PLONK

Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge

Presenter: Ninh Quoc Bao

Da Nang City, 12 July 2025

Reintroducing SNARKs

Scenario: A Prover wants to convince a Verifier that a certain computation was performed correctly. However:

- The Prover may want to keep parts of the computation secret
- The Verifier may want to verify the result without redoing the entire computation

Reintroducing SNARKs

Scenario: A Prover wants to convince a Verifier that a certain computation was performed correctly. However:

- The Prover may want to keep parts of the computation secret
- The Verifier may want to verify the result without redoing the entire computation

SNARKs (Succinct Non-interactive Arguments of Knowledge) allow the Prover to generate a *short proof* that:

- Convincingly proves correctness
- Hides specific parts of the input (if Prover want)
- Is quick for the Verifier to verify

Reintroducing SNARKs

SNARK : Succinct Non-interactive ARgument of Knowledge

- *Succinct* : The proof length is short and Verifier time is fast
- *Non-interactive* : The protocol does not require back-and-forth communication
- *ARgument* : Basically a proof
  (More precise: A computationally sound proof, which is secure against adversaries with bounded computational resources.)
- *of Knowledge* : The proof does not just show the system of equations has a solution; it also show that the Prover knows the solution

Zero-Knowledge SNARK (zk-SNARK): A SNARK where the Verifier learns nothing about the Prover's private input, except what the Prover intentionally reveals.

PLONK: A Modern zkSNARK

PLONK stands for:

Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge

It is a powerful type of **zkSNARK** that offers two major advantages:

- **Universal Trusted Setup:** A single setup can be reused for any circuit or program, regardless of the specific computation.
- **Updatable Setup:** Anyone can contribute randomness to the setup phase. As long as one participant is honest, the resulting setup remains secure - and can be updated over time to improve trust.

Programmable Cryptography

Programmable Cryptography refers to the ability to generate proofs for arbitrary computations.

Programmable Cryptography

Programmable Cryptography refers to the ability to generate proofs for arbitrary computations.

$\Rightarrow$ To achieve this, we need a kind of "programming language" to express the computation. *That is, we must represent the function in a form that the SNARK system can understand and generate a proof for.*

Programmable Cryptography

Programmable Cryptography refers to the ability to generate proofs for arbitrary computations.

$\Rightarrow$ To achieve this, we need a kind of "programming language" to express the computation. *That is, we must represent the function in a form that the SNARK system can understand and generate a proof for.*

The process of translating a computational problem into an algebraic form that a SNARK system can interpret and use to generate a proof called **Arithmetization**.

Circuit

A **circuit** (or Arithmetic Circuit) is an algebraic representation of a computational problem.

Given a problem, we can translate it into a circuit form that expresses its logic using arithmetic constraints.

Conversely, a circuit can be interpreted back into the original problem it was designed to represent.

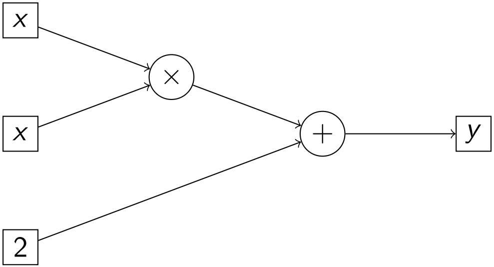
Circuit

A **circuit** (or Arithmetic Circuit) is an algebraic representation of a computational problem.

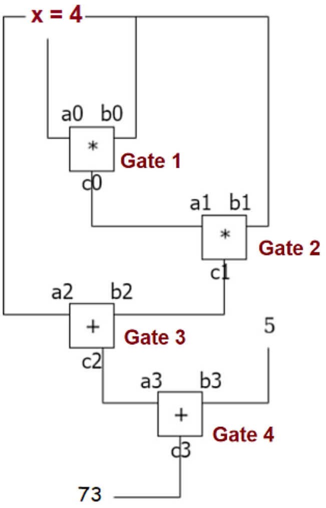
Given a problem, we can translate it into a circuit form that expresses its logic using arithmetic constraints.

Conversely, a circuit can be interpreted back into the original problem it was designed to represent.

Example: Arithmetic Circuit for $y = x^2 + 2$



Arithmetic Circuit for PLONK



Vectors	a	b	c
Gate 1	4	4	16
Gate 2	16	4	64
Gate 3	4	64	68
Gate 4	68	5	73

Source: PlonK — Risen Crypto

PLONK Instance

An instance of PLONK consists of two main components:

PLONK Instance

An instance of PLONK consists of two main components:

- **Gate Constraints:** Ensure that each gate's computation is performed correctly.
  e.g., $a_0 \cdot b_0 = c_0, \quad a_2 + b_2 = c_2, \quad \ldots$

PLONK Instance

An instance of PLONK consists of two main components:

- **Gate Constraints:** Ensure that each gate's computation is performed correctly.
  $e.g.,\ a_0 \cdot b_0 = c_0, \quad a_2 + b_2 = c_2, \quad \ldots$
- **Copy Constraints:** Ensure that values on connected wires are consistent across the circuit.
  $e.g.,\ c_0 = a_1, \quad c_1 = b_2, \quad \ldots$

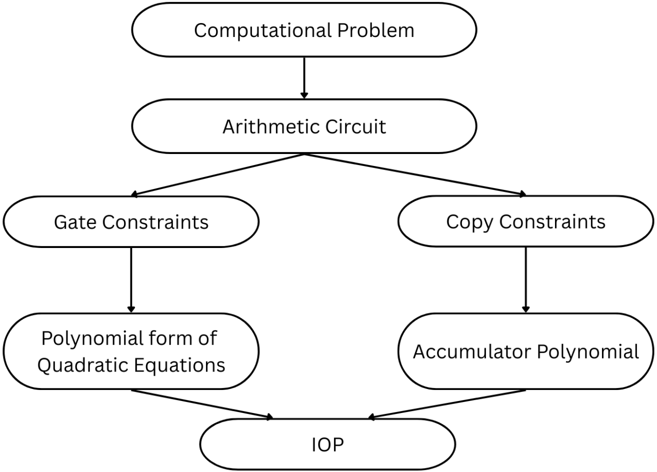
PLONK Instance

An instance of PLONK consists of two main components:

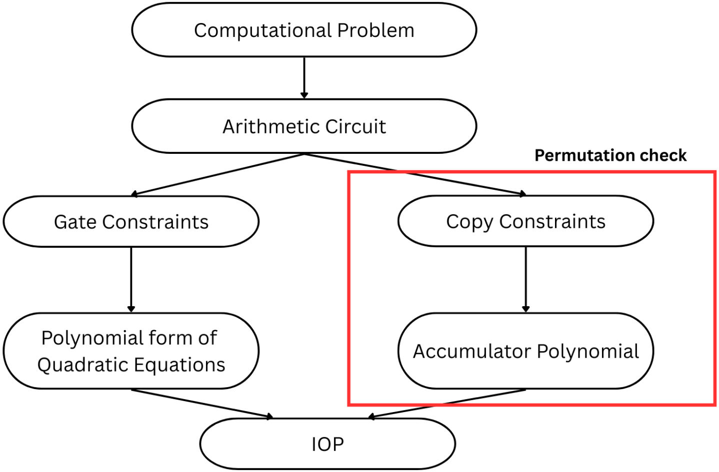
- **Gate Constraints:** Ensure that each gate's computation is performed correctly.
  $e.g.,\ a_0 \cdot b_0 = c_0, \quad a_2 + b_2 = c_2, \quad \ldots$
- **Copy Constraints:** Ensure that values on connected wires are consistent across the circuit.
  $e.g.,\ c_0 = a_1, \quad c_1 = b_2, \quad \ldots$

To prove a valid PLONK instance, the Prover must demonstrate that both the **Gate Constraints** and **Copy Constraints** are satisfied.

PLONK Overview



PLONK Overview



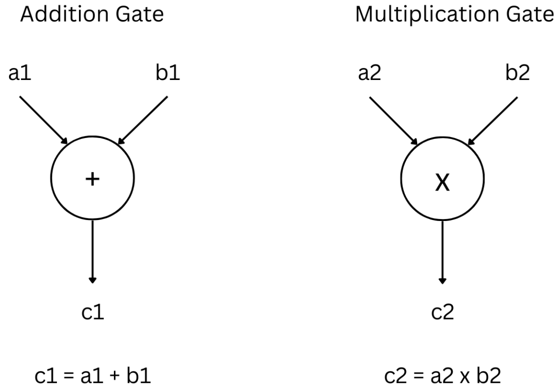
PLONK : All you need is a permutation check

Proving:

- **Gate Constraints**: $a_0 \cdot b_0 = c_0, a_2 + b_2 = c_2, \ldots$ is **easy**
- **Copy Constraints**: $c_0 = a_1, c_1 = b_2, \ldots$ is **hard**
- **Public input**: $a_2 = 5, c_3 = 9, \ldots$ is **easy**

So all you need is a permutation check - a technique to prove **Copy Constraints**.

PLONK Gate Constraint



PLONK Gate Constraint

The quadratic equation of Gate Constraint in PLONK has the form:

PLONK Gate Equation

$$q_{L,i} \cdot a_i + q_{R,i} \cdot b_i + q_{O,i} \cdot c_i + q_{M,i} \cdot a_i \cdot b_i + q_{C,i} = 0$$

For each equation i in the system:

- The $q_{*,i}$ are coefficients in $\mathbb{F}_p$, which are publicly known and often referred to as **selectors**.
- The subscripts denote the role of each term:
  - **L**: Left input
  - **R**: Right input
  - **O**: Output
  - **M**: Multiplication term
  - **C**: Constant term

PLONK Gate Constraint

Addition Gate : $a + b = c$

$$(q_{L_i}, q_{R_i}, q_{O,i}, q_{M,i}, q_{C,i}) = (1, 1, -1, 0, 0)$$

Multiplication Gate : $a \cdot b = c$

$$(q_{L_i}, q_{R_i}, q_{O,i}, q_{M,i}, q_{C,i}) = (0, 0, -1, 1, 0)$$

Assign Gate : $a = \tau$

$$(q_{L_i}, q_{R_i}, q_{O,i}, q_{M,i}, q_{C,i}) = (1, 0, 0, 0, -\tau)$$

PLONK Copy Constraints

We will explore the details of copy constraints later.

For now, it is enough to represent them as simple equalities between wires:

$$a_0 = b_1, \quad c_2 = a_3, \quad \ldots$$

PLONK Arithmetization Example

Goal: Encode the statement $y = x^2 + 2$ using PLONK constraints.

PLONK Arithmetization Example

Goal: Encode the statement $y = x^2 + 2$ using PLONK constraints.

Step 1: Break down the computation into basic operations

- **Multiplication:** $x \cdot x = x^2$
- **Constant assignment:** $t = 2$
- **Addition:** $x^2 + t = y$

PLONK Arithmetization Example

Step 2: Encode the operations using PLONK constraints

Gate Constraints:

- **Multiplication gate:** $(a_1, b_1, c_1) = (x, x, x^2)$

$$0 \cdot a_1 + 0 \cdot b_1 + (-1) \cdot c_1 + 1 \cdot a_1 \cdot b_1 + 0 = 0$$

- **Constant gate:** $(a_2, b_2, c_2) = (2, 0, 0)$

$$1 \cdot a_2 + 0 \cdot b_2 + 0 \cdot c_2 + 0 \cdot a_2 \cdot b_2 + (-2) = 0$$

- **Addition gate:** $(a_3, b_3, c_3) = (x^2, 2, y)$

$$1 \cdot a_3 + 1 \cdot b_3 - 1 \cdot c_3 + 0 \cdot a_3 \cdot b_3 + 0 = 0$$

Copy Constraints:

- $a_1 = b_1$ (for $x^2 = x \cdot x$)
- $c_1 = a_3$ (connect output of multiplication to input of addition)
- $a_2 = b_3$ (connect constant 2 to input of addition)

PLONK Flow: Setup

The Prover and Verifier are given a PLONK instance.

The Prover holds a **witness** — an assignment of values to each a_i, b_i, and c_i such that:

- All **gate constraints** are satisfied
- All **copy constraints** are satisfied

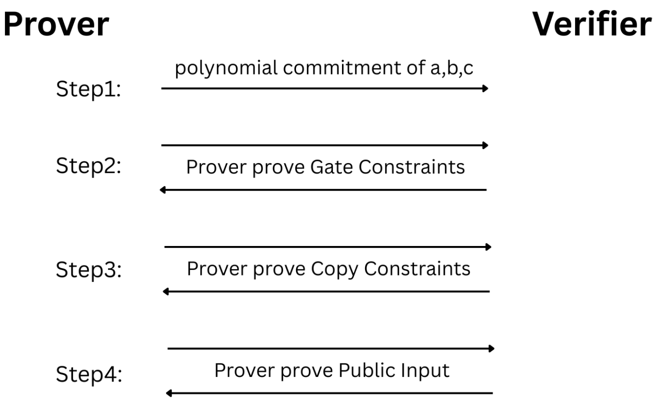
The goal: The Prover wants to convince the Verifier that a valid solution exists — **succinctly and without revealing the solution itself**.

PLONK Flow: Protocol Steps

The PLONK protocol proceeds as follows:

1. The Prover constructs and sends polynomial commitments corresponding to the wire assignments (a_i, b_i, c_i).
2. The Prover proves that these polynomials satisfy the **gate constraints** of the system.
3. The Prover proves that the same assignments also satisfy the **copy constraints**.
4. The Prover proves that the **Public Input** is match.

PLONK Flow


Prover
Verifier
Step1:
polynomial commitment of a,b,c
Step2:
Prover prove Gate Constraints
Step3:
Prover prove Copy Constraints
Step4:
Prover prove Public Input

Step 1: Commitment

We have n quadratic equations:

$$q_{L,1} \cdot a_1 + q_{R,1} \cdot b_1 + q_{O,1} \cdot c_1 + q_{M,1} \cdot a_1 \cdot b_1 + q_{C,1} = 0$$

$$q_{L,2} \cdot a_2 + q_{R,2} \cdot b_2 + q_{O,2} \cdot c_2 + q_{M,2} \cdot a_2 \cdot b_2 + q_{C,2} = 0$$

$$\vdots$$

$$q_{L,n} \cdot a_n + q_{R,n} \cdot b_n + q_{O,n} \cdot c_n + q_{M,n} \cdot a_n \cdot b_n + q_{C,n} = 0$$

Step 1: Commitments

To encode the witness, we move from field elements to polynomials.

Assume $n \mid (p-1)$ so that there exists a primitive n-th root of unity $\omega \in \mathbb{F}_p$:

$$\omega^n = 1 \quad \text{and} \quad \omega^i \neq 1 \text{ for } 1 \leq i < n$$

The Prover interpolates three polynomials:

$$\left.\begin{aligned} A(\omega^i) &= a_i \\ B(\omega^i) &= b_i \\ C(\omega^i) &= c_i \end{aligned}\right. \quad \text{for all } i = 1, \dots, n$$

These encode the witness values across all gates.

Step 1: Commitments

Commitment Protocol:

1. The Prover interpolates three polynomials: $A(X)$, $B(X)$, and $C(X)$ from the witness values.
2. The Prover sends $\texttt{Com}(A)$, $\texttt{Com}(B)$, $\texttt{Com}(C)$to the Verifier

These commitments hide the witness but bind the Prover to a specific assignment.

Step 2: Gate Constraint Check

Since the PLONK instance is public, both the Prover and Verifier can interpolate selector polynomials:

$$\begin{aligned} Q_L(\omega^i) &= q_{L,i} \\ Q_R(\omega^i) &= q_{R,i} \\ Q_O(\omega^i) &= q_{O,i} \quad \text{for } i = 1, \ldots, n \\ Q_M(\omega^i) &= q_{M,i} \\ Q_C(\omega^i) &= q_{C,i} \end{aligned}$$

These polynomials encode the gate constraints across the entire circuit.

Step 2: Gate Constraints Check

$$
\begin{aligned}
q_{L,1} \cdot a_1 + q_{R,1} \cdot b_1 + q_{O,1} \cdot c_1 + q_{M,1} \cdot a_1 \cdot b_1 + q_{C,1} &= 0 \\
q_{L,2} \cdot a_2 + q_{R,2} \cdot b_2 + q_{O,2} \cdot c_2 + q_{M,2} \cdot a_2 \cdot b_2 + q_{C,2} &= 0 \\
&\vdots \\
q_{L,n} \cdot a_n + q_{R,n} \cdot b_n + q_{O,n} \cdot c_n + q_{M,n} \cdot a_n \cdot b_n + q_{C,n} &= 0
\end{aligned}
$$

Q_L Q_R Q_O Q_M Q_C

Step 2: Gate Constraints Check

The Prover must convince the Verifier that the following polynomial identity holds:

$$Q_L(x)A(x) + Q_R(x)B(x) + Q_O(x)C(x) + Q_M(x)A(x)B(x) + Q_C(x) = 0$$

This equation must be satisfied at all n evaluation points:

$$x = \omega^1, \omega^2, \ldots, \omega^n$$

This ensures that all n gate constraints are satisfied simultaneously.

Vanishing Polynomial Proof Protocol

Goal: Prove that a polynomial $F(X)$ vanishes on a finite set $S \subseteq \mathbb{F}_p$:

$$F(x) = 0 \quad \forall x \in S$$

Vanishing Polynomial Proof Protocol

1. The Prover sends a commitment $\texttt{Com}(F)$ to the Verifier.
2. Both Prover and Verifier compute the vanishing polynomial over S:
$$Z(X) := \prod_{z \in S}(X - z)$$
3. The Prover computes:
$$H(X) := \frac{F(X)}{Z(X)}$$
and sends the commitment $\texttt{Com}(H)$ to the Verifier.
4. The Verifier chooses a random challenge $\lambda \in \mathbb{F}_p$ and requests an opening of both commitments at λ.
5. The Prover responds with $F(\lambda)$ and $H(\lambda)$, along with proofs that these values match their respective commitments.
6. The Verifier verify the proofs and checks that:
$$F(\lambda) = Z(\lambda) \cdot H(\lambda)$$

KZG Polynomial Commitment Scheme

The *Kate–Zaverucha–Goldberg (KZG)* Polynomial Commitment Scheme allows a Prover to convince a Verifier that a committed polynomial $P(X)$ evaluates to y at a point z, i.e., $P(z) = y$.

KZG Polynomial Commitment Scheme

The *Kate–Zaverucha–Goldberg (KZG)* Polynomial Commitment Scheme allows a Prover to convince a Verifier that a committed polynomial $P(X)$ evaluates to y at a point z, i.e., $P(z) = y$.

Protocol Overview:

- The Prover holds a secret polynomial $P(X)$, and the Verifier wants to learn its value at a point $z \in \mathbb{F}_p$.

KZG Polynomial Commitment Scheme

The *Kate–Zaverucha–Goldberg (KZG)* Polynomial Commitment Scheme allows a Prover to convince a Verifier that a committed polynomial $P(X)$ evaluates to y at a point z, i.e., $P(z) = y$.

Protocol Overview:

- The Prover holds a secret polynomial $P(X)$, and the Verifier wants to learn its value at a point $z \in \mathbb{F}_p$.
- The Prover sends a short commitment $\texttt{Com}(P)$ to the polynomial. This commitment binds the Prover to one specific polynomial and prevents them from changing their answer later.

KZG Polynomial Commitment Scheme

The *Kate–Zaverucha–Goldberg (KZG)* Polynomial Commitment Scheme allows a Prover to convince a Verifier that a committed polynomial $P(X)$ evaluates to y at a point z, i.e., $P(z) = y$.

Protocol Overview:

- The Prover holds a secret polynomial $P(X)$, and the Verifier wants to learn its value at a point $z \in \mathbb{F}_p$.
- The Prover sends a short commitment $\mathtt{Com}(P)$ to the polynomial. This commitment binds the Prover to one specific polynomial and prevents them from changing their answer later.
- The Verifier then queries the Prover for $P(z)$.

KZG Polynomial Commitment Scheme

The *Kate–Zaverucha–Goldberg (KZG)* Polynomial Commitment Scheme allows a Prover to convince a Verifier that a committed polynomial $P(X)$ evaluates to y at a point z, i.e., $P(z) = y$.

Protocol Overview:

- The Prover holds a secret polynomial $P(X)$, and the Verifier wants to learn its value at a point $z \in \mathbb{F}_p$.
- The Prover sends a short commitment $\texttt{Com}(P)$ to the polynomial. This commitment binds the Prover to one specific polynomial and prevents them from changing their answer later.
- The Verifier then queries the Prover for $P(z)$.
- The Prover responds with the claimed value $y = P(z)$, along with a short "proof" that convinces the Verifier that y is indeed the correct evaluation — without revealing $P(X)$.

Step 3: Proving Copy Constraints

Copy constraints ensure that values assigned to the same wire across different gates remain consistent.

To illustrate this, consider a circuit with 4 gates and the following copy constraints:

$$a_1 = a_4 = c_4 \quad \text{and} \quad b_2 = c_1$$

To prove that these constraints are satisfied, we use the technique named **permutation check**.

Permutation Check : Setup

Problem: Suppose we have polynomials P, Q with encode into two vectors of values

$$\vec{p} = \langle P(\omega^1), P(\omega^2), ..., P(\omega^n) \rangle$$

$$\vec{q} = \langle Q(\omega^1), Q(\omega^2), ..., Q(\omega^n) \rangle$$

Verify $\vec{p}$ is permutation of $\vec{q}$.

Permutation Check : Core Idea

Suppose $\vec{p}$ is a permutation of $\vec{q}$. Then the following identity holds:

$$(X + P(\omega^1))(X + P(\omega^2))...(X + P(\omega^n))$$
$$= (X + Q(\omega^1))(X + Q(\omega^2))...(X + Q(\omega^n))$$

$$\Leftrightarrow \prod_{i=1}^{n}(X + P(\omega^i)) = \prod_{i=1}^{n}(X + Q(\omega^i)) \qquad (*)$$

This is because both sides are degree-n polynomials in a single variable X, and they share the same set of roots (up to reordering).

Permutation Check : Randomness Argument

Lemma (Schwartz–Zippel Lemma)

If two degree-d polynomials over $\mathbb{F}_p$ are not equal, they can agree on at most d points in $\mathbb{F}_p$.

Therefore, if Equation $(*)$ from the previous slide does **not** hold, the probability that it still evaluates equally at a randomly chosen point $\lambda \in \mathbb{F}_p$ is:

$$\Pr[F(\lambda) = G(\lambda)] \leq \frac{n}{|\mathbb{F}_p|} \ll 1 \quad \text{(for large } p\text{, e.g., 256-bit field)}$$

This lets the verifier check polynomial equality using just one random evaluation.

Permutation Check : Accumulator Polynomial

We can use **Acummulator Polynomial** to let verifier check this equation at a random evaluation.

Create Accumulator Polynomials:

- Verifier picks a random challenge $\lambda \in \mathbb{F}_p$ and sends to Prover
- Prover then interpolate the **acummulator polynomial** F_P such that:

$$\begin{aligned}
F_P(\omega^1) &= \lambda + P(\omega^1) \\
F_P(\omega^2) &= (\lambda + P(\omega^1))(\lambda + P(\omega^2)) \\
&\dots \\
F_P(\omega^n) &= (\lambda + P(\omega^1))(\lambda + P(\omega^2))\dots(\lambda + P(\omega^n))
\end{aligned}$$

- Prover does similar for F_Q

Permutation Check — Protocol (Setup)

Suppose the Prover has committed to two polynomials via:

$$\texttt{Com}(P), \quad \texttt{Com}(Q)$$

Protocol Steps:

1. The Verifier sends a random challenge $\lambda \in \mathbb{F}_p$ to the Prover.
2. The Prover constructs two new polynomials:

$$F_P(X) := \prod_{i=1}^{n}(\lambda + P(\omega^i)), \quad F_Q(X) := \prod_{i=1}^{n}(\lambda + Q(\omega^i))$$

3. The Prover sends commitments $\texttt{Com}(F_P)$ and $\texttt{Com}(F_Q)$ to the Verifier.

Permutation Check : Verifier Checks

Using the **Vanishing Polynomial Proof Protocol** (described earlier), the Prover and Verifier now check the following statements:

1. $F_P(X) - (\lambda + P(X))$ vanishes at $X = \omega$
2. $F_P(\omega X) - (\lambda + P(X))F_P(X)$ vanishes at $X \in \{\omega, ..., \omega^{n-1}\}$
3. $F_Q(X) - (\lambda + Q(X))$ vanishes at $X = \omega$
4. $F_Q(\omega X) - (\lambda + Q(X))F_Q(X)$ vanishes at $X \in \{\omega, ..., \omega^{n-1}\}$
5. $F_P(X) - F_Q(X)$ vanishes at $X = 1$

Copy Constraint Check : Init Problem

Consider a circuit with 4 gates and the following copy constraints:

$$a_1 = a_4 = c_4 \quad \text{and} \quad b_2 = c_1$$

This means the assignments before and after applying the permutation (enforced by the copy constraints) must match:

$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \\ a_4 & b_4 & c_4 \end{bmatrix} = \begin{bmatrix} {\color{red}a_4} & b_1 & {\color{blue}b_2} \\ a_2 & {\color{blue}c_1} & c_2 \\ a_3 & b_3 & c_3 \\ {\color{red}c_4} & b_4 & {\color{red}a_1} \end{bmatrix} \tag{1}$$

Copy Constraint Check : Tag

To transform the equality in (1) into a permutation check, we **tag** each entry in the matrix with a unique offset using powers of η, ω, and a random scalar μ. Specifically:

Each element at row k and column j is transformed as:

$$\text{entry} \quad \longrightarrow \quad \text{entry} + \eta^j \omega^k \mu$$

for $j = 0, 1, 2$ and $k = 1, ..., n$, where:

- $\omega \in \mathbb{F}_p$ is an n-th root of unity
- $\eta \in \mathbb{F}_p$ is a random value such that η^2 is not a power of ω
- $\mu \in \mathbb{F}_p$ is a random challenge

Copy Constraint Check : Tag

This transforms both matrices in equation (1) into:

$$\begin{bmatrix} a_1 + \omega^1\mu & b_1 + \eta\omega^1\mu & c_1 + \eta^2\omega^1\mu \\ a_2 + \omega^2\mu & b_2 + \eta\omega^2\mu & c_2 + \eta^2\omega^2\mu \\ a_3 + \omega^3\mu & b_3 + \eta\omega^3\mu & c_3 + \eta^2\omega^3\mu \\ a_4 + \omega^4\mu & b_4 + \eta\omega^4\mu & c_4 + \eta^2\omega^4\mu \end{bmatrix} = \begin{bmatrix} {\color{red}a_4} + \omega^1\mu & b_1 + \eta\omega^1\mu & {\color{blue}b_2} + \eta^2\omega^1\mu \\ a_2 + \omega^2\mu & {\color{blue}c_1} + \eta\omega^2\mu & c_2 + \eta^2\omega^2\mu \\ a_3 + \omega^3\mu & b_3 + \eta\omega^3\mu & c_3 + \eta^2\omega^3\mu \\ {\color{red}c_4} + \omega^4\mu & b_4 + \eta\omega^4\mu & {\color{red}a_1} + \eta^2\omega^4\mu \end{bmatrix} \tag{2}$$

This transformation ensures that only if the copy constraints hold, both tagged matrices are a permutation of each other.

Copy Constraint Check : Change to Permutation Check

From Equation (2) we have:

$$\begin{bmatrix} a_1 + \omega^1\mu & b_1 + \eta\omega^1\mu & c_1 + \eta^2\omega^1\mu \\ a_2 + \omega^2\mu & b_2 + \eta\omega^2\mu & c_2 + \eta^2\omega^2\mu \\ a_3 + \omega^3\mu & b_3 + \eta\omega^3\mu & c_3 + \eta^2\omega^3\mu \\ a_4 + \omega^4\mu & b_4 + \eta\omega^4\mu & c_4 + \eta^2\omega^4\mu \end{bmatrix}$$

is a permutation of

$$\begin{bmatrix} a_1 + {\color{red}\eta^2\omega^4\mu} & b_1 + \eta\omega^1\mu & c_1 + {\color{blue}\eta\omega^2\mu} \\ a_2 + \omega^2\mu & b_2 + {\color{blue}\eta^2\omega^1\mu} & c_2 + \eta^2\omega^2\mu \\ a_3 + \omega^3\mu & b_3 + \eta\omega^3\mu & c_3 + \eta^2\omega^3\mu \\ a_4 + {\color{red}\omega^1\mu} & b_4 + \eta\omega^4\mu & c_4 + {\color{red}\omega^4\mu} \end{bmatrix} \tag{3}$$

Copy Constraint Check : Permutation Check

Because permutation is publicly known, then both parties can interpolate the 3 polynomial $\sigma_a, \sigma_b, \sigma_c$:

- $\sigma_a(\omega^1) = \eta^2\omega^4, \quad \sigma_b(\omega^1) = \eta\omega^1, \quad \sigma_c(\omega^1) = \eta\omega^2$
- $\sigma_a(\omega^2) = \omega^2, \quad \sigma_b(\omega^2) = \eta^2\omega^1, \quad \sigma_c(\omega^2) = \eta^2\omega^2$
- $\sigma_a(\omega^3) = \omega^3, \quad \sigma_b(\omega^3) = \eta\omega^3, \quad \sigma_c(\omega^3) = \eta^2\omega^3$
- $\sigma_a(\omega^4) = \omega^1, \quad \sigma_b(\omega^4) = \eta\omega^4, \quad \sigma_c(\omega^4) = \omega^4$

Copy Constraint Check : Permutation Check

Prover defining accumulator polynomials after receive the random challenge λ from Verifier.

- $F_a(\omega^k) = \prod_{i \leq k}(a_i + \omega^i \mu + \lambda)$
- $F_b(\omega^k) = \prod_{i \leq k}(b_i + \eta\omega^i \mu + \lambda)$
- $F_c(\omega^k) = \prod_{i \leq k}(c_i + \eta^2\omega^i \mu + \lambda)$
- $F'_a(\omega^k) = \prod_{i \leq k}(a_i + \sigma_a(\omega^i)\mu + \lambda)$
- $F'_b(\omega^k) = \prod_{i \leq k}(b_i + \sigma_b(\omega^i)\mu + \lambda)$
- $F'_c(\omega^k) = \prod_{i \leq k}(c_i + \sigma_c(\omega^i)\mu + \lambda)$

And similar to the **Permutation Check** we have:

6 initialization conditions :

- $F_a(\omega^1) = A(\omega^1) + \omega^1\mu + \lambda$
- $F_b(\omega^1) = B(\omega^1) + \eta\omega^1\mu + \lambda$
- $F_c(\omega^1) = C(\omega^1) + \eta^2\omega^1\mu + \lambda$
- $F'_a(\omega^1) = A(\omega^1) + \sigma_a(\omega^1)\mu + \lambda$
- $F'_b(\omega^1) = B(\omega^1) + \sigma_b(\omega^1)\mu + \lambda$
- $F'_c(\omega^1) = C(\omega^1) + \sigma_c(\omega^1)\mu + \lambda$

Copy Constraint Check : Permutation Check

6 accumulation conditions :

- $F_a(\omega X) = F_a(X)(A(\omega X) + X\mu + \lambda)$
- $F_b(\omega X) = F_b(X)(B(\omega X) + \eta X\mu + \lambda)$
- $F_c(\omega X) = F_c(X)(C(\omega X) + \eta^2 X\mu + \lambda)$
- $F'_a(\omega X) = F'_a(X)(A(\omega X) + \sigma_a(X)\mu + \lambda)$
- $F'_b(\omega X) = F'_b(X)(B(\omega X) + \sigma_b(X)\mu + \lambda)$
- $F'_c(\omega X) = F'_c(X)(C(\omega X) + \sigma_c(X)\mu + \lambda)$

Copy Constraint Check : Permutation Check

And one product condition:

$$F_a(1)F_b(1)F_c(1) = F'_a(1)F'_b(1)F'_c(1)$$

We use **Vanishing Polynomial Proof Protocol** described above to verify these conditions.

Step 4: Handling Public and Private Inputs

Let $S \subseteq \{1, 2, \ldots, n\}$ be the set of indices corresponding to **public** inputs (i.e., specific a_i values that the Prover wants to reveal).

The Prover constructs a new polynomial $A^{\text{public}}(X)$ such that:

$$A^{\text{public}}(\omega^i) = \begin{cases} a_i & \text{if } i \in S \\ 0 & \text{otherwise} \end{cases}$$

Then, the Prover uses the **Vanishing Polynomial Proof Protocol** to prove:

$$A^{\text{public}}(X) - A(X) \text{ vanishes on } S$$

This ensures that the committed polynomial $A(X)$ agrees with the claimed public values at exactly the specified positions, without revealing the private ones.

Fiat-Shamir Transform

Fiat-Shamir is a cryptographic technique that transforms an interactive protocol into a non-interactive one.

Originally used for identification schemes, it plays a central role in SNARKs.

Core Idea:

- Replace the Verifier's random challenge with a deterministic value.
- The challenge is computed by hashing the transcript (e.g., using SHA256).
- This makes the protocol non-interactive while maintaining soundness under the *Random Oracle Model (ROM)*.

From Interactive to Non-Interactive

Interactive Protocol:

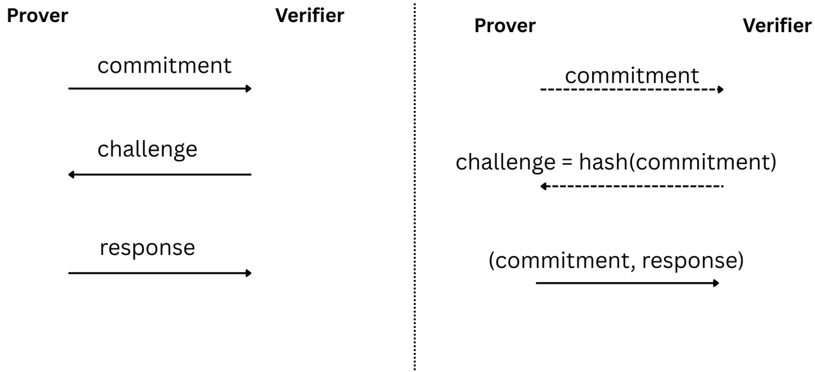
1. Prover sends a commitment to Verifier.
2. Verifier responds with a random challenge.
3. Prover replies with a response based on the challenge.

Fiat-Shamir:

1. Prover computes challenge $=$ Hash(commitment)
2. Uses this challenge to compute response.
3. Sends only (commitment, response) to Verifier.

No back-and-forth needed!

Fiat-Shamir Transform



Why Fiat-Shamir in SNARKs?

In SNARKs (e.g., PLONK, Groth16), many interactive steps involve the Verifier choosing random challenges.

To make the SNARK **succinct and non-interactive**, Fiat-Shamir is used to:

- Derive challenges deterministically from prior messages.
- Ensure the proof can be verified by anyone without interaction.
- Remove the need for trusted communication rounds.

Fiat-Shamir is what makes most zk-SNARKs practical!

KZG Polynomial Commitment Scheme

Structured Reference String (SRS)

$$g^1, g^\tau, g^{\tau^2}, \ldots, g^{\tau^d}$$

where $\tau \in \mathbb{F}_p$ is a secret scalar and g is a generator of an elliptic curve group.

Polynomial

$$p(x) = \sum_{i=0}^{d} a_i x^i$$

Commitment

$$\mathrm{Com}(p) = \prod_{i=0}^{d} (g^{\tau^i})^{a_i} = g^{\sum_{i=0}^{d} a_i \tau^i} = g^{p(\tau)}$$

KZG: Opening and Verification

Goal: Prove that a committed polynomial $p(x)$ evaluates to $p(z)$ at some point $z \in \mathbb{F}_p$.
Define:

$$h(x) := \frac{p(x) - p(z)}{x - z} \quad \Rightarrow \quad p(x) = h(x)(x - z) + p(z)$$

Opening Proof

$$\pi := g^{h(\tau)}$$

Verification Equation

$$e(\mathrm{Com}(p) - g^{p(z)}, g) \stackrel{?}{=} e(\pi, g^{\tau - z})$$

Why this works:

$$\begin{aligned} e(g^{p(\tau)-p(z)}, g) &= e(g^{h(\tau)(\tau-z)}, g) \\ &= e(g^{h(\tau)}, g^{\tau-z}) = e(\pi, g^{\tau-z}) \end{aligned}$$

Elliptic Curves

An elliptic curve over $\mathbb{F}_p$ is defined as:

$$E : y^2 = x^3 + ax + b$$

Properties:

- Points form a group with addition.
- Scalar multiplication: $g, 2g, 3g, \ldots$
- Secure due to hard discrete log problem.

Used in KZG: Commit to polynomials as group elements like $g^{p(\tau)}$.

Pairing Operation

A pairing is a map:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$$

Key properties:

- Bilinear: $e(g^a, g^b) = e(g, g)^{ab}$
- Non-degenerate and efficient

Used in KZG to verify:

$$e(g^{p(\tau)-p(z)}, g) = e(g^{h(\tau)}, g^{\tau-z})$$