Deep dive on Halo 2

Daira Hopwood (@feministPLT)
Ying Tong Lai (@therealyingtong)

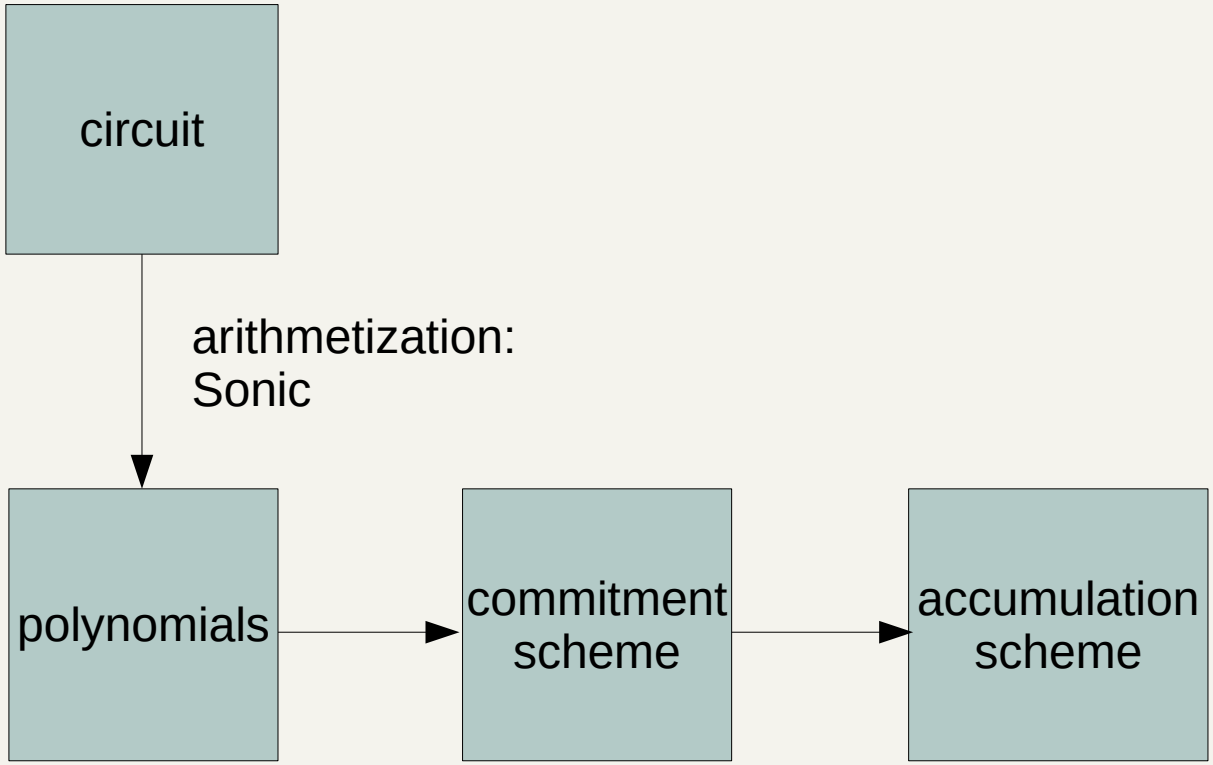
https://github.com/daira/halographs
(deepdive.odp)

Outline

- Halo 2
  - Polynomial circuits and PLONK
  - Polynomial commitments
  - Recursion and proof-carrying data
- Constructing 2-cycles of elliptic curves
  - Obstacles to finding cycles of curves
  - Complex multiplication to the rescue
  - 2-adicity
- Optimizations
  - Scalar multiplication
  - Fiat–Shamir and duplex sponges
  - Hashes (Sinsemilla).

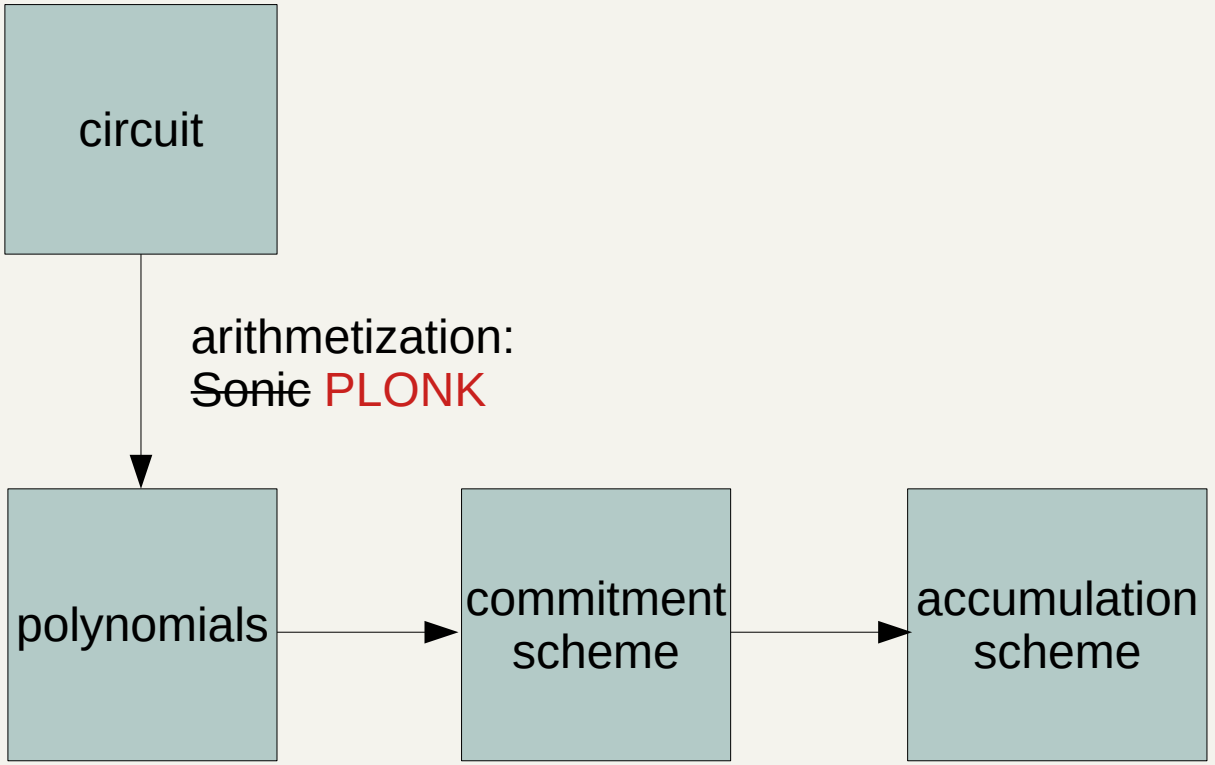
Halo

Halo 2 is like Halo but with a different arithmetization

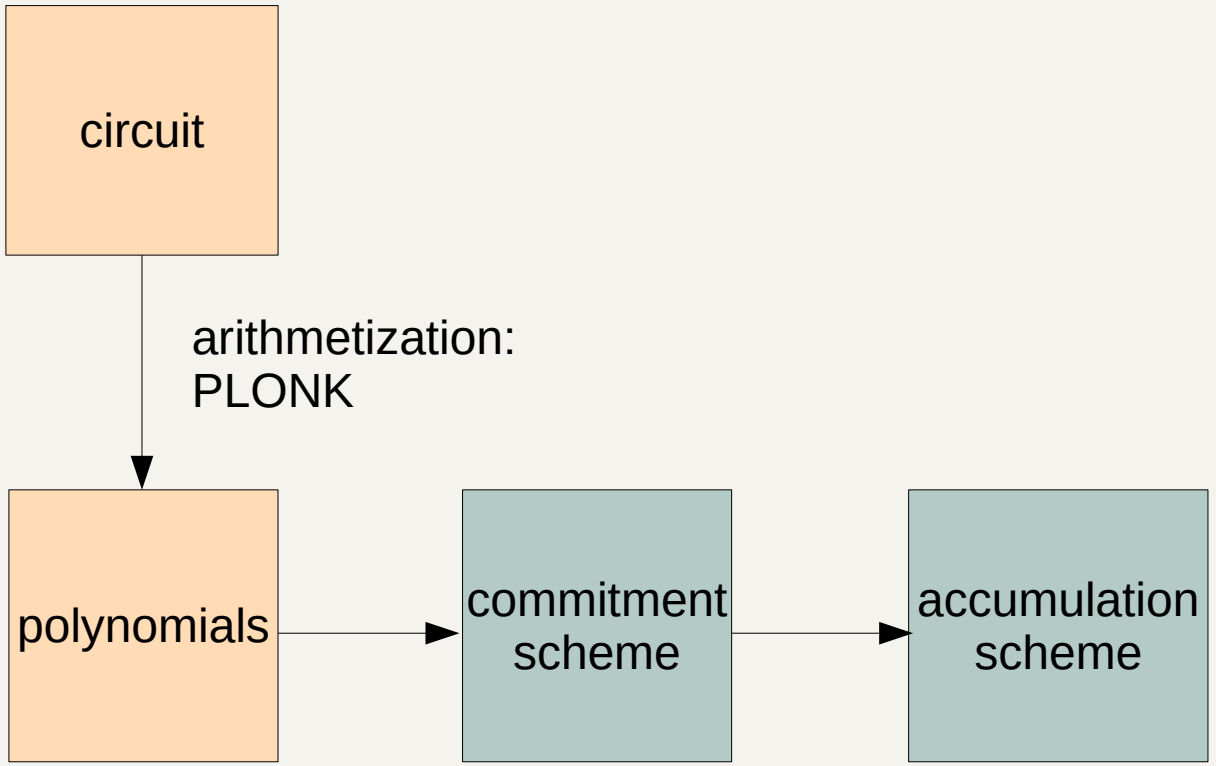


Halo 2

Halo 2 is like Halo but with a different arithmetization



Halo 2


circuit
arithmetization:
PLONK
polynomials
commitment
scheme
accumulation
scheme

Halo 2 and PLONK arithmetization

- Any generic proof system needs an “arithmetization”, which is a way of expressing statements as relations between variables.
- Halo 2 uses the PLONK [GWC2020] (ia.cr/2020/953) arithmetization, which is different from R1CS.
- In R1CS, each constraint is $A \times B = C$, where A, B, and C are linear combinations.
- PLONK generalises this to addition gates and public inputs using “selector polynomials”:

$$\mathbf{q}_L \cdot \mathbf{x}_a + \mathbf{q}_R \cdot \mathbf{x}_b + \mathbf{q}_O \cdot \mathbf{x}_c + \mathbf{q}_M \cdot (\mathbf{x}_a \mathbf{x}_b) + \mathbf{q}_C = 0$$

Halo 2 and PLONK arithmetization

standard PLONK: $\mathbf{q}_L \cdot \mathbf{x}_a + \mathbf{q}_R \cdot \mathbf{x}_b + \mathbf{q}_O \cdot \mathbf{x}_c + \mathbf{q}_M \cdot (\mathbf{x}_a \mathbf{x}_b) + \mathbf{q}_C = 0$

	x_{a_i}	x_{b_i}	x_{c_i}	q_{L_i}	q_{R_i}	q_{O_i}	q_{M_i}	q_{C_i}
i = 0	a_0	b_0	c_0	1	1	-1	0	0
i = 1	a_1	b_1	c_1	0	0	-1	1	0
i = 2	PI	0	0	1	0	0	0	-PI
i = 3	x	x	0	-1	0	0	1	0

$a_0 + b_0 = c_0$ (addition)
$a_1 \times b_1 = c_1$ (multiplication)
x_{a_2} = PI (public input)
$x \in \{0, 1\}$ (booleanity using $x^2 = x$)

Halo 2 and PLONK arithmetization

standard PLONK: $\mathbf{q}_L \cdot \mathbf{x}_a + \mathbf{q}_R \cdot \mathbf{x}_b + \mathbf{q}_O \cdot \mathbf{x}_c + \mathbf{q}_M \cdot (\mathbf{x}_a \mathbf{x}_b) + \mathbf{q}_C = 0$

	x_{a_i}	x_{b_i}	x_{c_i}	q_{L_i}	q_{R_i}	q_{O_i}	q_{M_i}	q_{C_i}	
i = 0	a_0	b_0	c_0	1	1	-1	0	0	$a_0 + b_0 = c_0$ (addition)
i = 1	a_1	b_1	c_1	0	0	-1	1	0	$a_1 \times b_1 = c_1$ (multiplication)
i = 2	PI	0	0	1	0	0	0	-PI	x_{a_2} = PI (public input)
i = 3	x	x	0	-1	0	0	1	0	$x \in \{0, 1\}$ (booleanity using $x^2 = x$)

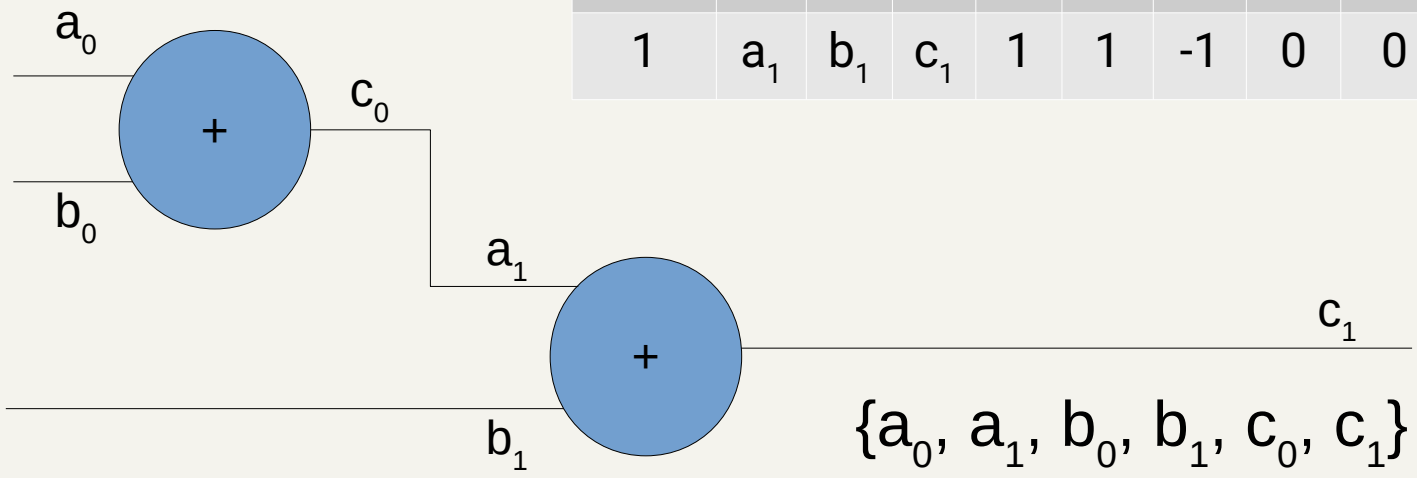
“running product” gate: $\mathbf{q}_P \cdot (\mathbf{x}_{a\_i} \mathbf{x}_{b\_i} \mathbf{x}_{c\_\{i-1\}} - \mathbf{x}_{c\_i}) = 0$

	x_{a_i}	x_{b_i}	x_{c_i}	q_{P_i}	
i = 0			c_0	0	
i = 1	a_1	b_1	c_1	1	$c_0 \times a_1 \times b_1 = c_1$
i = 2	a_2	b_2	c_2	1	$c_1 \times a_2 \times b_2 = c_2$
i = 3	a_3	b_3	c_3	1	$c_2 \times a_3 \times b_3 = c_3$

PLONK permutation check

- PLONK additionally introduces a permutation check to enforce “copy” operations between constraints.
- Suppose we have a toy circuit with two addition gates:

Row	x_a	x_b	x_c	q_L	q_R	q_O	q_M	q_C
0	a_0	b_0	c_0	1	1	-1	0	0
1	a_1	b_1	c_1	1	1	-1	0	0

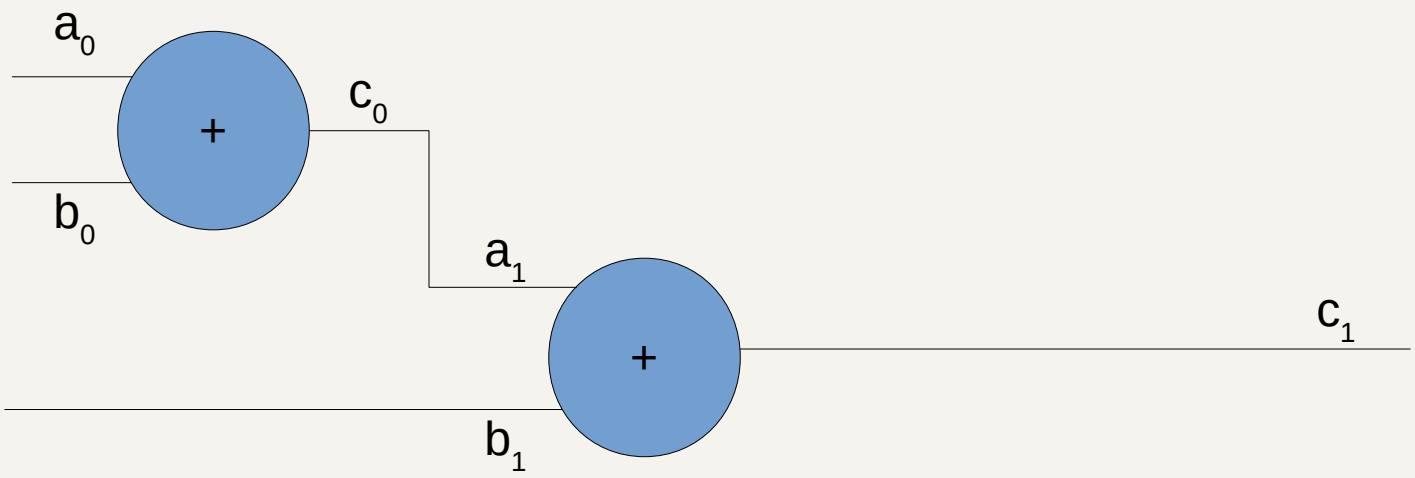


PLONK permutation check

- We want to enforce that c_0 is “copied” to a_1.
- We can do this by enforcing that the following two permutations are equal:

$$V = \{a_0, a_1, b_0, b_1, c_0, c_1\}$$
$$\sigma(V) = \{a_0, c_0, b_0, b_1, a_1, c_1\}$$

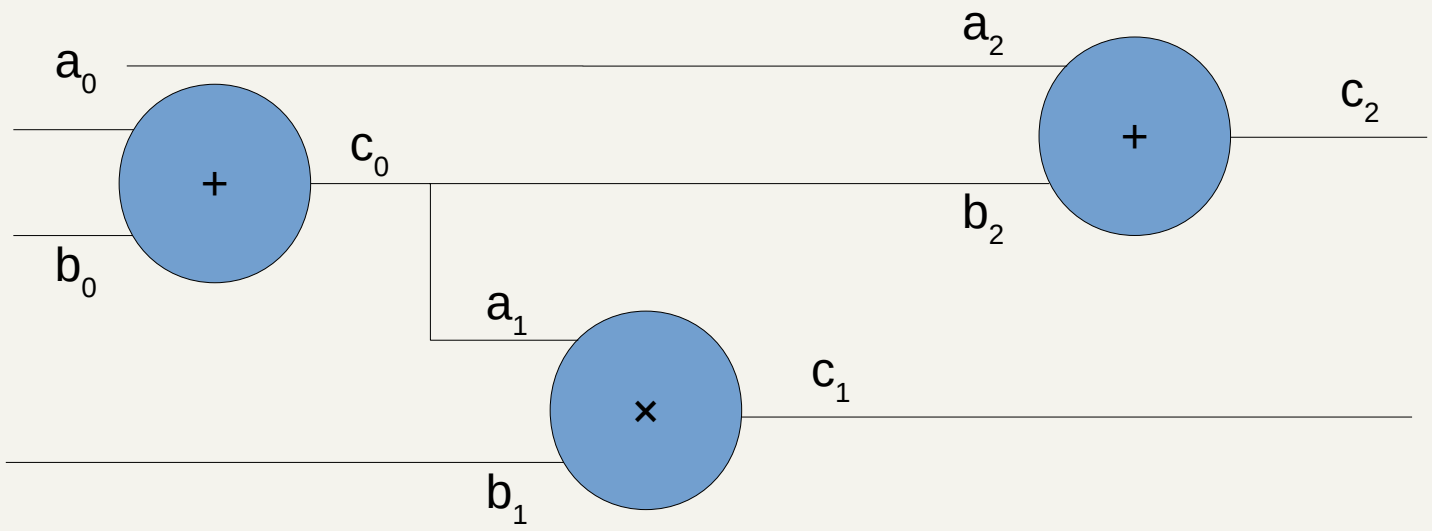
- In other words: $c_0 = a_1$ iff $V = \sigma(V)$ for permutation $\sigma = (1\ 4)$



PLONK permutation check

$V = \{a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2\}$

$\sigma = (0\ 2)(1\ 5\ 6)$



Row	x_a	x_b	x_c	q_L	q_R	q_O	q_M	q_C
0	a_0	b_0	c_0	1	1	-1	0	0
1	a_1	b_1	c_1	0	0	-1	1	0
2	a_2	b_2	c_2	1	1	-1	0	0

PLONK permutation check

- Permutation σ: $[n] \to [n]$ over multiplicative subgroup $\{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$ (where $\omega^n = 1$, $\omega^{n+k} = \omega^k$)
- Prover has sent $\mathbf{f} \in \mathbb{F}_n[X]$ and wants to prove $\mathbf{f} = \sigma(\mathbf{f})$:

$$\forall i \in [n], \mathbf{f}(\omega^i) = \mathbf{f}(\omega^{\sigma(i)})$$

- Define two permutation polynomials (which should be equal for a valid permutation):

$$\mathbf{f}' = \mathbf{f} + \beta \cdot \mathbf{f}_{ID} + \gamma,$$

$$\mathbf{g}' = \mathbf{f} + \beta \cdot \mathbf{f}_{\sigma} + \gamma,$$

- Compute their grand product:

$$\mathbf{Z}(X) \text{ s.t. } \mathbf{Z}(1) = 1, \mathbf{Z}(\omega^i) = \prod_{1 \le j < i} \mathbf{f}'(\omega^j) / \mathbf{g}'(\omega^j)$$

PLONK permutation check

- To check the equality of the two grand products, these identities must hold over the subgroup $\{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$:

1) $\mathbf{L}_0(X)(\mathbf{Z}(X) - 1) = 0$ (first term is $\mathbf{Z}(1) = 1$)

2) $\mathbf{Z}(X)\,\mathbf{f}'(X) = \mathbf{g}'(X)\,\mathbf{Z}(\omega \cdot X)$ (running product)

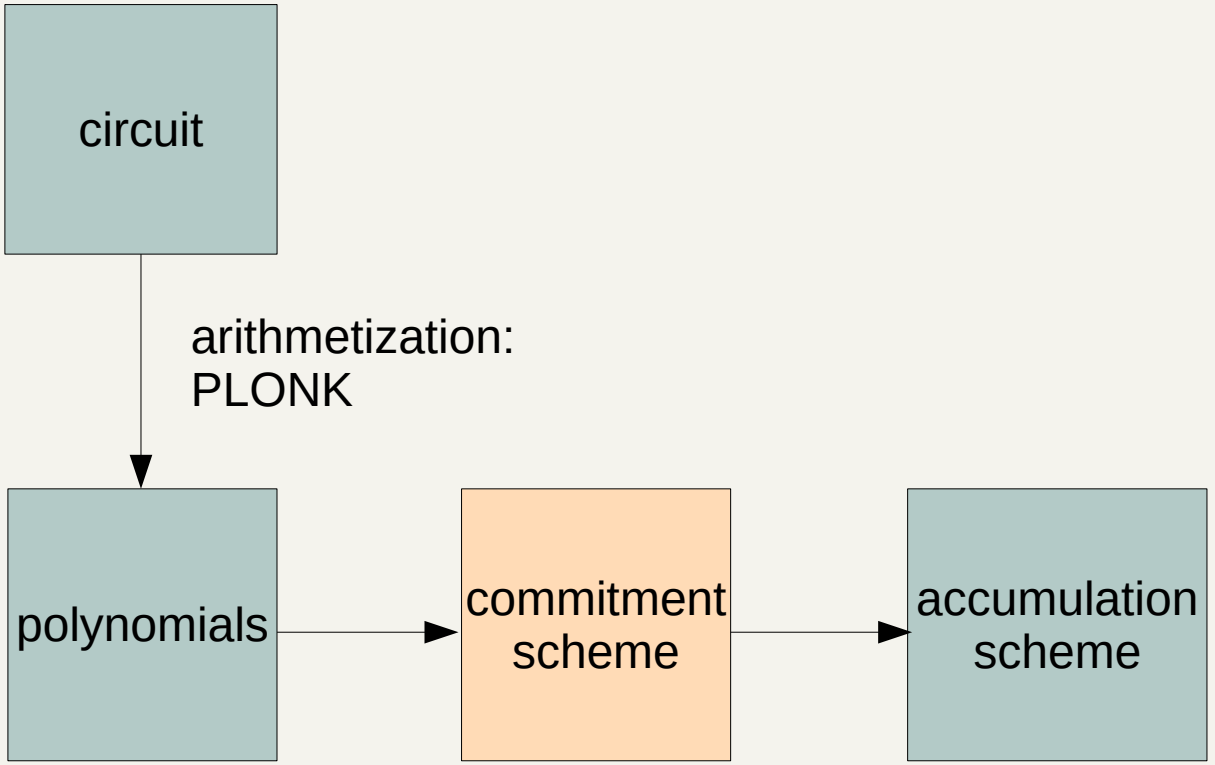
To see this, recall that $\mathbf{Z}(\omega^i) = \prod_{1 \le j < i} \mathbf{f}'(\omega^j) / \mathbf{g}'(\omega^j)$,

so $\mathbf{Z}(\omega^{i+1}) = \mathbf{Z}(\omega^i)\,(\mathbf{f}'(\omega^i) / \mathbf{g}'(\omega^i))$

$\Rightarrow\ \mathbf{Z}(\omega^i)\,\mathbf{f}'(\omega^i) = \mathbf{g}'(\omega^i)\,\mathbf{Z}(\omega^{i+1})$.

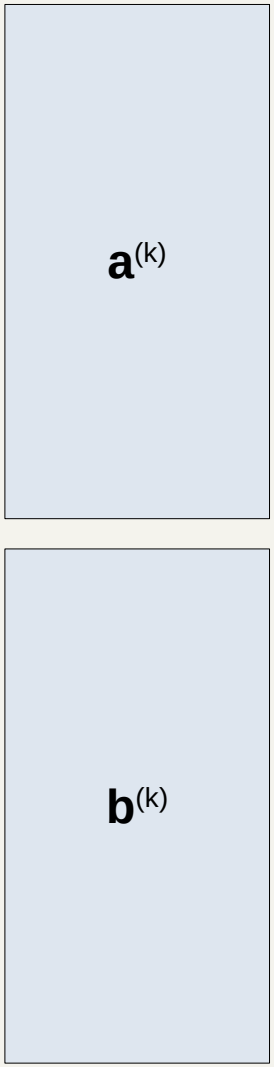
NB: the $\mathbf{L}_i(X)$'s are Lagrange basis polynomials, i.e. $\mathbf{L}_i(\omega^i) = 1$, and $\mathbf{L}_i$ is 0 elsewhere.
Thus, checking a condition using $\mathbf{L}_i$ is equivalent to checking it at the point ω^i.

Halo 2


circuit
arithmetization:
PLONK
polynomials
commitment
scheme
accumulation
scheme

Inner product argument

(originally from [Bootle et al, 2016])



We have vectors **a**, **b** each of length $n = 2^k$.

We want to prove that a given inner product c and a commitment P are related as:

$$c = \langle \mathbf{a}, \mathbf{b} \rangle$$
$$P = \langle \mathbf{a}, \mathbf{G} \rangle + \langle \mathbf{b}, \mathbf{H} \rangle$$

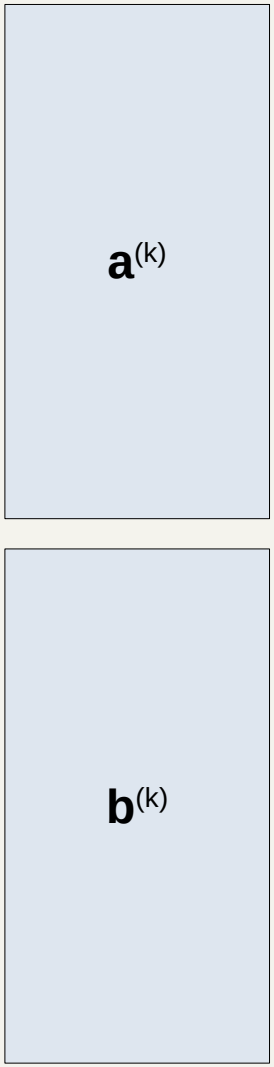
where **G**, **H** are length-n vectors of random group elements.

We can combine this into a single equation P_k using a random group element U:

$$P_k = P + [c]\, U = \langle \mathbf{a}, \mathbf{G} \rangle + \langle \mathbf{b}, \mathbf{H} \rangle + [\langle \mathbf{a}, \mathbf{b} \rangle]\, U$$

The naive solution would be to send the verifier **a**, **b** — but this requires sending $2n$ scalars. Instead, the inner product argument uses $O(\log(n)) = O(k)$ communication cost.

Modified inner product argument



We have vectors **a**, **b** each of length $n = 2^k$.

Fix **b** = $(1, x, x^2, \ldots, x^{n-1})$ for a chosen evaluation point x, known to both prover and verifier. Since **b** is known, no vector **H** is needed.

We want to prove that v, which is an evaluation of **a** at x, and a commitment P are related as:

$$v = \langle \mathbf{a}, (1, x, x^2, \ldots, x^{n-1}) \rangle$$
$$P = \langle \mathbf{a}, \mathbf{G} \rangle + [r] H$$

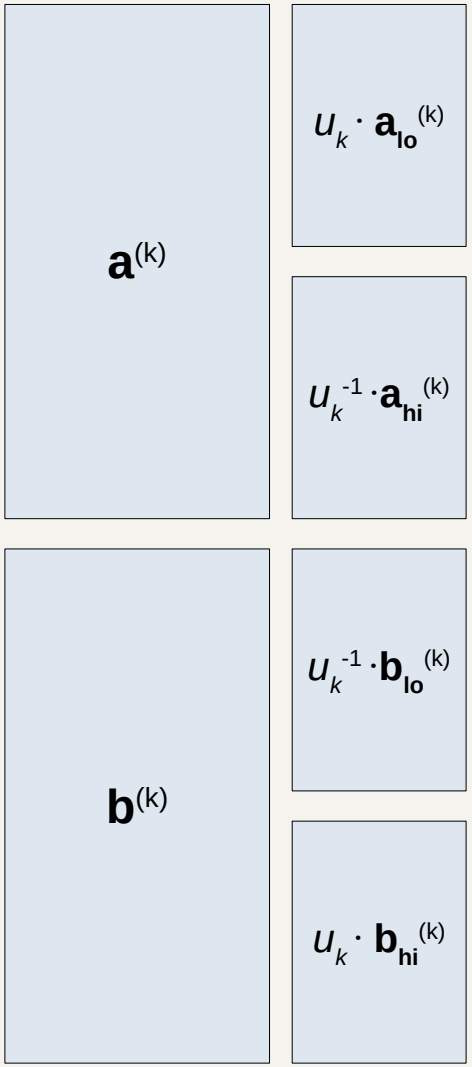
where **G** is a length-n vector of random group elements.

We can combine this into a single equation P_k using a random group element U:

$$P_k = P + [v] U = \langle \mathbf{a}, \mathbf{G} \rangle + [\langle \mathbf{a}, \mathbf{b} \rangle] U$$

Modified inner product argument

round k



We start at the k^{th} round, where we split **a**, **b**, **G** into **lo** and **hi** halves. We introduce a random challenge u_k and compress the vectors by adding the left and the right halves separated by u_k:

$$\mathbf{a}^{(k-1)} = u_k \cdot \mathbf{a}_{lo}^{(k)} + u_k^{-1} \cdot \mathbf{a}_{hi}^{(k)}$$

$$\mathbf{b}^{(k-1)} = u_k^{-1} \cdot \mathbf{b}_{lo}^{(k)} + u_k \cdot \mathbf{b}_{hi}^{(k)}$$

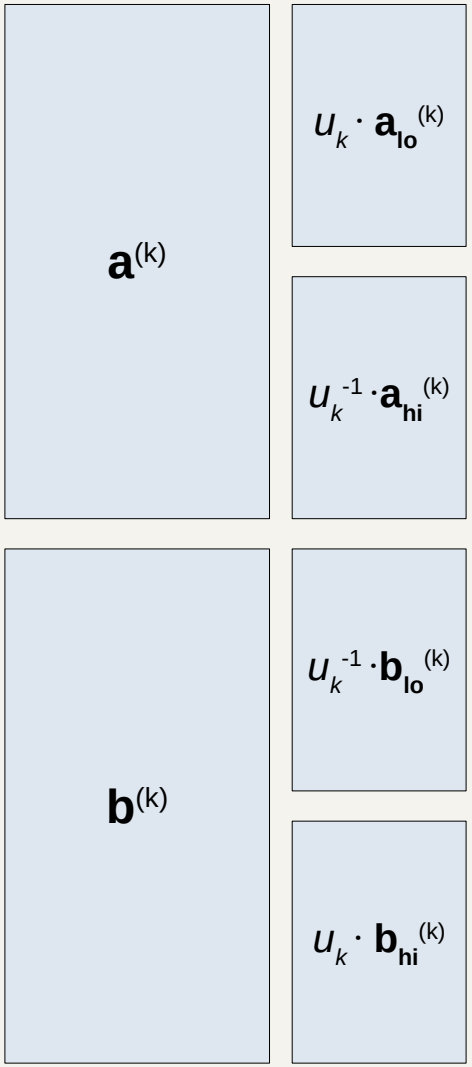
$$\mathbf{G}^{(k-1)} = [u_k^{-1}]\, \mathbf{G}_{lo}^{(k)} + [u_k]\, \mathbf{G}_{hi}^{(k)}$$

Now, we can write an equation P_{k-1} (of the same form as P_k) using the compressed vectors:

$$P_{k-1} = \langle \mathbf{a}^{(k-1)}, \mathbf{G}^{(k-1)} \rangle + [\langle \mathbf{a}^{(k-1)}, \mathbf{b}^{(k-1)} \rangle]\, U$$

Modified inner product argument

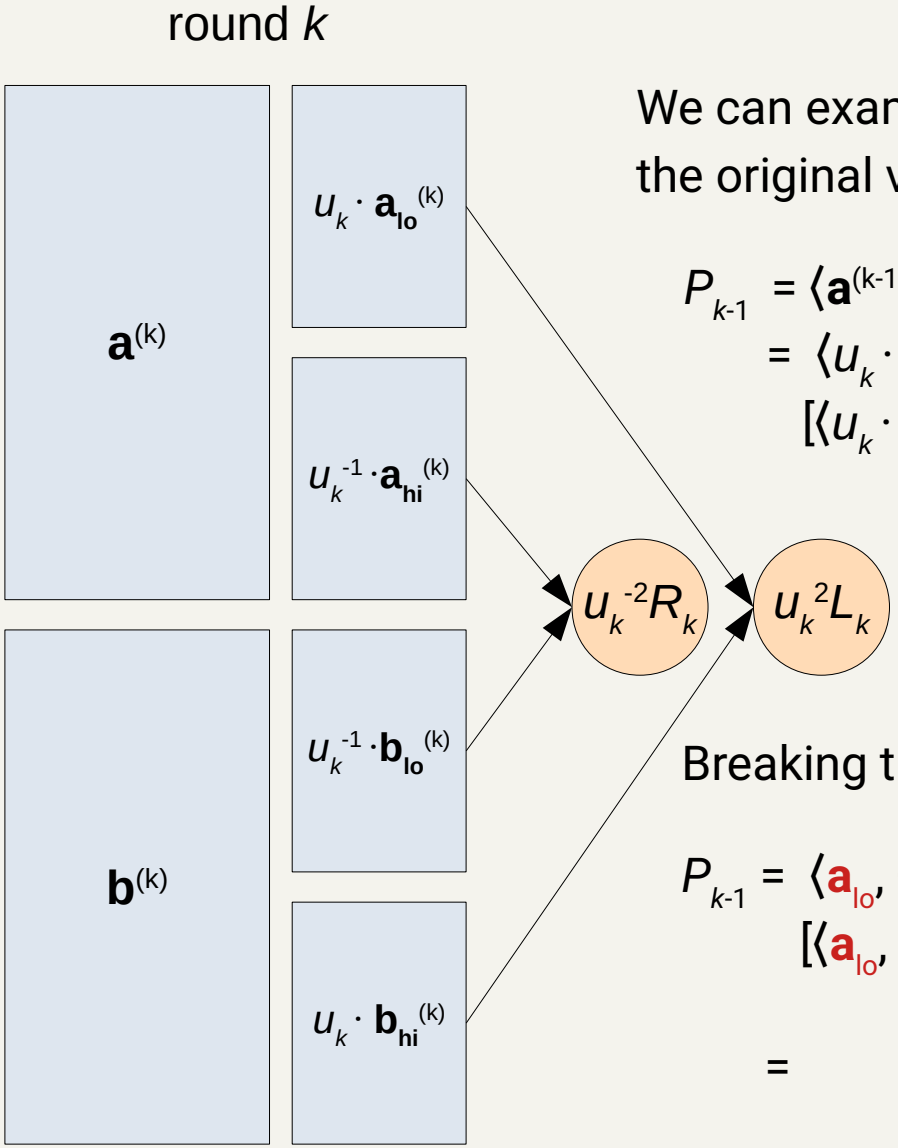
round k



We can examine the compressed P_{k-1} equation in terms of the original vectors:

$$
\begin{aligned}
P_{k-1} &= \langle \mathbf{a}^{(k-1)}, \mathbf{G}^{(k-1)} \rangle + [\langle \mathbf{a}^{(k-1)}, \mathbf{b}^{(k-1)} \rangle] \, U \\
&= \langle u_k \cdot \mathbf{a}_{lo} + u_k^{-1} \cdot \mathbf{a}_{hi}, [u_k^{-1}] \, \mathbf{G}_{lo} + [u_k] \, \mathbf{G}_{hi} \rangle + \\
&\quad [\langle u_k \cdot \mathbf{a}_{lo} + u_k^{-1} \cdot \mathbf{a}_{hi}, \; u_k^{-1} \cdot \mathbf{b}_{lo} + u_k \cdot \mathbf{b}_{hi} \rangle] \, U
\end{aligned}
$$

Modified inner product argument



We can examine the compressed $P_{k\text{-}1}$ equation in terms of the original vectors:

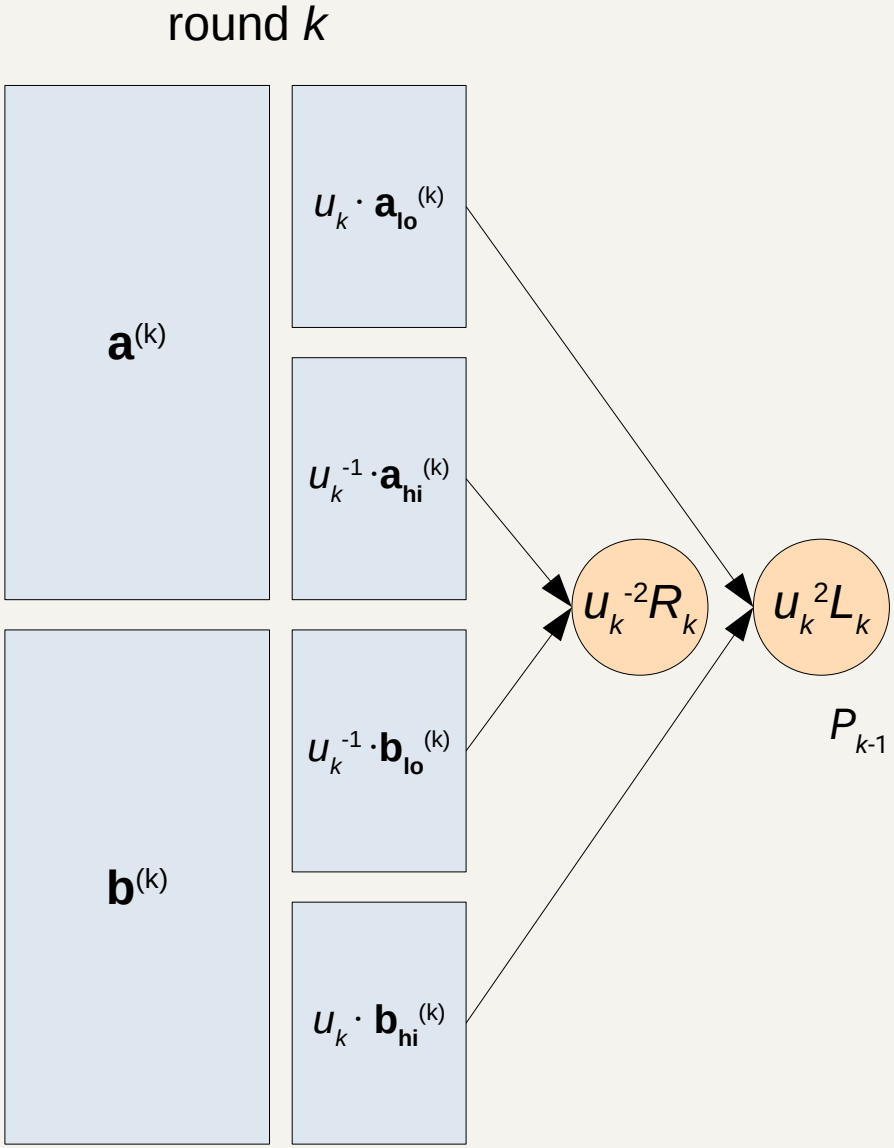
$$P_{k-1} = \langle \mathbf{a}^{(k-1)}, \mathbf{G}^{(k-1)} \rangle + [\langle \mathbf{a}^{(k-1)}, \mathbf{b}^{(k-1)} \rangle] U$$
$$= \langle u_k \cdot \mathbf{a}_{lo} + u_k^{-1} \cdot \mathbf{a}_{hi}, [u_k^{-1}] \mathbf{G}_{lo} + [u_k] \mathbf{G}_{hi} \rangle +$$
$$[\langle u_k \cdot \mathbf{a}_{lo} + u_k^{-1} \cdot \mathbf{a}_{hi}, \; u_k^{-1} \cdot \mathbf{b}_{lo} + u_k \cdot \mathbf{b}_{hi} \rangle] U$$

Breaking this down into simpler products:

$$P_{k-1} = \langle \mathbf{a}_{lo}, \mathbf{G}_{lo} \rangle + \langle \mathbf{a}_{hi}, \mathbf{G}_{hi} \rangle + u_k^2 \langle \mathbf{a}_{lo}, \mathbf{G}_{hi} \rangle + u_k^{-2} \langle \mathbf{a}_{hi}, \mathbf{G}_{lo} \rangle +$$
$$[\langle \mathbf{a}_{lo}, \mathbf{b}_{lo} \rangle + \langle \mathbf{a}_{hi}, \mathbf{b}_{hi} \rangle)] U + [u_k^2 \langle \mathbf{a}_{lo}, \mathbf{b}_{hi} \rangle + u_k^{-2} \langle \mathbf{a}_{hi}, \mathbf{b}_{lo} \rangle] U$$

$$= P_k + [u_k^2] L_k + [u_k^{-2}] R_k$$

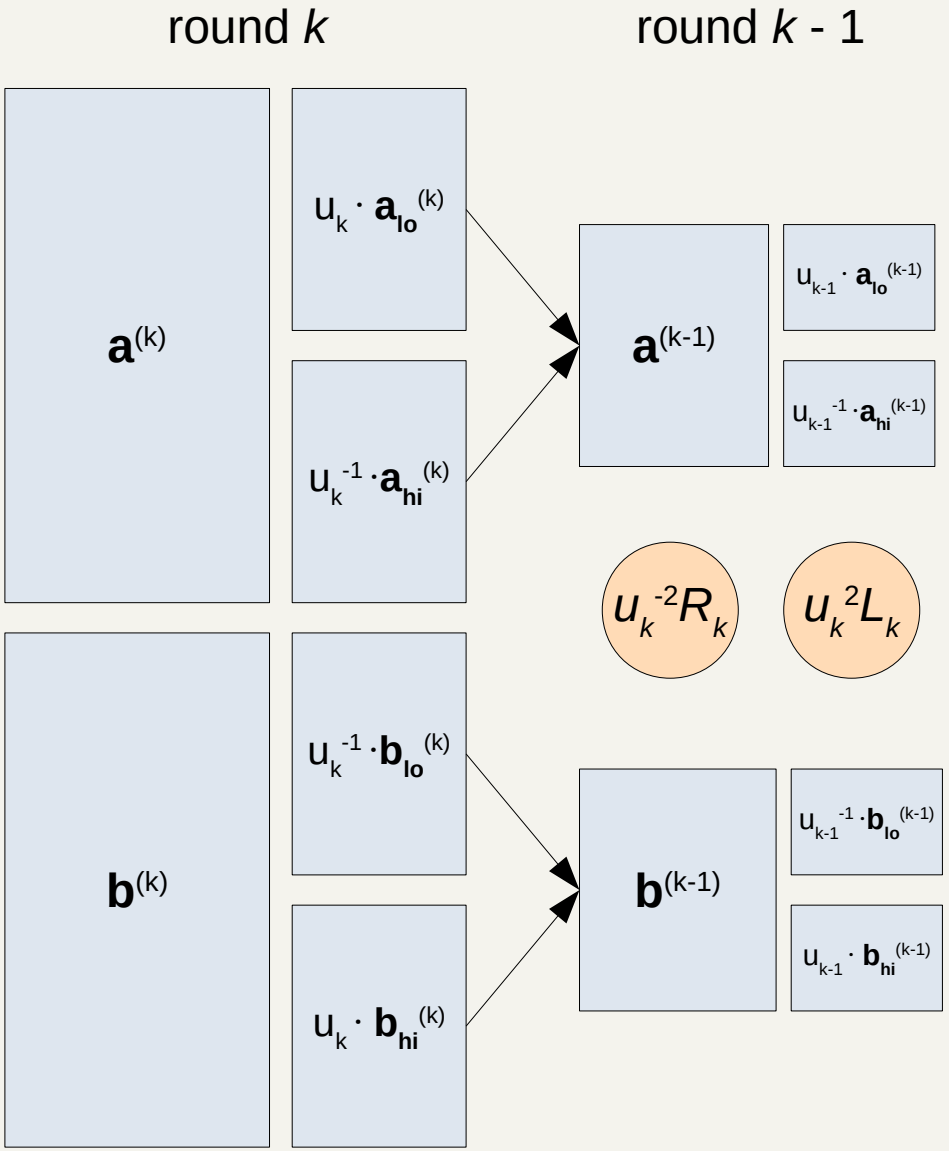
Modified inner product argument



$$P_{k-1} = P_k + [u_k^2]\, L_k + [u_k^{-2}]\, R_k$$

P_{k-1} is the sum of P_k and the cross-terms L_k, R_k (with coefficients from the round challenge u_k).

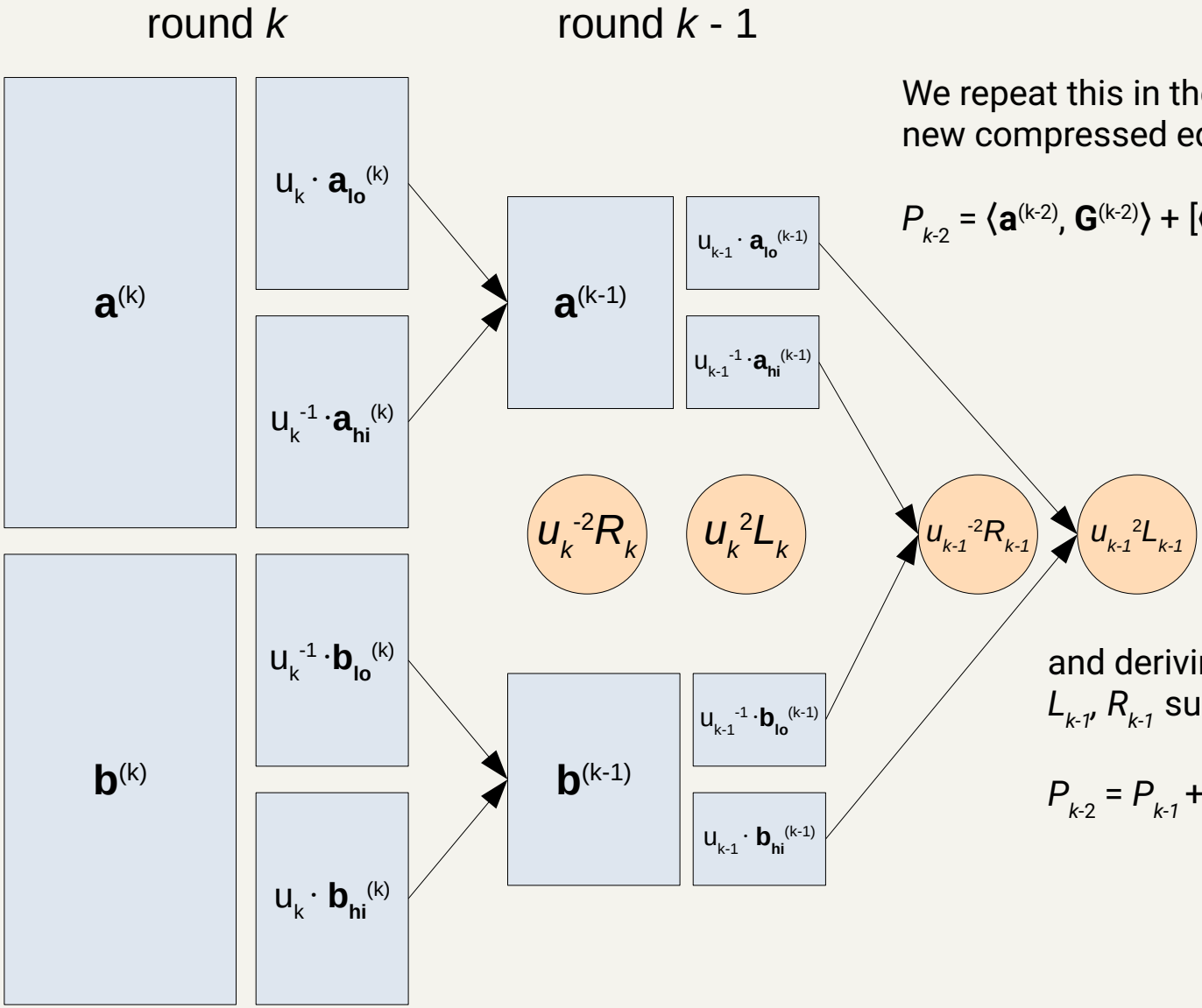
Modified inner product argument



We repeat this in the k-1st round, defining the new compressed equation

$$P_{k-2} = \langle \mathbf{a}^{(k-2)}, \mathbf{G}^{(k-2)} \rangle + [\langle \mathbf{a}^{(k-2)}, \mathbf{b}^{(k-2)} \rangle] \, U$$

Modified inner product argument



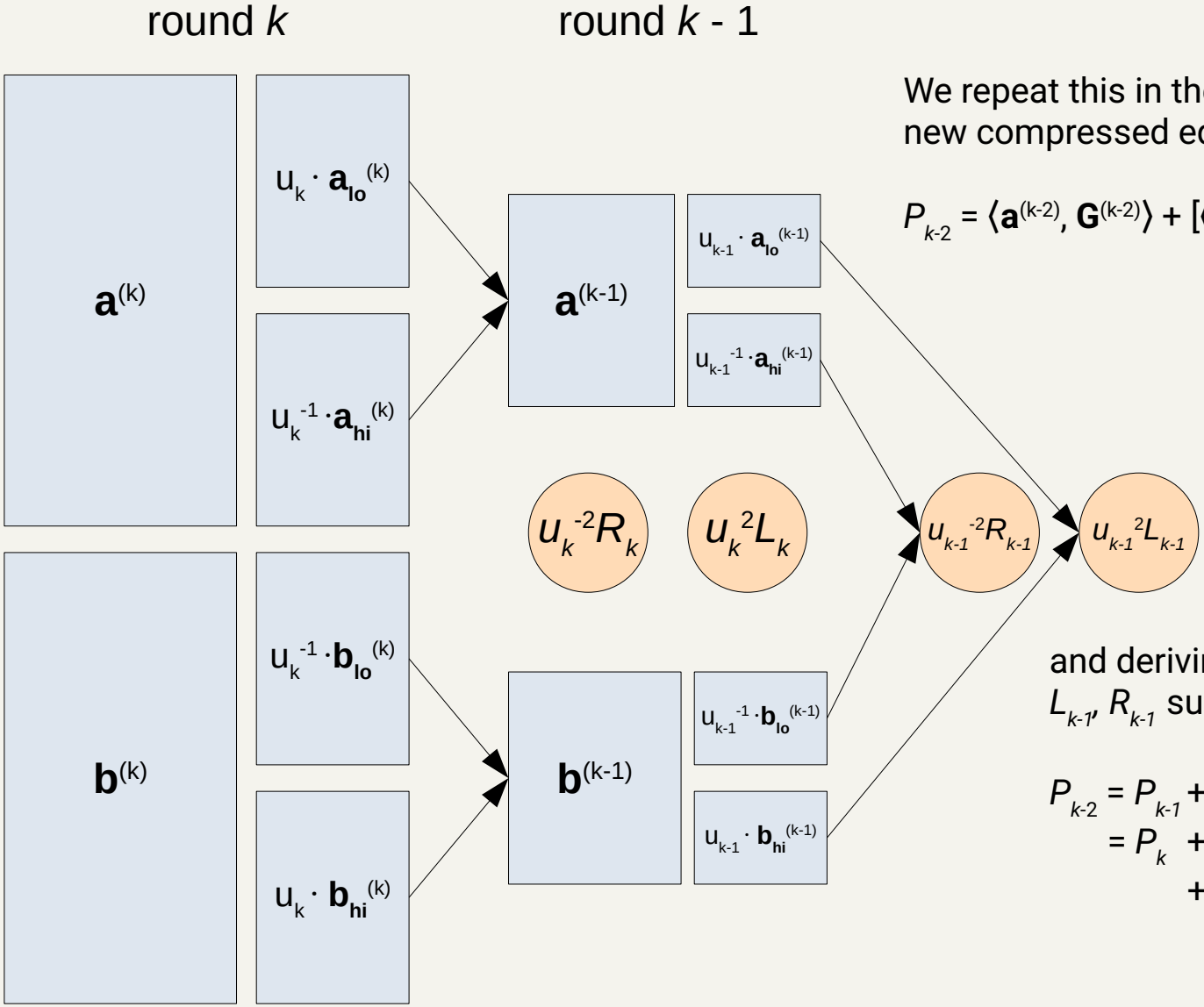
We repeat this in the k-1st round, defining the new compressed equation

$$P_{k-2} = \langle \mathbf{a}^{(k-2)}, \mathbf{G}^{(k-2)} \rangle + [\langle \mathbf{a}^{(k-2)}, \mathbf{b}^{(k-2)} \rangle] \, U$$

and deriving new cross-terms L_{k-1}, R_{k-1} such that

$$P_{k-2} = P_{k-1} + [u_{k-1}^{2}] \, L_{k-1} + [u_{k-1}^{-2}] \, R_{k-1}$$

Modified inner product argument



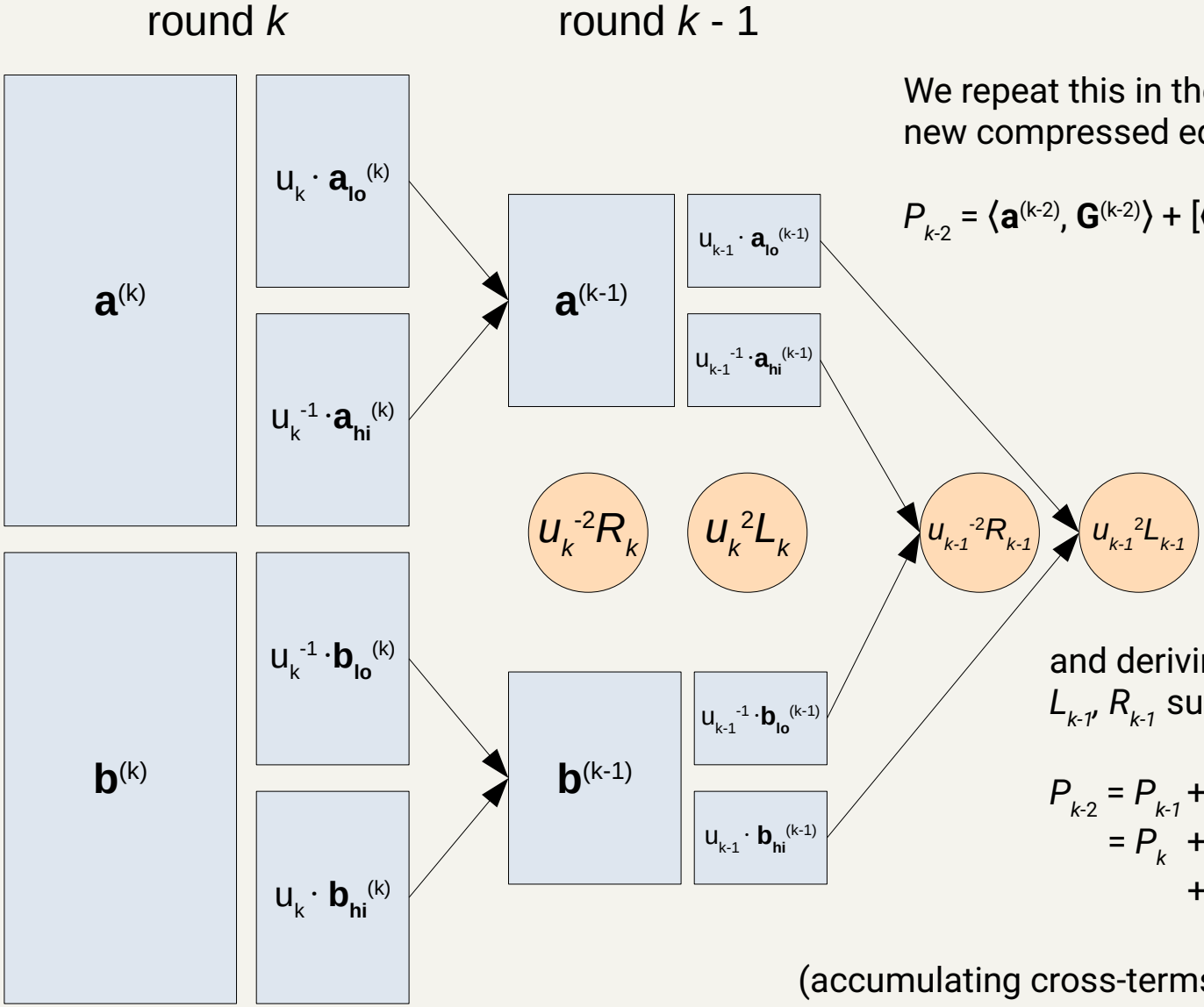
We repeat this in the k-1st round, defining the new compressed equation

$$P_{k-2} = \langle \mathbf{a}^{(k-2)}, \mathbf{G}^{(k-2)} \rangle + [\langle \mathbf{a}^{(k-2)}, \mathbf{b}^{(k-2)} \rangle] \, U$$

and deriving new cross-terms L_{k-1}, R_{k-1} such that

$$\begin{aligned} P_{k-2} &= P_{k-1} + [u_{k-1}^{2}] \, L_{k-1} + [u_{k-1}^{-2}] \, R_{k-1} \\ &= P_k + [u_k^{2}] \, L_k + [u_k^{-2}] \, R_k \\ &\quad + [u_{k-1}^{2}] \, L_{k-1} + [u_{k-1}^{-2}] \, R_{k-1} \end{aligned}$$

Modified inner product argument



We repeat this in the k-1st round, defining the new compressed equation

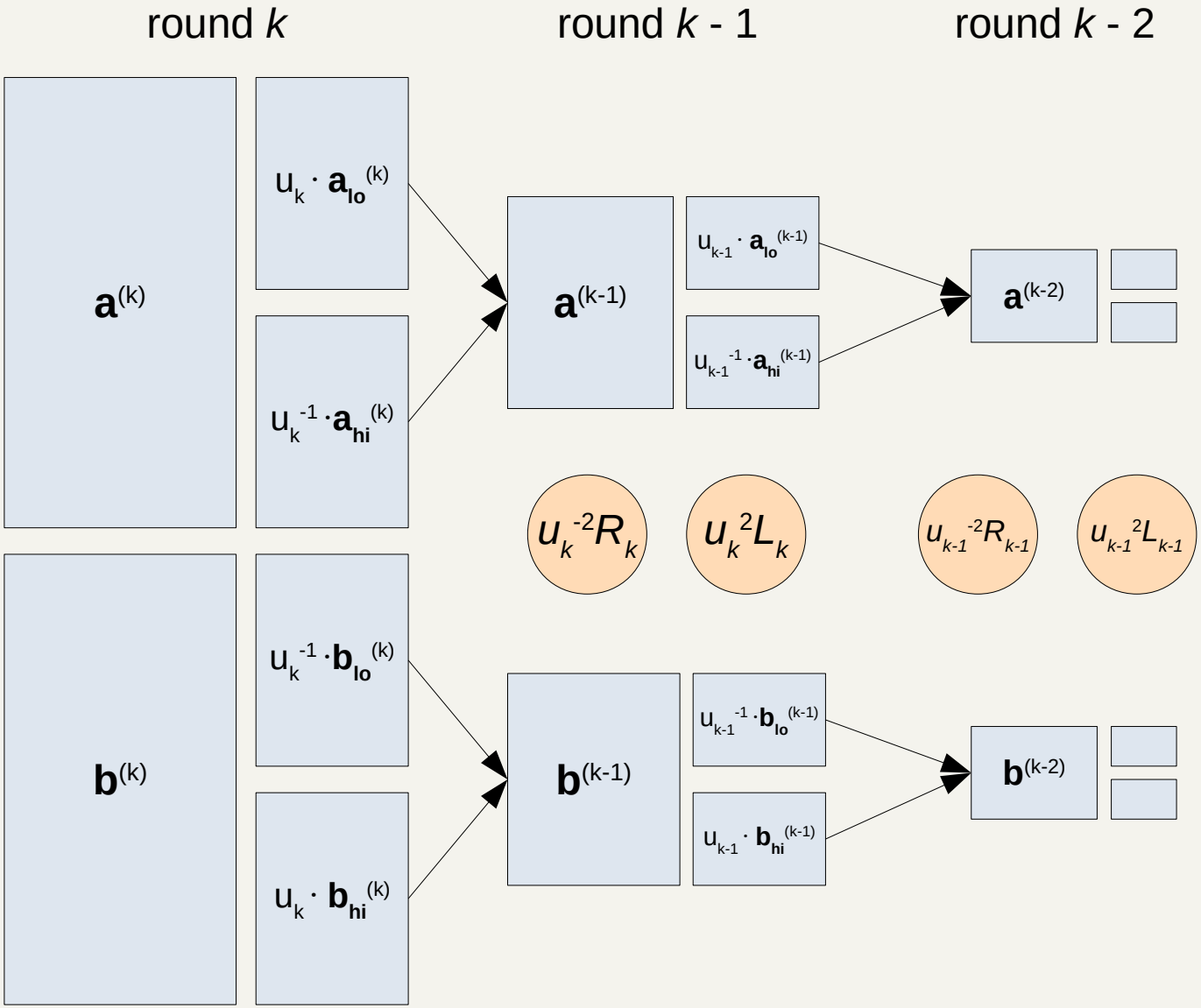
$$P_{k-2} = \langle \mathbf{a}^{(k-2)}, \mathbf{G}^{(k-2)} \rangle + [\langle \mathbf{a}^{(k-2)}, \mathbf{b}^{(k-2)} \rangle] \, U$$

and deriving new cross-terms L_{k-1}, R_{k-1} such that

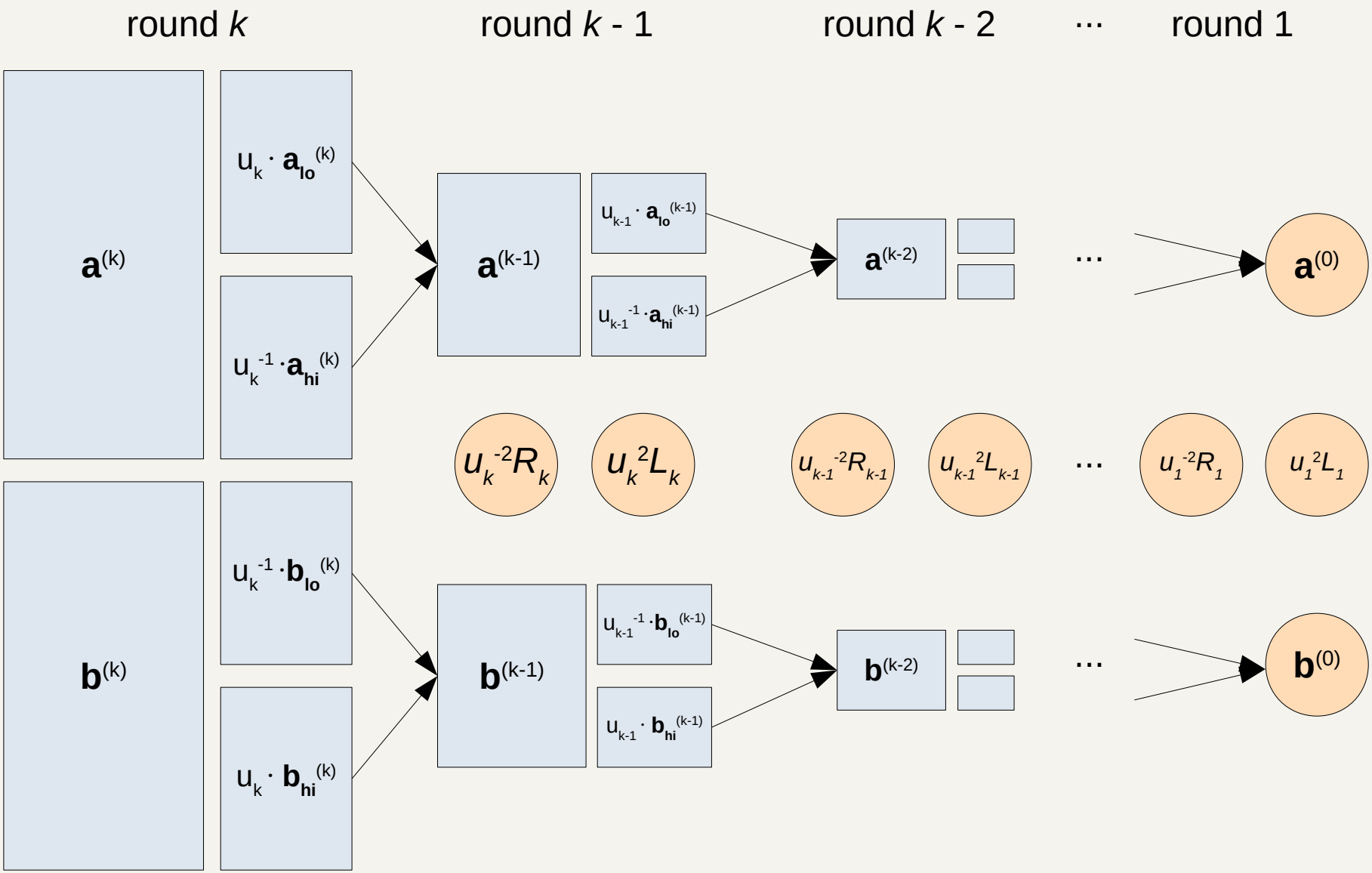
$$\begin{aligned} P_{k-2} &= P_{k-1} + [u_{k-1}^{2}] \, L_{k-1} + [u_{k-1}^{-2}] \, R_{k-1} \\ &= P_k + [u_k^{2}] \, L_k + [u_k^{-2}] \, R_k \\ &\quad + [u_{k-1}^{2}] \, L_{k-1} + [u_{k-1}^{-2}] \, R_{k-1} \end{aligned}$$

(accumulating cross-terms from previous round)

Modified inner product argument



Modified inner product argument



Modified inner product argument

round k | round k - 1 | round k - 2 | ⋯ | round 1

After k rounds, what we are left with is

$$P_0 = \mathbf{a}^{(0)}\mathbf{G}^{(0)} + [\mathbf{a}^{(0)}\mathbf{b}^{(0)}]\, U$$

$\mathbf{a}^{(0)}$

with $\mathbf{a}^{(0)}, \mathbf{G}^{(0)}, \mathbf{b}^{(0)}$ each containing a single term.

$u_k^{-2}R_k$ $u_k^{2}L_k$ $u_{k-1}^{-2}R_{k-1}$ $u_{k-1}^{2}L_{k-1}$ ⋯ $u_1^{-2}R_1$ $u_1^{2}L_1$

$\mathbf{b}^{(0)}$

Modified inner product argument

round k | round k - 1 | round k - 2 | ⋯ | round 1

After k rounds, what we are left with is

$$P_0 = \mathbf{a}^{(0)}\mathbf{G}^{(0)} + [\mathbf{a}^{(0)}\mathbf{b}^{(0)}]\, U$$

$\mathbf{a}^{(0)}$

with $\mathbf{a}^{(0)}, \mathbf{G}^{(0)}, \mathbf{b}^{(0)}$ each containing a single term.

$u_k^{-2}R_k$ $u_k^{2}L_k$ $u_{k-1}^{-2}R_{k-1}$ $u_{k-1}^{2}L_{k-1}$ … $u_1^{-2}R_1$ $u_1^{2}L_1$

Following the pattern of the previous rounds, we also know that

$$P_0 = P_k + \sum ([u_j^2]\, L_j + [u_j^{-2}]\, R_j) \quad \text{for } j = 1, \ldots, k$$

$\mathbf{b}^{(0)}$

meaning it contains all accumulated cross-terms at each step.

Modified inner product argument

round k | round k - 1 | round k - 2 | ⋯ | round 1

After k rounds, what we are left with is

$$P_0 = \mathbf{a}^{(0)}\mathbf{G}^{(0)} + [\mathbf{a}^{(0)}\mathbf{b}^{(0)}]\, U$$

$\mathbf{a}^{(0)}$

with $\mathbf{a}^{(0)}, \mathbf{G}^{(0)}, \mathbf{b}^{(0)}$ each containing a single term.

$u_k^{-2}R_k$ $u_k^{2}L_k$ $u_{k-1}^{-2}R_{k-1}$ $u_{k-1}^{2}L_{k-1}$ … $u_1^{-2}R_1$ $u_1^{2}L_1$

Following the pattern of the previous rounds, we also know that

$$P_0 = P_k + \sum ([u_j^2]\, L_j + [u_j^{-2}]\, R_j) \quad \text{for } j = 1, \ldots, k$$

$\mathbf{b}^{(0)}$

meaning it contains all accumulated cross-terms at each step.

The prover only has to send $\{L_j, R_j\}$, a_0 since b and G are known.

Modified inner product argument

round k | round k - 1 | round k - 2 | ⋯ | round 1

After k rounds, what we are left with is

$$P_0 = \mathbf{a}^{(0)}\mathbf{G}^{(0)} + \mathbf{a}^{(0)}\mathbf{b}^{(0)} \cdot Q$$

$\mathbf{a}^{(0)}$

with $\mathbf{a}^{(0)}, \mathbf{G}^{(0)}, \mathbf{b}^{(0)}$ each containing a single term.

$u_k^{-2}R_k$ $u_k^{2}L_k$ $u_{k-1}^{-2}R_{k-1}$ $u_{k-1}^{2}L_{k-1}$ ⋯ $u_1^{-2}R_1$ $u_1^{2}L_1$

Following the pattern of the previous rounds, we also know that

$$P_0 = P_k + \sum ([u_j^2]\, L_j + [u_j^{-2}]\, R_j) \quad \text{for } j = 1, \ldots, k$$

$\mathbf{b}^{(0)}$

meaning it contains all accumulated cross-terms at each step.

The prover only has to send $\{L_j, R_j\}$, a_0 since b and G are known. The verifier can check P_k with a communication complexity of $2k + 1 = O(k)$ terms.

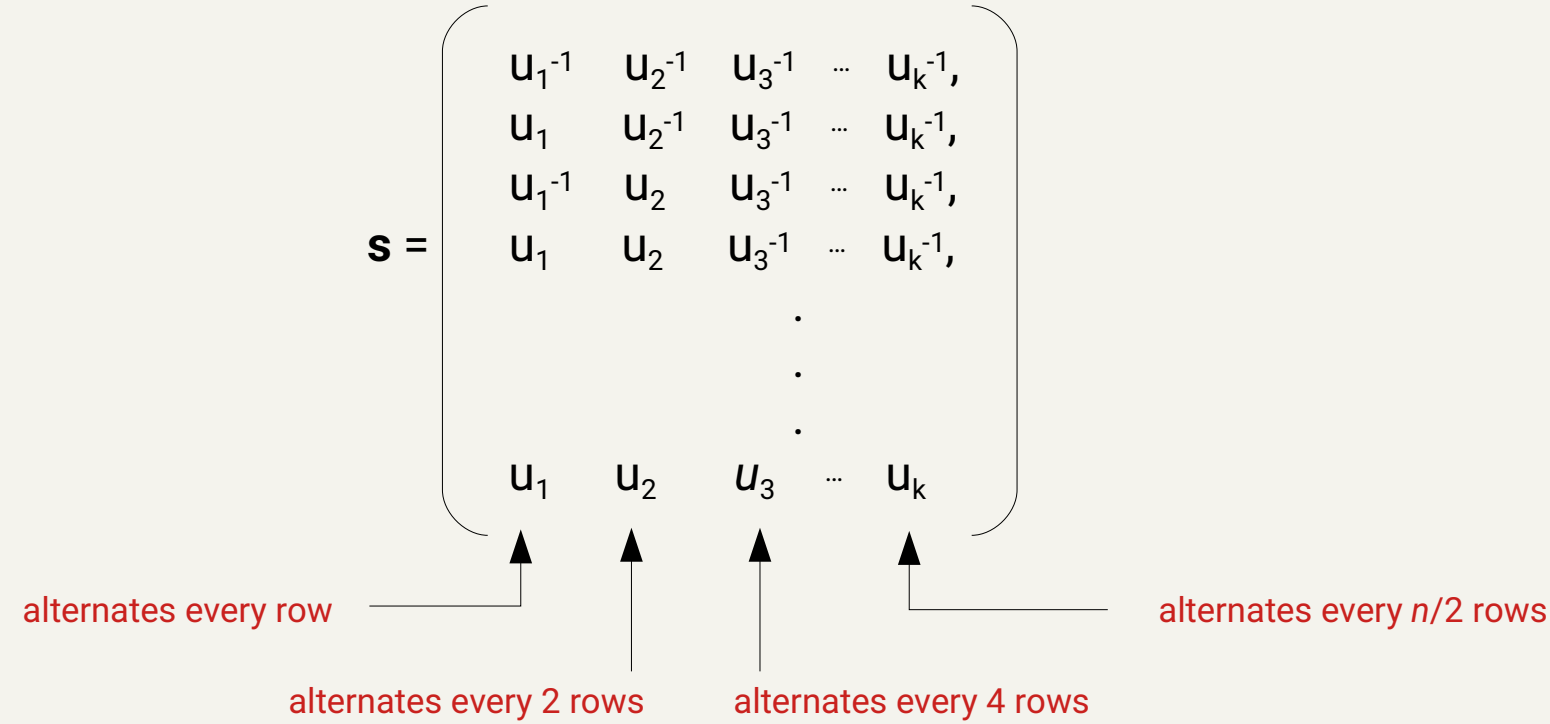
Modified inner product argument

Note that the verifier can compute $G = \mathbf{G}^{(0)}$ as $\langle \mathbf{s}, \mathbf{G} \rangle$ and $b = \mathbf{b}^{(0)}$ as $\langle \mathbf{s}, \mathbf{b} \rangle$ where

$$\mathbf{s} = \begin{pmatrix} u_1^{-1} & u_2^{-1} & u_3^{-1} & \cdots & u_k^{-1}, \\ u_1 & u_2^{-1} & u_3^{-1} & \cdots & u_k^{-1}, \\ u_1^{-1} & u_2 & u_3^{-1} & \cdots & u_k^{-1}, \\ u_1 & u_2 & u_3^{-1} & \cdots & u_k^{-1}, \\ & & \vdots & & \\ u_1 & u_2 & u_3 & \cdots & u_k \end{pmatrix}$$

Modified inner product argument

Note that the verifier can compute $G = \mathbf{G}^{(0)}$ as $\langle \mathbf{s}, \mathbf{G} \rangle$ and $b = \mathbf{b}^{(0)}$ as $\langle \mathbf{s}, \mathbf{b} \rangle$ where



which has a binary counting structure arising from the fact that in each round the inverted challenges are used to scale bases in the first half $\mathbf{G}_{\mathbf{lo}}$, while the ordinary challenges scale the bases in the second half $\mathbf{G}_{\mathbf{hi}}$.

Modified inner product argument

- $b = \langle \mathbf{s}, \mathbf{b} \rangle$ can be computed by the verifier in logarithmic time. This is achieved by defining a polynomial

$$g(X, u_1, u_2, \ldots, u_k) = \prod (u_i + u_i^{-1} X^{2i-1}),\ i = 1, \ldots, k$$

  such that $b = \langle \mathbf{s}, \mathbf{b} \rangle = g(x, u_1, u_2, \ldots, u_k)$ (recall $\mathbf{b} = (1, x, x^2, \ldots, x^{n-1})$).

- However, $G = \langle \mathbf{s}, \mathbf{G} \rangle$ still has to be computed in linear time:

$$\langle \mathbf{s}, \mathbf{G} \rangle = \begin{pmatrix} u_1^{-1} & u_2^{-1} & u_3^{-1} & \cdots & u_k^{-1}, \\ u_1 & u_2^{-1} & u_3^{-1} & \cdots & u_k^{-1}, \\ u_1^{-1} & u_2 & u_3^{-1} & \cdots & u_k^{-1}, \\ & & \vdots & & \\ u_1 & u_2 & u_3 & \cdots & u_k \end{pmatrix} \begin{pmatrix} \mathbf{g}_0, \\ \mathbf{g}_1, \\ \mathbf{g}_2, \\ \vdots \\ \mathbf{g}_{n-1} \end{pmatrix}$$

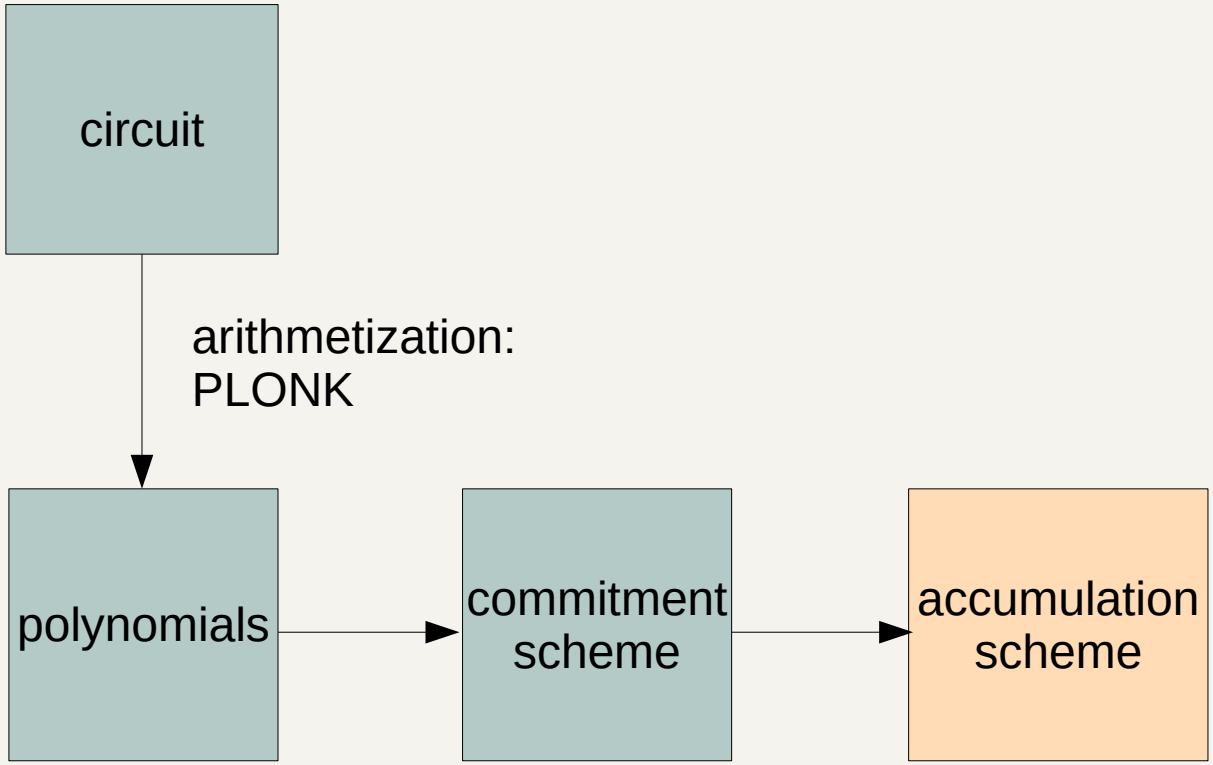
Modified inner product argument

- $b = \langle \mathbf{s}, \mathbf{b} \rangle$ can be computed by the verifier in logarithmic time. This is achieved by defining a polynomial
  - $g(X, u_1, u_2, \ldots, u_k) = \prod (u_i + u_i^{-1} X^{2i-1}), i = 1, \ldots, k$
- such that $b = \langle \mathbf{s}, \mathbf{b} \rangle = g(x, u_1, u_2, \ldots, u_k)$ (recall $\mathbf{b} = (1, x, x^2, \ldots, x^{n-1})$).
- However, $G = \langle \mathbf{s}, \mathbf{G} \rangle$ still has to be computed in linear time:

$$\langle \mathbf{s}, \mathbf{G} \rangle = \begin{pmatrix} u_1^{-1} & u_2^{-1} & u_3^{-1} & \cdots & u_k^{-1}, \\ u_1 & u_2^{-1} & u_3^{-1} & \cdots & u_k^{-1}, \\ u_1^{-1} & u_2 & u_3^{-1} & \cdots & u_k^{-1}, \\ & & \vdots & & \\ u_1 & u_2 & u_3 & \cdots & u_k \end{pmatrix} \begin{pmatrix} \mathbf{g}_0, \\ \mathbf{g}_1, \\ \mathbf{g}_2, \\ \vdots \\ \mathbf{g}_{n-1} \end{pmatrix}$$

We will amortise this final linear-time operation across multiple instances using an **accumulation scheme**.

Halo 2


circuit
arithmetization:
PLONK
polynomials
commitment
scheme
accumulation
scheme

Incrementally Verifiable Computation

- In IVC we have a computation with a “base step” and an “inductive step”. (PCD generalizes this to graphs of computation.)
- At any point we can represent the computation so far with a short proof.
- We can extend the computation incrementally. The new proof will attest to the validity of all previous steps.
- If a proof system is fully succinct (has both sublinear proof size and sublinear verification time), then there will be some circuit size at which the system can verify one of its own proofs.
- The main observation of Halo is that even if a NARK (non-interactive argument of knowledge) does not have sublinear verification time, it can still support recursive verification, provided (informally) that we can “compress” two proof states into one.
- This can be formalized using the concept of an “accumulation scheme”.

Accumulation schemes

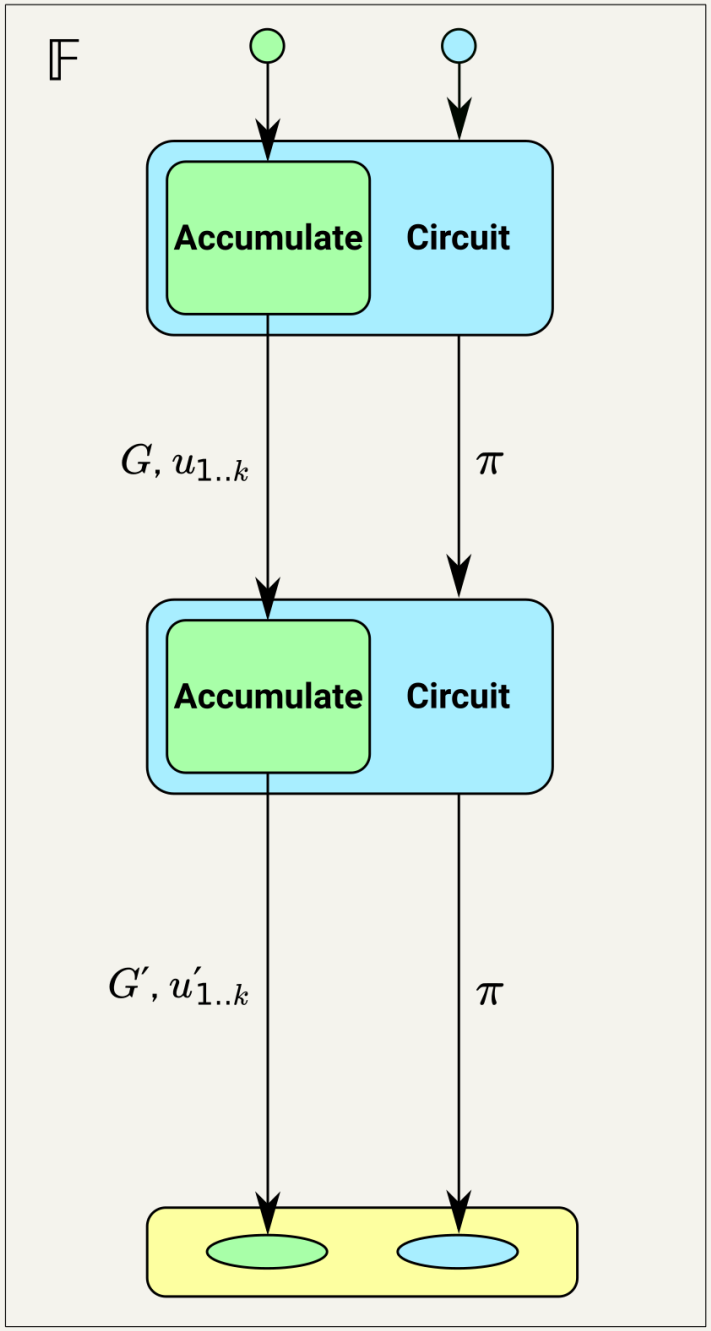
- In an accumulation scheme, it is expensive to "decide" validity of an instance, but cheap to combine two (or more) instances.
- This is how the idea is described in [BCMS2020] (ia.cr/2020/499).

Accumulator	Accumulation step	Decider
G, $\{u_1, u_2, \ldots, u_k\}$	Polynomial commitment opening of G. ($O(\log n)$) Accumulator is replaced with fresh instance of G', $\{u_1', u_2', \ldots, u_k'\}$	Check (in $O(n)$ time) that $G = \text{Commit}(\mathbf{g}(X, u_1, \ldots, u_k)) = \langle \mathbf{s}, \mathbf{G} \rangle$

Instead of computing $G = \langle \mathbf{s}, \mathbf{G} \rangle$ in the circuit, the prover supplies the purported value G as part of their witness, along with the challenges $\{u_1, u_2, \ldots, u_k\}$ which define $\mathbf{s}$.

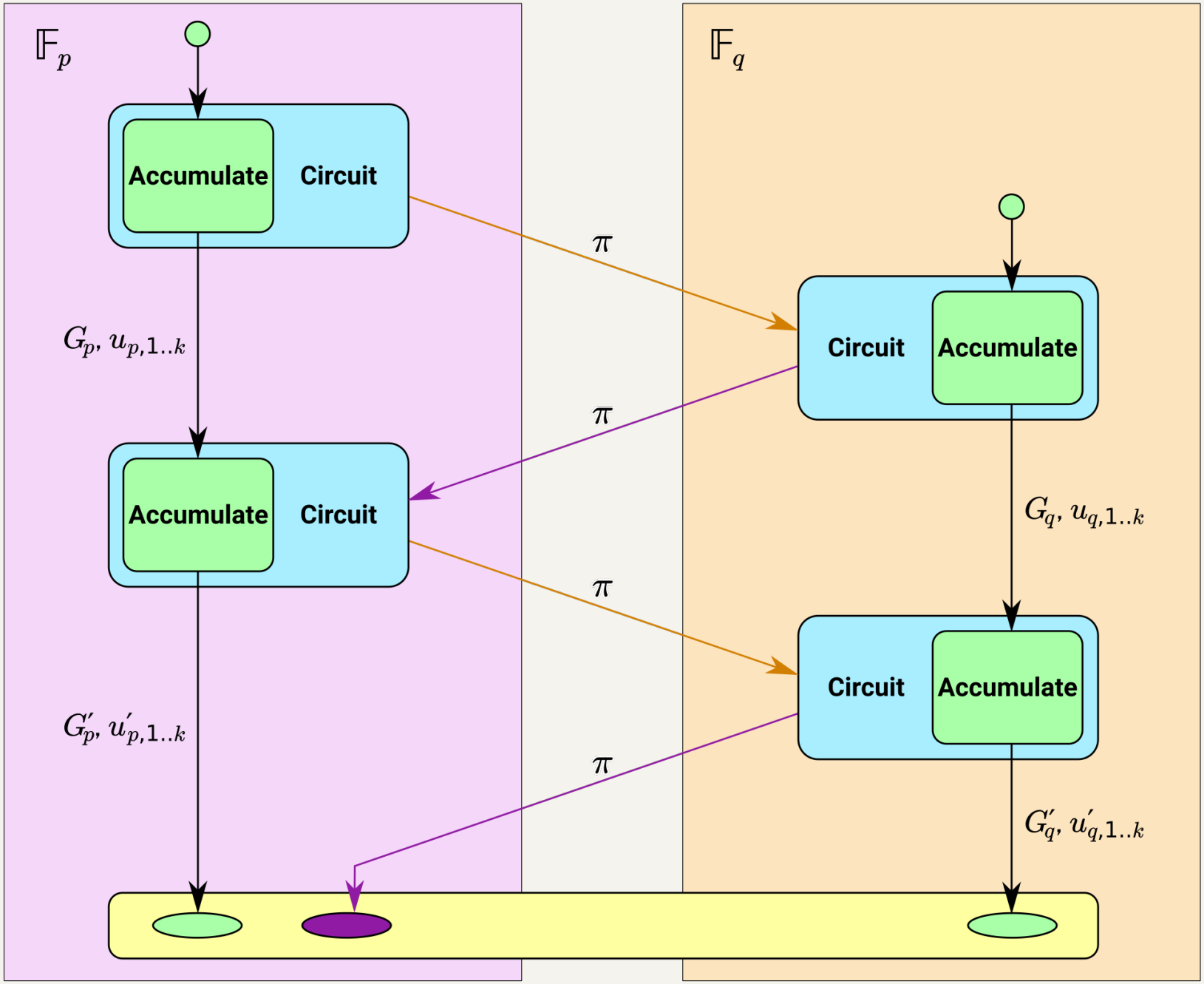
This value G is used to perform the rest of the checks.

Accumulation schemes

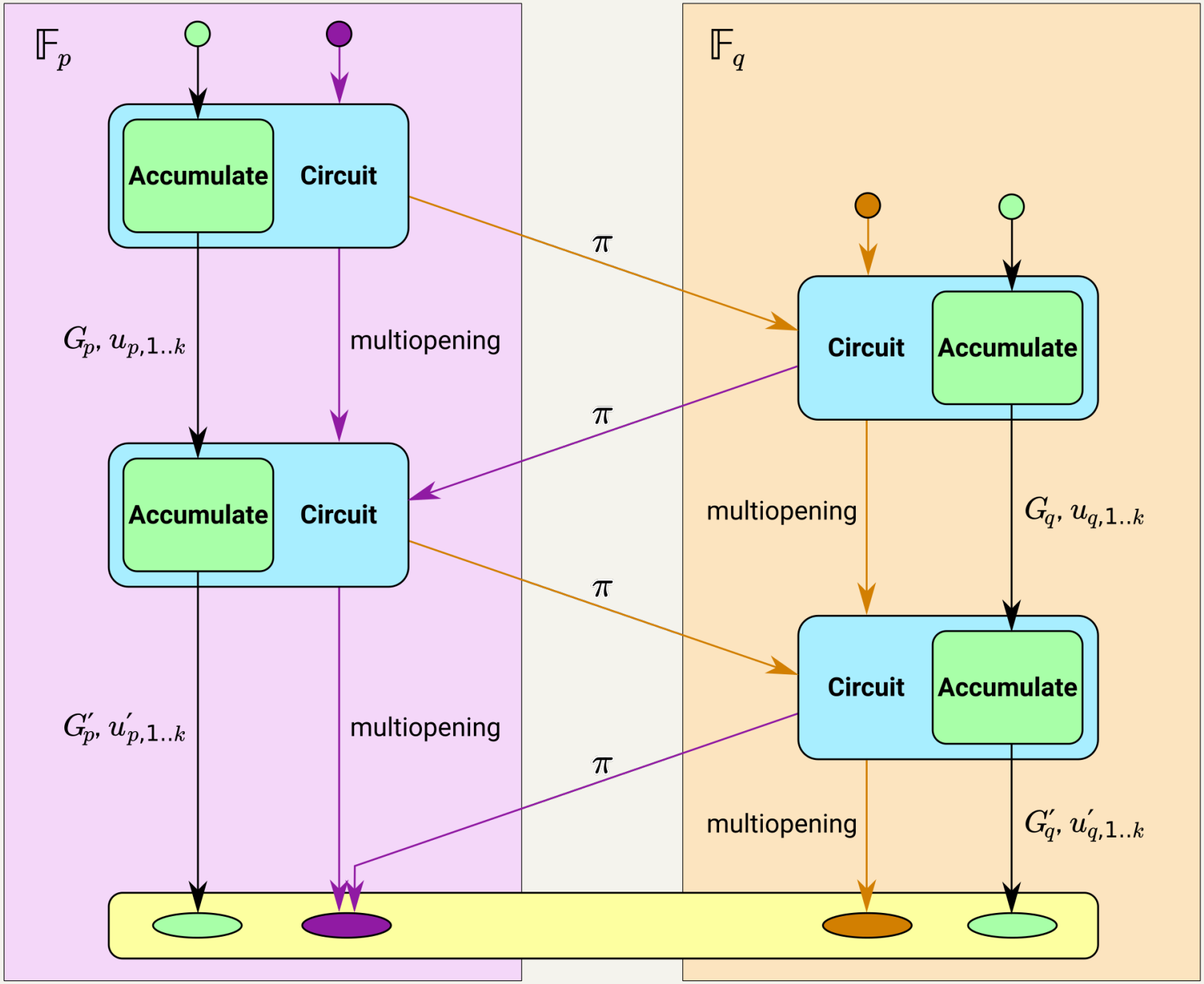


- In practice, accumulation schemes and proof systems use arithmetic in a particular field.
- “Wrong-field” arithmetic pays a circuit size penalty of ~100 times (order of magnitude). We’ll explain why in more detail later in the talk.
- If we tried to literally implement what is shown in the diagram, we’d have to do wrong-field arithmetic in the circuit.

Accumulation schemes on 2-cycles

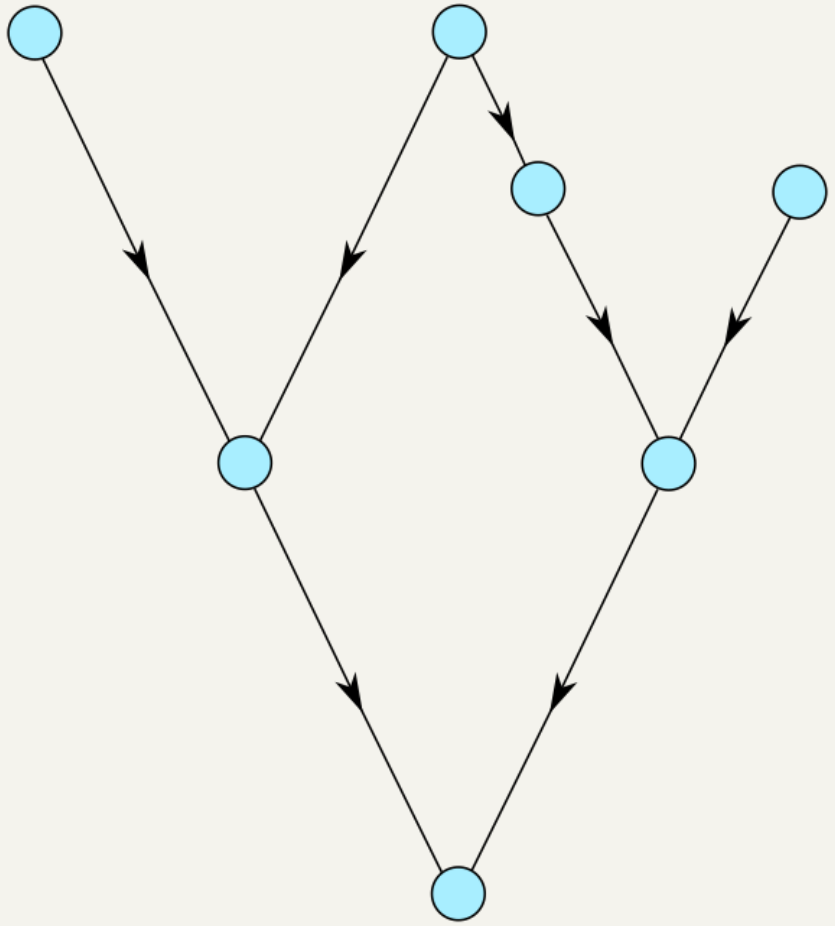


Accumulation schemes on 2-cycles



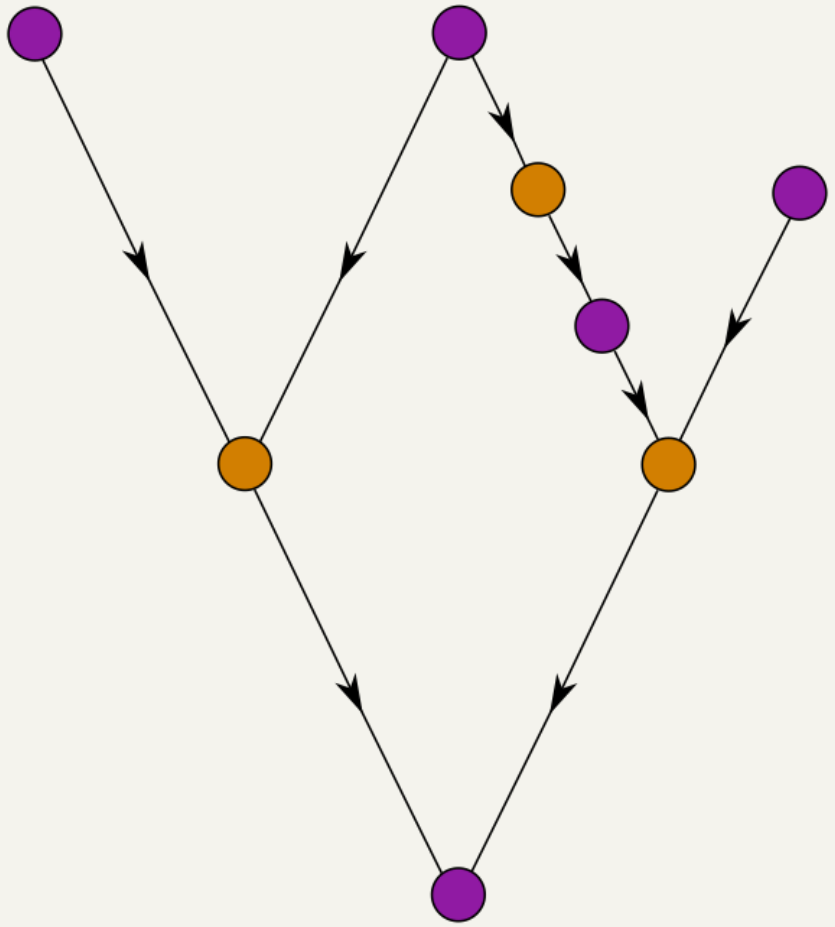
Encoding general Proof-Carrying Data

- Let’s consider how this goes for general PCD, where we have a directed acyclic graph of proofs.



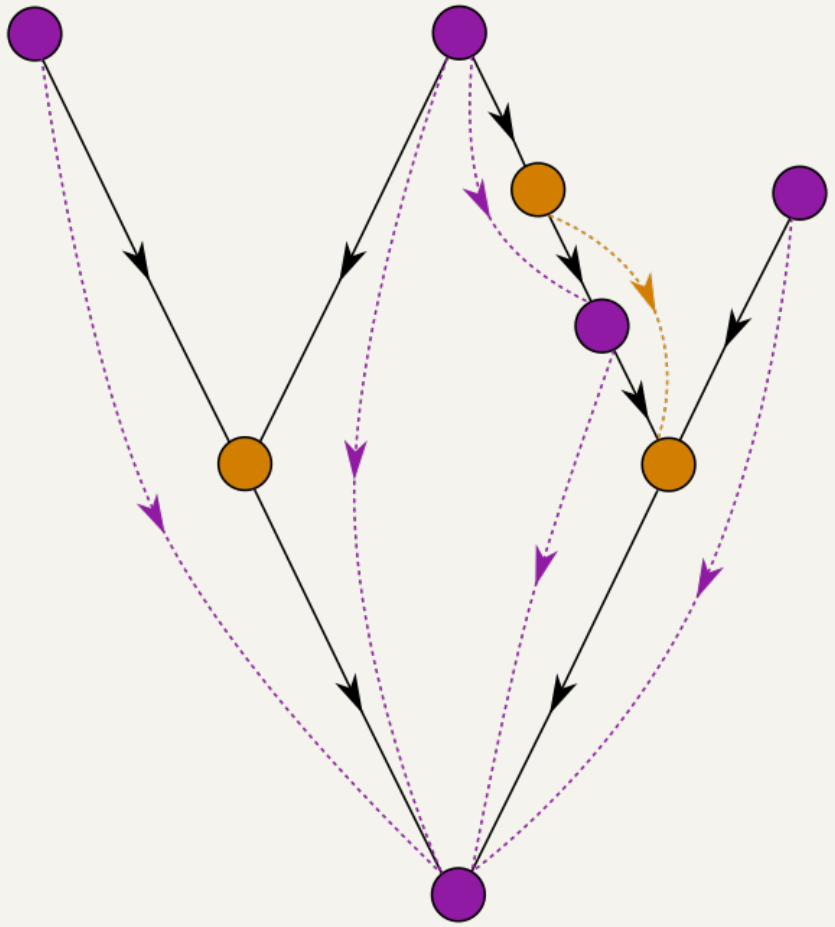
Encoding general Proof-Carrying Data

- We might need to add extra wrapper proofs so that $\mathbb{F}_p$ and $\mathbb{F}_q$ proofs alternate on all paths.



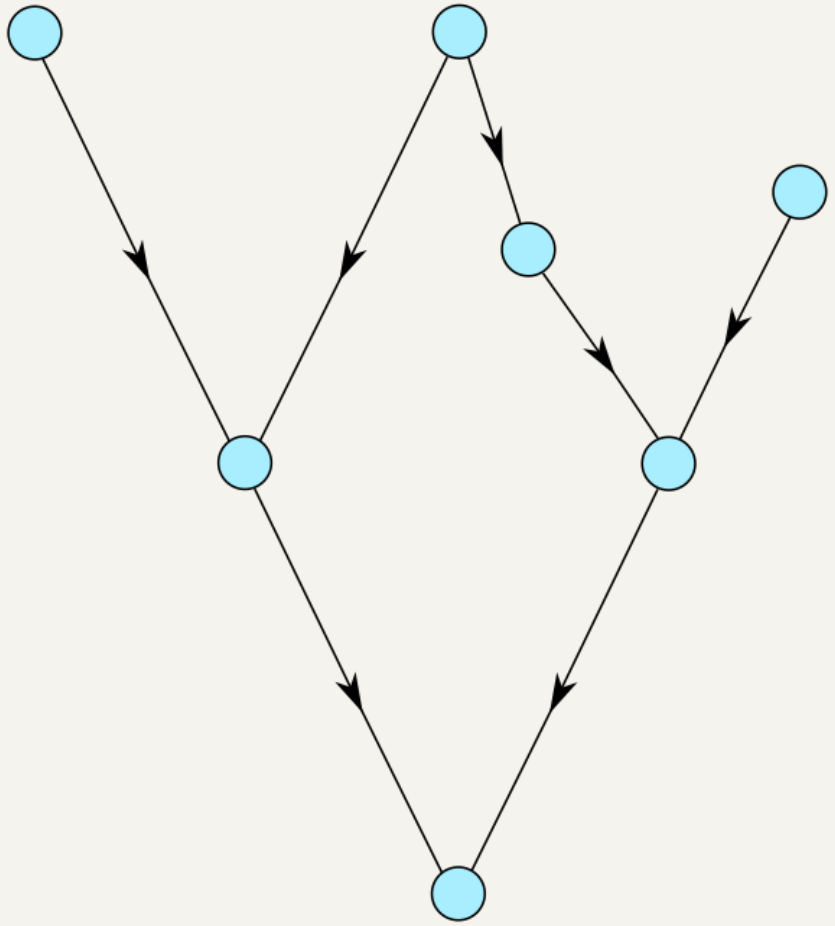
Encoding general Proof-Carrying Data

- We might need to add extra wrapper proofs so that $\mathbb{F}_p$ and $\mathbb{F}_q$ proofs alternate on all paths. Here we also show deferreds.



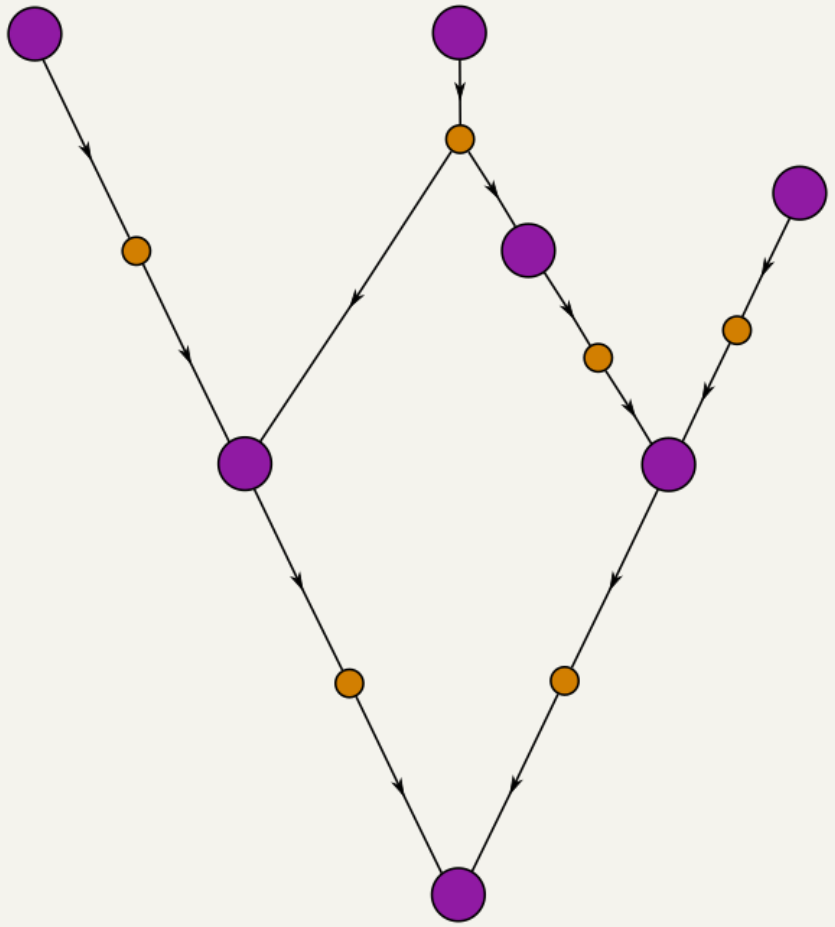
Encoding general Proof-Carrying Data

- Another option is to add wrapper proofs after every node.



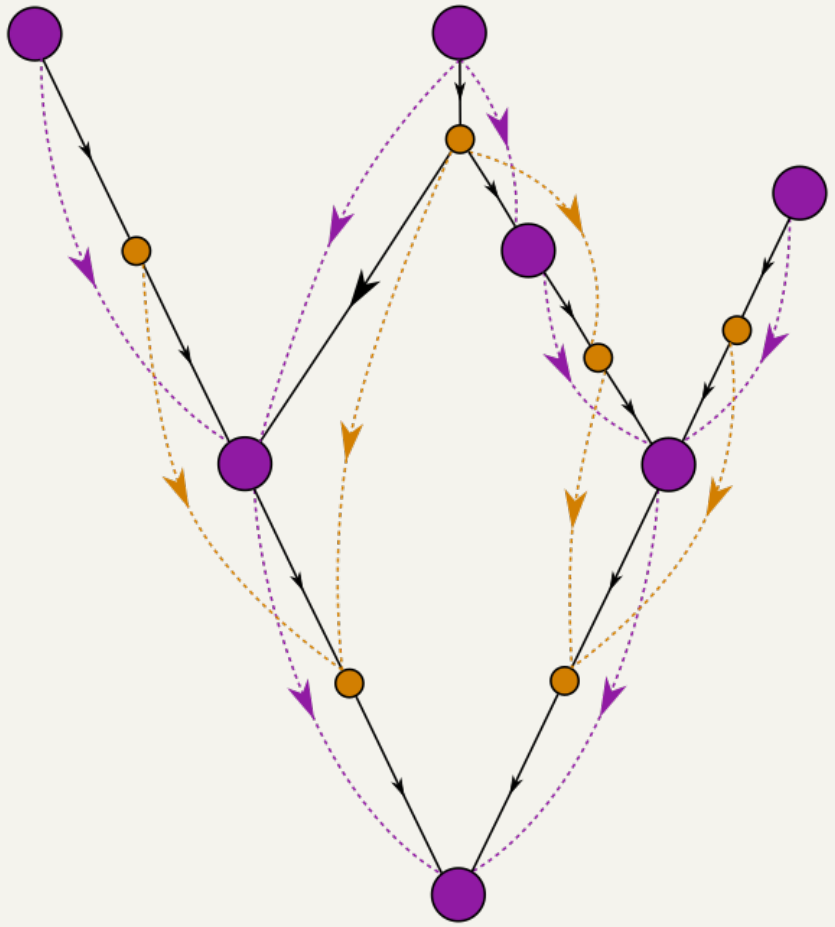
Encoding general Proof-Carrying Data

- Another option is to add wrapper proofs after every node, like this.



Encoding general Proof-Carrying Data

- The data flow for deferreds in PCD might look complicated, but it's fairly straightforward in practice: a proof just needs to include a lists of deferreds added in that proof, and a list of those added in its antecedent proof(s).



Elliptic curves

- An elliptic curve in affine form is:
  - a *group* of points (x, y)
  - over a *field of definition* $\mathbb{F}_p$
  - satisfying some cubic equation, say $y^2 = x^3 + ax + b$,
  - with (for this equation) a “point at infinity” as the group identity.
- If the group has a prime order q, we’ll name the curve $E_{p \to q}$.
- Because it’s a group, we can add points, or multiply them by a scalar.
- The *scalar field* is $\mathbb{F}_q$.
- We’ll assume that a proof system using $E_{p \to q}$ efficiently supports circuits with arithmetic over $\mathbb{F}_q$.
- A cycle of elliptic curves is a pair $E_{p \to q}$ and $E_{q \to p}$.

Motivation for cycles

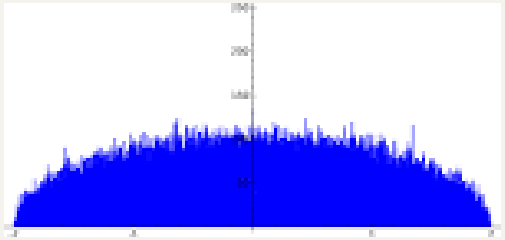
- A proof system using $E_{p \to q}$ directly supports arithmetic over $\mathbb{F}_q$.
- "Wrong-field" arithmetic has an overhead of over 1000 times.
  - This is using the sum of residues method for reduction: Zcash #4093
- Can't we solve this using lookups (e.g. Plookup)?
  - No; lookups help but wrong-field arithmetic probably still has an overhead of 10-100 times.
  - Nothing here conflicts with using lookups for other things in the same proof system.
- As we'll see later, it's not quite sufficient for recursive proofs to just do arithmetic in the field of definition of the other proof system. We'll need two instances of the proof system, on $E_{p \to q}$ and on $E_{q \to p}$.

The Tweedle cycle

- The Halo paper gives a pair of curves:
  - $E_{p \to q} : y^2 = x^3 + 5$ is called Tweedledum.
  - $E_{q \to p} : y^2 = x^3 + 5$ is called Tweedledee.
  - $p = 2^{254} + \texttt{0x38AA1276C3F59B9A14064E200000001}$
  - $q = 2^{254} + \texttt{0x38AA127696286C9842CAFD400000001}$
  - Both have 126-bit Pollard rho security, maximal embedding degree, and 2-adicity ≥ 33.
  - They have cubic endomorphisms (we'll explain what that means).
  - $\gcd(p - 1, 5) = \gcd(q - 1, 5) = 1$
- We're going to explain the construction that found them, and some generalizations of it that allow finding more complicated graphs of curves.

Constructing cycles – the problem

- By the Hasse bound, the order of an elliptic curve over $\mathbb{F}_p$ lies in the range $[\, p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}\,]$. For Tweedle this range is of size $\sim 2^{129}$.
- The Sato–Tate conjecture* concerns the distribution of the order in this range. We don't need to go into detail, but here's a picture:

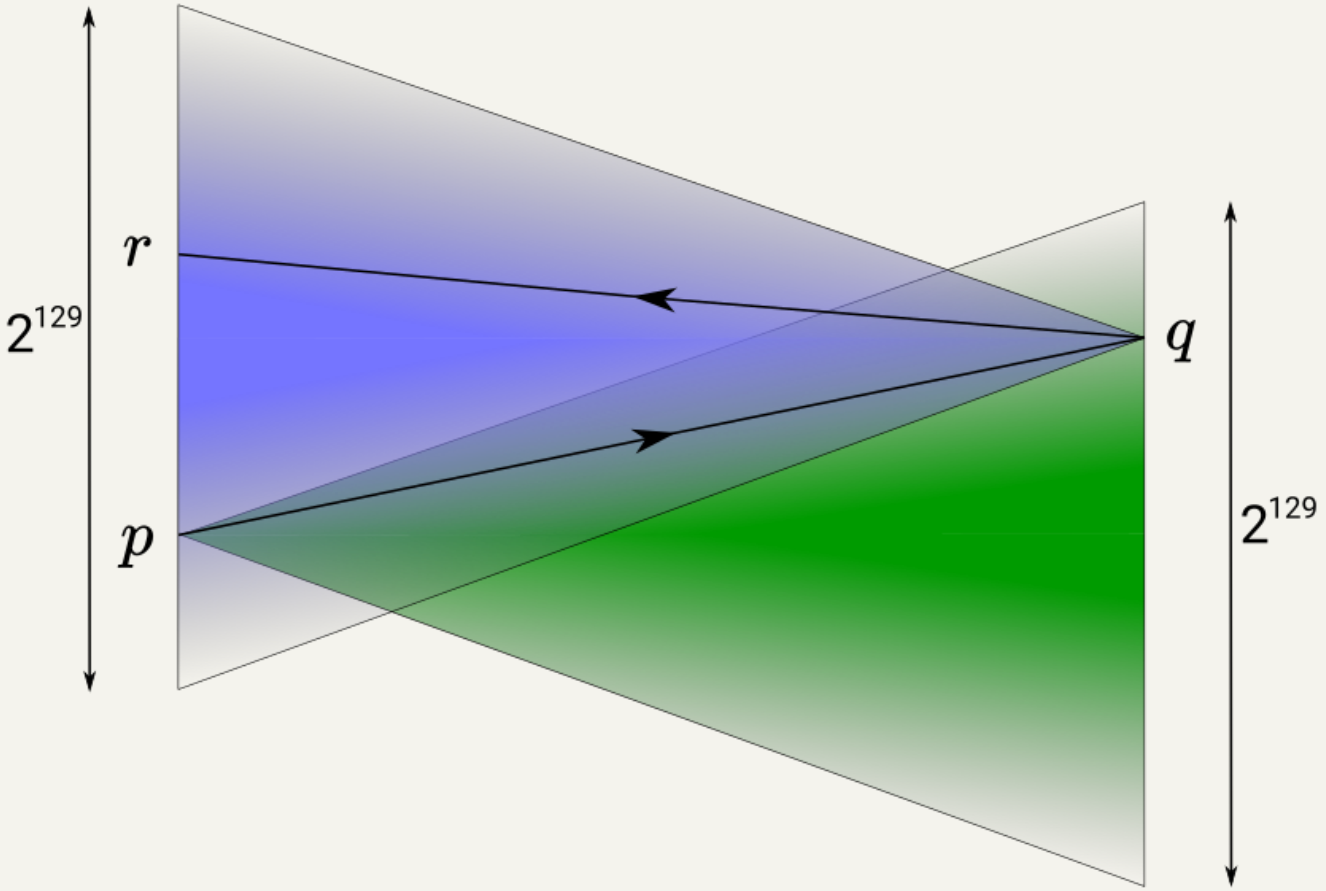


* proven by Taylor et al in 2006.

- That is, the order n could be anywhere in the range.
- And (if n is a prime q) when we construct a curve $E_{q \to r}$ it could also have order anywhere in its Hasse range.
- So, it's exceptionally unlikely that $E_{q \to r}$ has order p.

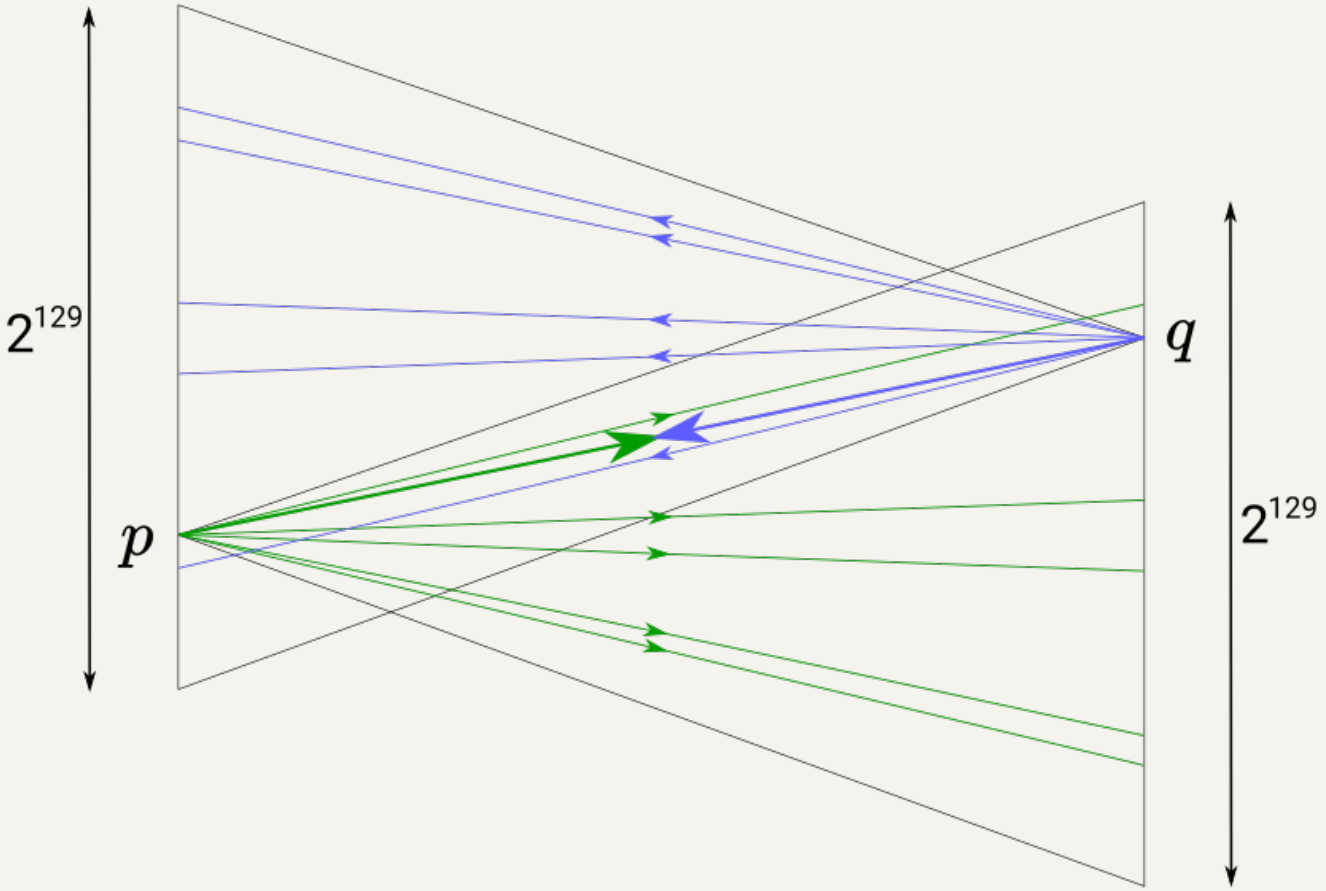
Constructing cycles – the problem

- By the Hasse bound, the order of an elliptic curve over $\mathbb{F}_p$ lies in the range $[\, p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}\,]$. For Tweedle this range is of size $\sim 2^{129}$.



Constructing cycles – the solution

- Suppose we were able to restrict the orders to a small number of possibilities, one of which was guaranteed to form a cycle...



CM curves to the rescue

- Katherine Stange and Joe Silverman noticed that CM curves have precisely that property [SS2011].
- What is a CM curve?
  - This is not intended to be a fully precise definition, just to give intuition and make the concept less mysterious.
- An endomorphism is a group homomorphism (meaning that it preserves the group structure) from the curve group to itself.
- An example of an endomorphism in an elliptic curve group is scalar multiplication by a constant integer.
- Each curve has an “endomorphism ring”, i.e. the ring of all possible endomorphisms for that curve.
  - Why is it a ring? Because you can compose and “add” endomorphisms, and there is an identity endomorphism.
- We can think of endomorphisms on elliptic curves as being generalized scalars.

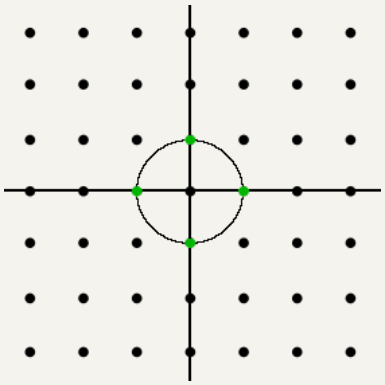
Complex multiplication

- All endomorphisms of an ordinary elliptic curve *over a finite field* are equivalent to scalar multiplication by an integer.
- But an elliptic curve over $\mathbb{F}_p$ is a reduction of a curve with the same equation over the complex numbers $\mathbb{C}$.
- "Complex multiplication" refers to scalar-multiplying points in the curve over $\mathbb{C}$ by complex numbers.
  - E.g. consider $E : y^2 = x^3 + ax$ and let $[i]\ (x, y) = (-x, iy)$. Then $[i^2]\ (x, y) = [-1]\ (x, y) = (x, -y)$, which is the same as applying the $[i]$ map twice.
  - So these scalars are numbers in a complex lattice L, such as $\mathbb{Z}[i]$ or $\mathbb{Z}[\sqrt[3]{1}]$.
- Are yous still with me? If not then don't worry because it all gets a bit simpler again when we map back to finite fields.

Structure of the endomorphism ring

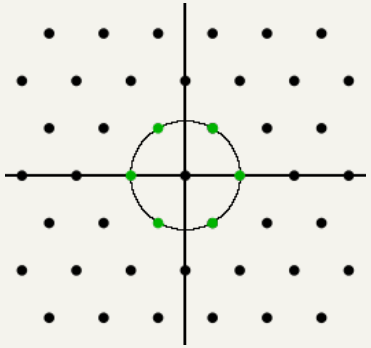
- We said that a generalized scalar can be a number in a complex lattice.
- An elliptic curve over $\mathbb{C}$ has CM if that lattice has more than one dimension (i.e. it's bigger than $\mathbb{Z}$). The lattice structure depends on something called the curve's j-invariant.
- For example, j = 1728 gives a quadratic lattice ($\mathbb{Z}[i]$, also called the Gaussian integers):

  The example we saw on the previous slide is of this case.



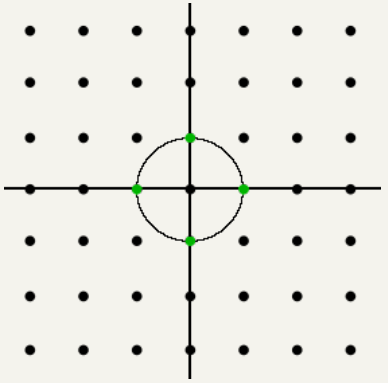
- And j = 0 gives a hexagonal lattice ($\mathbb{Z}[\sqrt[3]{1}]$, also called the Eisenstein integers):

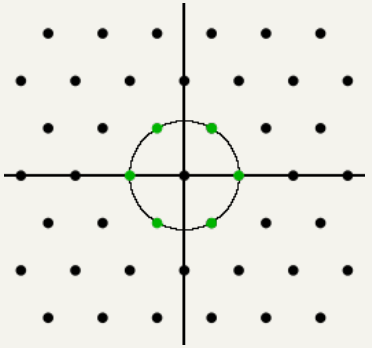
  This case turns out to be really nice for cryptography. This is the case that secp256k1, used in Bitcoin, falls into.



Moving from $E_{\mathbb{C}\to L}$ to $E_{p\to q}$

- The lattice L is a ring, and its units are the elements that have multiplicative inverses. The roots of unity are evenly distributed around the circle of radius 1 in the complex plane.
- The units of L will correspond to possible orders of E_p as we vary the curve coefficients.
- Remember that each point in L is a generalized scalar. But a complex root s can also have a corresponding root s' in $\mathbb{F}_q$.
- Then when we move to the curve over $\mathbb{F}_p$, complex multiplication by s corresponds to ordinary scalar multiplication by the integer s'.
- Also, the units that form a basis for the lattice correspond to endomorphisms that provide “shortcuts” to scalar multiplication in the curve over $\mathbb{F}_p$.





Security of CM curves

- There are only 13 elliptic curves over $\mathbb{C}$ with complex multiplication, up to isomorphism. Whether a curve has CM depends only on its equation.
- Even though CM curves are a small fraction of all elliptic curves, there's no reason to believe they are any weaker. All known pairing-friendly curve constructions produce CM curves, and they are very well-studied.
- Note that isomorphism between curves over $\mathbb{C}$ isn't what determines the security of the Discrete Log Problem; that depends mainly on the sizes of p and q.
- Curves with the same curve equation over different fields are not isomorphic.

But what does it all mean?

- What happens when we map a CM curve back to $\mathbb{F}_p$ is that it can only have a small number of possible orders.
- How many orders it can have is dependent on the number of units in the lattice we saw earlier, which is in turn determined by the j-invariant.
  - For details see On Orders of Elliptic Curves over Finite Fields.
- Curves $y^2 = x^3 + b$ (with no x or x^2 terms) have j-invariant 0.
- These curves are interesting because, over $\mathbb{F}_p$,
  - they have 6 possible orders;
  - they have efficiently computable endomorphisms;
  - it's usually easy to solve the CM norm equation.

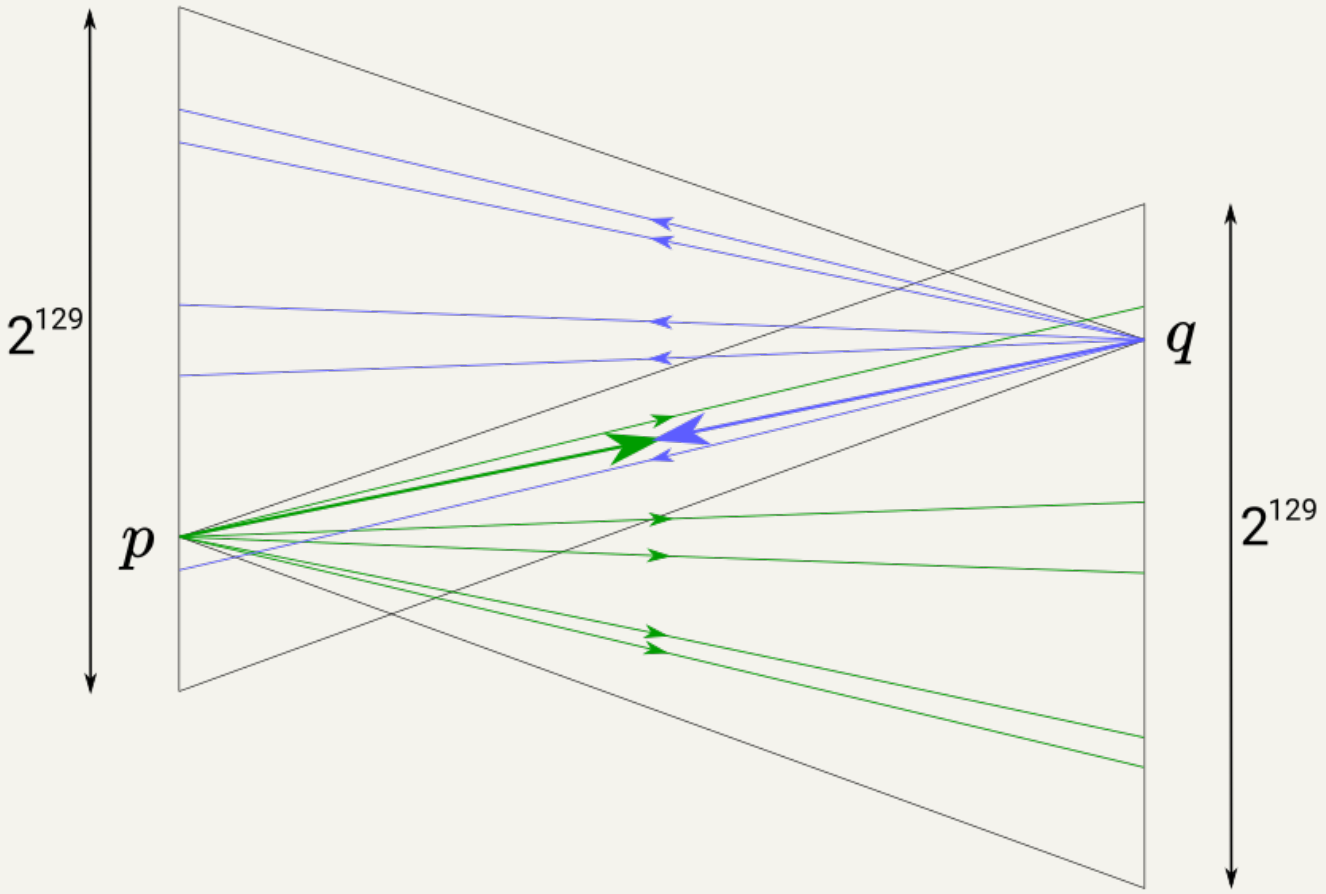
The CM norm equation

- $|D|V^2 = 4p - T^2$
- $|D|$ is the absolute fundamental discriminant (D is negative).
- p is the field size.
- V and T are integers.
- V and T determine the trace t, where $q = p + 1 - t$.
- In fact $\pm T$ are two of the possible traces.
- The Tweedle curves were found using this construction:
  - set $|D| = 3$, pick V and T, then

    $p = \frac{1}{4}(|D|V^2 + T^2)$ and $q = p + 1 - T$.
- The reason for this approach is that we can choose V and T so that *both* curves in the cycle have high 2-adicity.

Constructing cycles – the solution

- For each possible order of $E_{p\to q}$ one of the possible orders of $E_{p\to q}$ is p.
- This is proven by Stange and Silverman, and again with a more elementary proof in the accompanying notes. For more detail on CM see Chenal's thesis.



But why?

- The norm equation helps to explain why CM curves form cycles.

The CM norm equation of $E_{p \to q}$ is $4p = |D|V^2 + T_p^2$ for integers V and T_p.

Theorem: One of the possible orders for E_q is p.

Case for $q = p + 1 \pm T_p$

We have

$$\begin{aligned} 4q &= 4p + 4 \pm 4T_p \\ &= |D|V^2 + T_p^2 \pm 4T_p + 4 \\ &= |D|V^2 + (T_p \pm 2)^2 \end{aligned}$$

This is a norm equation for E_q, with the same $|D|$ and V, and with $T_q = T_p \pm 2$.

In the positive case we have $T_q = T_p + 2$ and $q + 1 - T_q = (p + 1 + T_p) + 1 - (T_p + 2) = p$.

In the negative case we have $T_q = T_p - 2$ and $q + 1 + T_q = (p + 1 - T_p) + 1 + (T_p - 2) = p$.

That is, one of the possible orders of E_q, specifically $q + 1 \mp T_q$, is p in both cases. $\square$

2-adicity

- Protocols that use Lagrange basis, or that need to efficiently multiply polynomials, benefit from $\mathbb{F}^*_{p,\,q}$ having a “large enough” multiplicative subgroup of size z^c. The simplest option is z = 2.
- In other words, we need $p \equiv 1$ and $q \equiv 1 \pmod{2^c}$.

We can freely choose V_p and T_p. So choose $\frac{V_p-1}{2}$ and $\frac{T_p-1}{2}$ as multiples of 2^c. Then

$$\begin{aligned} 4p &= (3(V_p-1)^2 + 6(V_p-1) + 3) + ((T_p-1)^2 + 2(T_p-1) + 1) \\ &= 3(V_p-1)^2 + 6(V_p-1) + (T_p-1)^2 + 2(T_p-1) + 4 \\ p &= 3\left(\frac{V_p-1}{2}\right)^2 + 3\frac{V_p-1}{2} + \left(\frac{T_p-1}{2}\right)^2 + \frac{T_p-1}{2} + 1 \end{aligned}$$

So $p-1$ will be a multiple of 2^c, and so will $q-1$ for $q \in \left\{p+1-T_p,\, p+1-\frac{3V_p}{2}+\frac{T_p}{2}\right\}$.

Recap

- In Ying Tong's section, we described how Halo 2 is constructed from PLONK and a variation of the inner product argument.
- We explained how this is used in an accumulation scheme to give an efficient recursive proof system.
- We explained the difficulty with constructing cycles of curves, and how to solve it by using curves with complex multiplication.
- We described (approximately) what complex multiplication is, and how the units of a lattice defined by the endomorphism ring correspond to the possible orders of curves over a given field.
- Along the way, we discussed endomorphisms and how they correspond to "shortcuts" for scalar multiplication.
  - That part will be useful for an optimization we're going to describe in this section of the talk.

Circuit optimization for Halo 2

- Practical circuits tend to be dominated by a small number of operations that are repeated many times. In the Halo verification circuit, we need:
  - Multi-scalar multiplications
  - A hash function, to generate Fiat–Shamir challenges.
- We also need to include the “application” circuit(s), and so any optimizations to the proof system should be made with that in mind.
- As an application-level example, we’ll describe a Pedersen-like hash function called Sinsemilla, and show how to implement it using lookups.

Scalar multiplication in circuits

- Halo uses prime-order short Weierstrass curves, $E : y^2 = x^3 + b$.
- We start with a variation on the scalar multiplication algorithm in Zcash #3924, which takes 6 muls per scalar bit.
- This is based on an idea from Kirsten Eisentrager, Kristin Lauter, and Peter Montgomery. Instead of computing [2] $A + P$ directly in a double-and-add algorithm, we compute $(A + P) + A$.
- To avoid needing two constraints for the conditional, we actually compute $(A \pm P) + A$.
- The λ for the outer addition can be computed from the λ for the inner one, saving one mul.
- Naively we need 4 muls for the doubling, 3 for the addition, and 2 for the conditional. This technique costs 3 for the inner addition, 2 for the outer addition, and 1 for conditional negation.

Endoscaling

- It turns out that in PLONK we can process two bits per row, by using the endomorphism we dicussed earlier. (In Halo 1 this technique gave us 7 constraints per two bits.)
- We only require the multiplications to be of some scalar with at least 128 bits of entropy (depending on the verifier challenges).
- The basic idea is to add one of $\{ -P,\ P,\ -\phi(P),\ \phi(P) \}$ at each step. This requires two curve additions for every two bits of entropy.
- This has the effect of multiplying by $\zeta a + b$, where a and b depend on random $\mathbf{r}$.

Algorithm 1

Inputs: $\mathbf{r} \in \{0,1\}^\lambda, P \in E \setminus \{\mathcal{O}\}$

$\mathsf{Acc} := [2](\phi(P) + P)$

for i from $\lambda/2 - 1$ down to 0:

$$\text{let } S_i = \begin{cases} [2\mathbf{r}_{2i} - 1]P, & \text{if } \mathbf{r}_{2i+1} = 0 \\ \phi\big([2\mathbf{r}_{2i} - 1]P\big), & \text{otherwise} \end{cases}$$

$\mathsf{Acc} := (\mathsf{Acc} + S_i) + \mathsf{Acc}$

Output Acc

Endoscaling

- We want the mapping $\mathbf{r} \to \zeta a + b$ not to lose any entropy, i.e. to be 1-to-1.
- To prove this we show that $\mathbf{r} \to (a, b)$ and $(a, b) \to \zeta a + b$ are both 1-to-1.
- For the first part:

Lemma 2. *For* $k \geq 0$, $(\mathbf{c}, \mathbf{d}) \in M_k \mapsto \left(\sum_{j=0}^{k-1} \mathbf{c}_j 2^j, \sum_{j=0}^{k-1} \mathbf{d}_j 2^j\right)$ *is injective.*

Proof (sketch). If $(\mathbf{c}, \mathbf{d})$ and $(\mathbf{c}', \mathbf{d}')$ coincide on a prefix of length m, then the statement reduces to a smaller instance of the lemma with that prefix deleted and k reduced by m. So we need only consider the case $(\mathbf{c}_0, \mathbf{d}_0) \neq (\mathbf{c}'_0, \mathbf{d}'_0)$ and show that the resulting sums always differ. In fact they always differ modulo 4:

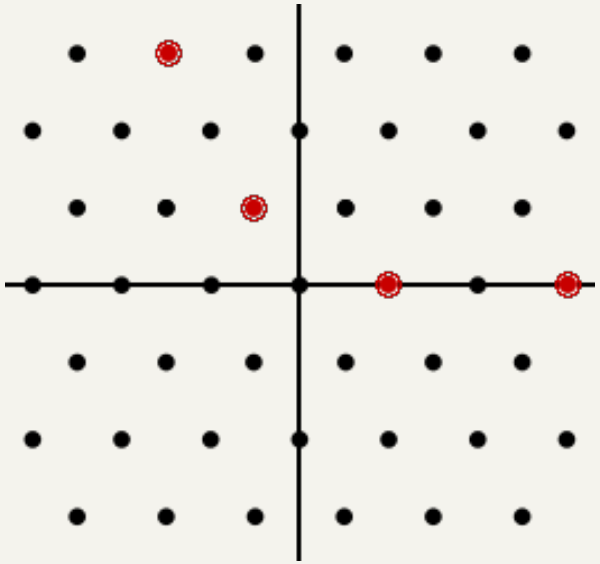
$$\begin{aligned}(\mathbf{c}_0, \mathbf{d}_0) \neq (\mathbf{c}'_0, \mathbf{d}'_0) \implies & \left(\sum_{j=0}^{k-1} \mathbf{c}_j 2^j \bmod 4, \sum_{j=0}^{k-1} \mathbf{d}_j 2^j \bmod 4\right) \\ & \neq \left(\sum_{j=0}^{k-1} \mathbf{c}'_j 2^j \bmod 4, \sum_{j=0}^{k-1} \mathbf{d}'_j 2^j \bmod 4\right)\end{aligned}$$

Therefore, it suffices to verify this property exhaustively for $k = 1$ and $k = 2$ [26, `injectivitylemma.py`], since $j \geq 2$ terms do not affect the sums modulo 4. □

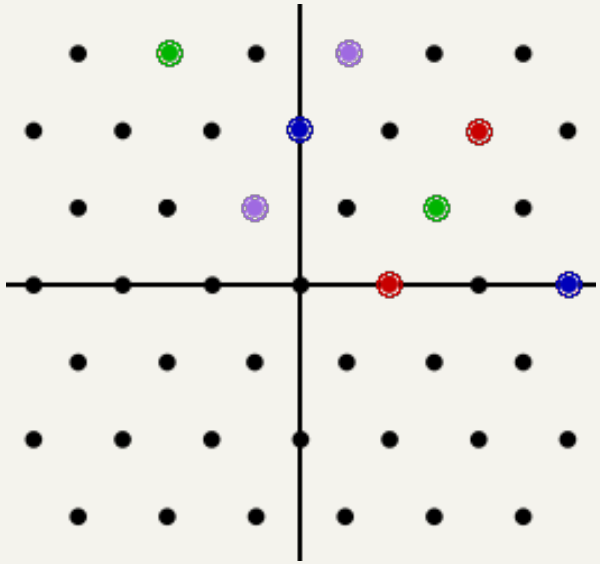
Optimized scalar multiplication

- We want the mapping $\mathbf{r} \to \zeta a + b$ not to lose any entropy, i.e. to be 1-to-1.
- To prove this we show that $\mathbf{r} \to (a, b)$ and $(a, b) \to \zeta a + b$ are both 1-to-1.
- For the first part:

$k = 1$

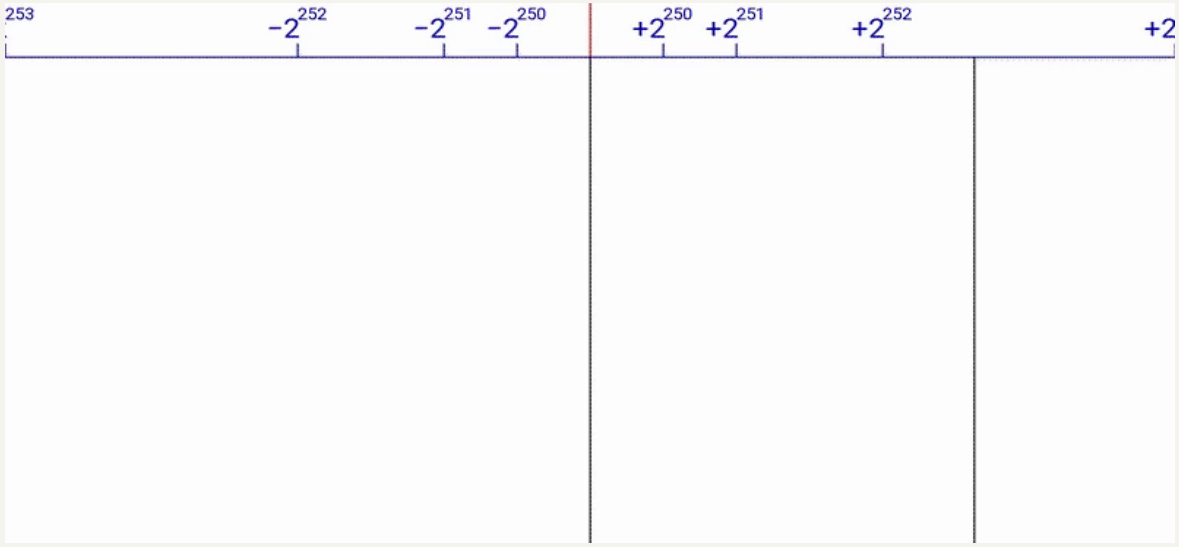


$k = 2$



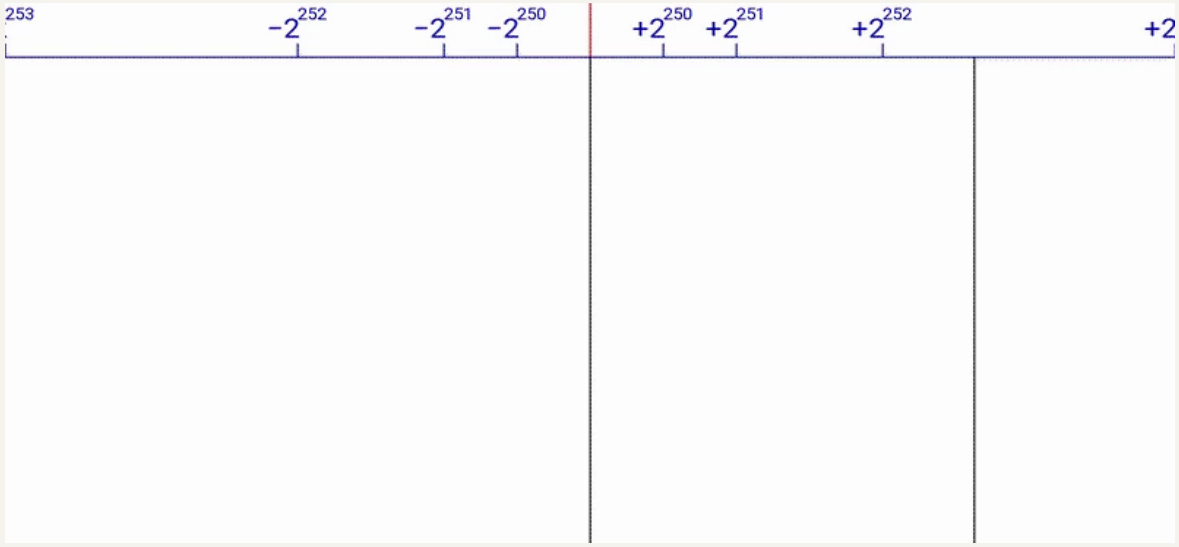
Endoscaling

- We want the mapping $\mathbf{r} \to \zeta a + b$ not to lose any entropy, i.e. to be 1-to-1.
- To prove this we show that $\mathbf{r} \to (a, b)$ and $(a, b) \to \zeta a + b$ are both 1-to-1.
- For the second part, we find the smallest distance between two values of $\zeta a \pmod q$ for a in $A = [0, 2^{65} + 2^{64})$. If this is $\geq 2^{65} + 2^{64}$, then a copy of A will "fit between the gaps" of ζA, and so $(a, b) \to \zeta a + b$ must be 1-to-1.
- `checksumsets.py` does this check and also generates this video:



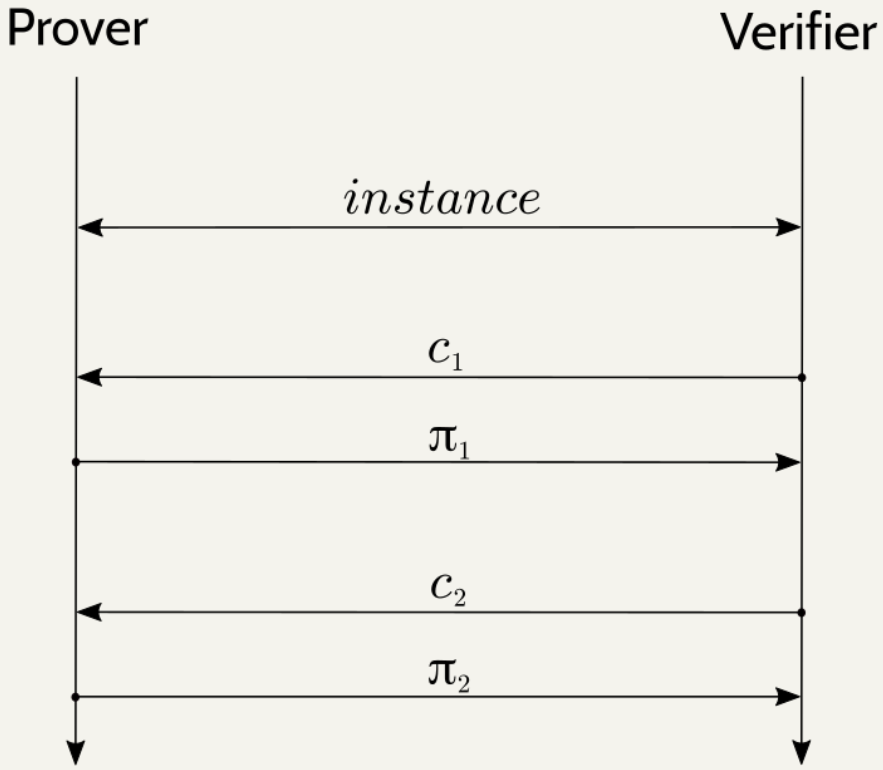
Endoscaling

- We want the mapping $\mathbf{r} \to \zeta a + b$ not to lose any entropy, i.e. to be 1-to-1.
- To prove this we show that $\mathbf{r} \to (a, b)$ and $(a, b) \to \zeta a + b$ are both 1-to-1.
- For the second part, we find the smallest distance between two values of $\zeta a \pmod q$ for a in $A = [0, 2^{65} + 2^{64})$. If this is $\geq 2^{65} + 2^{64}$, then a copy of A will "fit between the gaps" of ζA, and so $(a, b) \to \zeta a + b$ must be 1-to-1.
- `checksumsets.py` does this check and also generates this video:



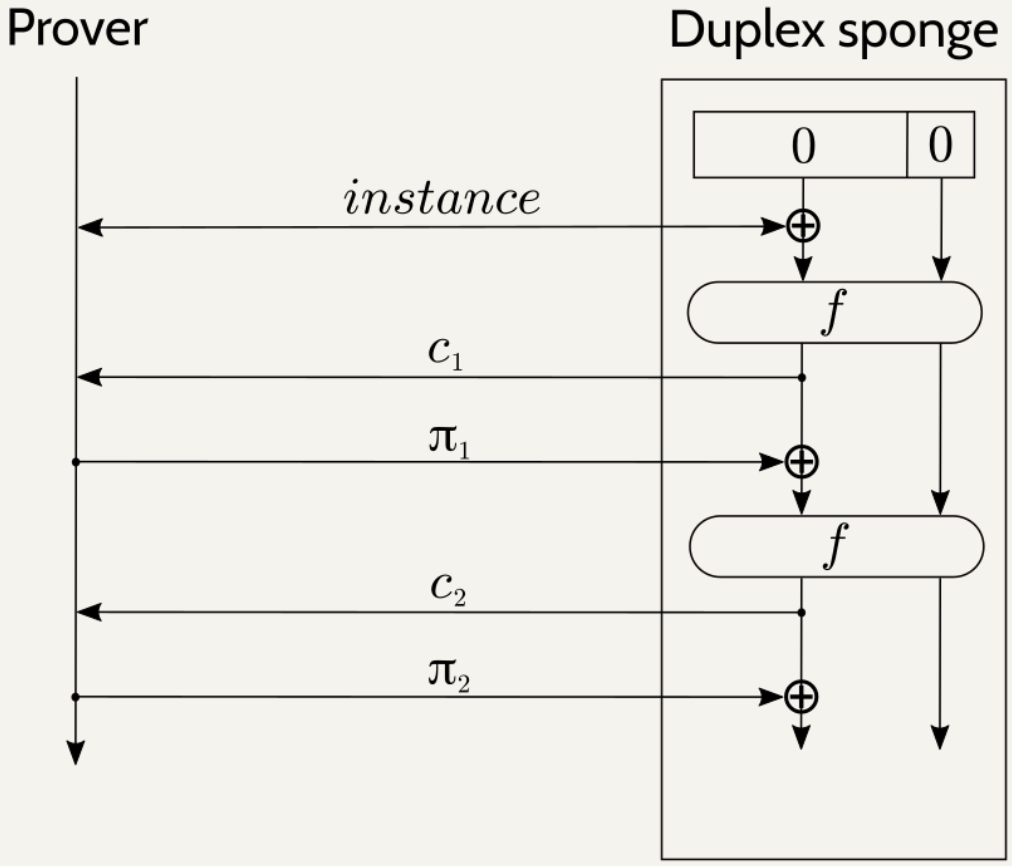
Fiat–Shamir and duplex sponges

- The Fiat–Shamir construction takes an interactive public-coin protocol, ...



Fiat–Shamir and duplex sponges

- The Fiat–Shamir construction takes an interactive public-coin protocol, and replaces the verifier with a hash function.



- Using a duplex sponge basically halves the number of f evaluations relative to other hash constructions.

Optimizations for any duplex sponge

- Use addition in the field for ⊕, rather than XOR.
- Compress the absorbed inputs.
  - There's a way of probabilistically compressing two curve points to three field elements that is *much* less expensive than standard point compression (see accompanying notes).

Theorem: Let $E/\mathbb{F}_p : y^2 = x^3 + b$ be an elliptic curve with a large number of points $\#E$. Except with negligible probability over random choices of P_1 and P_2, $x_1^3 \neq x_2^3$ and $x_1^3 + b \neq 0$ and $x_2^3 + b \neq 0$ and $y_1 + y_2 \neq 0$, and in that case $(x_1, x_2, y_1 + y_2)$ determines P_1 and P_2.

- Pick a "rate" that is just large enough that we only need one f evaluation per round.
  - For the inner product argument we need $\lg(N)$ rounds, each of which absorbs two curve points and squeezes out one challenge.

Algebraic hashes

- To instantiate f in the duplex sponge, we need a permutation that is efficient in the circuit.
- The options are:
  - Rescue
  - Poseidon (recently wounded by eprint 2020/188, but more efficient outside a circuit than Rescue)
- Optimization that we can use with either Rescue or Poseidon:
  - Choose curves with $\gcd(p-1, 5) = 1$ so that $x \mapsto x^5$ is a permutation.
  - $x \mapsto x^3$ cannot be a permutation for curves with the endomorphism (and the latter gives us more of a performance advantage).

Sinsemilla

- What if we aren't sufficiently confident about Rescue and Poseidon yet?
- Zcash Sapling used Bowe–Hopwood Pedersen hashes, with security tightly reducible to the Discrete Logarithm Problem.
- Sinsemilla is a new Pedersen-like hash by Sean Bowe and me, designed for circuit efficiency when lookups are available.
- It is **not** suitable for Fiat–Shamir since it only provides collision-resistance.
- I'll describe a simplified version (this is likely to change, and its security proof is unfinished – don't use it yet!)

Sinsemilla

Let $k \geq 1$ be an integer chosen based on efficiency considerations (the table size will be 2^k). Let n be a **fixed** integer such that messages are kn bits, where $2^n \leq \frac{p-1}{2}$. Message padding is out of scope for this note.

Setup: Choose Q and $P[0..n-1]$ as $n+1$ independent, verifiably random generators of $\mathbb{G}$, using a suitable hash into $\mathbb{G}$.

$\mathsf{Hash}(M)$:

Split M into n groups of k bits. Interpret each group as a k-bit little-endian integer m_i.
$A_0 := Q$
for i from 0 up to $n-1$:
    $A_{i+1} := [2]A_i + P[m_i]$
return A_n

If $\mathbb{G}$ is a prime-order elliptic curve in short Weierstrass form, then let $\mathsf{ShortHash}(M)$ be the x-coordinate of $\mathsf{Hash}(M)$.

Security of Sinsemilla

- The structure is equivalent to a Pedersen hash “on its side”:

i \ j	0	1	2	3	4	5	6	7	m_i
0	0	0	1	0	0	0	0	0	2
1	0	0	0	0	0	0	1	0	6
2	0	0	0	0	1	0	0	0	4
3	0	0	1	0	0	0	0	0	2
4	0	0	0	0	0	1	0	0	5
5	1	0	0	0	0	0	0	0	1

- Reading each column in binary (big end at the top) gives the scalar by which each base $P[j]$ is multiplied.
- Informally, each message gives a unique table and so a unique input to the Pedersen hash (if we’re careful about ranges).

Sinsemilla constraints

Let $\mathcal{P} = \Big\{(j,\, x_{P,j},\, y_{P,j}) \text{ for } j \in \{0..2^k - 1\}\Big\}$.

Input: $z_n \in \{0..2^{kn} - 1\}$ such that $z_n = \sum_{i=0}^{n-1} m_i \cdot 2^{ki}$.

$z_0 = 0$

$(x_{A,0},\, y_{A,0}) = Q$

for i from 0 up to $n - 1$:

$(z_{i+1} - 2^k \cdot z_i,\, x_{P,i},\, y_{P,i}) \in \mathcal{P}$

$\lambda_{1,i} \cdot (x_{A,i} - x_{P,i}) = y_{A,i} - y_{P,i}$

$\lambda_{1,i}^2 = x_{R,i} + x_{A,i} + x_{P,i}$

$(\lambda_{1,i} + \lambda_{2,i}) \cdot (x_{A,i} - x_{R,i}) = 2y_{A,i}$

$\lambda_{2,i}^2 = x_{A,i+1} + x_{R,i} + x_{A,i}$

$\lambda_{2,i} \cdot (x_{A,i} - x_{A,i+1}) = y_{A,i} + y_{A,i+1}$

Output $(x_{A,n},\, y_{A,n})$

Implementing lookups

- Here I'll describe a lookup protocol that is similar to Plookup, but simpler and optimized for smaller circuits.
- Let's say we have a relation $\mathcal{R} : (\mathbb{F}_p)^w$, e.g. $(i, x_i, y_i) : (x_i, y_i)$ = P[i].
- Compress the entries using a random challenge r, e.g. to $i + rx_i + r^2y_i$.
- We can implement the lookups using a "subset argument".
- Assume there are 2^k rows (numbered from 0) and columns A and S.
- The goal is to enforce that every cell in A is equal to *some* cell in S.
- We allow the prover to supply columns A′ and S′, and enforce that:
  - A′ is a permutation of A, and S′ is a permutation of S.
  - Each cell in A′ has to either have the same value as the one above it, or the same value as in the corresponding row of S′.
  - $A'_0 = S'_0$.

An example

- Let’s use the prime numbers { 2, 3, 5, … 19 } for our set.

A	A´	S´	S
11			2
2			3
5			5
19			7
2			11
11			13
7			17
3			19

An example

- Let’s use the prime numbers { 2, 3, 5, … 19 } for our set.

A	A´	S´	S
11	2		2
2	2		3
5	3		5
19	5		7
2	7		11
11	11		13
7	11		17
3	19		19

An example

- Let’s use the prime numbers { 2, 3, 5, … 19 } for our set.

A	A´		S´	S
11	2	=	2	2
2	2			3
5	3	=	3	5
19	5	=	5	7
2	7	=	7	11
11	11	=	11	13
7	11			17
3	19	=	19	19

An example

- Let’s use the prime numbers { 2, 3, 5, … 19 } for our set.

A	A´		S´	S
11	2	=	2	2
2	2	≠	17	3
5	3	=	3	5
19	5	=	5	7
2	7	=	7	11
11	11	=	11	13
7	11	≠	13	17
3	19	=	19	19

An example

- Let’s use the prime numbers { 2, 3, 5, … 19 } for our set.

A	A´		S´	S
11	2	=	2	2
2	2		17	3
5	3	=	3	5
19	5	=	5	7
2	7	=	7	11
11	11	=	11	13
7	11		13	17
3	19	=	19	19

$$(11+\beta)(2+\beta)(5+\beta)(19+\beta)(2+\beta)(11+\beta)(7+\beta)(3+\beta)\ (2+\gamma)(3+\gamma)(5+\gamma)(7+\gamma)(11+\gamma)(13+\gamma)(17+\gamma)(19+\gamma)$$
$$= (2+\beta)(2+\beta)(3+\beta)(5+\beta)(7+\beta)(11+\beta)(11+\beta)(19+\beta)\ (2+\gamma)(17+\gamma)(3+\gamma)(5+\gamma)(7+\gamma)(11+\gamma)(13+\gamma)(19+\gamma)$$

Arbitrary permutation argument

- Enforcing the permutations is like the main PLONK permutation argument, except that we don't want to enforce a *specific* permutation, we want to allow an arbitrary one.

We'll start by allowing the prover to supply permutation columns of A and S. Let's call these A' and S', respectively. We can enforce that they are permutations using a permutation argument with product column Z with the rules:

$$Z(X)(A(X)+\beta)(S(X)+\gamma) - Z(\omega^{-1}X)(A'(X)+\beta)(S'(X)+\gamma) = 0$$

$$\ell_0(X)(Z(X)-1) = 0$$

This is a version of the permutation argument which allows A' and S' to be permutations of A and S, respectively, but doesn't specify the exact permutations. β and γ are separate challenges so that we can combine these two permutation arguments into one without worrying that they might interfere with each other.

Credits

- Halo and Halo 2 are joint work with Sean Bowe and Jack Grigg.
- PLONK is work by Ariel Gabizon, Zach Williamson, and Oana Ciobotaru.
- The original inner product argument is due to Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit, with later improvements by Benedikt Bünz, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell.
- Numerous people have contributed to the science of zero-knowledge proving systems, but we would particularly like to acknowledge the work of Shafı Goldwasser, Silvio Micali, Oded Goldreich, Charles Rackoff, Rosario Gennaro, Bryan Parno, Jon Howell, Craig Gentry, Mariana Raykova, Jens Groth, Mary Maller, Rafail Ostrovsky, and Amit Sahai.

Questions?

Daira Hopwood (@feministPLT)
Ying Tong Lai (@therealyingtong)

https://github.com/daira/halographs
(deepdive.odp)