logarithm problem.

**Definition 6** (DL Problem)**.** Let $G$ be a multiplicative group. The discrete logarithm problem (DLP) is: Given $g, h \in G$ to find $a$, if it exists, such that $h = g^a$.

**Elliptic Curve Discrete Logarithm Problem (ECDLP)** Let $E$ be an elliptic curve of the Weierstrass form defined over a finite field $F_q$. Let $S$ and $T$ be two points in $E(F_q)$. Find an integer $m$ such that: $T = mS$

The fastest method to solve the ECDLP problem in $E(F_q)$ is the Pollard Rho method which has exponential complexity $O(\sqrt{|G|})$. In order for this algorithm to be exponential, we need to define elliptic curves over very large fields $|F_p| = 2^{256}$.

### 2.3.2 Pairings

Pairing based-cryptography is used in many cryptographic applications like signature schemes, key agreement, zero knowledge... etc. For example, pairings are used to create efficient circuit-based zero knowledge proofs.

**Definition 7** (Pairing)**.** A pairing $e$ is a bilinear map $\langle ;\dot{;}\rangle$ defined as:

$$e : G_1 \times G_2 \to G_T$$

Such that $G_1, G_2$ and $G_T$ are abelian groups.
The bilinear property means that:

$$e(P + P', Q) = e(P, Q) + e(P', Q)$$
$$\text{e(P,Q+Q')=e(P,Q)+e(P, Q')}$$

From which, we deduce the following transformations:

$$e([a]P, [b]Q) = e(P, [b]Q)^a = e([a]P, Q)^b = e(P, Q)^{ab} = e([b]P, [a]Q)$$

for $P, P' \in G_1$ and $Q, Q' \in G_2$ and $a, b \in \mathbb{Z}$

### 2.3.3 Pairing Friendly Curves

In zk-SNARK schemes such as Placeholder, we need to manipulate very large polynomials in order to perform efficient multi-point evaluation/interpolation with fast fourier transforms. We therefore target a subfamily of curves that have:

- Optimal extension field towers.
- Simple twisting isomorphisms.

**Montgomery Curve** Montgomery curves were first introduced by Peter L. as a way to accelerate elliptic curve methods of factorization and then became central to elliptic curve cryptography. Later, these same qualities also led to many efficient implementations of elliptic curve cryptosystems, most notably Bernstein's Curve25519 software.

**Definition 8** (Montgomery Curve)**.** A Montgomery curve over $F_q$ is an elliptic curve defined as $E_{(A,B)}$ : $By^2 = x(x^2 + Ax + 1)$ where $A$ and $B$ are parameters in $F_q$ satisfying $B \neq 0$ and $A^2 \neq 4$.

**Edwards Curves and Twisted Edwards Curves** Edwards curves are a family of elliptic curves that uses Edwards form instead of the more well known Weierstrass form used by elliptic curves. Edwards curves provide better efficiency and security than a general elliptic curve. For example the edwards25519 curve is faster than secp256k1 curve and considered to be more secure according to the safe curve security assessments. The edwards25519 is considered to be secure against twist attacks (small group and invalid group attacks) whereas the secp256k1 is considered to be vulnerable to those.

**Definition 9** (Edwards Curves)**.** An Edwards curve over $k$ is a curve: $E : x^2 + y^2 = 1 + dx^2y^2 - x^2$ where $d \in k - \{0, 1\}$ and $k$ is a field with $char(k) \neq 2$.

**Twisted Edward Curves** Fix a field $k$ with $char(k) \neq 2$ and distinct non-zero elements $a, d \in k$. The Twisted Edwards curve with coefficients $a$ and $d$ is the curve $E_{E_{a,b}} : ax^2 + y^2 = 1 + dx^2y^2$ An Edwards curve is a Twisted Edwards Curve with $a = 1$.

**Correspondence with Montgomery curves** Every Twisted Edwards curve is birationally equivalent to an elliptic curve in Montgomery form, and vice versa.

**Curve25519** Curve25519 is a Montgomery curve providing 128 bits of security defined as: $y^2 = x^3 + ax^2 + x$ over prime field $p$ where: $b = 1$.

The curve is birationally equivalent to a twisted Edwards curve used in the Ed25519 signature scheme.

**Barreto-Naehrig curves(BN curves)** The BN-curve is a pairing friendly elliptic curve built over a field $F_q$ for $q \geq 5$ that achieves both high security and efficiency and has optimal ate pairing. BN curves with 256-bit (for example BN256) were believed to provide a 128-bit security level, but due to recent research exTNFS this number dropped to 100-bits of security.

**Barreto-Lynn-Scott curves (BLS curves)** A BLS curve is a pairing over BLS curves that constructs optimal Ate pairings. BLS12-381 is optimal for zk-SNARKs at the 128-bit security level. Bls12-381 has an embedded Jubjub curve.

BLS-12 curves are a more efficient choice than BN curves in terms of optimal Ate pairings, and they have a better security level (Bls12-381 provides 128-bits security whereas BN256 provides only 100-bits security)

**Definition 10** (Jubjub Curve)**.** Jubjub curve is a twisted Edwards curve of the form $-x^2 + y^2 = dx^2y^2$ built over the BLS12-381 scalar field.

### 2.3.4 Lagrange Basis and Polynomial Interpolation

Polynomial interpolation is a process where a given set of points $(x_i, y_i)$, $i \in [n]$ allows us to construct a polynomial $f(x)$ that passes through all of them. We will assume that $x_i \neq x_j$ for all distinct $i, j$ pairs, otherwise there is a repeated pair or it is not possible to construct the polynomial as it would have to take two different values at the same $x$-point.

Notice that for a set of 2 points, we can find a line that crosses both of them, for a set of 3 points, a parabola, and in general, for a set of $n$ points there is a polynomial of degree $n-1$ that contains all of them.

The Lagrange interpolation consists of 2 steps:

1. Construct a **Lagrange basis**. This is a set of $n$ polynomials of degree $n-1$ that take the value 0 at all points of the set except one, where their value is 1. Expressed in a formula:

$$L_i(x) = \begin{cases} 0, & \text{if } \ x = x_j, j \in [n], j \neq i \\ 1, & \text{if } \ x = x_i \end{cases}$$

   The polynomials $L_i$ can be constructed in the following way:

$$L_i(x) = \prod_{0 \leq j < n,\ j \neq i} \frac{x - x_j}{x_i - x_j}$$

   Notice that this product has $n-1$ terms, and therefore results in a degree $n-1$ polynomial.

2. Scale and sum the polynomials of the basis.

$$f(x) = \sum_{i=0}^{n-1} y_i \cdot L_i(x)$$

   The properties of the Lagrange basis now allow us to scale each polynomial to its target vale – multiplying by $y_i$ and then add up all the terms.

The important observation we can extract from the Lagrange interpolation is that given a fixed set of points $x_1, \ldots, x_n$ (an evaluation domain) we can represent any polynomial of degree $d < n$ by its evaluations $f(x_i)$ at $d+1$ points in the set. As it turns out, this representation is much more convenient than the usual coefficient representation as it provides a very simple and fast way of computing sums and multiplication of polynomials. However, the coefficient form is still useful for evaluating the polynomial at points outside the evaluation domain.

Switching between these two forms of representation is very useful. The coefficient form is preferred when the polynomial must be evaluated at a random point (outside of the evaluation domain). The evaluation form is better suited for operations between polynomials such as addition, multiplication and exact quotients. The algorithm that allows us to efficiently switch between representations is the Fast Fourier Transform (FFT). This is an efficient algorithm for the more general discrete Fourier Transform (DFT). It has a complexity of $\mathcal{O}(n \cdot log(n))$ with $n$ being the degree of the polynomial.

**Fast Fourier Transform algorithm** The FFT is generally defined over the complex numbers but in the crypto context it is always used over a finite field $\mathbb{F}$. The only requisite for $\mathbb{F}$ is that it have a large multiplicative subgroup $H$ of order $n = 2^k$ for some $k \in \mathbb{N}$. This subgroup $H$ will be the evaluation domain and it will consist of the $n^{th}$ roots of unity $H = \{\omega, \omega^2, \ldots, \omega^n\} = x \in \mathbb{F} | x^n - 1 = 0\}$

## 2.4 Commitment Schemes

A commitment scheme $C$ is a protocol between two parties: a prover $P$ and a verifier $V$. The goal of such a scheme is to satisfy the following security properties:

- **Hiding:** $P$ should be able to commit to a value $m$ by encoding it using a key $P_K$ without $V$ learning any information about $m$.
- **Binding:** $P$ cannot "cheat" by sending a different key $P'_K$ which opens $C$ to a different value $m'$.

This is very useful in various cryptographic applications including zero-knowledge proofs. A commitment scheme can either be interactive or non-interactive depending on the use case. Their security assumption also varies between perfect or computational security with respect to the hiding and binding properties.

There are different combinations of these properties but the most famous ones are perfectly binding and computationally hiding commitment schemes, and computationally binding and perfectly hiding commitment schemes.
In the first scheme, $P$ generates the public key and sends it to $V$. The perfectly binding means $P$ is unable to change the commitment value after it has been committed to. The computational hiding means the probability of $V$ being able to guess the commitment value is negligible.
In the second scheme, $V$ generates the public key and sends it to $P$. The computational binding means the chance of being able to change the commitment is negligible. The perfectly hiding means that a commitment to a message $m$ reveals no information about $m$.

For a detailed explanation of those properties, check this article.

**Definition 11** (Group)**.** A non-interactive commitment scheme has the following three algorithms:

1. **Key generation (setup)** $(1^k)$: The algorithm key outputs a pair of keys $(P_K, V_K)$ that is sent to the prover and verifier respectively for a given security parameter $k$.

2. **Commitment** $(P_K, m)$: The algorithm com takes as input the prover key $P_K$ and the message $m$ and outputs the commitment $C$ and an opening value $d$ which will be used by the verifier.

3. **Verification** $(V_K, C, m, d)$: The algorithm takes the verification key $V_K, C, m$ and $d$ as input and outputs yes or no depending on whether the verification is successful.

## 2.5 Zero Knowledge Proof Systems

Before we start talking about zero knowledge proof systems, let's first define what a proof system is: A proof system is a protocol by which one party (prover) wants to convince another party (verifier) that a given statement is true.

**Definition 12** (Zero Knowledge Proof System)**.** In zero-knowledge proofs, the prover convinces the verifier about the truthfulness of the statement without revealing any information about the statement itself.

**Properties** A zero-knowledge proof needs to fulfill each of the following properties to be fully described:

1. **Completeness:** An honest prover is always able to convince the verifier of the truthfulness of their claim.

2. **Soundness:** If the prover's claim is false (malicious prover), the verifier is not convinced.

3. **Zero knowledge:** The proof should not reveal any information to the verifier beyond the truthfulness of the given claim.

**Types of Zero Knowledge Proofs** There are two types of zero knowledge proofs (interactive and non interactive ones) Interactive zero-knowledge proof systems were first introduced in 1985 by Goldwasser, Micali and Rackoff. Non-interactive schemes were introduced later on by Blum et al. The main difference between both schemes is that interactive proofs require interaction between both parties which means both have to be online in order to do so; this can be seen as inconvenient, especially for modern cryptography applications, while non-interactive proofs need a shared setup preprocessing phase instead. The shared setup phase will allow the participating parties to know which statement is being proved and what protocol is being used.

**Interactive Zero Knowledge Proofs** A prover $P$ has a secret $s$ and correctly responds to challenges to convince a verifier $V$ it has knowledge of $s$ using rounds of interaction between the two parties.

**Non interactive zero knowledge proofs (NIZK)** Non-interactive zero-knowledge proofs, also known as NIZKs are another type of zero-knowledge proof which require no interaction between the prover and the verifier. In order to transform interactive proofs into NIZK proofs, cryptographers used the Fiat-Shamir heuristic hash function. This hash function allows one to compute the verifier challenges and offer very efficient NIZK arguments that are secure in the random oracle model. More recent works have started using bilinear groups to improve efficiency.

zkSNARKs stands for zero-knowledge Succinct Non-interactive ARguments of Knowledge:

- **Succinct**: proof length needs to be short
- **Non-interactive**: needs to be verifiable in a short amount of time
- **ARKs**: need to show that we know an input (witness) which yields to a certain computation.

zkSNARKs cannot be applied to any computational problem directly; rather, you have to convert the problem into the right "form" for the problem to operate on.

## 2.6 PLONK

PLONK stands for Permutations over Lagrange-bases for Oecumenical Non Interactive Arguments of Knowledge. From hereon we shall simply write PLONK.

PLONK is an arithmetization method which solves a huge issue inherited from more established arithmetizatons - low constraints degree which disabled compact representaton of complex expressions which reduced the efficiency of a statement proved representation.

PLONK-alike arithmetization enforces the following pipeline of a proof system:

**Program (code) → Arithmetic circuit → Constraint systems → Permutation checks → Commitment Schemes**

We will use the following example to be able to explain each step and the transition from one to the other. For a more detailed explanation, please check the original PLONK paper as it contains formal definitions and some interesting insights regarding efficiency etc.

**Problem definition:** Prover wants to prove to Verifier that she knows the solution to the equation:

$$x^3 + x + 5 = 0$$

The goal is for Prover to evaluate the above function without revealing anything about the secret value $x$ (solution to the equation).

Prover creates a program to represent the problem in a function code, which then will be translated into an arithmetic circuit.

### 2.6.1 Arithmetic Circuit

This step transforms a program into an arithmetic circuit where two basic components are being used: wires and gates. PLONK uses fan-in two gates; therefore each gate has a left input, a right input, and an output. A circuit with $n$ gates will have $3n$ wires.

PLONK is, unlike R1CS-alike arithmetizations, is a gate-based arithmetization. A primary difference between the two systems is in how they handle addition gates; in R1CS, addition gates are cheap since

wires that go from an addition to a multiplication gate are not labeled, which is not the case for a gate-based system. The reason why PLONK uses a gate-based system rather than an R1CS system is that the linear constraints (which are just wiring constraints) can be reduced to a permutation check. To better understand the advantages and disadvantages of each of those designs check this article.

The following circuit translates the previous equation $x^3 + x + 5 = 0$, giving 2 multiplication gates and 2 addition gates.

#### 2.6.2 Constraint System

The circuit is converted into a system of equations where the variables are the values on each of the wires, and there is one equation per gate. Let's take our previous example:

$$\begin{cases} & = 0 \\ a_2 * b_2 - c_2 & = 0 \\ a_3 + b_3 - c_3 & = 0 \\ a_4 + b_4 - c_4 & = 0 \\ \end{cases}$$

The final result is $x^3 + x + 5 - c_4 = 0$, which represents the program we wanted to solve for $c_4 = x^3 + x + 5 = 0$. The setup for each of those equations is of the form:

$$(Q_{L_i})a_i + (Q_{R_i})b_i + (Q_{O_i})c_i + (Q_{M_i})a_i b_i + Q_{C_i} = 0$$

for $L$= left, $R$ = right, $O$= output, $M$= multiplication, $C$= constant The arithmetic gates are modeled with the selector vectors: $(Q_L, Q_R, Q_O, Q_C, Q_M)$

Each $Q$ value is a constant. We define $Q$ for an additive gate and a multiplicative one and a constant gate as follow: For an addition gate, we set:

$$Q_{L_i} = 1, Q_{R_i} = 1, Q_{M_i} = 0, Q_{O_i} = -1, Q_{C_i} = 0$$

For a multiplication gate, we set:

$$Q_{L_i} = 0, Q_{R_i} = 0, Q_{M_i} = 1, Q_{O_i} = -1, Q_{C_i} = 0$$

For a constant gate setting $a_i$ to some constant $x$, we set:

$$Q_{L_i} = 1, Q_{R_i} = 0, Q_{M_i} = 0, Q_{O_i} = 0, Q_{C_i} = -x$$

# 3 Proposal

This section describes Placeholder proof system proposed.

## 3.1 PLONK Arithmetization with Custom Gates

Here we describe our instantiation of PLONK with custom gates.

The computation sequence that needs to be proved is called **Circuit**. **Circuit** is defined by **Table**, **Basic Constraints**, **Copy Constraints**, **Lookup Constraints**. Note that **Circuit** does not include witnesses, public input, and intermediate values.

It all starts with a rectangular matrix, which we'll refer as **Table**, represents a structure of the computations. **Rows**, **Columns** and **Cells** of this matrix are used with the conventional meanings.

There are four types of columns:

- **Witness Columns** contain witness input and intermediate calculations.
  Witness Columns differ between proof instances (because they depend on input). They are not known to the verifier.

| Column Type | Constant Values | Contains Public Data | Values |
|---|---|---|---|
| Witness | ✗ | ✗ | $\mathbb{F}$ |
| Public | ✗ | ✓ | $\mathbb{F}$ |
| Fixed | ✓ | ✓ | $\mathbb{F}$ |
| Selectors | ✓ | ✓ | $\{1, 0\} \in \mathbb{F}$ |

- **Public Columns** contain public input.
  Public Columns differ between proof instances (because they depend on input). They are known to the verifier.
- **Fixed Columns** contain circuit-depended data. Fixed Columns do not differ between proof instances. They are known to the verifier.
- **Selectors** are special case of Fixed Columns. Selectors define to which rows of the Table the basic constraint is applied. Selectors' values can be only ones or zeroes. We provide details on Selectors later in this Section.

Table values are bound by three types of assertions:

1. Assertions that are imposed relation between Table values are referred as **Basic Constraints**. Basic Constraints are expressions (multivariate polynomials) over Cells.
2. **Copy Constraints** defines equality assertions between Cells.
3. **Lookup Constraints** assert that the chosen tuples of Cells of the Table are equal to some rows in **Lookup Table**. Note that Lookup Constraint does not define the precise place of the tuple in the Lookup Table. It is the main difference between Lookup and Copy constraints.

Table stores values used during computations, Constraints define relationships between these values.

Basic and Lookup Constraints may use Cells from the different Rows. If Constraint references values from the neighboring row, we call that reference as **Offset Reference** and the difference between constraint's row and referenced row as **Offset**. We also use **Absolute References** in Copy Constraints. **Absolute References** point to the number of the row instead of the difference between rows.

| Constraint Type | Example |
|---|---|
| Basic Constraint | $2 \cdot T_{i,j} + T_{i+1,j} = 0$ |
| Basic Constraint | $T_{i,j} \cdot T_{i+1,j} + T^2_{i+1,j+1} = 1$ |
| Copy Constraint | $T_{i,j} = T_{1,0}$ |
| Lookup Constraint | $(T_{i,j}, T_{i+1,j}, T_{i+2,j+1}) \in L$ |

*Constraints examples for Table $T$ and Lookup Table $L$*

**Selectors** are used to include/exclude a Basic Constraint check to/from the Row. Selectors are included as a part of the assertion to do this. A set of Basic Constraints used with the same Selector is called **Gate**. A gate may contain one or more constraints. Each Row has to satisfy all Gates of the Circuit.

To include Gate as an assertion to the particular Row, the value of the corresponding Selector's column of the Row has to be set to 1. Otherwise, it is set to 0.

Appendix A contains example of the circuit construction.

## 3.2 Public Input

Public Input takes a separate column or columns (in the case if one column is not enough, that is very unlikely) in the table. It is enforced in the witness columns via copy constraints.

## 3.3 Placeholder Protocol

Let's introduce definitions which will be used all over the paper as follows:

| | |
|---|---|
| $N_{\texttt{rows}}$ | Number of rows |
| $N_{\texttt{witness}}$ | Number of witness columns |
| $N_{\texttt{perm}}$ | Number of witness columns that are included in |
| the permutation argument | |
| $N_{\texttt{sel}}$ | Number of selectors used in the circuit |
| $N_{\texttt{lookup}}$ | Number of lookup constraints |
| $N_{\mathbf{c}}$ | Number of constraints polynomials |
| $N_{PI}$ | Number of public input columns |
| $w_i$ | Witness polynomials, $0 \leq i < N_{\texttt{witness}}$ |
| $\texttt{c}_j^{(i)}$ | Constraint polynomials, $0 \leq i < N_{\texttt{sel}}$ |
| $\texttt{gate}_i$ | Gate polynomials for selector $\mathbf{q}_i(X)$ and constraints $\{\mathbf{c}_j^{(i)}\}_{j=0}^{n_i'-1}$ |
| $PI_i$ | Public input polynomials, $0 \leq i < N_{PI}$ |
| $\sigma(\text{col} : i, \text{row} : j) = (\text{col} : i', \text{row} : j')$ | Permutation over the table |
| $\mathbf{o}$ | Set of all offsets (see Section 3.1) |

For details on polynomial commitment scheme and polynomial evaluation scheme, we refer the reader to [1].

We need to define how gates are constructed from constraints. Remind that gates contains a set of constraints with the same selector.

Each constraint may include different columns and offsets. Let constraint polynomial $\texttt{c}_j$ includes columns $w_{j,0}, \ldots, w_{j,k-1}$ for some $k \in \mathbb{N}$ with the corresponding offsets $d_{j,0}, \ldots, d_{j,k-1}$. It has the following form:

$$\texttt{c}_j = a_{j,0} \cdot w_{j,0}(\omega^{d_{j,0}} X) + \ldots a_{j,k} \cdot w_{j,k-1}(\omega^{d_{j,k-1}}),\ a_{j,i} \in \mathbb{F}$$

Denote by $k_i$ the number of constraints used in $i$-th gate. Let $\nu_i$ be the initial degree of the random challenge for $i$-th gate.

$$\begin{aligned} \nu_{i+1} &= \nu_i + k_i \\ \nu_0 &= 0. \end{aligned}$$

Then we can represent $i$-th gate as:

$$\texttt{gate}_i(X) = q_{l_i}(X) \cdot (\theta^{k_i - 1 + \nu_i} \texttt{c}_{0_i}(X) + \cdots + \theta^{\nu_i} \texttt{c}_{k_i - 1}(X))$$

**Preprocessing:**

Preprocessing from table to polynomials

1. $\mathcal{L}' = (\mathbf{q}_0, ..., \mathbf{q}_{N_{\texttt{sel}}})$
2. Let $\omega$ be a $2^k = N_{\texttt{rows}}$ root of unity.
3. Let $\delta$ be a $T$ root of unity, where $T \cdot 2^S + 1 = p$, $k \leq S$, $T$ odd and $p$ is a size of the field.
4. Compute $N_{\texttt{perm}}$ permutation polynomials $S_{\sigma_i}(X)$ such that $S_{\sigma_i}(\omega^j) = \delta^{i'} \cdot \omega^{j'}$
5. Compute $N_{\texttt{perm}}$ identity permutation polynomials: $S_{id_i}(X)$ such that $S_{id_i}(\omega^j) = \delta^i \cdot \omega^j$
6. Let $H = \{\omega^0, ..., \omega^{N_{\texttt{rows}} - 1}\}$ be a cyclic subgroup of $\mathbb{F}^*$
7. Let $A_i$ be a witness columns, which are used for a lookup, and $S_i$ be a table columns for some lookup, $i = 0, .., m$.
8. $Z(X) = \prod_{a \in H}(X - a) = X^{N_{\texttt{rows}}} - 1$

interpolation from $\omega^0$

### 3.4 Prover View

Denote polynomial commitment generation function as **Commit**(.) The commitment scheme is described in Section 4. Details on the commitment scheme optimizations are in Section 9.

Define params and how to add params to transcript

Details on transcript

1. $\texttt{transcript.append}(\texttt{circuit\_params})$
2. $\texttt{transcript.append}(\textbf{Commit}(w_i(X)))$ for $0 \leq i < N_{\texttt{witness}}$
3. Denote witness polynomials included in permutation argument and public input polynomials as follows

$$f_0 \coloneqq w_0, f_1 \coloneqq w_1, \ldots, f_{N_{\texttt{perm}}+N_{PI}-1} = PI_{N_{PI}-1}$$

4. $F_0(X), F_1(X), F_2(X) = \texttt{permutation\_argument}(\text{transcript}, f_0, \ldots, f_{N_{\texttt{perm}}+N_{PI}-1}, \texttt{circuit\_params})$
5. Denote witness polynomials included in lookup argument as follows

$$a_0 \coloneqq A_0, a_1 \coloneqq A_1, \ldots, a_{m-1} = A_{m-1}$$

6. $F_3(X), F_4(X), F_5(X), F_6(X), F_7(X) = \texttt{lookup\_argument}(\text{transcript}, a_0, \ldots, a_{m-1}, \texttt{circuit\_params})$
7. Constraint-satisfability processing:
   - 7.1 $\tau = \texttt{transcript.get\_challenge}()$
   - 7.2 For $i = 0, \ldots, N_{\texttt{sel}} - 1$ (the details on the gate construction are presented above):
     - 7.2.1 $\texttt{gate}_i(X) = q_{l_i}(X) \cdot (\theta^{k_i-1+\nu_i}\texttt{c}_{0_i}(X) + \cdots + \theta^{\nu_i}\texttt{c}_{k_i-1}(X))$
   - 7.3 Calculate a constraints-related numerator of the quotinent polynomial:

$$F_8(X) = \sum_{0 \leq i < N_{\texttt{sel}}} (\texttt{gate}_i(X))$$

8. Quotient polynomial calculation:
   - 8.1 $\alpha_0, \ldots, \alpha_8 = \texttt{transcript.get\_challenge}()$
   - 8.2 Compute quotient polynomial $T(X)$:

$$F(X) = \sum_{i=0}^{8} \alpha_i F_i(X)$$
$$T(X) = \frac{F(X)}{Z(X)}$$

   - 8.3 $N_T \coloneqq \max(N_{\texttt{perm}} + N_{PI}, \texttt{deg}_{\texttt{gates}} - 1)$, where $\texttt{deg}_{\texttt{gates}}$ is the highest degree of the degrees of gate polynomials

     check lookup degree

   - 8.4 Split $T(X)$ into separate polynomials $T_0(X), ..., T_{N_T-1}(X)$ to fit them into commitments [1]
   - 8.5 $\texttt{transcript.append}(\textbf{Commit}(T_i(X)))$ for $0 \leq i < N_T - 1$
9. Run evaluation proof:
   - 9.1 $y = \texttt{transcript.get\_challenge\_from}(\mathbb{F}/H)$, $y \in \mathbb{F}/H$
   - 9.2 Run evaluation scheme with the committed polynomials and the corresponding points from the set $\{y, y\omega^{-1}, y\omega, y\omega^d\}$ for $d \in \mathbf{o}$
   - 9.3 The proof is $(\pi_{\texttt{comm}}, \pi_{\texttt{eval}})$, where:
     - $\pi_{\texttt{comm}} = \{w_{0,\texttt{comm}}, \ldots, w_{N_{\texttt{witness}-1},\texttt{comm}}, V_{P,\texttt{comm}}, T_{0,\texttt{comm}}, ..., T_{N_T-1,\texttt{comm}}, A_{\texttt{perm},\texttt{comm}}, S_{\texttt{perm},\texttt{comm}}, V_{L,\texttt{comm}}\}$

[1] Commit scheme supposes that polynomials should be degree $\leq n$

- $\pi_{\texttt{eval}}$ is evaluation proofs for $w_i(y), w_i(y\omega^d), V_P(y), V_P(y\omega)$, $T_0(y), \ldots, T_{N_T-1}(y), A_{\texttt{perm}}(y), A_{\texttt{perm}}(y\omega^{-1}), S_{\texttt{perm}}(y), V_L(y), V_L(y\omega)$ for all corresponding $d \in \mathbf{o}$

---

**Algorithm 1** Permutation Argument

**Input**: transcript, $f_0, \ldots, f_k$, `circuit_params`
**Output**: $F_0, F_1, F_2$

1. $\beta_1, \gamma_1 = \texttt{transcript.}\textbf{get\_challenge}()$
2. For $0 \leq j \leq N_{\texttt{rows}} - 1$

$$\texttt{id\_binding}_j = \prod_{i=0}^{N_{\texttt{perm}}+N_{PI}-1} \left(f_i(\omega^j) + \beta_1 \cdot S_{id_i}(\omega^j) + \gamma_1\right)$$
$$\sigma\texttt{\_binding}_j = \prod_{i=0}^{N_{\texttt{perm}}+N_{PI}-1} \left(f_i(\omega^j) + \beta_1 \cdot S_{\sigma_i}(\omega^j) + \gamma_1\right)$$

   **Remark**: Note that $\texttt{id\_binding}_j, \sigma\texttt{\_binding}_j$ are elements of $\mathbb{F}$, not polynomials.

3. Calculate $V_P$:

$$V_P(\omega) = V_P(\omega^{N_{\texttt{rows}}}) = 1$$
$$V_P(\omega^j) = \prod_{i=0}^{j-1} \frac{\texttt{id\_binding}_i}{\sigma\texttt{\_binding}_i} \text{ for } 0 < j < N_{\texttt{rows}}$$

4. $\texttt{transcript.}\textbf{append}(\textbf{Commit}(V_P))$
5. Calculate $g_{\texttt{perm}}(X), h_{\texttt{perm}}(X)$:

$$g_{\texttt{perm}}(X) \coloneqq \prod_{i=0}^{N_{\texttt{perm}}+N_{PI}-1} (f_i(X) + \beta_1 \cdot S_{id_i}(X) + \gamma_1)$$
$$h_{\texttt{perm}}(X) \coloneqq \prod_{i=0}^{N_{\texttt{perm}}+N_{PI}-1} (f_i(X) + \beta_1 \cdot S_{\sigma_i}(X) + \gamma_1)$$

6. Calculate permutation-related numerators of the quotinent polynomial:

$$F_0(X) = L_0(X)(1 - V_P(X))$$
$$F_1(X) = (1 - (q_{\texttt{last}}(X) + q_{\texttt{blind}}(X))) \cdot (V_P(\omega X) \cdot h_{\texttt{perm}}(X) - V_P(X) \cdot g_{\texttt{perm}}(X))$$
$$F_2(X) = q_{\texttt{last}}(X) \cdot (V_P(X)^2 - V_P(X))$$

---

**Algorithm 2** Lookup Argument

**Input**: transcript, $a_0, \ldots, a_{m-1}$, `circuit_params`
**Output**: $F_3, F_4, F_5, F_6, F_7$

1. $\theta = \texttt{transcript.get\_challenge}()$
2. For $i = 0, \ldots, N_{\texttt{lookup}} - 1$ (see Section 8.3 for details):
   2.1 $\texttt{lookup\_gate}_i(X) = q_{l_i}(X) \cdot (\theta^{k_i - 1 + \nu_i} A_{0_i}(\omega^{d_{0_i}} X) + \cdots + \theta^{\nu_i} A_{k_i - 1}(\omega^{d_{k_i - 1}} X))$
   2.2 $\texttt{table\_value}_i(\omega^j) = q_{l_i}(\omega^j) \cdot (\theta^{k_i - 1 + \nu_i} S_{0_i}(\omega^j) + \cdots + \theta^{\nu_i} S_{k_i - 1}(\omega^j))$
3. Construct the input lookup compression and table compression values for $1 \leq j \leq N_{\texttt{rows}}$:

$$\mathbf{A}_{\texttt{compr}}(\omega^j) = \sum_{0 \leq i < N_{\texttt{lookup}}} \texttt{lookup\_gate}_i(\omega^j)$$
$$\mathbf{S}_{\texttt{compr}}(\omega^j) = \sum_{0 \leq i < N_{\texttt{lookup}}} \texttt{table\_value}_i(\omega^j)$$

4. Interpolate polynomials $A_{\texttt{compr}}(X), S_{\texttt{compr}}(X)$ from $\mathbf{A}_{\texttt{compr}}, \mathbf{S}_{\texttt{compr}}$
5. Produce the permutation polynomials $S_{\texttt{perm}}(X)$ and $A_{\texttt{perm}}(X)$ according to Section 8.1.
6. $\texttt{transcript.}\textbf{append}(\textbf{Commit}(A_{\texttt{perm}}))$, $\texttt{transcript.}\textbf{append}(\textbf{Commit}(S_{\texttt{perm}}))$
7. Compute $V_L(X)$ such that:

$$V_L(1) = V_L(\omega^{N_{\texttt{rows}}}) = 1$$
$$V_L(\omega^j) = \prod_{i=0}^{j-1} \frac{(A_{\texttt{compr}}(\omega^i) + \beta)(S_{\texttt{compr}}(\omega^i) + \gamma)}{(A_{\texttt{perm}}(\omega^i) + \beta)(S_{\texttt{perm}}(\omega^i) + \gamma)} \text{ for } 0 < j < N_{\texttt{rows}}$$

8. $\texttt{transcript.}\textbf{append}(\textbf{Commit}(V_L))$
9. $\beta_2, \gamma_2 = \texttt{transcript.get\_challenge}()$
10. Calculate $g_L(X), h_L(X)$:

$$g_L(X) = (A_{\texttt{compr}}(X) + \beta_2) \cdot (S_{\texttt{compr}}(X) + \gamma_2)$$
$$h_L(X) = (A_{\texttt{perm}}(X) + \beta_2) \cdot (S_{\texttt{perm}}(X) + \gamma_2)$$

11. Calculate lookup-related numerators of the quotinent polynomial:

$$F_3(X) = L_0(X)(1 - V_L(X))$$
$$F_4(X) = V_L(\omega X) \cdot h_L(X) - V_L(X) \cdot g_L(X)$$
$$F_5(X) = q_{\texttt{last}}(X) \cdot (V_L(X)^2 - V_L(X))$$
$$F_6(X) = L_0(X)(A_{\texttt{perm}}(X) - S_{\texttt{perm}}(X))$$
$$F_7(X) = (1 - (q_{\texttt{last}}(X) + q_{\texttt{blind}}(X))) \cdot (A_{\texttt{perm}}(X) - S_{\texttt{perm}}(X)) \cdot (A_{\texttt{perm}}(X) - A_{\texttt{perm}}(\omega^{-1} X))$$

## 3.5 Verifier View

1. Parse proof $\pi$ into:
   - $\pi_{\texttt{comm}} = \{w_{0,\texttt{comm}}, \ldots, w_{N_{\texttt{witness}}-1,\texttt{comm}}, V_{P,\texttt{comm}}, T_{0,\texttt{comm}}, ..., T_{N_T-1,\texttt{comm}}, A_{\texttt{perm},\texttt{comm}}, S_{\texttt{perm},\texttt{comm}}, V_{L,\texttt{comm}}\}$
   - $\pi_{\texttt{eval}}$ is evaluation proofs for $w_(y), w_i(y\omega^d), V_P(y), V_P(y\omega)$, $T_0(y), \ldots, T_{N_T-1}(y), A_{\texttt{perm}}(y), A_{\texttt{perm}}(y\omega^{-1}), S_{\texttt{perm}}(y), V_L(y), V_L(y\omega)$ for all corresponding $d \in \mathbf{o}$
2. $\texttt{transcript.}\textbf{append}(\texttt{circuit\_params})$
3. $\texttt{transcript.}\textbf{append}(w_{i,\texttt{comm}})$ for $0 \leq i < N_{\texttt{witness}}$
4. Denote witness polynomials included in permutation argument and public input polynomials as

$$f_0 \coloneqq w_0, f_1 \coloneqq w_1, \ldots, f_{N_{\texttt{perm}}+N_{PI}-1} = PI_{N_{PI}-1}$$

5. $F_0(y), F_1(y), F_2(y) = \texttt{permutation\_argument}(\text{transcript}, \texttt{circuit\_params})$

6. $F_3(y), F_4(y), F_5(y), F_6(y), F_7(y) = \texttt{lookup\_argument}(\text{transcript}, \texttt{circuit\_params})$
7. Constraints-satisfability processing:
   7.1 $\theta = \texttt{transcript.get\_challenge}()$
   7.2 For $i = 0, \ldots, N_{\texttt{sel}} - 1$:
      7.2.1 $\texttt{gate}_i(X) = q_{l_i}(X) \cdot (\theta^{k_i - 1 + \nu_i} \texttt{c}_{0_i}(X) + \cdots + \theta^{\nu_i} \texttt{c}_{k_i - 1}(X))$.
   7.3 Calculate:
$$F_8(y) = \sum_{0 \leq i < N_{\texttt{sel}}} (\texttt{gate}_i(y))$$
8. $\alpha_0, \ldots, \alpha_8 = \texttt{transcript.get\_challenge}()$
9. Evaluation proof check:
   9.1 $N_T := \texttt{max}(N_{\texttt{perm}} + N_{PI}, \texttt{deg}_{\texttt{gates}} - 1)$, where $\texttt{deg}_{\texttt{gates}}$ is the highest degree of the degrees of gate polynomials
   9.2 Let $T_{0,\texttt{comm}}, ..., T_{N_T - 1,\texttt{comm}}$ be commitments to $T_0(X), ..., T_{N_T - 1}(X)$
   9.3 $\texttt{transcript.append}(T_{i,\texttt{comm}})$ for $0 \leq i < N_T$
   9.4 $y = \texttt{transcript.get\_challenge\_from}(\mathbb{F}/H)$, $y \in \mathbb{F}/H$
   9.5 Run evaluation scheme verification with the committed polynomials and the points from the set $\{y, y\omega^{-1}, y\omega, y\omega^d\}$ for all corresponding $d \in \mathbf{o}$ to get values $w_i(y), w_i(y\omega^d), V_P(y), V_P(y\omega), T_j(y), A_{\texttt{perm}}(y), S_{\texttt{perm}}(y), V_L(y), V_L(y\omega^{-1}), V_L(y\omega)$
10. Quotient Polynomial Check:
   10.1 Check the identity:
$$\sum_{i=0}^{10} \alpha_i F_i(y) = Z(y)T(y)$$

**Algorithm 3** Permutation Argument Verification

1. $\beta_1, \gamma_1 = \texttt{transcript.get\_challenge}()$
2. $\texttt{transcript.append}(V_{P,\texttt{comm}})$,
3. Denote (see Step 3 of the Prover's view for details on $f_i$):
$$g_{\texttt{perm}}(y) := \prod_{i=0}^{N_{\texttt{perm}} + N_{PI} - 1} (f_i(y) + \beta \cdot S_{id_i}(y) + \gamma)$$
$$h_{\texttt{perm}}(y) := \prod_{i=0}^{N_{\texttt{perm}} + N_{PI} - 1} (f_i(y) + \beta \cdot S_{\sigma_i}(y) + \gamma)$$
4. Calculate:
$$F_0(y) = L_0(y)(1 - V_P(y))$$
$$F_1(y) = (1 - (q_{\texttt{last}}(y) + q_{\texttt{blind}}(y))) \cdot (V_P(\omega y) \cdot h_{\texttt{perm}}(y) - V_P(y) \cdot g_{\texttt{perm}}(y))$$
$$F_2(y) = q_{\texttt{last}}(y) \cdot (V_P(y)^2 - V_P(y))$$

**Algorithm 4** Lookup Argument Verification

1. $\theta = \texttt{transcript.get\_challenge}()$
2. $\texttt{transcript.append}(A_{\texttt{perm,comm}})$, $\texttt{transcript.append}(S_{\texttt{perm,comm}})$, $\texttt{transcript.append}(V_{L,\texttt{comm}})$
3. For $i = 0, \ldots, N_{\texttt{lookup}} - 1$ (see Section 8.3 for details):
   3.1 $\texttt{lookup\_gate}_i(y) \coloneqq q_{l_i}(y) \cdot (\theta^{k_i - 1 + \nu_i} A_{0_i}(\omega^{d_{0_i}} y) + \cdots + \theta^{\nu_i} A_{k_i - 1}(\omega^{d_{k_i - 1}} y))$
   3.2 $\texttt{table\_value}_i(y) \coloneqq q_{l_i}(y) \cdot (\theta^{k_i - 1 + \nu_i} S_{0_i}(y) + \cdots + \theta^{\nu_i} S_{k_i - 1}(y))$
4. Construct the input lookup compression and table compression:

$$A_{\texttt{compr}}(y) \coloneqq \sum_{0 \le i < N_{\texttt{lookup}}} \texttt{lookup\_gate}_i(y)$$
$$S_{\texttt{compr}}(y) \coloneqq \sum_{0 \le i < N_{\texttt{lookup}}} \texttt{table\_value}_i(y)$$

5. Denote (see Step 3 of the Prover's view for details on $f_i$):

$$g_L(y) = (A_{\texttt{compr}}(y) + \beta) \cdot (S_{\texttt{compr}}(y) + \gamma)$$
$$h_L(y) = (A_{\texttt{perm}}(y) + \beta) \cdot (S_{\texttt{perm}}(y) + \gamma)$$

6. $\beta_2, \gamma_2 = \texttt{transcript.get\_challenge}()$
7. Calculate:

$$F_3(y) = L_0(y)(1 - V_L(y))$$
$$F_4(y) = (1 - (q_{\texttt{last}}(y) + q_{\texttt{blind}}(y))) \cdot (V_L(\omega y) \cdot h_L(y) - V_L(y) \cdot g_L(y)$$
$$F_5(y) = q_{\texttt{last}}(y) \cdot (V_L(y)^2 - V_L(y))$$
$$F_6(y) = L_0(X)(A_{\texttt{perm}}(y) - S_{\texttt{perm}}(y))$$
$$F_7(y) = (1 - (q_{\texttt{last}}(y) + q_{\texttt{blind}}(y))) \cdot (A_{\texttt{perm}}(y) - S_{\texttt{perm}}(y)) \cdot (A_{\texttt{perm}}(y) - A_{\texttt{perm}}(\omega^{-1} y))$$

# 4 Placeholder Commit / Evaluation Schemes

**WIP**

In this section, we define different structures that contains data (proofs and params). These structures only defines the data contained in the proof (commit scheme parameters). We do not specify implementation details on the data structures that are used to store the data.

## 4.1 Witness Polynomials

Generilize tha algorithm with $\eta$

**Algorithm 5** Setup

- Field $\mathbb{F}$
- Folding map $q(X) = X^2$
- Localization factor $m$. Default value $m = 2$.
- Domains $D_0, \ldots D_{r-1}$, such that:
  - $D_i \subset \mathbb{F}$
  - $D_0 = [\omega, \ldots, \omega^n]$.
  - $D_{i+1} = q(D_i)$
  - $|D_{i+1}| = \frac{|D_i|}{m} = \frac{|D_0|}{m^{i+1}}$
- Error-bound $\delta > 0$
- Bound for degree of polynomial $d \in \mathbb{N}$
- The number of FRI rounds $r$

Details on commit

**Algorithm 6** Commit

1. Calculate $f$ over all elements of $D$: $\mathbf{y} = \{f(H)\}_{H \in D}$
2. Build a Merkle-tree $T$ for set $\mathbf{y}$.
3. Root of $T$ is commitment

Define $T$.`path`()

Details on `fri_params`

FRI paramerters `fri_params` are defined by the following structure:

- $D_0, \ldots D_{r-1}$
- $q(X)$
- $m = 2$
- `max_degree` $= d$
- The number of points to open $k$
- FRI rounds number $r$
- $\lambda$

LPC proof $\mathcal{P}$:

- Evaluation vales $z_0, \ldots, z_{k-1}$
- Merkle proofs $p_{z_0}, \ldots, p_{z_{k-1}}$
- FRI proofs $\texttt{fri\_proof}_0, \ldots \texttt{fri\_proof}_{\texttt{fri\_params}.\lambda - 1}$

details

FRI proof $\pi$ contains:

- $\texttt{round\_proof}_i$ for $0 \le i < \texttt{fri\_params}.r - 1$
- $\texttt{final\_polynomial} = \{c_0, \ldots, c_k\}$, $k = 2^{\log d' - \texttt{fri\_params}.r}$

**Algorithm 7** Proof Eval

**Input**: $k$ points for evaluation $g$ over them: $\{\xi_j\}_{j=0}^{k-1}$, commitment to $g$ (root of Merkle tree $T$), `transcript`

1. Open $z_j = g(\xi_j)$ for $0 \le j < k$ from the commitment and add the along with Merkle paths $p_{z_j}$ to the proof
2. `MultiEval`:
   - 2.1 Interpolate polynomial $U(X)$ such that $U(\xi_j) = z_j$ for $0 \le j < k$.
     **Remark**: Notice, that $U(X) \neq g(X)$ since $deg(U) < deg(g)$.
   - 2.2 Calculate $Q(X) = \frac{g(X) - U(X)}{\prod_{j=0}^{k-1}(X - \xi_j)}$
     **Remark**: $deg(Q) = d' = d - k$.
   - 2.3 for $i$ from 0 to $\texttt{fri\_params}.\lambda - 1$:
     - 2.3.1 $\texttt{fri\_proof}_i = \texttt{FRI.Eval}(Q(X), g(X), T, \texttt{transcript}, \texttt{fri\_params})$ with rate $\rho = \frac{d'}{|D|}$ and error-dound $\delta$
   - 2.4 $\mathcal{P} = \{z_0, \ldots, z_{k-1}, p_{z_0}, \ldots, p_{z_{k-1}}, \texttt{fri\_proof}_0, \ldots \texttt{fri\_proof}_{\lambda - 1}\}$

details

Round proof for FRI:

- $y_0, \ldots, y_m$ polynomial values
- $p_0, \ldots, p_m$ Merkle tree paths
- $T$ Merkle tree root (commitment)
- `colinear_value`, `colinear_path`

**Algorithm 8** FRI.Eval

**Input**: $Q(X), T, \texttt{transcript}, \texttt{fri\_params}$
**Output**: $\pi$

1. $f(X) \coloneqq Q(X)$, $f(X)$ can be represented as $\sum_{i=0}^{d-1} c_i X^i$
2. $x = \texttt{transcript}.\textbf{get\_challenge}()$
3. $r = \texttt{fri\_params}.r$
4. for $i = 0..r-1$:
   - 4.1 $\alpha = \texttt{transcript}.\textbf{get\_challenge\_from}(\mathbb{F})$, $\alpha \in \mathbb{F}$
   - 4.2 $x_{\texttt{next}} = \texttt{fri\_params}.q(x)$
   - 4.3 $d = \textbf{deg}(f(X))$
     **Remark**: $d \leq \texttt{fri\_params.max\_degree}$
   - 4.4 Fold an intermediate polynomial (here for $\texttt{fri\_params}.m = 2$):
     $$f_{\texttt{even}}(X^2) = \sum_{i=0}^{\frac{d+1}{2}-1} c_{2i} X^{2i}$$
     $$f_{\texttt{odd}}(X^2) = \sum_{i=0}^{\frac{d+1}{2}-1} c_{2i+1} X^{2i}$$
     $$f_{\texttt{next}}(X) = f_{\texttt{even}}(X^2) + \alpha \cdot f_{\texttt{odd}}(X^2)$$
   - 4.5 Get points for interpolation:
     - 4.5.1 Find all $s_j$ from coset $S = \{s_j \in D_i : \texttt{fri\_params}.q(s_j) = x_{\texttt{next}}\}$, $|S| = \texttt{fri\_params}.m$
       **Remark**: For the case $\texttt{fri\_params}.m = 2$, all $s_j$ can be found from the equation $x_{\texttt{next}} - X^2 = 0$. In other words, $s_0 = x$, $s_1 = -x$.
     - 4.5.2 $y_j = f(s_j)$ for $0 \leq j < \texttt{fri\_params}.m$
     - 4.5.3 Get paths to the openings:
       - if $i = 0$:
         $$p_j = T.\textbf{path}(g(s_j)) \text{ for } 0 \leq j < m$$
         **Remark**: During the first iteration, there is not commitment for $Q(X)$, only for $g(X)$.
       - Otherwise:
         $$p_j = T.\textbf{path}(y_j) \text{ for } 0 \leq j < m$$
   - 4.6 if $i < r - 2$:
     - 4.6.1 $T_{\texttt{next}} = \textbf{Commit}(f_{\texttt{next}}(X))$, the commit is calculated over domain $\texttt{fri\_params}.D_{i+1}$
     - 4.6.2 $\texttt{transcript}.\textbf{append}(T_{next})$
   - 4.7 Form a round proof:
     - 4.7.1 if $i < r - 1$:
       - $\texttt{colinear\_value} = f_{\texttt{next}}(x_{\texttt{next}})$
       - $\texttt{colinear\_path} = T_{\texttt{next}}.\textbf{path}(\texttt{colinear\_value})$
       - $\texttt{round\_proof}_i = \{y_0, \ldots, y_m, p_0, \ldots, p_m, T, \texttt{colinear\_value}, \texttt{colinear\_path}\}$
     - 4.7.2 else:
       - $d' = \textbf{deg}(Q(X))$
       - $\texttt{final\_polynomial} = \{c_0, \ldots, c_{2^{\log d' - r}}\}$, where $f_{\texttt{next}}(X) = \sum_{i=0}^{2^{\log d' - r}} c_i X^i$
   - 4.8 $x = x_{\texttt{next}}$
   - 4.9 $f = f_{\texttt{next}}$
   - 4.10 $T = T_{\texttt{next}}$
5. $\pi = \{\texttt{round\_proof}_0, \ldots \texttt{round\_proof}_{r-2}, \texttt{final\_polynomial}\}$

**Algorithm 9** Verify Eval

**Input**: queries $\xi_0, \dots \xi_{k-1}$, proof $\mathcal{P}$, `fri_params`, `transcript`

1. Check Merkle proofs for $\mathcal{P}.p_{z_i}$ for $0 \le i < k$
2. `MultiEvalVerify`:
   2.1 Interpolate polynomial $U(X)$ such that $U(\xi_j) = z_j$ for $0 \le j < k$
   2.2 $V(X) = \prod_{j=0}^{k-1}(X - \xi_j)$
   2.3 for $i$ from 0 to $\texttt{fri\_params}.\lambda - 1$:
      2.3.1 Abort if $\texttt{FRI.Verify}(\mathcal{P}.\texttt{fri\_proof}_i, \texttt{transcript}, \texttt{fri\_params}, U(X), V(X))$ returns 0

**Algorithm 10** FRI.Verify

**Input**: FRI proof $\pi, \texttt{transcript}, \texttt{fri\_params}, U(X), V(X)$

1. $x = \texttt{transcript}.\textbf{get\_challenge}()$
2. $r = \texttt{fri\_params}.r$
3. for $i = 0..r-2$:
   3.1 $\alpha = \texttt{transcript}.\textbf{get\_challenge\_from}(\mathbb{F})$, $\alpha \in \mathbb{F}$
   3.2 $x_{\texttt{next}} = q(x)$
   3.3 Find all $s_j \in S = \{s_j \in D_i : \texttt{fri\_params}.q(s_j) = x_{\texttt{next}}\}$, $|S| = \texttt{fri\_params}.m$
   **Remark**: For the case $\texttt{fri\_params}.m = 2$, all $s_j$ can be found from the equation $x_{\texttt{next}} - X^2 = 0$. In other words, $s_0 = x$, $s_1 = -x$.
   3.4 $\pi.\texttt{round\_proof}_i.T.\textbf{verify}(\pi.\texttt{round\_proof}_i.p_j)$ for $0 \le j < m$
   3.5 Get the polynomial values for $0 \le j < \texttt{fri\_params}.m$:
   - if $i = 0$:
   $$y_j = \frac{\pi.\texttt{round\_proof}_0.y_j - U(s_j)}{V(s_j)}$$
   **Remark**: $\pi.\texttt{round\_proof}_0.y_j$ are values of the original polynomial, not $Q(X)$. For this reason, we need to recompute them.
   - Otherwise:
   $$y_j = \pi.\texttt{round\_proof}_i.y_j$$
   3.6 if $i < r - 2$:
      3.6.1 $\texttt{transcript}.\textbf{append}(\pi.\texttt{round\_proof}_{i+1}.T)$
   3.7 Colinearity check:
      3.7.1 Interpolate $\texttt{interpolant}(X)$ from $(s_j, y_j)$
      3.7.2 $\pi.\texttt{round\_proof}_{i+1}.T.\textbf{verify}(\pi.\texttt{round\_proof}_i.\texttt{colinear\_path})$
      3.7.3 Check that $\texttt{interpolant}(\alpha) = \pi.\texttt{round\_proof}_i.\texttt{colinear\_value}$
   3.8 $x = x_{\texttt{next}}$
4. for the last round $r - 1$:
   4.1 Check that $\pi.\texttt{final\_polynomial}$ contains $2^{\log d' - r}$ elements
   4.2 $f(X) \coloneqq \sum_{i=0}^{2^{\log d' - r}} \pi.\texttt{final\_polynomial}.c_i \cdot X^i$
   4.3 Check that $f(x) = \pi.\texttt{round\_proof}_{r-2}.\texttt{colinear\_value}$

## 4.2 Circuit Polynomials

In the previous scheme commit/opening for a polynomial $f$ describe a $\delta$-list of functions $f'$ such that

$$\Delta(f, f') < \delta,$$

where $\Delta$ is Hamming weight function.

For the polynomials that define a circuit, we require one more check. Each commit/opening should describe exactly one polynomial from the list. For that, Preprocessing step is used.

**Algorithm 11** Preprocessing

**Input**: Set of polynomials $g_i(X)$ that are required to be preprocessed

1. Prover and Verifier agree on separation points $x_i \in \mathbb{F}$ and values $\nu_i = g_i(\mu_i)$, such that:

$$\forall g_i' \in L_\delta(g_i) : g_i'(\mu_i) \neq g_i(\mu_i)$$

During the Proof Eval / Verify Eval algorithms for $g_i(X)$, additional pair $(\mu_i, \nu_i)$ should be added to the list of evaluation points $(\xi_j, z_j)$.

# 5 Optimized Placeholder Commit / Evaluation Schemes

**WIP**

In this section, we describe optimized commitment protocol. This description based on current protocol implementation. We define different structures that contains data (proofs and params). These structures only defines the data contained in the proof (commit scheme parameters). We do not specify implementation details on the data structures that are used to store the data.

## 5.1 Setup

Setup algorithm is similar to algorithm from previous section

**Algorithm 12** Setup

- Field $\mathbb{F}$.
- Folding map $q(X) = X^2$, denote $q_j(X)$ a result of $j$ times application map $q$ applied on $X$. In our case it $q_j(x) = X^{2^j}$.
- Localization factor $m$. Default value $m = 2$.
- Domains $D_0, \dots D_{r-1}$, such that:
  - $D_i \subset \mathbb{F}$
  - $D_0 = [\omega, \dots, \omega^n]$.
  - $D_{i+1} = q(D_i)$
  - $|D_{i+1}| = \frac{|D_i|}{m} = \frac{|D_0|}{m^{i+1}}$
- Error-bound $\delta > 0$
- Bound for degree of polynomial $d \in \mathbb{N}$
- Number of FRI rounds $r$

## 5.2 Merkle trees usage

To commit polynomial values protocol uses Merkle trees. Algorithm needs following operations:

- `make_merkle_tree(leaves)` – each leaf of Merkle tree commits values of polynomial $f$ on some coset $S \subset D_i$. Coset has the following structure $S = \forall s_i, s_j \in S : q_r(s_i) = q_r(s_j)$ for some $r$.
- `make_merkle_proof(tree, leaf)` – generates Merkle proof for the leaf.
- `validate(proof, leaf)` – checks if `proof` corresponds `leaf` data. Input of this function should contain $f(s_j) \forall s_j \in S$.

## 5.3 Data structures

In optimized version of algorithm FRI-proof $\pi$ doesn't contain all $r$ round proofs. There are only *steps* of them. Algorithm executes $r_i$ FRI-rounds on $i$-th step to produce $\texttt{round\_proof}_i$. Total number of FRI-rounds is $r$.

Another algorithm change based on the fact that $\texttt{round\_proof}_i.y$ contained $\texttt{round\_proof}_{i-1}.\texttt{colinear\_value}$ in the basic version of protocol. Polynomials' values are stored in common data structure `values` in FRI proof $\mathcal{P}$ now.

$$\texttt{round\_proof}_0.y == \texttt{values}_0, \tag{1}$$
$$\texttt{round\_proof}_0.\texttt{colinear\_value} \in \texttt{values}_1, \tag{2}$$
$$\texttt{round\_proof}_i.y == \texttt{values}_i, \tag{3}$$
$$\texttt{round\_proof}_i.\texttt{colinear\_value} \in \texttt{values}_{i+1}. \tag{4}$$

$\texttt{Values}_i$ contains polynomial values on coset $S$ of one of the domains. $S$ structure was described earlier. We can conviniently create Merkle proof for $\texttt{values}_i$ by calling `make_merkle_proof` function on corresponding precommitment.

LPC paramerters `lpc_params` are defined by the following structure:

- $\lambda$ – number of calls FRI-protocol are necessary for constructing LPC-proof;
- $r$ — total number of FRI-rounds in constructing FRI-proof;
- $m$ — localization factor;
- $k$ — number of points to open.

FRI paramerters `fri_params`:

- $D_0, \ldots D_{r-1}$
- $q(X)$
- $m = 2$
- $\texttt{max\_degree} = d$
- *steps* — number of FRI round proofs in FRI proof.
- $r_0, \ldots r_{steps-1} : \sum_{i=0}^{steps-1} r_i = r$, number of rounds in each step.

LPC proof $\mathcal{P}$:

- Evaluation values $z_0, \ldots, z_{k-1}$
- Merkle tree root `T_root`
- $\texttt{fri\_proof}_0, \ldots, \texttt{fri\_proof}_{\lambda-1}$

FRI proof $\pi$:

- $\texttt{round\_proof}_i$ for $0 \leq i < \texttt{fri\_params}.steps$
- $\texttt{final\_polynomial} = \{c_0, \ldots, c_k\}$, $k = 2^{\log d' - \texttt{fri\_params}.r}$
- $\texttt{values}_0 \ldots \texttt{values}_{\texttt{fri\_params}.steps}$ — sets of polynomials' values.

Round proof for FRI:

- Merkle tree path $p$
- Merkle tree root `T_root`
- Merkle tree paths `colinear_path`

## 5.4 Commit

**Algorithm 13** Precommit

**Input:** polynomial $f(X)$, domain $D$, $r$ — number of rounds in the first FRI-step.
**Ounput:** Merkle tree $T$.

1. Split domain $D$ into cosets $\{S \subset D : \forall s_i, s_j \in S, q_r(s_i) = q_r(s_j)\}$. If $m = 2$ $|S| = 2^r$
2. For each coset $S$ calculate $f$ over all elements of $S : \texttt{leaf}_S = \{f(x)\}_{x \in S}$. Number of leaves is $\frac{|D|}{|S|}$
3. Build Merkle tree $T = \texttt{make\_merkle\_tree}(\{\texttt{leaf}_S\})$.
4. $T$ is precommitment.

Root of $T$ is commitment

## 5.5 Proof eval

**Algorithm 14** LPC.Eval

**Input**: $k$ points for evaluation $g$ over them: $\{\xi_j\}_{j=0}^{k-1}$, polynomial $g$, Merkle tree $T$ (precommitment to $g$), `transcript`
**Output**: Proof $\mathcal{P}$

1. Calculate $z_j = g(\xi_j)$ for $0 \leq j < k$
2. `MultiEval`:
   - 2.1 Interpolate $U(X)$ such that $U(\xi_j) = z_j$ for $0 \leq j < k$.
     **Remark**: Notice, that $U(X) \neq g(X)$ since $deg(U) < deg(g)$.
   - 2.2 Calculate $Q(X) = \frac{g(X)-U(X)}{\prod_{j=0}^{k-1}(X-\xi_j)}$
     **Remark**: $deg(Q) = d' = d - k$.
   - 2.3 for $i$ from 0 to $\texttt{fri\_params}.\lambda - 1$:
     - 2.3.1 $\texttt{fri\_proof}_i = \texttt{FRI.Eval}(Q(X), g(X), T, \texttt{transcript}, \texttt{fri\_params})$ with rate $\rho = \frac{d'}{|D|}$ and error-dound $\delta$
   - 2.4 $\mathcal{P} = \{z_0, \ldots, z_{k-1}, T.\texttt{root()}, \texttt{fri\_proof}_0, \ldots \texttt{fri\_proof}_{\lambda-1}\}$

Details about $\delta$ error bound and rate $\rho$

**Algorithm 15** FRI.Eval

**Input**: Polynomials $Q(X), g(x)$, Merkle tree $T$, `fri_params`, `transcript`
**Output**: Proof $\pi$

1. `transcript`(`commit`($T$))
2. $f(X) \coloneqq Q(X)$, $f(X)$ can be represented as $\sum_{i=0}^{d-1} c_i X^i$
3. $x :=$ `transcript`.**get_challenge**()
4. $D := D_0$
5. $steps =$ `fri_params`.$steps, m =$ `fri_params`.$m, q(X) =$ `fri_params`.$q(X), r_i =$ `fri_params`.$r_i$
6. for $i = 0, \ldots, steps - 1$:
   - 6.1 $x_{\texttt{next}} := q_{r_i}(x)$
   - 6.2 If $i = 0$
     - 6.2.1 Construct coset $S = \{s \in D_0 | q_{r_i}(s) = x_{\texttt{next}}\}$.
     - 6.2.2 $\texttt{values}_0 := \{g(s)\}_{s \in S}$.
   - 6.3 Compute $f_{\texttt{next}(X)}$.
     - 6.3.1 $f_{\texttt{next}}(X) := f(X)$
     - 6.3.2 Repeat $r_i$ times:
       - 6.3.2.1 let $f_{\texttt{next}} = \sum_{i=0}^{d-1} c_i X^i$
       - 6.3.2.2 $\alpha :=$ `transcript`.**get_challenge_from**($\mathbb{F}$), $\alpha \in \mathbb{F}$
       - 6.3.2.3 $d = \mathbf{deg}(f(X))$
       - 6.3.2.4 Fold an intermediate polynomial (here for $m = 2$):

$$f_{\texttt{even}}(X^2) = \sum_{i=0}^{\frac{d+1}{2}-1} c_{2i} X^{2i}$$
$$f_{\texttt{odd}}(X^2) = \sum_{i=0}^{\frac{d+1}{2}-1} c_{2i+1} X^{2i}$$
$$f_{\texttt{next}}(X) = f_{\texttt{even}}(X^2) + \alpha \cdot f_{\texttt{odd}}(X^2)$$

   - 6.4 $D :=$ `fri_params`.$D_{\sum_{j=0}^{i} r_j}$
   - 6.5 Construct coset $S = \{s \in D_{\texttt{next}} | q_{r_{i+1}}(s) = x_{\texttt{next}}\}$. $|S| =$ `fri_params`.$m^{r_{i+1}}$
     **Remark**: For the case $m = 2$, all $s_j$ can be found from the equation

$$x_{\texttt{next}} - X^{2^{r_{i+1}}} = 0.$$

   - 6.6 $\texttt{values}_{i+1} := \{f_{\texttt{next}}(s)\}_{s \in S}$
   - 6.7 If $i < steps - 2$, then $T_{\texttt{next}} :=$ `Precommit`$(f_{\texttt{next}}(X), D, r_i)$
   - 6.8 Form a round proof:
     - 6.8.1 $\texttt{values}_i$ is data for one leaf of tree $T$. So we can construct Merkle proof for it:

$$p := \texttt{make\_merkle\_proof}(T, \texttt{values}_i)$$

       **Remark**: $\texttt{values}_i$ where computed on previous step for $i > 0$.
     - 6.8.2 $\texttt{colinear\_path} := \texttt{make\_merkle\_proof}(T_{\texttt{next}}, \texttt{values}_{i+1})$
     - 6.8.3 $\texttt{round\_proof}_i := \{p, T.\texttt{root}(), \texttt{colinear\_path}\}$
   - 6.9 $x := x_{\texttt{next}}, f := f_{\texttt{next}}, T := T_{\texttt{next}}, D := D_{\texttt{next}}$
7. Compute final polynomial:
   - $d' = \mathbf{deg}(Q(X))$
   - $\texttt{final\_polynomial} = \{c_0, \ldots, c_{2^{\log d' - r}}\}$, where $f_{\texttt{next}}(X) = \sum_{i=0}^{2^{\log d' - r}} c_i X^i$
8. $\pi = \{\texttt{round\_proof}_0, \ldots \texttt{round\_proof}_{steps-1}, \texttt{final\_polynomial}, \texttt{values}_0, \ldots \texttt{values}_{steps}\}$

## 5.6 Verify eval

**Algorithm 16** LPC.Verify

**Input**: queries $\xi_0, \ldots \xi_{k-1}$, proof $\mathcal{P}$, root of Merkle tree `t_polynomials`, `fri_params`, `transcript`

1. If $\mathcal{P}$.`T_root`! = `t_polynomials`, then abort.
2. `MultiEvalVerify`:
   - 2.1 Interpolate polynomial $U(X)$ such that $U(\xi_j) = z_j$ for $0 \leq j < k$
   - 2.2 $V(X) = \prod_{j=0}^{k-1}(X - \xi_j)$
   - 2.3 for $i$ from 0 to $\lambda - 1$:
     - 2.3.1 Abort if `FRI.Verify`($\mathcal{P}$.`fri_proof`$_i$, `fri_params`, `t_polynomials`, $U(X), V(X)$, `transcript`) returns `false`.

Why have we thrown away Merkle proofs $p_{z_0} \ldots p_{z_{k-1}}$?

**Algorithm 17** FRI.Verify

**Input**: FRI proof $\pi$, `fri_params`, Merkle tree root `t_polynomials`, $U(X), V(X)$, `transcript`

1. `transcript`(`t_polynomials`)
2. $x :=$ `transcript`.**get_challenge**()
3. $steps :=$ `fri_params`.$steps, m :=$ `fri_params`.$m, r_i :=$ `fri_params`.$r_i, q(X) :=$ `fri_params`.$q(X)$
4. for $i = 0, \ldots, steps - 2$:
   - 4.1 $x_{\texttt{next}} := q_{r_i}(x)$
   - 4.2 $D :=$ `fri_params`.$D^{\sum_{j=0}^{i} r_j}$
   - 4.3 Construct coset $S = \{s \in D : q_{r_i}(s) = x_{\texttt{next}}\}$, coset size $|S| = m^{r_i}$
   - 4.4 Get polynomial values $y = \{y_j\}_{j=0}^{m^{r_i}-1}$ from proof $\pi$
     - if $i = 0$ then $y_j = \frac{\pi.\texttt{values}_0[j] - U(s_j)}{V(s_j)}$
     - otherwise $y_j = \pi.\texttt{values}_i[j]$

     **Remark**: $\pi.\texttt{values}_0$ are values of the original polynomial, not $Q(X)$. For this reason, we need to recompute them.
   - 4.5 If not `validate`($\pi.\texttt{round\_proof}_i, y$), return `false`.
   - 4.6 Check correctness of Merkle trees' roots in round proofs.
     - if $i = 0$ then if proof $\pi.\texttt{round\_proofs}_i.p$ is not from Merkle tree with root `t_polynomials` then return `false`.
     - if $i \neq 0$ then if proof $\pi.\texttt{round\_proofs}_{i-1}.\texttt{colinear\_path}$ is not from Merkle tree with root $\pi.\texttt{round\_proofs}_i.T$ then return `false`.
   - 4.7 Colinearity check.
     - 4.7.1 Let $S$ is coset for values on $i$-th step.
     - 4.7.2 For each $j = 0, \ldots, r_i - 2$ round in $i$-th step:
       - 4.7.2.1 $\alpha =$ `transcript`.**get_challenge_from**($\mathbb{F}$), $\alpha \in \mathbb{F}$
       - 4.7.2.2 For each pair $s, -s \in S$ compute $\texttt{interpolant}_s(X)$ using values from $\mathcal{P}.\texttt{values}_i$ and results from previous iteration.
       - 4.7.2.3 $S_{\texttt{next}}$ is set of $s_j^2$
       - 4.7.2.4 $S := S_{\texttt{next}}$, $y := y_{\texttt{next}}$
     - 4.7.3 after this operation $S$ consisits of single element $\mathbf{x}$
     - 4.7.4 $\alpha =$ `transcript`.**get_challenge_from**($\mathbb{F}$), $\alpha \in \mathbb{F}$
     - 4.7.5 Compute $\texttt{interpolant}_{\mathbf{x}}(X)$
     - 4.7.6 Retrieve $\texttt{colinear\_value}_i$ from $\mathcal{P}.\texttt{values}_{i+1}$
     - 4.7.7 If $\texttt{interpolant}_{\mathbf{x}}(\alpha)$ is not equal to $\texttt{colinear\_value}_i$ then return `false`.
   - 4.8 `validate`($\pi.\texttt{round\_proof.colinear\_path}, \pi.\texttt{values}[i+1]$)
   - 4.9 `transcript`(`colinear_path.root()`)
   - 4.10 $x = x_{\texttt{next}}$
5. for the last step $step - 1$:
   - 5.1 Check that $\pi.\texttt{final\_polynomial}$ contains $2^{\log d' - r}$ elements
   - 5.2 $f(X) := \sum_{i=0}^{2^{\log d' - r}} \pi.\texttt{final\_polynomial}.c_i \cdot X^i$
   - 5.3 Check that $f(x) = \pi.\texttt{round\_proof}_{r-2}.\texttt{colinear\_value}$

How it works if m!=2? Then interpolant won't be a straight line?

# 6 Zero Knowledge

## 6.1 Cosets

As a part of modifications to achieve zero-knowlege property, [1] proposes to use a cosets of the subdomains $D^{(i)}$ introduced in Section 4. Let $h \in \mathbb{F}^*/D$. Define domains $D^{(0)'} = hD^{(0)}, \ldots, D^{(r)'} = hD^{(r)}$. FRI protocol works with new domains in the same way as described in Section 4.

## 6.2 Hiding Commitments

introduce $\lambda$ earlier

We use Merkle tree commitments with a pivacy adjustements from [2]. Each Merkle tree leaf contrains concatenation of the original leaf data and a random value of the size $2\lambda$ for the given security parameter $\lambda$.

## 6.3 Random Rows

Use citation

We use the same approach as Mina[2] and Halo[3]. The zero-knowledge adjustment is already included in the protocol in Section 3.3. In this section, we provide details on a PLONK-trace table preprocessing.

The basic idea is to fill the last $t$ rows of the table with uniformly distributed random values. In this case, the values of the polynomials constructed during the protocol are uniformly distributed random values as well. The same is true for the last $t$ evaluations of permutation and lookup polynomials. However, this change affects the permutation and lookup arguments.

Denote the number of usable rows by $N_{\texttt{usable}} = N_{\texttt{rows}} - t - 1$. Now we introduce two additional selectors:

- $q_{\texttt{blind}}$, $q_{\texttt{blind}}(\omega^i) = 1$ for $N_{\texttt{usable}} < i \leq N_{\texttt{rows}}$ and $q_{\texttt{blind}}$ is equal to zero elsewhere.
- $q_{\texttt{last}}$, $q_{\texttt{last}}(\omega^{N_{\texttt{usable}}}) = 1$ and and $q_{\texttt{last}}$ is equal to zero elsewhere.

The new selectors and corresponding calculations are included in the protocol in Section 3.3.

Details on how to calculate $t$

# 7 Permutation Argument Details

Here we describe the transformation from copy constraints to permutation argument described as a part of the protocol in Section 3.3.

## 7.1 Cells as Permutation Cycles

Let $c_{i,j}, c_{i',j'}$ be two cells of the table representation of the circuit's trace. Denote by $\texttt{value}(c)$ value of the cell $c$ during circuit's trace computation. Copy constraint $\texttt{Cp}(c_{i,j}, c_{i',j'})$ asserts that $\texttt{value}(c_{i,j}) = \texttt{value}(c_{i',j'})$.

Using copy constraints, we define permutation over the table's cells. The permutation can be presresented as a set of cycles[4]. Note that distinct cycles are disjoint. For each set of equal cells $\{c_i, \ldots, c_{i+k}\}$ define a cycle $C = (c_i, \ldots, c_k)$. $C$ is 'sub-permutation' $\delta_C$ such that $\delta_C(c_j) = c_{j+1}$ for $i \leq j < k$ and $\delta_C(c_k) = c_i$.

Thus, we can split all copy constraints into a set of cycles such that cells in the same cycles are supposed to have the same circuit's trace value. The circuit's permutation is defined as a composition of these cycles.

Example

## 7.2 Permutation Construction Algorithm

We use the same algorithm as Halo[5].

The state is represented as:

[2] https://minaprotocol.com/blog/a-more-efficient-approach-to-zero-knowledge-for-plonk
[3] https://zcash.github.io/halo2/design/proving-system/lookup.html#zero-knowledge-adjustment
[4] https://en.wikipedia.org/wiki/Permutation#Cycle_notation
[5] https://zcash.github.io/halo2/design/proving-system/permutation.html#algorithm

- a map **mapping** for the permutation itself;
- a map **aux** that keeps track of a distinguished element of each cycle;
- a map **sizes** that keeps track of the size of each cycle.

If $x$, $y$ belong to the same cycle, then $\mathbf{aux}(x) = \mathbf{aux}(y)$. $\mathbf{sizes}(\mathbf{aux}(x))$ containts the size of the cycle containing $x$.

**Remark**: Here, we use one label for the element of the permutation for simplicity. However, it is $x = (i, j)$, where $i$ is the cell's column and $j$ is the cell's row.

**Algorithm 18** Copy State Initialization

1. For all $x$ (each $x$ is one-element cycle):
   1.1 $\mathbf{mapping}(x) = x$
   1.2 $\mathbf{aux}(x) = x$
   1.3 $\mathbf{sizes}(x) = 1$

**Algorithm 19** Add Copy Constraint $\texttt{Cp}(x, y)$

1. `if` $\mathbf{aux}(x) = \mathbf{aux}(y)$:
   1.1 `return` // don't do anything if $x, y$ belong to the same cycle
2. Let `left` be an input with a larger cycle (defined by **sizes**) and `right` the other one.
3. $\mathbf{sizes}(\mathbf{aux}(\texttt{left})) = \mathbf{sizes}(\mathbf{aux}(\texttt{left})) + \mathbf{sizes}(\mathbf{aux}(\texttt{right}))$ // the right cycle will be merged into the left cycle
4. $z = \mathbf{aux}(\texttt{right})$
5. `do`: // set all pointers from `right` cycle to `left` cycle
   5.1 $\mathbf{aux}(z) = \mathbf{aux}(\texttt{left})$
   5.2 $z = \mathbf{mapping}(z)$

   `while`$(z \neq \mathbf{aux}(\texttt{right}))$
6. `tmp` $= \mathbf{mapping}(\texttt{left})$ // actually merge cycles in mapping
7. $\mathbf{mapping}(\texttt{left}) = \mathbf{mapping}(\texttt{right})$
8. $\mathbf{mapping}(\texttt{right}) = tmp$

Example

## 7.3 Permutation Polynomial

The algorithm 19 outputs permutation $\sigma$ as a resulting copy state. Now we need to transform the permutation to the permutation polynomials.

Let $\sigma(\text{col} : i, \text{row} : j) = (\text{col} : i', \text{row} : j')$. We can construct it using $\mathbf{mappring}(x = (i, j)) = (i', j')$. Let $\omega$ be a $2^k$ root of unity, $\delta$ be a $T$ root of unity, where $T \cdot 2^S + 1 = p$, $k \leq S$, $T$ odd and $p$ is a size of the field.

Define $k$

Now we can interpolate permutation polynomials as:

$$S_{id_i}(\omega^j) = \delta^i \cdot \omega^j \text{ for } i = 0, \ldots, N_{\texttt{perm}} - 1$$
$$S_{\sigma_i}(\omega^j) = \delta^{i'} \cdot \omega^{j'} \text{ for } i = 0, \ldots, N_{\texttt{perm}} - 1$$

$S_{id_i}(X)$ is called identity permutation polynomials, and $S_{\sigma_i}(X)$ is called permutation polynomials. The constructed polynomials are used in the main protocol described in Section 3.3.

Do we need a separate explanation for permutation argument?

# 8 Lookup Argument

Here we describe the transformation from lookup constraints to lookup argument described as a part of the protocol in Section 3.3. We use the lookup argument proposed in Halo[6].

Let $T$ be a PLONK-trace table. Let $\mathbf{S} = S_0, \ldots, S_{m-1}$ be a table with $m$ columns and $N_{\mathbf{rows}}$ rows. Note, that $N_{\mathbf{rows}}$ is equal to the number of usable rows in $T$. For `lookup input` cells $(T_{i_0,j_0}, \ldots, T_{i_{m-1},j_{m-1}})$, lookup constraint allow to assert that $\mathbf{S}$ contains a row `lookup value` with values of these cells.

Denote columns of $T$ that are included in lookup argument as $\mathbf{A} = A_0, \ldots, A_{m-1}$. We refer to $A_i$ as input columns and to $S_i$ as lookup columns.

The $\mathbf{A}$ and $\mathbf{S}$ contain the same number of rows. Moreover, each value in $\mathbf{A}$ has to be presented in $\mathbf{S}$ (we'll ). Both $\mathbf{A}$ and $\mathbf{S}$ can contain duplicate. If it is necessary to extend one of the sets, we extend $\mathbf{S}$ with duplicates and $\mathbf{A}$ with dummy values known to be in $\mathbf{S}$.

Let $\theta \in \mathbb{F}$ is the verifier's challenge. We compress the columns $A_i$, $S_i$ into two columns $A$, $S$ as follow:

$$A_{\texttt{compr}} = \theta^{m-1}A_0 + \cdots + \theta A_{m-2} + A_{m-1}$$
$$S_{\texttt{compr}} = \theta^{m-1}S_0 + \cdots + \theta S_{m-2} + S_{m-1}$$

There are two parts of lookup argument similar to the original PLONK argument: permutation and assertion check. Firstly, the prover permutes $\mathbf{A}$, $\mathbf{S}$ in a such way that verification of inclusion lookup queries into $\mathbf{S}$ is relatively simple task. After that, the prover provides a permutation argument for the permuted columns. Finally, they proves that the values from the permuted $\mathbf{A}$ is subset of the values from the permuted $\mathbf{S}$.

## 8.1 Permutation

Firstly, the prover calculates two additional columns $A_{\texttt{perm}}$ and $S_{\texttt{perm}}$ that are permutations of $A_{\texttt{compr}}$ and $S_{\texttt{compr}}$ respectively.

The permutations for the new columns are defined by the following rules:

- All the cells of column $A_{\texttt{perm}}$ are arranged so that like-valued cells are vertically adjacent to each other. The order of these like-valued groups is not matter.
- The first row in a sequence of like values in $A_{\texttt{perm}}$ is the row that has the corresponding value in $S_{\texttt{perm}}$. The order of the other values in $S_{\texttt{perm}}$ can be arbitrary.

Similarly to Section 7, we use a grand product argument [3] to prove that $A_{\texttt{perm}}$, $S_{\texttt{perm}}$ are permutations of $A_{\texttt{compr}}$, $S_{\texttt{compr}}$ in the step **??**.

## 8.2 Assertion Check

The permuted columns are constructed in a such way that we can assert that all elements from $A_{\texttt{perm}}$ are presented in $S_{\texttt{perm}}$ with the following rules:

1. $(A_{\texttt{perm}}(X) - S_{\texttt{perm}}(X)) \cdot (A_{\texttt{perm}}(X) - A_{\texttt{perm}}(\omega^{-1}X))$ to ensure that either $A_{\texttt{perm}}[j] = S_{\texttt{perm}}[j]$ or $A_{\texttt{perm}}[j] = A_{\texttt{perm}}[j-1]$
2. $L_1(X) \cdot (A_{\texttt{perm}}(X) - S_{\texttt{perm}}(X))$. We need it because $(A_{\texttt{perm}}(X) - A_{\texttt{perm}}(\omega^{-1}X))$ is not a valid check on the first row.

In order to archieve zero-knowledge we use the following constraints:

1. $(1 - (q_{\texttt{last}}(X) + q_{\texttt{blind}}(X))) \cdot (V_L(\omega X) \cdot (A_{\texttt{perm}}(X) + \beta) \cdot (S_{\texttt{perm}}(X) + \gamma) - V_L(X) \cdot (A_{\texttt{compr}}(X) + \beta) \cdot (S_{\texttt{compr}}(X) + \gamma)) = 0$
2. $(1 - (q_{\texttt{last}}(X) + q_{\texttt{blind}}(X))) \cdot (A_{\texttt{perm}}(X) - S_{\texttt{perm}}(X)) \cdot (A_{\texttt{perm}}(X) - A_{\texttt{perm}}(\omega^{-1}X))$
3. $q_{\texttt{last}}(X) \cdot (V_L(X)^2 - V_L(X)) = 0$

[6] https://zcash.github.io/halo2/design/proving-system/lookup.html

## 8.3 Generalization

Each lookup input's cell can be any polynomial expression and use the relative references in the lookup constraint. It influences on the way how the column $A_{\texttt{compr}}$ is calculated.

To combine multiple lookup constraints into one argument, we add one more random challenge. Denote a random challenges by $\theta$.

Now we need to introduce notations for general lookup representation.

- **Lookup Table**: a table of values with columns $S_i$
- **Lookup Input**: a set of cells of the PLONK table of the form $(a_0, \dots, a_k)$
- **Compressed Lookup Table**: a column that represents jointed columns of all Lookup Tables.
- **Compressed Lookup Input**: a column that represents jointed column of all lookup inputs.
- **Lookup Constraint**: $(a_0, \dots, a_{k_i-1}) \in S$ for some lookup table $S$
- **Lookup Expression**: Polynomial representation of the lookup constraint

Suppose the circuit $C$ contains $N_{\texttt{tables}}$ lookup tables and $N_{\texttt{lookup}}$ lookup constraints.

Each lookup constraint may contain different width of input. Denote by $k_i$ the number of elements in the lookup input of the $i$-th lookup constraint. Let $d_{j_i}$ be the rotation of each element in the lookup constraint (i.e. shift by $d_{j_i}$ rows). Let $\nu_i$ be the initial degree of the random challenge for $i$-th lookup constraint.

$$\begin{gathered}\nu_{i+1} = \nu_i + k_i \\ \nu_0 = 0.\end{gathered}$$

Thus, the lookup expression would be:

$$\texttt{lookup\_gate}_i(X) = (\theta^{k_i-1+\nu_i} A_{0_i}(\omega^{d_{0_i}} X) + \cdots + \theta^{\nu_i} A_{k_i-1}(\omega^{d_{k_i-1}} X))$$

The compressed lookup input:

$$A_{\texttt{compr}}(\omega^j) = \sum_{0 \le i < N_{\texttt{lookup}}} \texttt{lookup\_gate}_i(\omega^j)$$

Note that each column $A_{i,j}$, which represents one of the columns in table $T$, is not necessarily different from each other.

Let $\nu_i$ for a table value be defined in the same way as for lookup inputs. The compressed lookup table is computed similarly to compressed lookup input:

$$\begin{gathered}\texttt{table\_value}_i(\omega^j) = (\theta^{k_i-1+\nu_i} S_{0_i}(\omega^j) + \cdots + \theta^{\nu_i} S_{k_i-1}(\omega^j)) \\ S_{\texttt{compr}}(\omega^j) = \sum_{0 \le i < N_{\texttt{lookup}}} \texttt{table\_value}_i(\omega^j)\end{gathered}$$

We need to prove the security of the lookup aggregation

## 8.4 Small Tables

Note that we can arrange multiple tables in the same columns using tag column.

For instance, let $\mathbf{S}_1 = S_{1_0}, S_{1_1}$, $\mathbf{S}_2 = S_{2_0}, S_{2_1}$ be two lookup tables with 4 rows. Two lookup expression corresponds to $\mathbf{S}_1, \mathbf{S}_2$. These tables can be located in the separate way:

| $ql_1$ | $S_{1_0}$ | $S_{1_1}$ | $ql_2$ | $S_{2_0}$ | $S_{2_1}$ |
|---|---|---|---|---|---|
| ... | 0 | 1 | ... | 4 | 5 |
| ... | 1 | 1 | ... | 5 | 6 |
| ... | 2 | 1 | ... | 6 | 7 |
| ... | 3 | 0 | ... | 7 | 8 |

However, $N_{\texttt{rows}} >> 4$ in a typical case. This means that the prover has to complete these columns to $N_{\texttt{rows}}$ and commit all of them. Instead of this, we can arrange the tables in the following way:

| $q_{l_1}$ | $q_{l_2}$ | tag | $S_0$ | $S_1$ |
|---|---|---|---|---|
| ... | ... | 1 | 0 | 1 |
| ... | ... | 1 | 1 | 1 |
| ... | ... | 1 | 2 | 1 |
| ... | ... | 1 | 3 | 0 |
| ... | ... | 2 | 4 | 5 |
| ... | ... | 2 | 5 | 6 |
| ... | ... | 2 | 6 | 7 |
| ... | ... | 2 | 7 | 8 |

It allows saving up to $(N_{\texttt{con\_tables}} - 1) \cdot \texttt{max\_columns} - 1$ columns, where `max_columns` is the maximum number of columns in all $N_{\texttt{con\_tables}}$ concatenated tables.

### 8.5 Non-Fixed Lookup Tables

The table **S** has not to be fixed. Any columns from **S** can be witness columns. This does not change the argument above.

## 9 Optimizations

**WIP**

### 9.1 Batched FRI

Instead of checking each commitment individualy, it is possible to aggregate them for FRI. For polynomials $f_0, \ldots, f_k$:

1. Get $\theta$ from transcript
2. $f = f_0 \cdot \theta^{k-1} + \cdots + f_k$
3. Run FRI over $f$, using oracles to $f_0, \ldots, f_k$

Thus, we can run only one FRI instance for all commited polynomials.
See [1] for details.

### 9.2 Hash By Column

Instead of committing each of the polynomials, it is possible to use the same Merkle tree for several polynomials. This leads to the decrease of the number of Merkle tree paths which are required to be provided by the prover.
See [**?**], [1] for details.

### 9.3 Hash By Subset

Each $i + 1$ FRI round supposes the prover to send all elements from a coset $H \in D^{(i)}$. Each Merkle leaf is able to contain the whole coset instead of separate values.
See [**?**] for details. Similar approach is described in [1]. However, the authors of [1] use more values per leaf, that leads to better performance.

### 9.4 FRI PoW

**WIP**

## 10 Placeholder Parameters

In this section, we discuss Placeholder parameters and their influence on the protocol security and performance.

## 10.1 Circuit Influence

Let $C$ be a circuit that should be proven. Recall some notations from Section 3.3.

| | |
|---|---|
| $N_{\texttt{rows}}$ | Number of rows (unpadded to power of two) in $C$ |
| $N_{\texttt{witness}}$ | Number of witness columns |
| $N_{\texttt{perm}}$ | Number of witness columns that are included in the permutation argument |
| $N_{\texttt{sel}}$ | Number of selectors used in the circuit |
| $N_{\texttt{lookups}}$ | Number of lookups |

PADDING TO POWER OF TWO

FAKE ROWS

## 10.2 FRI Parameters

Let $\mathbf{RS}[\mathbb{F}, D, \rho]$ be Reed-Solomon code family. Here $|D| = n = 2^k$, $\rho = 2^{-R}(k, R\mathbb{N})$. This implies degree bound for commiting polynomials $d = 2^{k-R}$. Fix $r \in [1, \log d = n]$ — the number of FRI inner rounds. Let $l$ be a repetition parameter.

Prover: $\mathcal{O}(n)$, Verifier: $\mathcal{O}(\log n)$.

For every $\epsilon \in (0, 1]$, let $J_\epsilon : [0, 1] \to [0, 1]$ be the function

$$J_\epsilon(X) = 1 - \sqrt{1 - X(1 - \epsilon)}$$

USE THE LATEST SOUNDNESS BOUND

Suppose that $\Delta(f, \mathbf{RS}) = \delta > 0$, then soundness error is bounded by:

$$\mathbf{err}(\delta) = \frac{2 \log |D|}{\epsilon^3 |\mathbb{F}|} + \left(1 - \mathtt{min}\{\delta_0, J_\epsilon(J_\epsilon(1 - \rho))\} + \epsilon \log |D|\right)^l$$

## 10.3 Placeholder Parameters

Now we can apply the circuit parameters to FRI commitments.

Let $d$ be the smallest power of two such that $d \geq N_{\texttt{rows}}$. In Placeholder, $d$ defines the highest degree of polynomials that can be committed by FRI instance.

CAN WE JUST USE NOTATION $N_{\texttt{rows}}$ FOR PADDED ROWS?

Let $\omega$ be $d$ root of unity. Recall that $d$ has a form $d = 2^{\hat{n}}$ and $\hat{n} \geq N_{\texttt{rows}}$.

We use $\mathbf{RS}[\mathbb{F}, D, \rho]$ for FRI commitments, where

- $\mathbb{F}$ is the same field that is used in PLONK arithmetization.
- $D$ is the domain of $n = 2^k = 2^{\hat{n}+R}$ root of unity.
- $\rho = 2^{-R}$ is the parameter that can be adjusted.

Let $f$ be a polynomial used during Placeholder proof. To interpolate $f$, we use the powers of $\omega$. To commit $f$, we need to build low degree extension $f|_D$.

CHANGE THE NOTATION

An additional root of unity is used for permutation polynomials interpolation. Let $\delta$ be $T$ root of unity where $T \cdot 2^S + 1 = p$, $T$ odd and $k \leq S$

$\delta^i$ are all distinct quadratic non-residues

LOOKUP PARAMETERS SHOULD BE HERE

IT'S FOR PASTA CURVES

$m$ for lookup

I suppose, we need also fix $\epsilon \in (0, 1]$?! - $\epsilon$-ball is defined for LPC $L_\delta(f)$ is $\delta$-list of $f$. $\Delta(f, g) < \delta$. $\delta > 0$ - error-bound

"distinguishing" point $z$

# 11 Circuit Performance Estimation

Recall and update notations:

| | |
|---|---|
| $n$ | Number of rows |
| $N_{\texttt{witness}}$ | Number of witness columns (‘advice columns‘) |
| $N_{\texttt{perm}}$ | Number of witness columns that are included in the permutation argument |
| $N_{\texttt{sel}}$ | Number of selectors used in the circuit |
| $N_{\texttt{const}}$ | Number of constant columns |
| $\mathbf{f}_i$ | Witness polynomials, $0 \leq i < N_{\texttt{witness}}$ |
| $\mathbf{f}_{c_i}$ | Constant-related polynomials, $0 \leq i < N_{\texttt{const}}$ |
| $\mathbf{gate}_i$ | Gate polynomials, $0 \leq i < N_{\texttt{sel}}$ |
| $\sigma(\text{col} : i, \text{row} : j) = (\text{col} : i', \text{row} : j')$ | Permutation over the table |
| $H_c$ | Commitment hash |
| $H_r$ | Random Oracle hash |
| $l_{H_c}$ | Number of bits in commitment hash |
| $l_{H_r}$ | Number of bits in random oracle hash |

Public data:

- ???

## 11.1 Proof Size

Proof contains: ALICE CHECK LOOKUP

- $f_{0,\texttt{comm}}, \ldots, f_{N_{\texttt{witness}}-1,\texttt{comm}}$ - commitments for witness polynomials
- $A'_{\texttt{comm}}, S'_{\texttt{comm}}$ - lookup commitments
- $P_{\texttt{comm}}, Q_{\texttt{comm}}$
- $V_{\texttt{comm}}$ - lookup related
- $T_{0,\texttt{comm}}, ..., T_{N_{\texttt{perm}}-1,\texttt{comm}}$
- Values and paths with size $\log n$:
  - $f_i(y)$ for $i \in [0, N_{\texttt{witness}} - 1]$
  - $P(y), P(y\omega), Q(y), Q(y\omega)$
  - $T_j(y)$ for $j \in [0, N_{\texttt{perm}} - 1]$
  - $A'(y), S'(y), V(y), A'(y\omega^{-1}), V(y\omega)$
  - Gate-depending $f_i(y\omega^{\mu})$
- For circuit polynomials: distinguishing point values
- Evaluation proof for the values above ($l$ times):
  - for $i \in [0, r-1]$: // i = 0 makes sense because it's for $U$ polynomial,
    - $m + 1$ values
    - $m$ merkle paths of size $\log n - i$
  - in $r$ round: $\log n - r$ values

Firstly, we define the size of basic structures in the proof. Each commitment has size $l_{H_c}$ as a Merkle tree root. Values are taken from the field $\mathbb{F}$ with a bit length $l_p$.

FRI proofs size depends on FRI parameters. Fix RS code familty $\text{RS}[\mathbb{F}, D, \rho]$, where $|D| = n = 2^k$ and rate $\rho = 2^{-R}$. This implies that degree bound $d$ is $2^{k-R}$

Let $J_\epsilon(X) = 1 - \sqrt{1 - X(1 - \epsilon)}$

For any $\epsilon \in (0, 1])$, FRI soundness error $\mathbf{err}(\delta)$ is bounded by:

$$\frac{2\log|D|}{\epsilon^3|\mathbb{F}|} + (1 - min\{\delta_0, J_\epsilon(J_\epsilon(1-\rho))\} + \epsilon\log|D|)^l$$

Here $l$ us repetition parameter. The soundness bound was improved in [4] (something around $1/2x$). CHECK IT LATER

Proof size (original):

1. Commitments: $\texttt{comm\_n} = N_{\texttt{witness}} + 2 + N_{\texttt{perm}}$
2. LookUp Commitments: $\texttt{lookup\_comm\_n} = 3$
3. Evaluations: $\texttt{eval\_n} = N_{\texttt{witness}} + 4 + N_{\texttt{perm}} + GATES$

4. Evaluations Merkle paths: $(\log n \cdot |H_c|) \cdot \texttt{eval\_n}$
5. LookUp Evaluations: 5
6. Evaluations proofs: $l \cdot [r \cdot (m+1) \cdot |p| + \sum_{i=0}^{r-1}((m+1) \cdot (\log n - i) \cdot |H_c|) + (\log n - r) \cdot |p|]$

# Appendices

## A Circuit Example

## B Get It All Together

Protocol description with optimizations, zk, etc.

## References


1. Kattis A., Panarin K., Vlasov A. RedShift: Transparent SNARKs from List Polynomial Commitment IOPs. Cryptology ePrint Archive, Report 2019/1400. 2019. https://ia.cr/2019/1400.
2. Ben-Sasson E., Chiesa A., Spooner N. Interactive Oracle Proofs. Cryptology ePrint Archive, Report 2016/116. 2016. https://ia.cr/2016/116.
3. Gabizon A., Williamson Z. J., Ciobotaru O. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953. 2019. https://ia.cr/2019/953.
4. Ben-Sasson E., Carmon D., Ishai Y. et al. Proximity Gaps for Reed-Solomon Codes. Cryptology ePrint Archive, Report 2020/654. 2020. https://ia.cr/2020/654.
5. Fast Reed-Solomon interactive oracle proofs of proximity / E. Ben-Sasson, I. Bentov, Y. Horesh et al. // 45th international colloquium on automata, languages, and programming (icalp 2018) / Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.
6. Gabizon A., Williamson Z. J. Proposal: The Turbo-PLONK program syntax for specifying SNARK programs. https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf.
7. Gabizon A., Williamson Z. J. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315. 2020. https://ia.cr/2020/315.
8. PLONKish Arithmetization - The halo2 book. https://zcash.github.io/halo2/concepts/arithmetization.html.
9. Lookup argument - The halo2 book. https://zcash.github.io/halo2/design/proving-system/lookup.html.