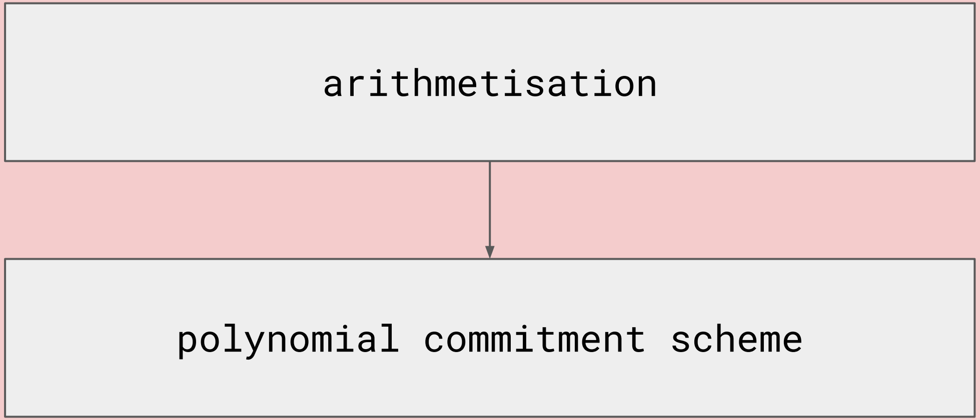
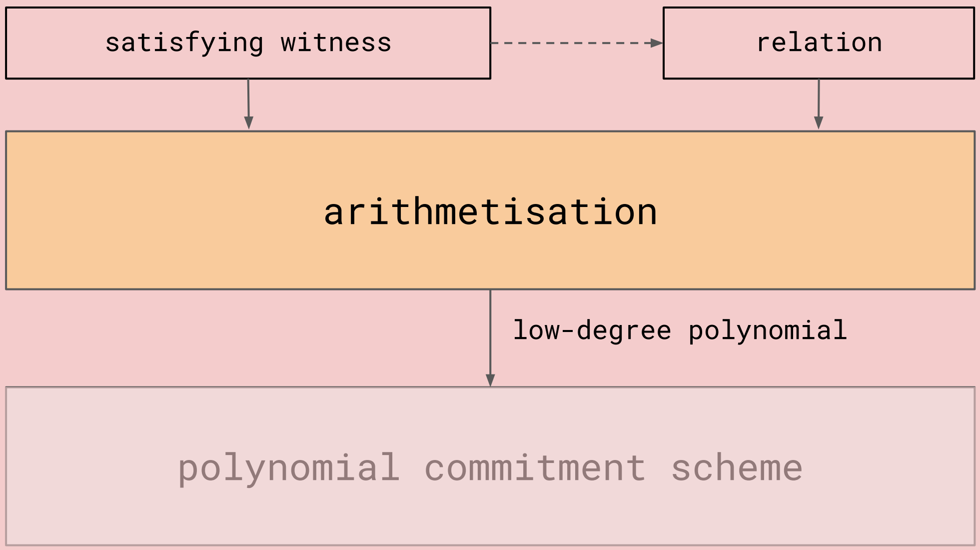
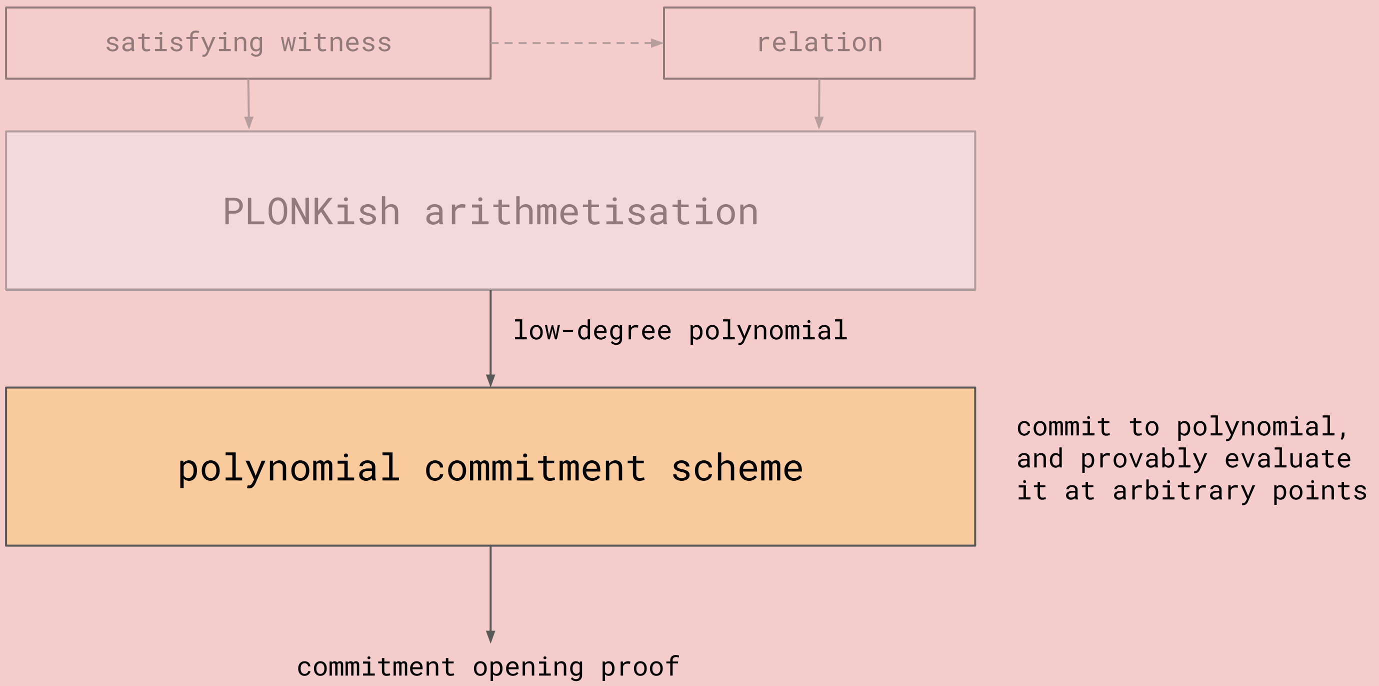
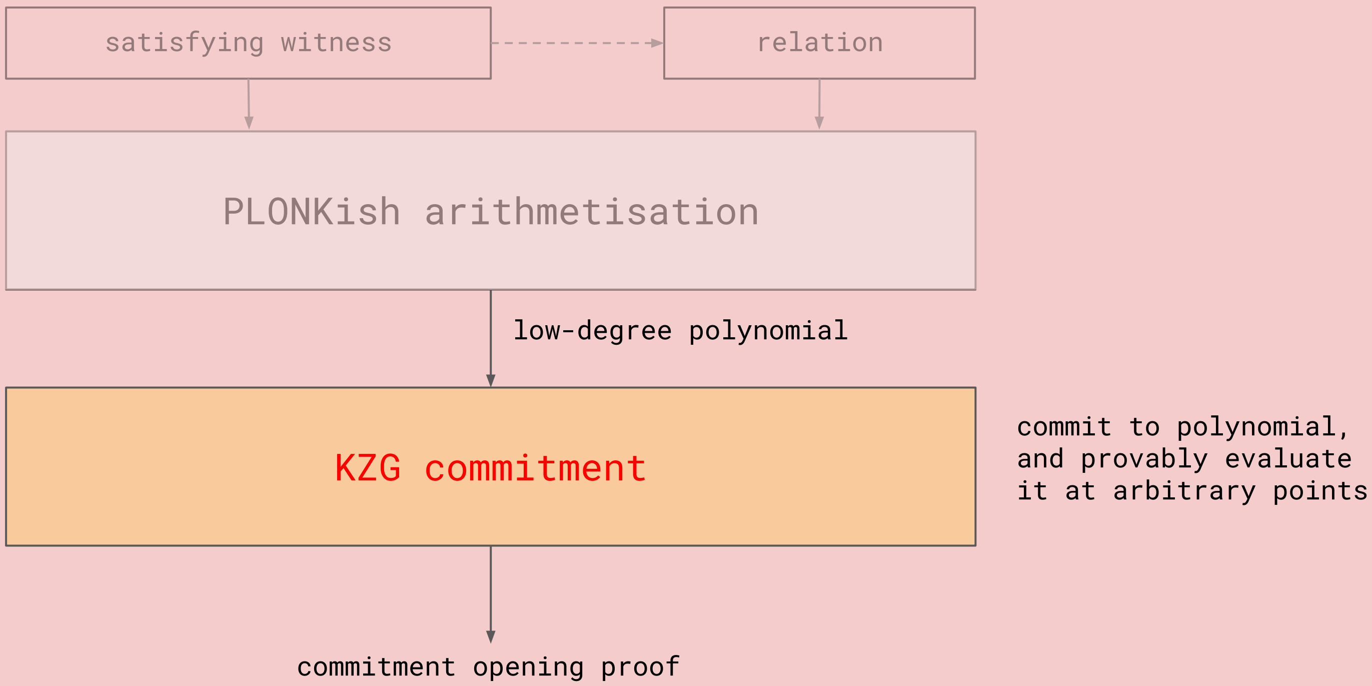
intro to PlonK

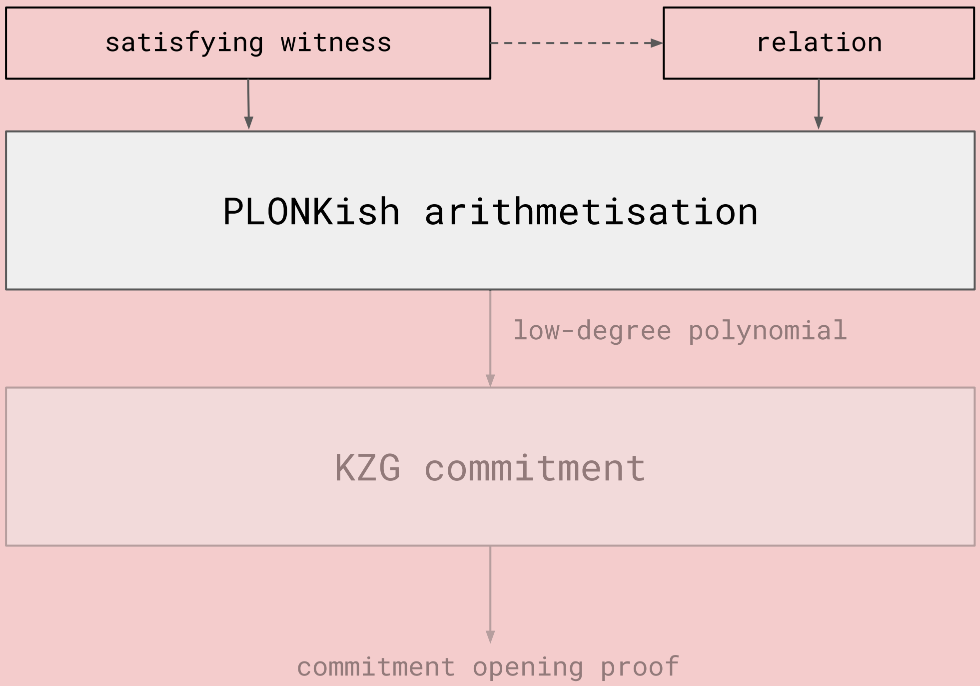
PSE ZK Workshop
15 Feb 2025


arithmetisation
polynomial commitment scheme

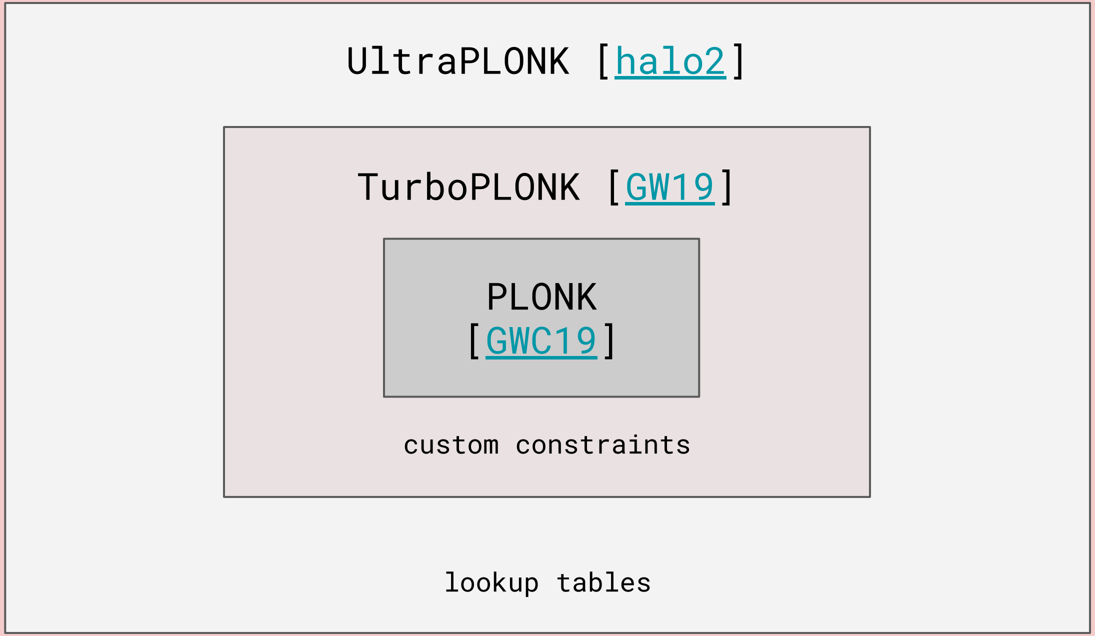

satisfying witness
relation
arithmetisation
low-degree polynomial
polynomial commitment scheme


satisfying witness
relation
PLONKish arithmetisation
low-degree polynomial
polynomial commitment scheme
commit to polynomial,
and provably evaluate
it at arbitrary points
commitment opening proof

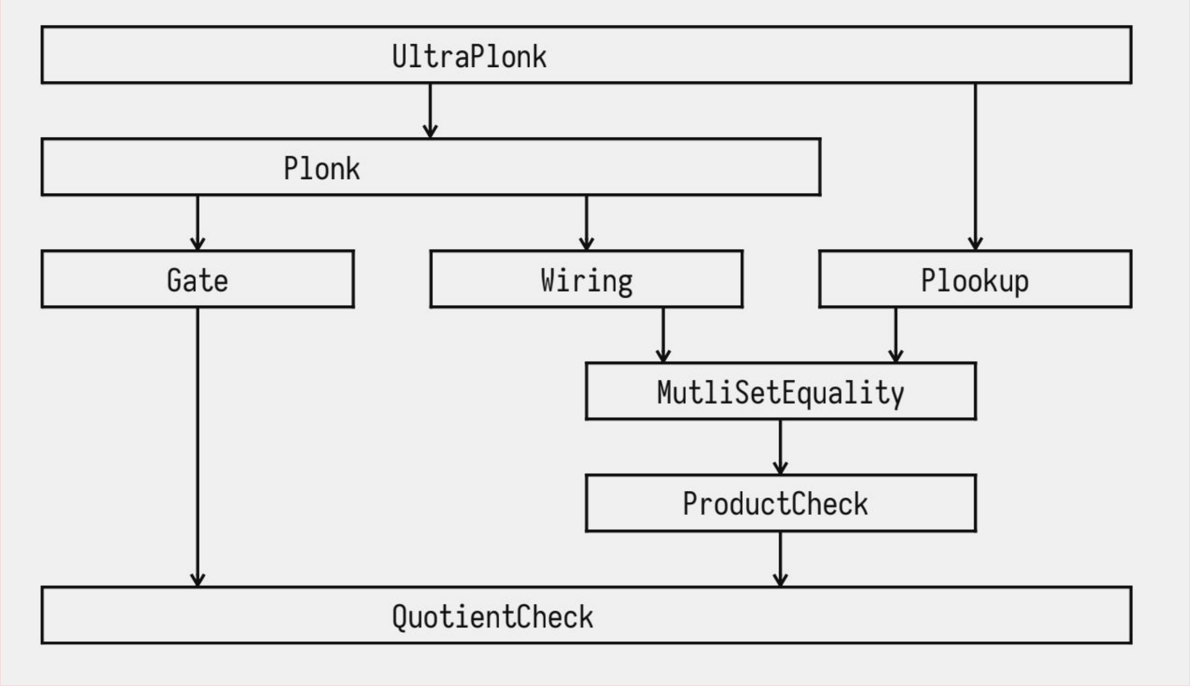

satisfying witness
relation
PLONKish arithmetisation
low-degree polynomial
KZG commitment
commit to polynomial,
and provably evaluate
it at arbitrary points
commitment opening proof


satisfying witness
relation
PLONKish arithmetisation
low-degree polynomial
KZG commitment
commitment opening proof

PLONKish arithmetisation (univariate)

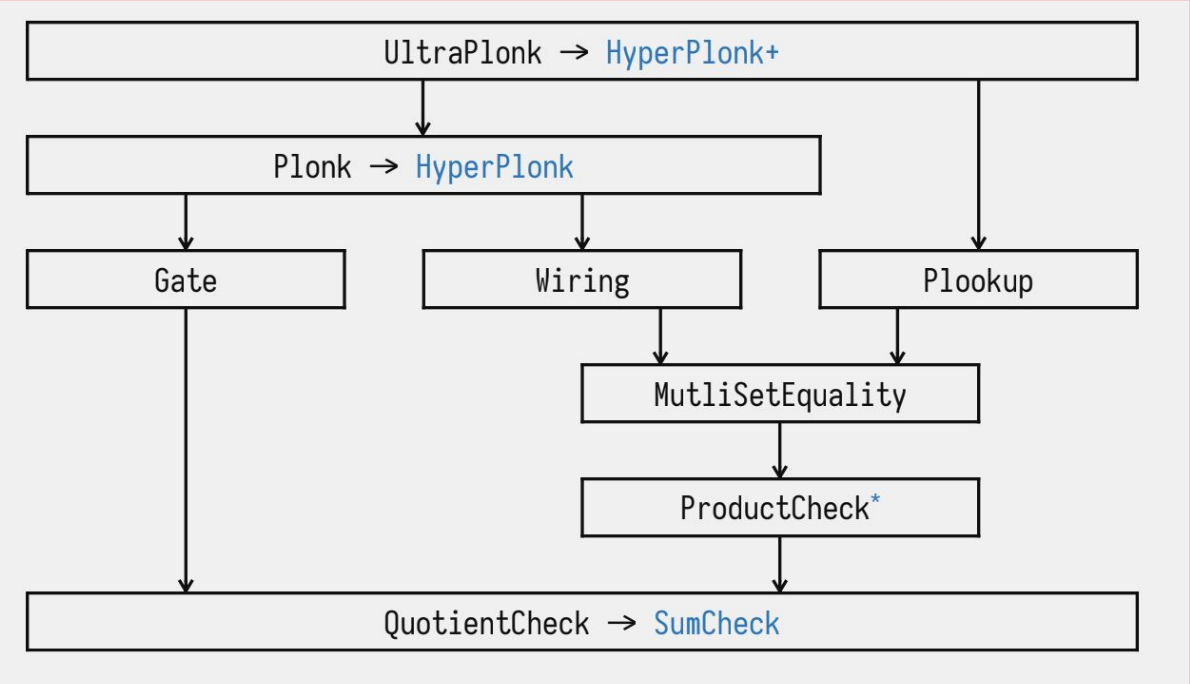


PLONKish arithmetisation



(slide from han0110)

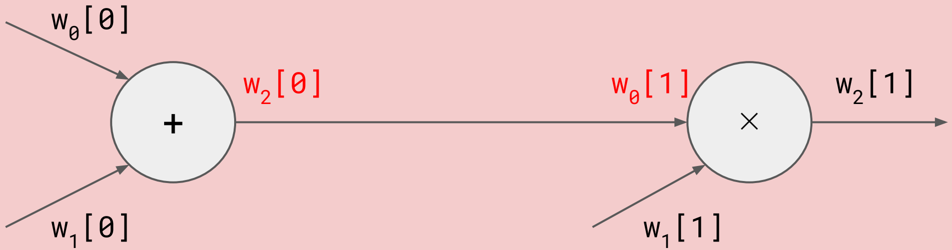
PLONKish arithmetisation



(slide from han0110)

PLONKish arithmetisation (vanilla)

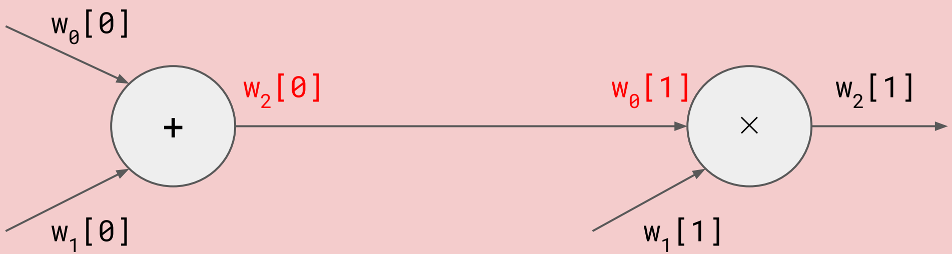
PLONK
[GWC19]



- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

PLONKish arithmetisation (vanilla)

PLONK
[GWC19]

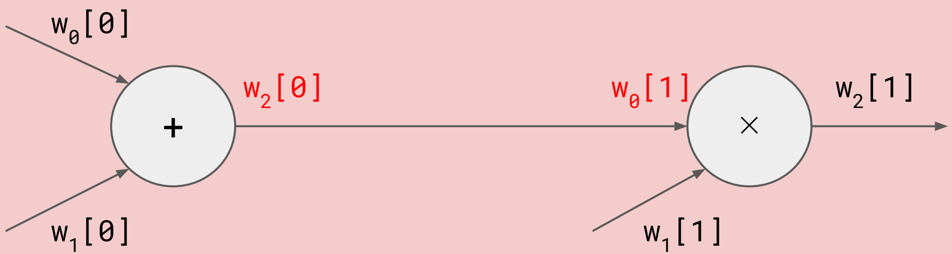


- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

"local" consistency check: are all gate equations satisfied?

PLONKish arithmetisation (vanilla)

PLONK
[GWC19]



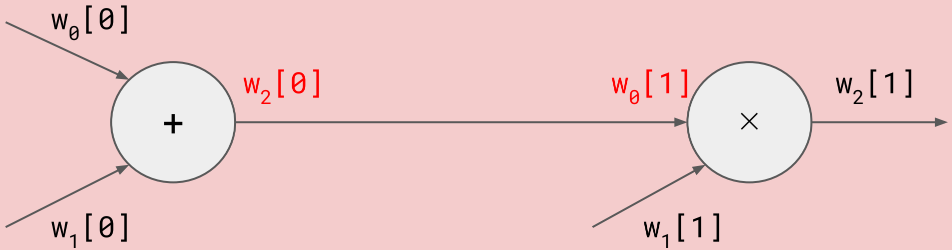
- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

"local" consistency check: are all gate equations satisfied?

$$q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$$

PLONKish arithmetisation (vanilla)

PLONK
[GWC19]



- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

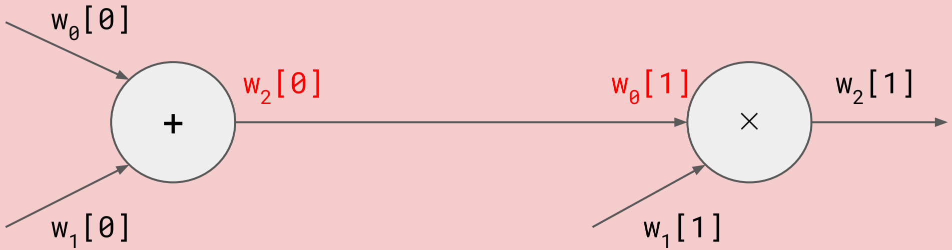
"local" consistency check: are all gate equations satisfied?

$$\boxed{q_L}\cdot x_a + \boxed{q_R}\cdot x_b + \boxed{q_O}\cdot x_c + \boxed{q_M}\cdot (x_a x_b) = 0$$

preprocessed selector polynomials

PLONKish arithmetisation (vanilla)

PLONK
[GWC19]



- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

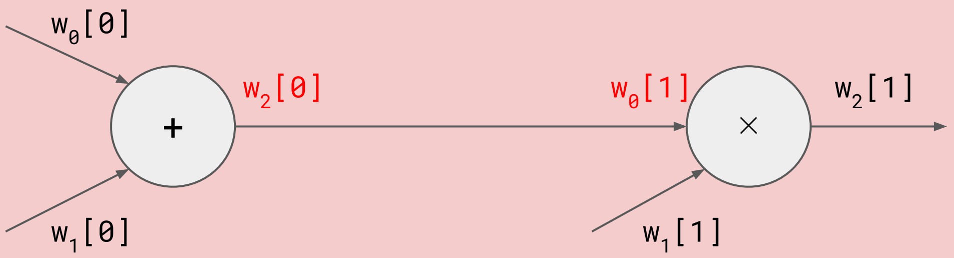
"local" consistency check: are all gate equations satisfied?

$$q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$$

$$\text{add: } 1 \cdot x_a + 1 \cdot x_b + (-1) \cdot x_c + 0 \cdot (x_a x_b) = 0$$

PLONKish arithmetisation (vanilla)

PLONK
[GWC19]



- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

"local" consistency check: are all gate equations satisfied?

$$q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$$

$$\text{add: } 1 \cdot x_a + 1 \cdot x_b + (-1) \cdot x_c + 0 \cdot (x_a x_b) = 0$$

$$\text{mul: } 0 \cdot x_a + 0 \cdot x_b + (-1) \cdot x_c + 1 \cdot (x_a x_b) = 0$$

PLONKish arithmetisation (custom gates)

TurboPLONK
[GW19]

vanilla PLONK gate: $q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$

custom gates (arbitrary linear combinations):

$$\underbrace{q_{add} \cdot (a_0 + a_1 - a_2)}_{\text{add gate}} = 0$$

PLONKish arithmetisation (custom gates)

TurboPLONK
[GW19]

vanilla PLONK gate: $q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$

custom gates (arbitrary linear combinations):

$$\underbrace{q_{add} \cdot (a_0 + a_1 - a_2)}_{\text{add gate}} + \underbrace{q_{mul} \cdot (a_0 \cdot a_1 - a_2)}_{\text{mul gate}} = 0$$

PLONKish arithmetisation (custom gates)

TurboPLONK
[GW19]

vanilla PLONK gate: $q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$

custom gates (arbitrary linear combinations):

$$\underbrace{q_{add} \cdot (a_0 + a_1 - a_2)}_{\text{add gate}} + \underbrace{q_{mul} \cdot (a_0 \cdot a_1 - a_2)}_{\text{mul gate}} + \underbrace{q_{bool} \cdot (a_0 \cdot a_0 - a_0)}_{\text{bool gate}} = 0$$

PLONKish arithmetisation (custom gates)

TurboPLONK
[GW19]

vanilla PLONK gate: $q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$

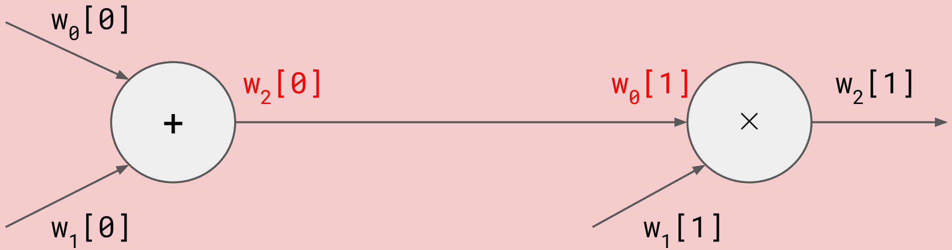
custom gates (arbitrary linear combinations):

$$\underbrace{q_{add} \cdot (a_0 + a_1 - a_2)}_{\text{add gate}} + y \cdot \underbrace{q_{mul} \cdot (a_0 \cdot a_1 - a_2)}_{\text{mul gate}} + y^2 \cdot \underbrace{q_{bool} \cdot (a_0 \cdot a_0 - a_0)}_{\text{bool gate}} = 0$$

verifier challenge to keep gates linearly independent

PLONKish arithmetisation (permutation)

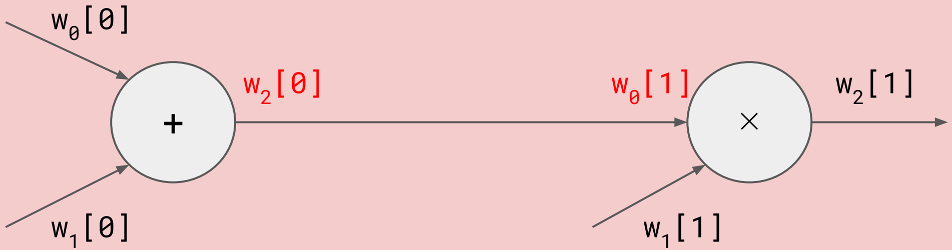
PLONK
[GWC19]



- **wires** carry values into and out of gates

PLONKish arithmetisation (permutation)

PLONK
[GWC19]



- **wires** carry values into and out of gates

"global" consistency check: do the wires correctly join the gates together?

* in Groth16, routing is baked into the trusted setup; we can't do this for universal SNARKs

PLONKish arithmetisation (permutation)

PLONK
[GWC19]

w_0	w_1	w_2	gate
$w_0[0]$	$w_1[0]$	$w_2[0]$	+
$w_0[1]$	$w_1[1]$	$w_2[1]$	×

each wire (column i) is encoded as a Lagrange polynomial w_i over the powers (rows) of an n^{th} root of unity $\{1, \omega, \ldots, \omega^{n-1}\}$, where $\omega^n = 1$:

$$\boxed{w_i(\omega^j) = w_i[j]}$$

PLONKish arithmetisation (permutation)

PLONK
[GWC19]

w_0	w_1	w_2	gate
$w_0[0]$	$w_1[0]$	$w_2[0]$	+
$w_0[1]$	$w_1[1]$	$w_2[1]$	×

each wire (column i) is encoded as a Lagrange polynomial w_i over the powers (rows) of an n^{th} root of unity $\{1, \omega, \ldots, \omega^{n-1}\}$, where $\omega^n = 1$:

$$\boxed{w_i(\omega^j) = w_i[j]}$$

to enforce equality of wires, use permutation argument (deep-dive); show that swapping $w_2(\omega^0)$ with $w_0(\omega^1)$ doesn't change the polynomials.

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1
42	SHA(42)
0	0
69	SHA(69)
…	…
0	0

problem: SHA is expensive to do in-circuit

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

solution: load precomputed SHA (e.g. for 8-bit values) as lookup table

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

$(q_{lookup} \cdot w_0,\ t_0)$
$(q_{lookup} \cdot w_1,\ t_1)$

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

$$(q_{lookup} \cdot w_0 + (1 - q_{lookup}) \cdot 0, \quad t_0)$$
$$(q_{lookup} \cdot w_1 + (1 - q_{lookup}) \cdot \mathrm{SHA}(0), \quad t_1)$$

lookup default value when q_{lookup} is not enabled,
so that lookup argument passes on every row

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

the lookup argument is a more permissive version of the permutation argument. it enforces that:

every cell in a set of **input columns** is equal to
some cell in a set of **table columns**

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

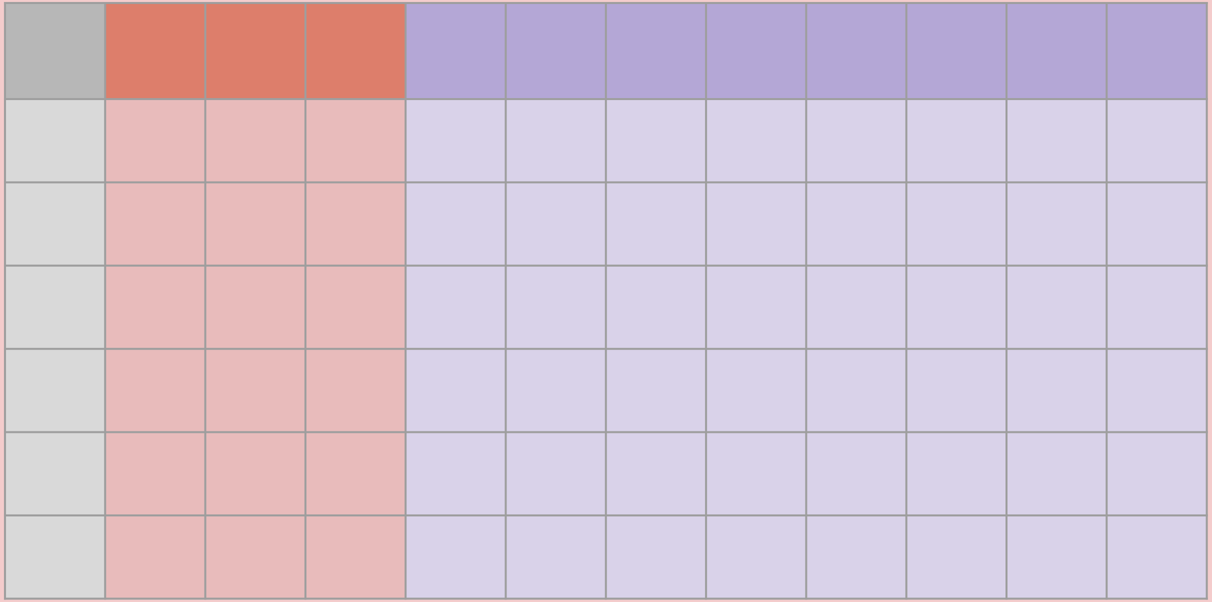
w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

the lookup argument is a more permissive version of the permutation argument. it enforces that:

every expression in a set of **input columns** is equal to
some expression in a set of **table columns**

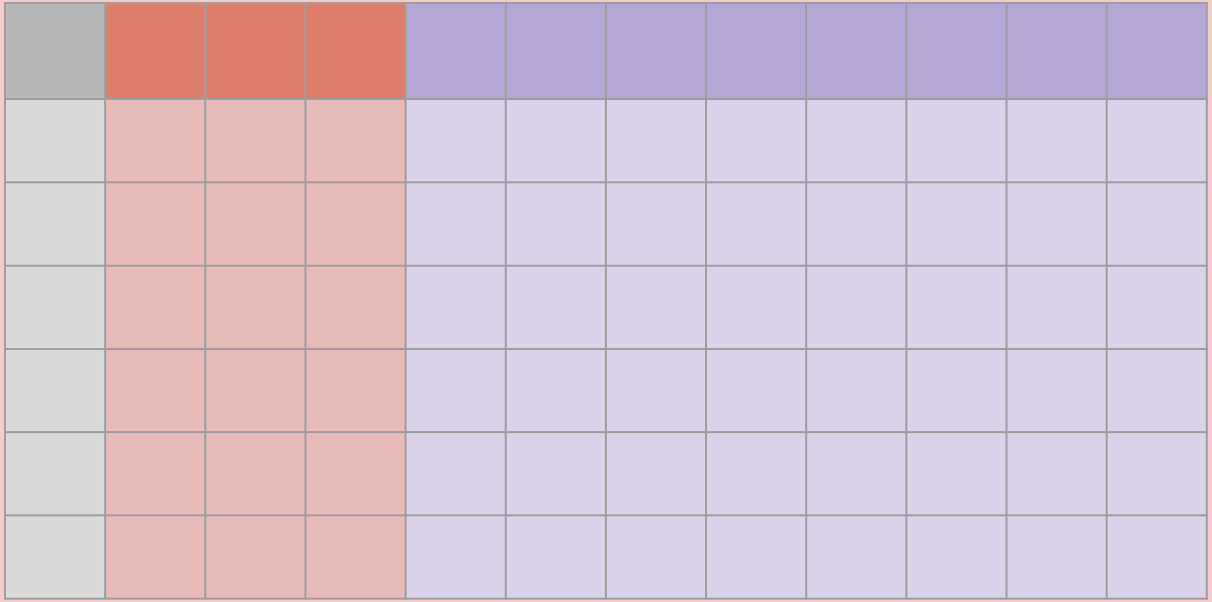
more on this in the deep-dive!

PLONKish arithmetisation



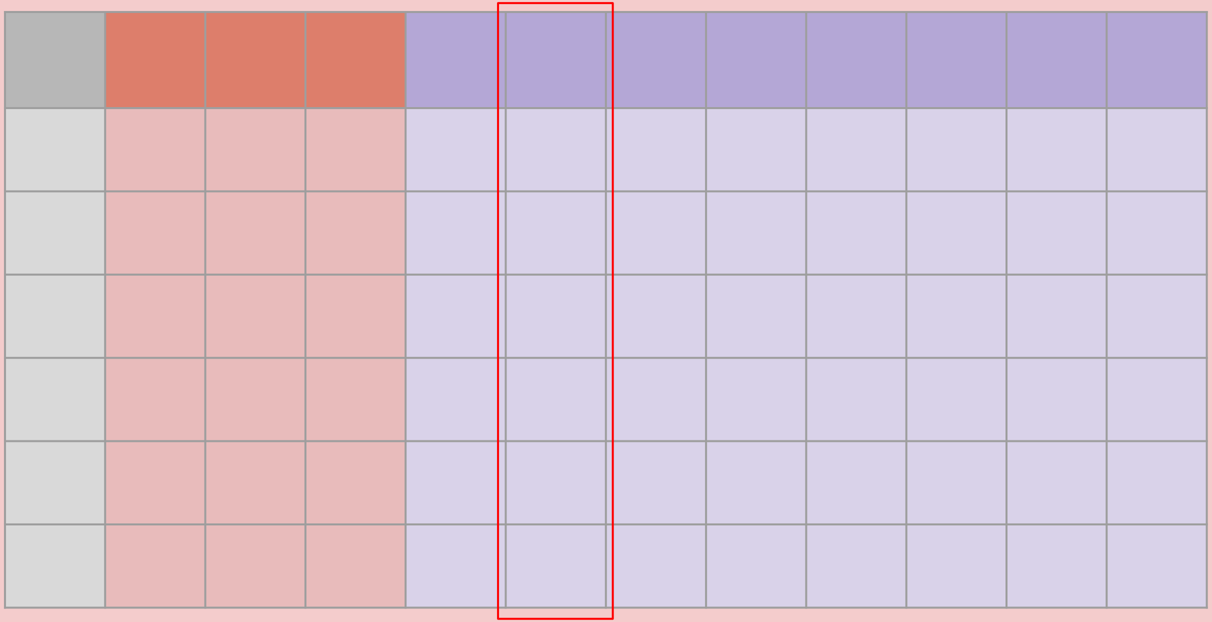
we conceptualise the circuit as a **matrix** of *m* columns and *n* rows

PLONKish arithmetisation



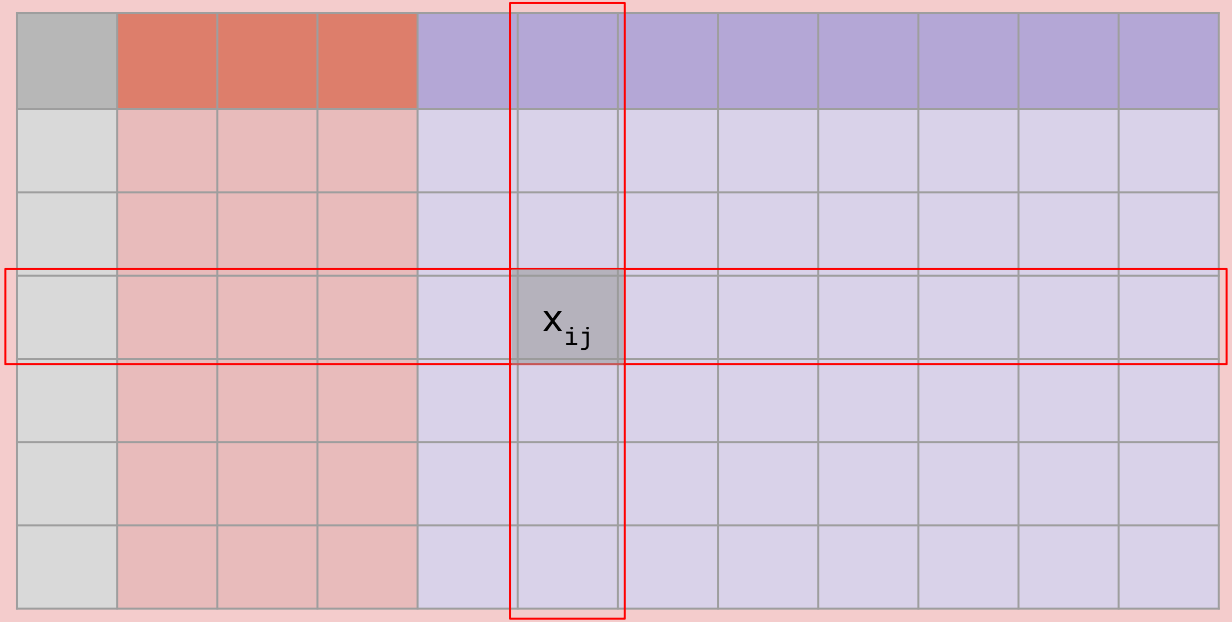
we conceptualise the circuit as a **matrix** of m columns and n rows, over a given **finite field** $\mathbb{F}$ (so the cells contain elements of $\mathbb{F}$)

PLONKish arithmetisation



each column j corresponds to a Lagrange interpolation polynomial $p_j(X)$

PLONKish arithmetisation



each column j corresponds to a Lagrange interpolation polynomial $p_j(X)$ evaluating to $\mathbf{p_j(\omega^i) = x_{ij}}$, where ω is the n^{th} primitive root of unity.

aside: fast Fourier transform (FFT)

how to encode vector $[a_0, a_1, \ldots, a_{n-1}]$ as polynomial $p(X)$?

treat each a_i as the **evaluation** of $p(X)$ at a certain point x_i.
(for efficiency, we pick x_i to be the ith power of the root of unity ω^i, where $\omega^n = 1$.)

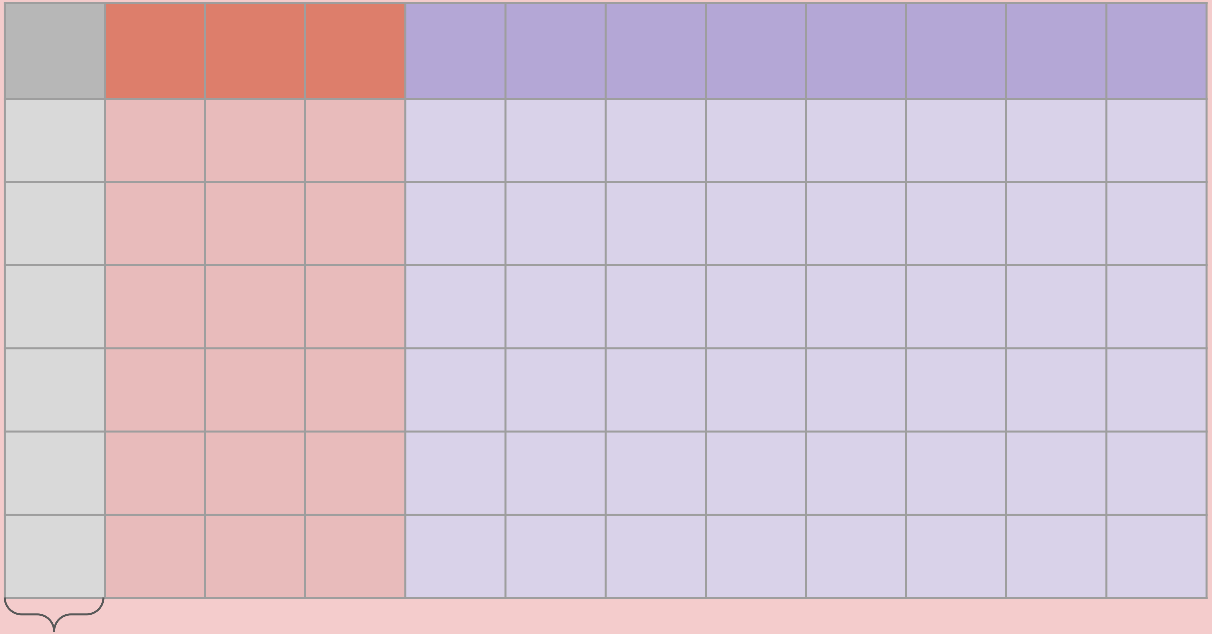
$$\boxed{p(X) := \sum a_i L_i(X)},$$

where $L_i(X)$'s are the Lagrange bases

$$L_i(X) := \frac{\prod_{j \neq i}(\omega^i - \omega^j)}{\prod_{j \neq i}(X - \omega^j)} = \begin{cases} 1 \text{ if } X = \omega^i, \\ 0 \text{ otherwise} \end{cases}$$

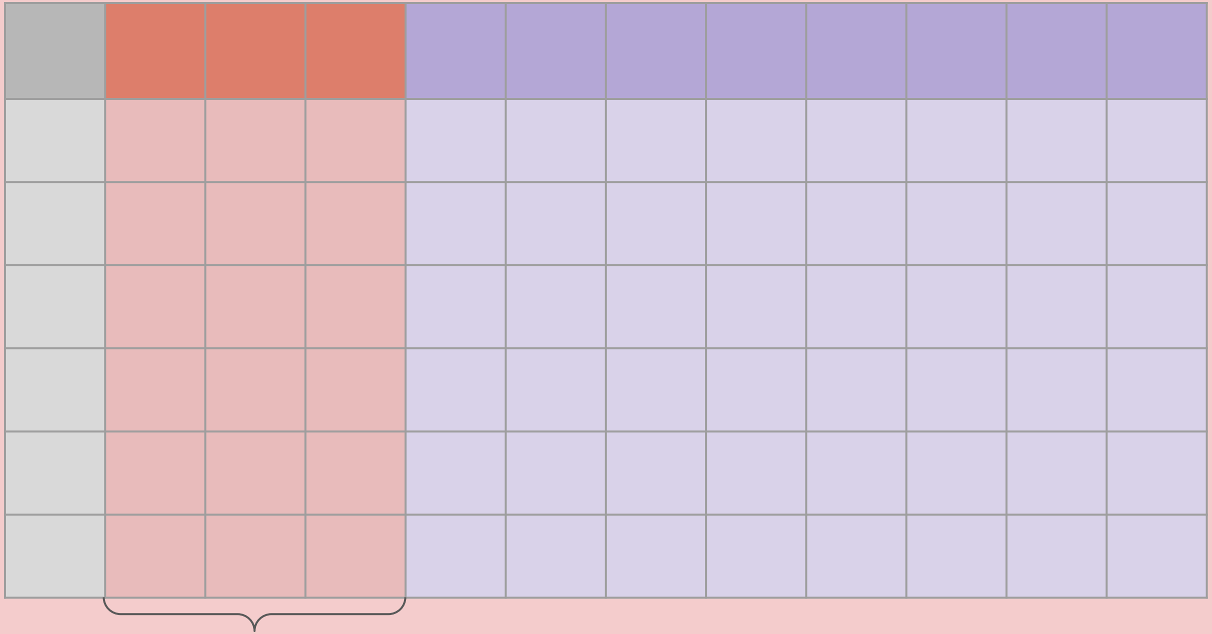
we will be working over the evaluation domain $H = \{\omega^i\}$, $i = 0..N$

PLONKish arithmetisation



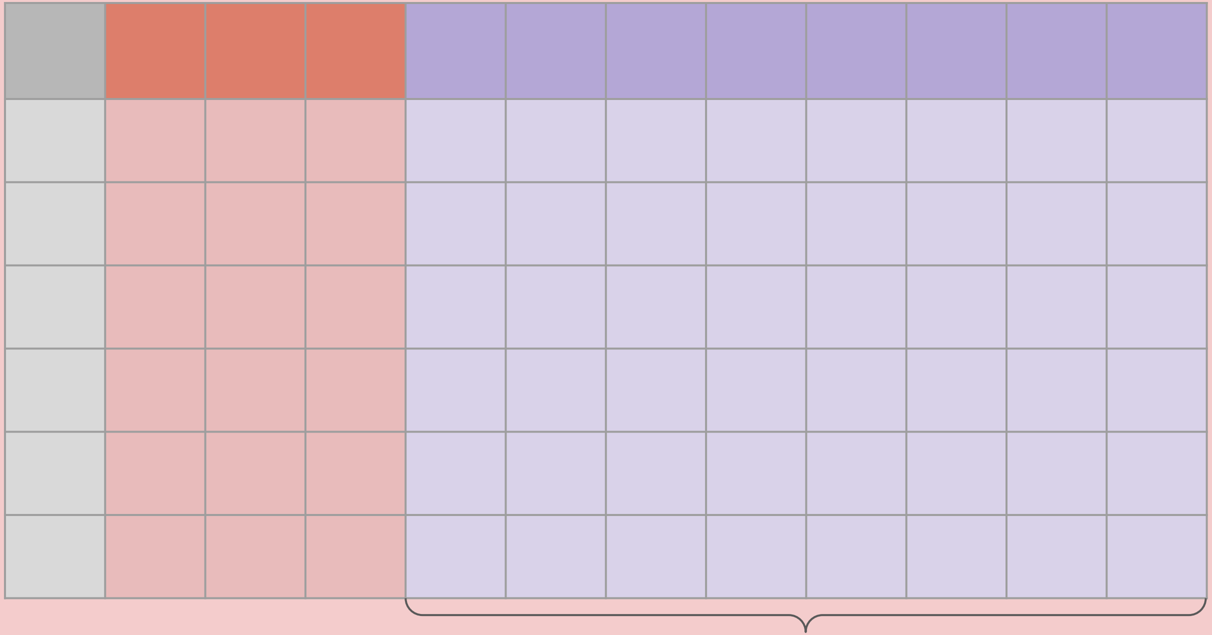
instance columns contain inputs **shared** between prover/verifier (e.g. public inputs)

PLONKish arithmetisation



advice columns contain private
values witnessed by the prover

PLONKish arithmetisation



fixed columns contain preprocessed
values set at key generation

example: Fibonacci sequence

write this in tomorrow's session!

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	2	1
	2	3	5	1
13	5	8	13	0

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1 +	1 =	2	1
	2	3	5	1
13	5	8	13	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	2	1
	2	3	5	1
13	5	8	13	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$$

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	2	1
	2	3	5	1
13	5	8	13	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$$

$$q_{fib} \cdot (a_{1,cur} + a_{2,cur} - a_{1,next}) = 0$$

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	2	1
	2	3	5	1
13	5	8	13	0

$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$

$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$

$q_{fib} \cdot (a_{1,cur} + a_{2,cur} - a_{1,next}) = 0$

global permutation: $a_2[i] = a_0[i + 1]$

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	2	1
	2	3	5	1
13	5	8	13	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$$

$$q_{fib} \cdot (a_{1,cur} + a_{2,cur} - a_{1,next}) = 0$$

global permutation: $a_2[i] = a_0[i + 1]$

exercise: can you see how to constrain this locally (using q_{fib})?

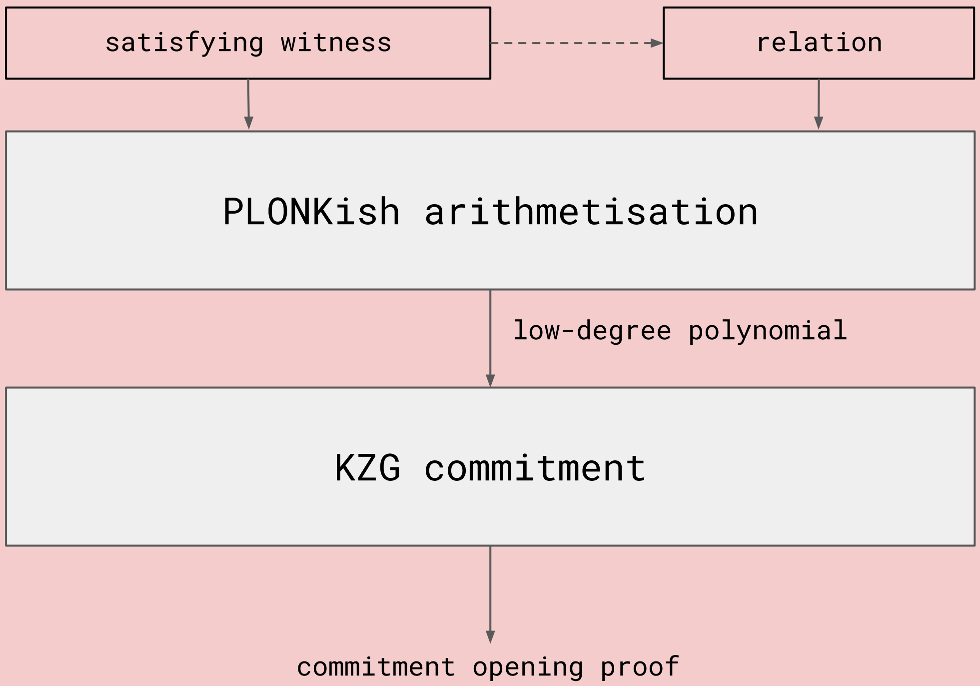
example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	2	1
	2	3	5	1
$\boxed{13}$	5	8	$\boxed{13}$	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$
$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$$
$$q_{fib} \cdot (a_{1,cur} + a_{2,cur} - a_{1,next}) = 0$$

global permutation:

- $i_0[0] = a_0[0]$ // initialisation
- $i_0[0] = a_1[0]$ // initialisation
- $i_0[2] = a_2[2]$ // output


satisfying witness
relation
PLONKish arithmetisation
low-degree polynomial
KZG commitment
commitment opening proof

Prover

Verifier

(slide from han0110)

Prover

Verifier

$Z_H(X) = X^n - 1,$

$f_0(X),\ f_1(X),\ \dots$

$\longleftrightarrow$

(slide from han0110)

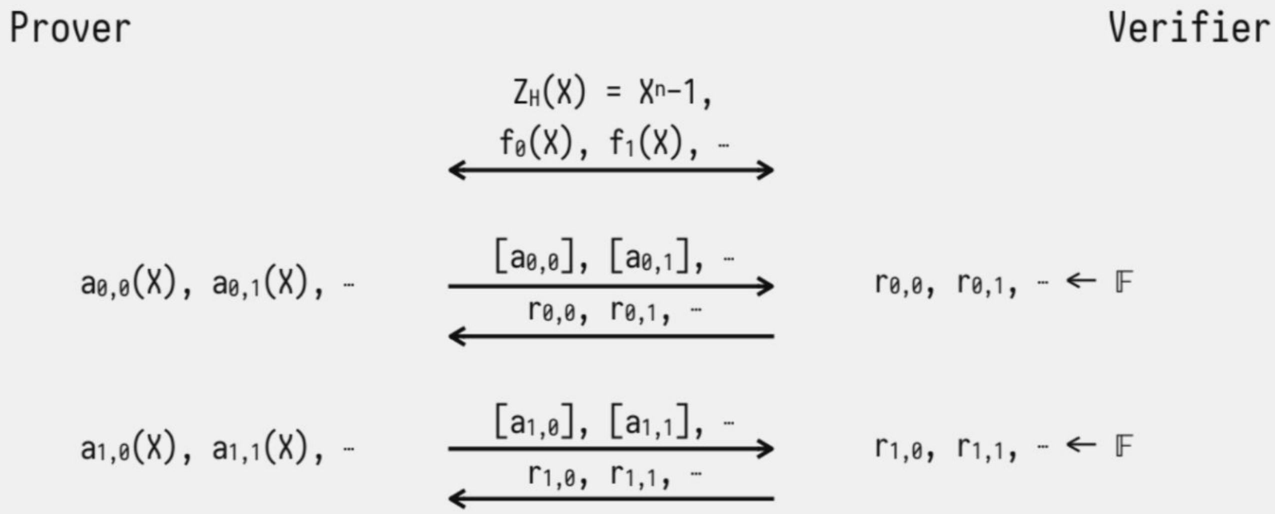
Prover | **Verifier**

$$\xleftrightarrow{Z_H(X) = X^n-1,\ f_0(X),\ f_1(X),\ \dots}$$

$$a_{0,0}(X),\ a_{0,1}(X),\ \dots \qquad \xrightarrow{[a_{0,0}],\ [a_{0,1}],\ \dots} \qquad r_{0,0},\ r_{0,1},\ \dots \leftarrow \mathbb{F}$$

$$\xleftarrow{r_{0,0},\ r_{0,1},\ \dots}$$

(slide from han0110)



(slide from han0110)

Prover **Verifier**

$$\xleftrightarrow{Z_H(X) = X^n - 1,\ f_0(X),\ f_1(X),\ \ldots}$$

$$a_{0,0}(X),\ a_{0,1}(X),\ \ldots \quad \xrightarrow{[a_{0,0}],\ [a_{0,1}],\ \ldots} \quad r_{0,0},\ r_{0,1},\ \ldots \leftarrow \mathbb{F}$$

$$\xleftarrow{r_{0,0},\ r_{0,1},\ \ldots}$$

$$a_{1,0}(X),\ a_{1,1}(X),\ \ldots \quad \xrightarrow{[a_{1,0}],\ [a_{1,1}],\ \ldots} \quad r_{1,0},\ r_{1,1},\ \ldots \leftarrow \mathbb{F}$$

$$\xleftarrow{r_{1,0},\ r_{1,1},\ \ldots}$$

$$\ldots$$

$$q(X) = \frac{(gate_0(X) + y \cdot gate_1(X) + \ldots)}{Z_H(X)} \quad \xrightarrow{[q_0],\ [q_1],\ \ldots} \quad x \leftarrow \mathbb{F}$$

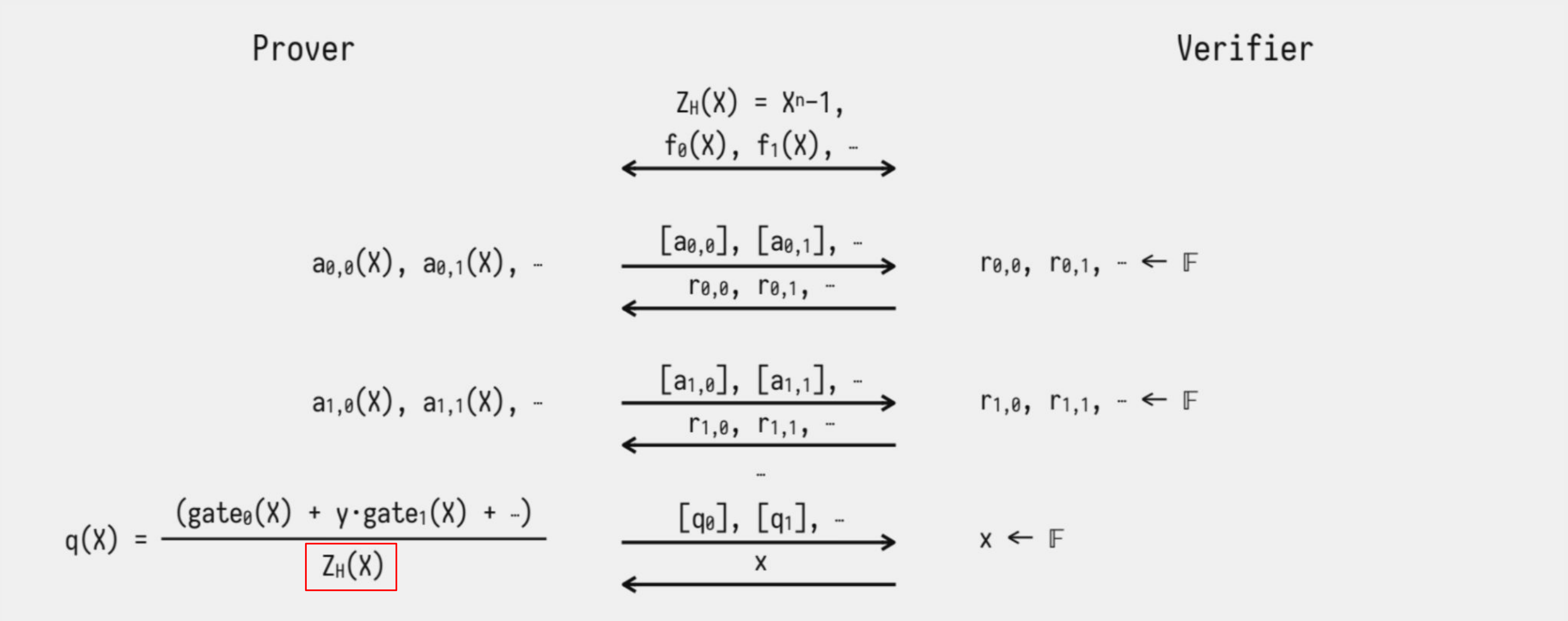
$$\xleftarrow{x}$$

(slide from han0110)

Prover		Verifier
	$Z_H(X) = X^n - 1,$ $f_0(X),\ f_1(X),\ \ldots$ $\longleftrightarrow$	
$a_{0,0}(X),\ a_{0,1}(X),\ \ldots$	$\xrightarrow{[a_{0,0}],\ [a_{0,1}],\ \ldots}$ $\xleftarrow{r_{0,0},\ r_{0,1},\ \ldots}$	$r_{0,0},\ r_{0,1},\ \ldots \leftarrow \mathbb{F}$
$a_{1,0}(X),\ a_{1,1}(X),\ \ldots$	$\xrightarrow{[a_{1,0}],\ [a_{1,1}],\ \ldots}$ $\xleftarrow{r_{1,0},\ r_{1,1},\ \ldots}$	$r_{1,0},\ r_{1,1},\ \ldots \leftarrow \mathbb{F}$
	$\ldots$	
$q(X) = \dfrac{(gate_0(X) + y \cdot gate_1(X) + \ldots)}{\boxed{Z_H(X)}}$	$\xrightarrow{[q_0],\ [q_1],\ \ldots}$ $\xleftarrow{x}$	$x \leftarrow \mathbb{F}$

recall: our evaluation domain is $H = \{H = \{\omega^i\},\ i = 0..N\}$, where $\omega^N = 1$

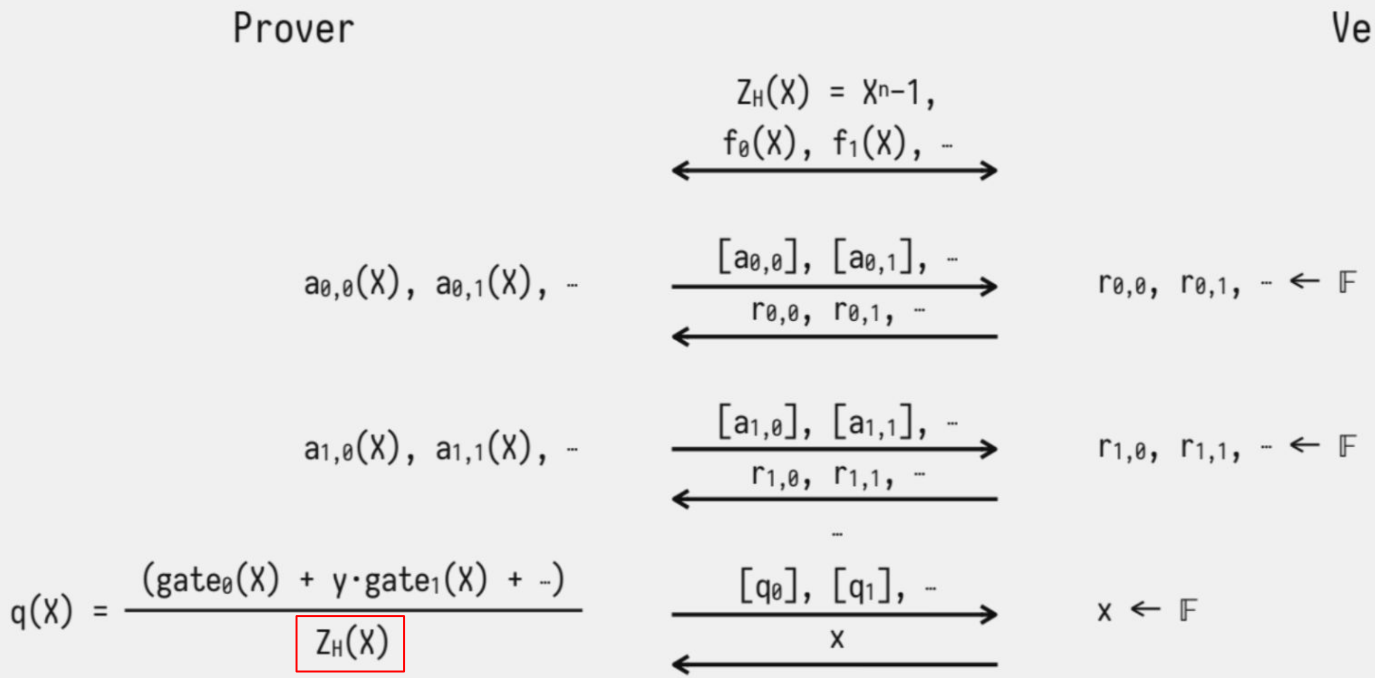
(slide from han0110)



recall: our evaluation domain is H = $\{H = \{\omega^i\}, i = 0..N\}$, where $\omega^N = 1$

$Z_H(X) = X^N - 1$ evaluates to zero (i.e. vanishes) over H

(slide from han0110)



recall: our evaluation domain is $H = \{H = \{\omega^i\}, i = 0..N\}$, where $\omega^N = 1$

$Z_H(X) = X^N - 1$ evaluates to zero (i.e. vanishes) over H

if $f(X) \,/\, Z_H(X) = q(X)$ some polynomial $\Rightarrow f(\omega^i) = 0$ for ω^i in H

(slide from han0110)

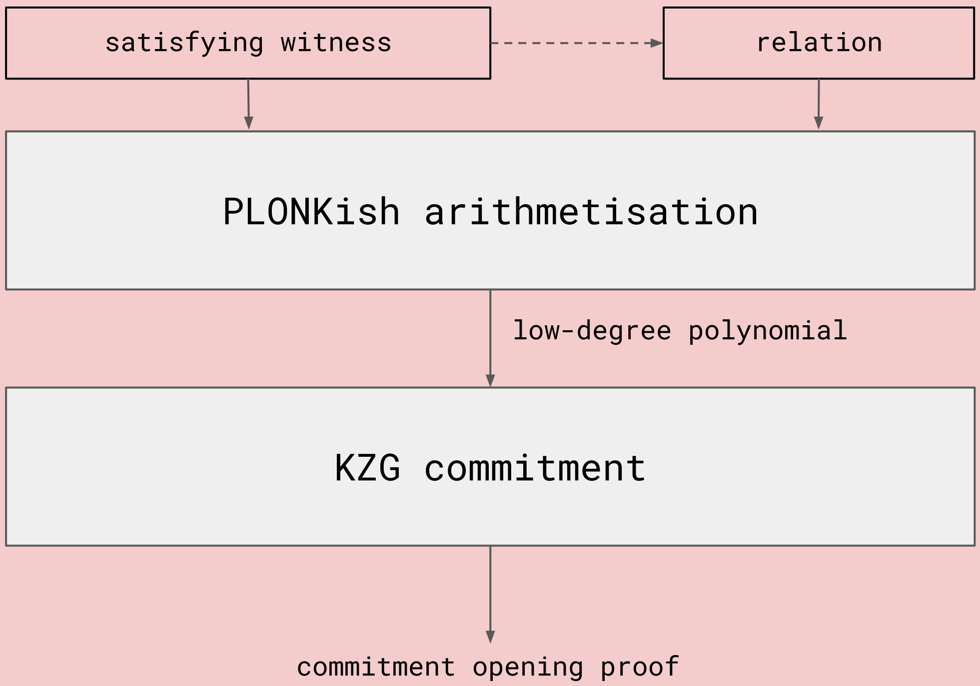
Prover		Verifier
	$Z_H(X) = X^n - 1,$ $f_0(X),\ f_1(X),\ \dots$ $\longleftrightarrow$	
$a_{0,0}(X),\ a_{0,1}(X),\ \dots$	$\xrightarrow{[a_{0,0}],\ [a_{0,1}],\ \dots}$ $\xleftarrow{r_{0,0},\ r_{0,1},\ \dots}$	$r_{0,0},\ r_{0,1},\ \dots \leftarrow \mathbb{F}$
$a_{1,0}(X),\ a_{1,1}(X),\ \dots$	$\xrightarrow{[a_{1,0}],\ [a_{1,1}],\ \dots}$ $\xleftarrow{r_{1,0},\ r_{1,1},\ \dots}$	$r_{1,0},\ r_{1,1},\ \dots \leftarrow \mathbb{F}$
	$\dots$	
$q(X) = \dfrac{(gate_0(X) + y \cdot gate_1(X) + \dots)}{Z_H(X)}$	$\xrightarrow{[q_0],\ [q_1],\ \dots}$ $\xleftarrow{x}$	$x \leftarrow \mathbb{F}$
	$\xrightarrow{\text{evals}}$	$q(x) \cdot Z_H(x) \stackrel{?}{=} gate_0(x) + y \cdot gate_1(x) + \dots$

(slide from han0110)

Prover		Verifier
	$\xleftrightarrow{Z_H(X) = X^n - 1,\ f_0(X),\ f_1(X),\ \dots}$	
$a_{0,0}(X),\ a_{0,1}(X),\ \dots$	$\xrightarrow{[a_{0,0}],\ [a_{0,1}],\ \dots}$	$r_{0,0},\ r_{0,1},\ \dots \leftarrow \mathbb{F}$
	$\xleftarrow{r_{0,0},\ r_{0,1},\ \dots}$	
$a_{1,0}(X),\ a_{1,1}(X),\ \dots$	$\xrightarrow{[a_{1,0}],\ [a_{1,1}],\ \dots}$	$r_{1,0},\ r_{1,1},\ \dots \leftarrow \mathbb{F}$
	$\xleftarrow{r_{1,0},\ r_{1,1},\ \dots}$	
	$\dots$	
$q(X) = \dfrac{(gate_0(X) + y \cdot gate_1(X) + \dots)}{Z_H(X)}$	$\xrightarrow{[q_0],\ [q_1],\ \dots}$	$x \leftarrow \mathbb{F}$
	$\xleftarrow{x}$	
	$\xrightarrow{evals}$	$q(x) \cdot Z_H(x) \stackrel{?}{=} gate_0(x) + y \cdot gate_1(x) + \dots$

$\Rightarrow f(x) := q(x) \cdot Z_H(x) - gates(x) \ ?= 0$

(slide from han0110)


satisfying witness
relation
PLONKish arithmetisation
low-degree polynomial
KZG commitment
commitment opening proof

polynomial commitment scheme

allows prover to convince verifier that $f(z) = y$, without revealing f

Setup$(1^\lambda, N)$: generates a setup pp

Commit(pp, f): creates a commitment C to $f(X)$

Prove(pp, f, z): generates an opening proof π

Verify(pp, C, z, y, π): checks if $y = f(z)$ using π

polynomial commitment scheme

allows prover to convince verifier that f(z) = y, without revealing f

Setup$(1^\lambda, N)$: generates a setup pp

Commit(pp, f): creates a commitment C to $f(X)$

Prove(pp, f, z): generates an opening proof π

Verify(pp, C, z, y, π): checks if $y = f(z)$ using π

recall that at the end of arithmetisation, we wanted to check:

$$f(x) := q(x) \cdot Z_H(x) - gates(x) \mathrel{?=} 0$$

setup: KZG commitment scheme

$[a^0]_1$ $[a^1]_1$ $[a^2]_1$ $[a^3]_1$ $[a^4]_1$ $[a^5]_1$ $[a^6]_1$ … $[a^{N-1}]_1$ $[a^N]_1$

- $pp = ([a^0]_1, \ldots, [a^N]_1, [a]_2) \in (\{\mathbb{G}_1\}^N, \mathbb{G}_2) \leftarrow \mathrm{Setup}(1^\lambda, N)$, $\mathbb{G}_1$, $\mathbb{G}_2$ *cryptographic groups*

commit: KZG commitment scheme

$$\text{Commit}(\; c_0 \;|\; c_1 \;|\; c_2 \;|\; c_3 \;|\; c_4 \;|\; c_5 \;|\; c_6 \;|\; \dots \;|\; c_{N-1} \;|\; c_N \;)$$

$$c_0[a^0]_1 + c_1[a^1]_1 + c_2[a^2]_1 + c_3[a^3]_1 + c_4[a^4]_1 + c_5[a^5]_1 + c_6[a^6]_1 \;\dots\; + c_{N-1}[a^{N-1}]_1 + c_N[a^N]_1$$

$$= \Sigma[c_i][a^i]_1 = C$$

- $pp = ([a^0]_1, \dots, [a^N]_1, [a]_2) \in (\{\mathbb{G}_1\}^N, \mathbb{G}_2) \leftarrow \text{Setup}(1^\lambda, N)$, $\mathbb{G}_1$, $\mathbb{G}_2$ *cryptographic groups*
- $C \in \mathbb{G}_1 \leftarrow \text{Commit}(pp; \boldsymbol{f}) = \Sigma[c_i][a^i]_1$

aside: discrete logarithm hardness

the **discrete log problem** is defined as follows. given:

- a group element $G \in \mathbb{G}$, and
- a group element $[a]_1 = [a]G$,

to recover a, the discrete logarithm of $[a]_1$ in G.

this problem is assumed to be **hard** in **cryptographic groups** (e.g. **elliptic curves**)

so, given $G_1 \in \mathbb{G}$ and an encoding **srs** $= [\tau^0]G_1, [\tau^1]G_1, \ldots, [\tau^d]G_1$

$= [\tau^0]_1, [\tau^1]_1, \ldots, [\tau^d]_1,$

it's **hard** to recover the powers of the secret point τ.

prove: KZG commitment scheme

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

- pp = $([\alpha^0]_1, \ldots, [\alpha^N]_1, [\alpha]_2) \in (\{\mathbb{G}_1\}^N, \mathbb{G}_2) \leftarrow$ Setup(1^λ, N), $\mathbb{G}_1$, $\mathbb{G}_2$ *cryptographic groups*
- $C \in \mathbb{G}_1 \leftarrow$ Commit(pp; $\boldsymbol{f}$) = $\sum[c_i][\boldsymbol{\alpha}^i]_1$
- $\boxed{\Pi \leftarrow \text{Prove}(\text{pp}, C, i)}$ proof size: $O(1)$

prove: KZG commitment scheme

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

$\Pi := [q(X)]_1 = \sum [q_i][a^i]_1$

- $pp = ([a^0]_1, \ldots, [a^N]_1, [a]_2) \in (\{\mathbb{G}_1\}^N, \mathbb{G}_2) \leftarrow \mathrm{Setup}(1^\lambda, N)$, $\mathbb{G}_1$, $\mathbb{G}_2$ *cryptographic groups*
- $C \in \mathbb{G}_1 \leftarrow \mathrm{Commit}(pp; \boldsymbol{f}) = \sum [c_i][a^i]_1$
- $\boxed{\Pi \leftarrow \mathrm{Prove}(pp, C, i)}$ proof size: $O(1)$

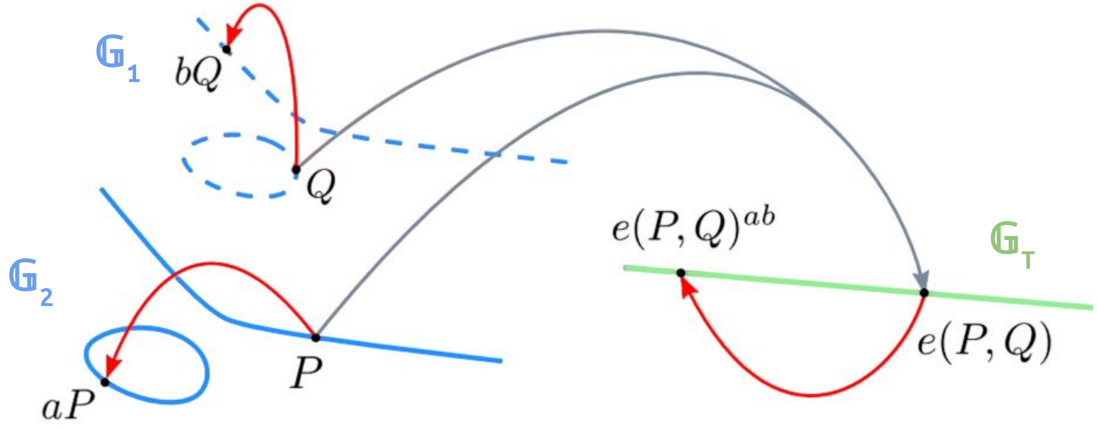
verify: KZG commitment scheme

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

$\Pi := [q(X)]_1 = \sum [q_i][\alpha^i]_1$, check: $e(\pi, [\alpha - z]_2) = e(C - [y]_1, [1]_2)$

- pp = $([\alpha^0]_1, \dots, [\alpha^N]_1, [\alpha]_2) \in (\{\mathbb{G}_1\}^N, \mathbb{G}_2) \leftarrow$ Setup(1^λ, N), $\mathbb{G}_1$, $\mathbb{G}_2$ *cryptographic groups*
- $C \in \mathbb{G}_1 \leftarrow$ Commit(pp; $\boldsymbol{f}$) = $\sum [c_i][\alpha^i]_1$
- $\Pi \leftarrow$ Prove(pp, C, i)
- $\boxed{\{0,1\} \leftarrow \text{Verify(pp, C, } i\text{; }\Pi)}$ verification time: $O(1)$

aside: bilinear pairings



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

$$e([a]P,\ [b]Q) = [ab]\ e(P,\ Q)$$

image from https://blog.cloudflare.com/circl-pairings-update/

thank you!

any questions?