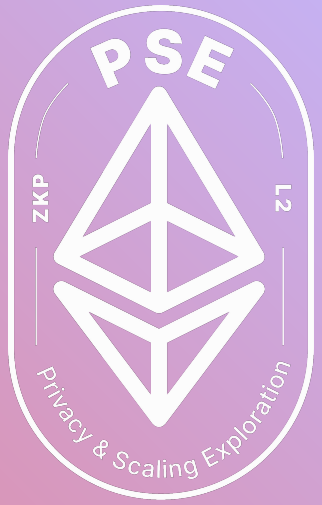


ZK school

About PLONK & Halo2

DoHoon Kim

PSE

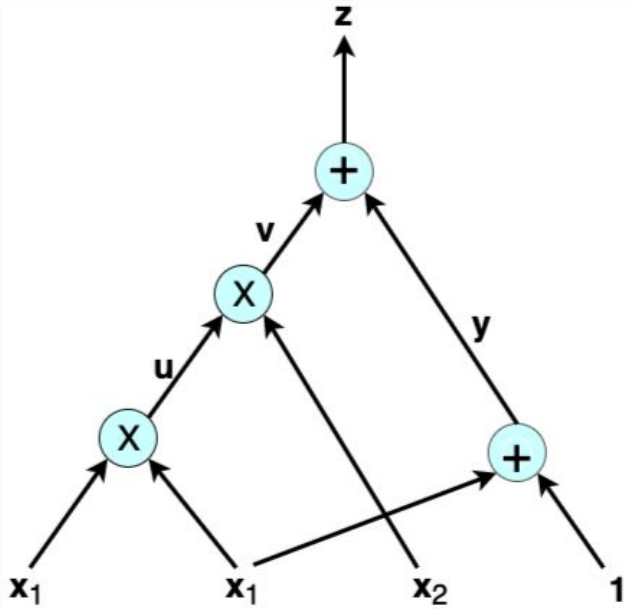
What is PLONK...?

1. Introduction

Intro

```
R1CS 형태 a*b-c == 0
```

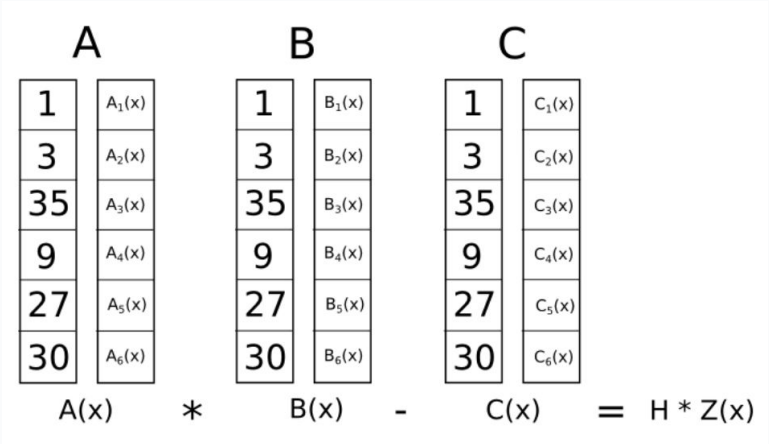
- A * B - C == 0 is what we call Rank-1 Constraint System(R1CS)
- **Arithmetization** of circuit by R1CS
- Left, Right, Output **wires**
- **Constraints** between wires
- Simple, and strong



PSE

Quadratic Arithmetic Program (a.k.a QAP)

- Arithmetic circuit → R1CS → QAP → ... → zk-SNARK!
- **QAP** is A(x) * B(x) - C(x) == 0
- Wire values are **interpolated** into polynomials
- In this sense, **witness** of the circuit == **satisfying assignment** of QAP



PSE

Groth16

- How to convert QAP into zk-SNARK?
- For Groth16, they use **pairing** based cryptography, which is very powerful
- Groth16 requires **Phase 2 trusted setup**

2. PLONK

PLONK [GWC19]

- **Universal** trusted setup, no circuit specific trusted setup
- Based on **Polynomial Commitment Scheme**
- Enables **modular** SNARK
- First introduce vanilla PLONK and then Turbo PLONK with custom gates

PLONK arithmetization

- PLONK circuit has gates and wires
- Each gate has **gate constraint** between wires
- Gates are wired each other, introducing **copy constraints**

Gate constraint

- In vanila PLONK, we have addition and multiplication gates
- We can toggle between addition and multiplication gates with **selectors**
- i-th gate constraint is expressed as follows

$$(\mathbf{q_L})_i \cdot \mathbf{x_{a_i}} + (\mathbf{q_R})_i \cdot \mathbf{x_{b_i}} + (\mathbf{q_O})_i \cdot \mathbf{x_{c_i}} + (\mathbf{q_M})_i \cdot (\mathbf{x_{a_i}} \mathbf{x_{b_i}}) + (\mathbf{q_C})_i = 0.$$

Copy constraint

- We need to **connect** the gates with wires
- This is where **copy** constraint comes in
- We should enforce some wire values to be same, as they are copied each other
- We use **permutation argument**

Grand product argument

- How can we show 2 sets are equal?
- Simple solution: multiply the two sets respectively and compare the results

$$\prod_{i\in[n]} a_i \stackrel{?}{=} \prod_{i\in[n]} b_i,$$

- For soundness, we need a **randomness**

$$\prod_{i\in[n]} (a_i + \gamma) \stackrel{?}{=} \prod_{i\in[n]} (b_i + \gamma)$$

- This check can be achieved with great efficiency on **multiplicative subgroup**

Permutation argument

- How can we show 2 sets are equal under the certain permutation?
- This is where **permutation argument** comes in
- Combine the set elements with randomness and apply grand product argument
- This technique will be used in **lookup argument** later

Again copy constraint

- Think of wire values as sets
- Swap the elements that have the same value → this will generate specific **permutation!**
- We can use permutation argument to show that certain set of wire values are the same

PLONK arithmetization

- Keep **Gate constraints**, **Copy constraints** in mind
- Easiest way to understand PLONK is drawing **matrix**

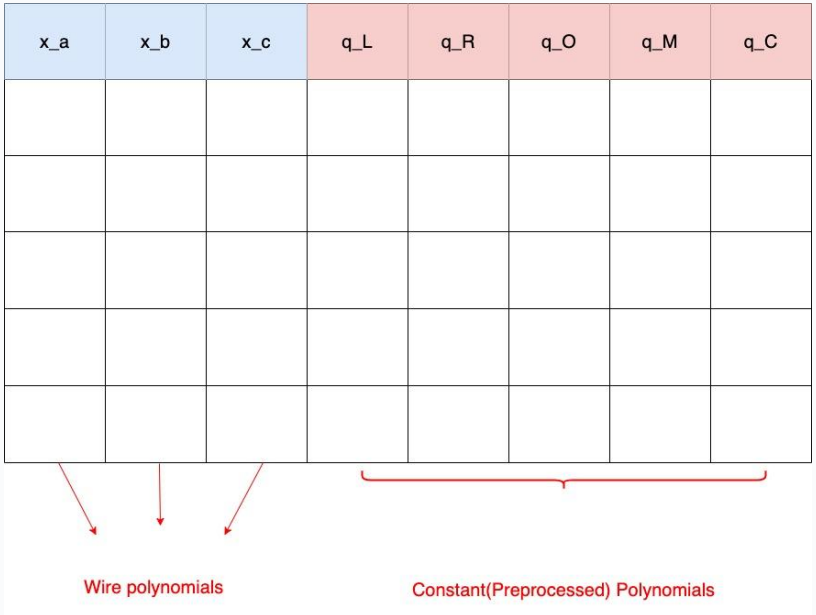
x_a	x_b	x_c	q_L	q_R	q_O	q_M	q_C

$$(\mathbf{q_L})_i \cdot \mathbf{x_{a_i}} + (\mathbf{q_R})_i \cdot \mathbf{x_{b_i}} + (\mathbf{q_O})_i \cdot \mathbf{x_{c_i}} + (\mathbf{q_M})_i \cdot (\mathbf{x_{a_i}} \mathbf{x_{b_i}}) + (\mathbf{q_C})_i = 0.$$

PSE

PLONK arithmetization

- Each column corresponds to a **polynomial**
- Some columns are **preprocessed**
- Some columns should be provided by **prover**



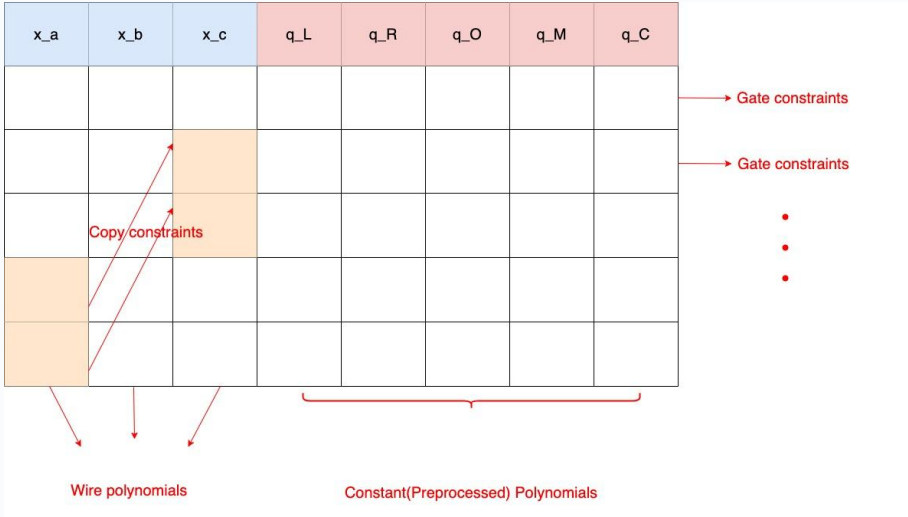
PSE

PLONK arithmetization

- **Prover** fills in wire values that satisfies constraints
- **Proof** is consisted of each column polynomial
- **Verifier** checks the constraints by checking polynomial identities
- Is this **succinct** or **zero-knowledge**?

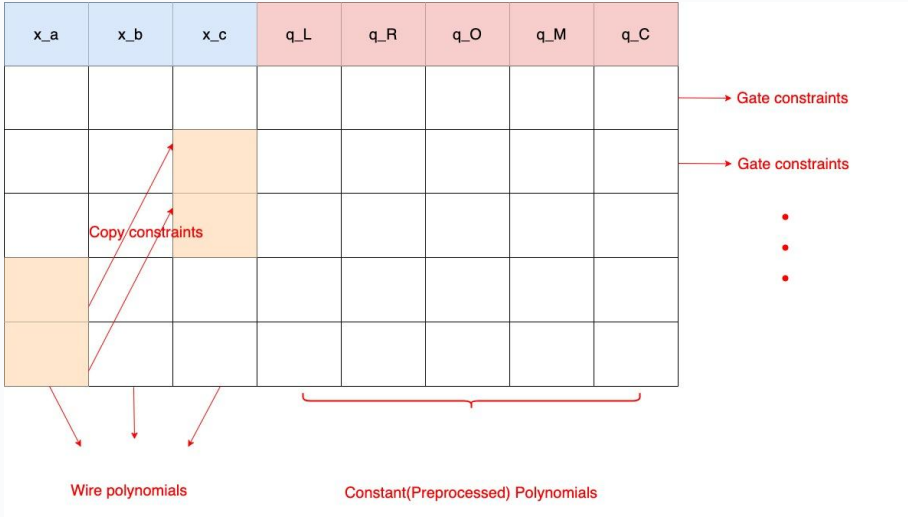
프로버가 각 로우에 값을 넣는다

베리파이어는 조건이 맞는지 검증



PLONK arithmetization

- **Prover** fills in wire values that satisfies constraints
- **Proof** is consisted of commitments of each column polynomial
- **Verifier** checks the constraints by checking polynomial identities on certain point



PSE

Polynomial commitment scheme

- How can verifier **convinced** prover provided the evaluation on polynomial
- Prover can **commit** to the polynomial which is **binding** and **hiding**
- Prover can **open** the commitment in certain point after given commitment & **opening proof**
- Uses **elliptic curve pairing**

KZG10

- Vanilla PLONK uses **KZG10** as polynomial commitment scheme
- It's based on basic mathematical fact

$$q(X) = \frac{p(X) - y}{X - z}$$

- For polynomial **p(X)** if **p(z) = y**, then **(X - z) | (p(X) - y)**
- Prover can calculate **quotient** polynomial
- Prover provides commitment of quotient polynomial

KZG10

- Each commitment and proof are elliptic curve **points**
- Prover provides elliptic curve points as a proof
- Prover commits to **p(X)**
- Prover provides commitment **q(X)** for an opening proof of **p(z) = y**
- Verifier can check the polynomial identity by elliptic curve **pairing**

PSE

KZG10

- Prepare 2 **pairing-friendly** elliptic curves
- Pairing enables **multiplication** between 2 committed values without knowing them
- Verifier checks the following pairing equation

$$e([p(s)]_1 - [y]_1, [1]_2) = e([q(s)]_1, [s]_2 - [z]_2)$$

PLONK + ?

- PLONK arithmetization can be used with another polynomial commitment schemes
- PLONK + **I**nner **P**roduct **A**rgument(**IPA**) ⇒ zcash/halo2
- PLONK + KZG ⇒ pse/halo2
- PLONK + **F**ast **R**eed-Solomon **I**OPP(**FRI**) ⇒ polygon/plonky2
- Swap out backends, replace it with another scheme

Further study

- Recommend reading PLONK paper (**[GWC19]**)
- Turbo-PLONK with **custom gates**
- Ultra-PLONK with lookup arguments
- Dive into **Halo2** argument system

Q & A