Groth16 - SNARK Fundamentals

PSE ZK Workshop

Flying Nobita

Sat, Feb 15, 2025

Agenda

1. Groth16 Overview
2. Hands-on with `circom` on zkREPL

Feel free to ask questions at anytime.

What's Special About Groth16?

- One of the most widely used proof system
- **Smallest** proof size (128 to 192 bytes)
- **Fastest** verification time (~1 ms)
- Used in ZCash, one of the earliest cryptocurrency projects that use ZK

Groth16 Workflow

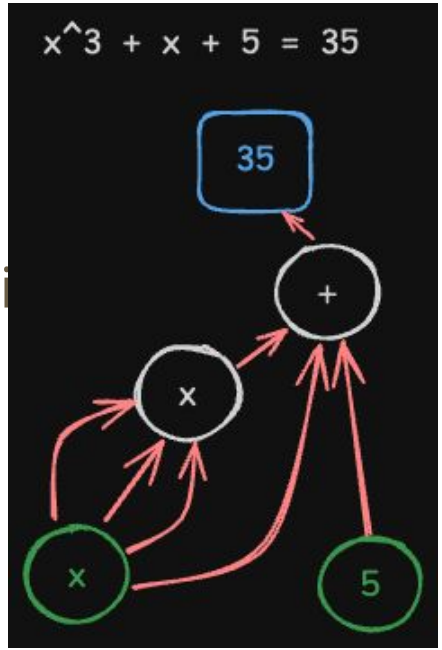
Problem -> Arithmetic Circuit -> R1CS -> QAP -> Groth16

Problem / NP Relations

- Groth16 is a zero-knowledge proof (ZKP) system for the satisfiability of any NP Relations
  - i.e. given a public statement **x**, it proves knowledge of a secret **w** such that **R(x; w) = 1}**
- e.g. **Sudoku(puzzle, solution) = 1**
  - a ZKP convinces the **Verifier** that the **Prover** knows a **solution** (secret, i.e. only prover has this) to a **puzzle** such that **Sudoku(puzzle, sol) = 1**

Arithmetic Circuit

- directed acyclic graph (DAG)
- $C : \mathbb{F}^n \to F$
- a function that takes n elements in a Field and output an element in the Field
- |C| = size of the circuit = the number of gates i
- any problem in NP can be modeled with an arithmetic circuit
- example:
  - what is x?
    - x = 3
  - given witness: {x=3}, circuit C is satisfied
- we will look and write more circuits in a bit



Groth16 Workflow

Problem -> Arithmetic Circuit -> R1CS -> QAP -> Groth16

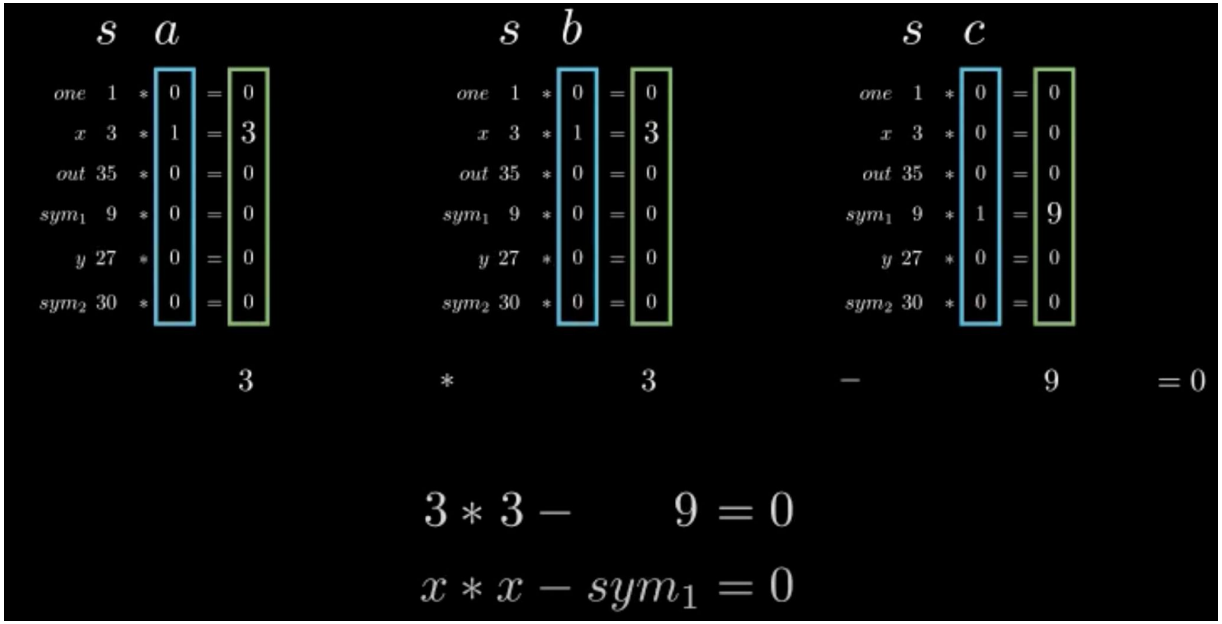
Arithmetic Circuit -> R1CS: Step 1: Flattening

- transform the circuit equation to a set of equations that has at most 1 multiplication
- each equation has 2 input and 1 output
- this set of equations is equivalent to our original circuit equation if `out = 35`

$$sym_1 = x * x$$
$$y = sym_1 * x$$
$$sym_2 = y + x$$
$$out = sym_2 + 5$$

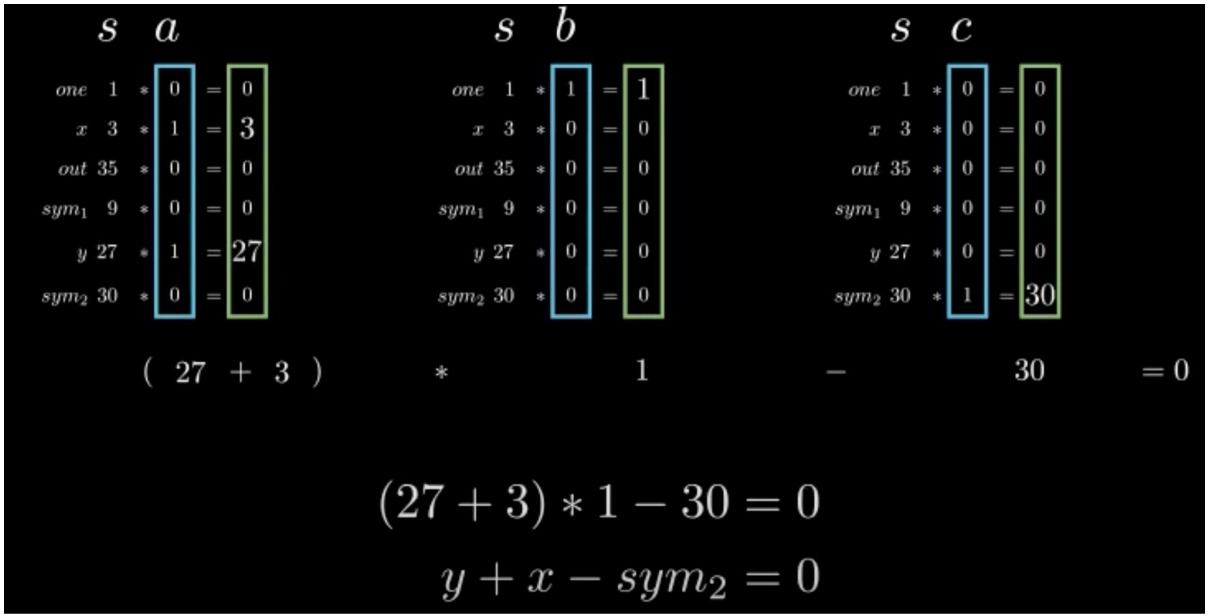
- thus proving these 4 equations is the same as proving the original circuit equation
- Output: $[one, x, out, sym_1, y, sym_2]$

Example: Gate 1



Source: [R1CS1 by 0xPARC]

Example: Gate 3



Source: [R1CS3 by 0xPARC]

R1CS

$R(x; w) = 1 \iff U(s) \circ V(s) - W(s) = 0$ where: U, V, W are of size $(\mathrm{m} + 1)$ rows and $(\mathrm{n} + 1)$ columns

s, the witness, is a column vector of size $(n + 1)$

- $\circ$ is the Hadamard product (element-wise multiplication)

Groth16 Workflow

Problem -> Arithmetic Circuit -> R1CS -> QAP -> Groth16

Prerequisites: FFT / Lagrange Interpolation

Polynomial Representation

Coefficient Representation

- $A(x) = a0 + a1x + a2x^2 + \ldots + a_nx^n$

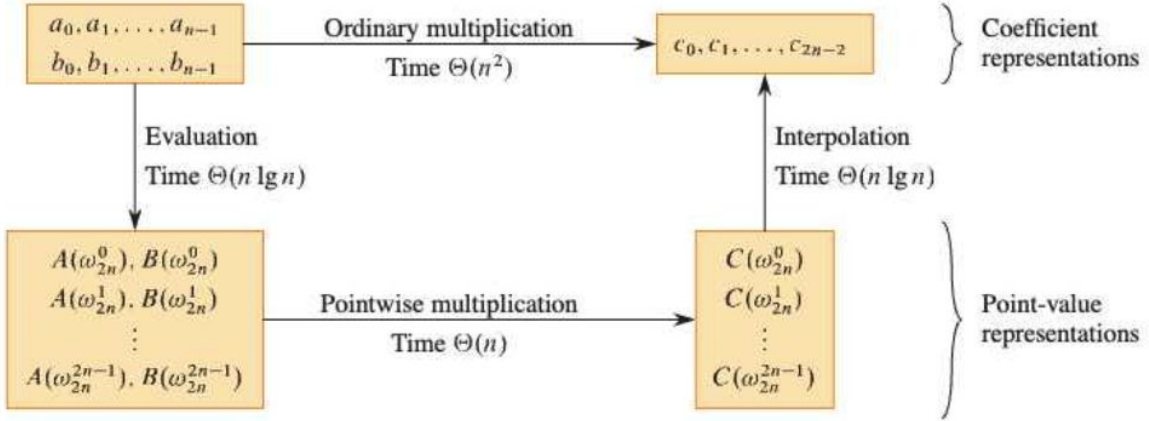
Evaluation Representation / Point-Value Representation

- represented as a set of n pairs $(x0, y0), (x1, y1), \ldots, (x_n, y_n)$ such that:
  - $x_i \neq x_j, \forall\, i \neq j$ (i.e. the points are unique)
  - $y_k = A(x_k), \forall\, k$ where $k = 0 \ldots n-1$

FFT / Lagrange Interpolation Applications

Applications

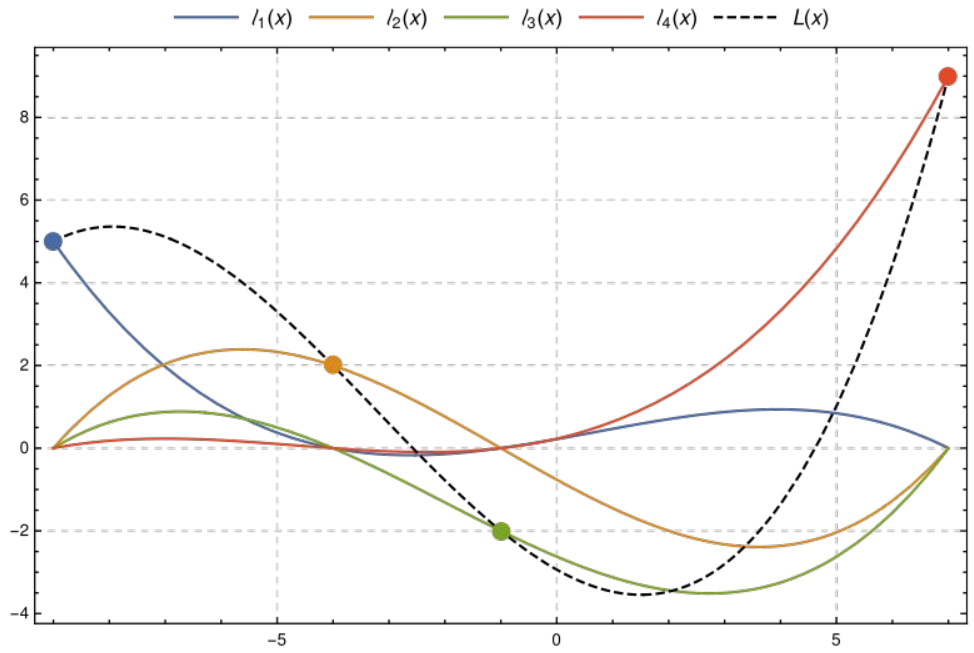
- some operations (e.g. **polynomial multiplication and division**) compute more **efficiently in evaluation representation**
- useful for **FRI**-based systems (e.g. STARKs)



Source: *Introduction to Algorithms by CLSR*

Lagrange Interpolation

- **evaluation -> coefficient** of a given polynomial



Fast Fourier Transform (FFT)

- **coefficient -> evaluation** of a given polynomial:
- Using divide and conquer to achieve O(n log(n)) instead of O(n^2):
  a. split the polynomial into 2 parts, odd and even coefficients
  b. evaluate each of the part
  c. combine the 2 parts

iFFT

- **evaluation -> coefficient** of a given polynomial
  - find the inverse of the NTT matrix
  - use the inverse matrix to find the coefficients

FFT vs Lagrange Interpolation

- Lagrange Interpolation is slower than FFT, but can take any arbitrary point-values, and any arbitrary field
- FFT requires:
  - a finite field
  - domain must be a multiplicative subgroup of the field (i.e. elements are from a generator G to a range of powers)
  - multiplicative subgroup is size 2^n, which let us create a recursive algorithm that calculates the results with much less work

Prerequisite: Pairing / Bilinear Map

- Pairing is the function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ over 3 groups with their respective generators
- Pairing allows us to multiply 2 “hidden numbers”

Pairing Rules

1. $e(nP, mQ) = e(P, mQ)^n = e(nP, Q)^m = e(P, Q)^{nm}$
2. $e(P, Q)^R * e(P, Q)^S = e(P, Q)^{R+S}$
3. $e(P, Q)^0 = 1$
4. $e(P, Q + R) = e(P, Q) * e(P, Q)$
5. $e(P + S, Q) = e(P, Q) * e(S, Q)$

Prerequisite: Pairing / Bilinear Map - Example

Question

Let's suppose Alice want to prove to Bob that she know some integer a that satisfies the equation: $a^2 - 2027a + 16152 = 0$ without revealing a to Bob. How would Alice do this using an elliptic curve pairing?

Answer:

1. Alice choose 2 *public* generators G, and H and share to Bob.
2. Alice Compute aG and aH and send results to Bob (Bob cannot compute a due to ECDLP)
3. Bob computes:

   $$e(aG, aH) * e(G, (-2027) * (aH)) * e(G, 16152 * H) = e(G, H)^{a^2 - 2027a + 16152}$$

If above result == 1, then Bob know $a^2 - 2027a + 16152 = 0$ with high probability.

QAP

- R1CS can be converted to QAP
- Instead of using dot products, it uses polynomials with Lagrange interpolation
- Rather than checking the constraints in R1CS individually, one can check all of the constraints at the same time by doing the dot product check on the polynomial
- to check correctness, we divide $U(s) \circ V(s) - W(s)$ by $(X^n - 1)$ and ensure there is no remainder
- Groth16 is a ZKP for the QAP satisfiability relation:

$$\frac{\sum_{j=0}^{m} u_j(X)a_j \cdot \sum_{j=0}^{m} v_j(X)a_j + \sum_{j=0}^{m} w_j(X)a_j}{(X^n - 1)} = h(X)$$

where:

- $h(X)$ is the quotient polynomial
- $X^n - 1 = \prod_{i=0}^{n}(X - \omega^i)$ where $(\omega, \omega^2, \ldots, \omega^n)$ denote the nth roots of unity

Groth16 Workflow

Problem -> Arithmetic Circuit -> R1CS -> QAP -> Groth16

Groth16 Protocol

Notation

$$\begin{aligned}\text{commitment to a that results in a point in } G_1 &:= [a]_1 \\ &:= a \cdot G_1 \in \mathbb{G}_1 \\ &:= aG_1 \\ &:= \underbrace{G_1 + G_1 + \ldots + G_1}_{a \text{ times}}, \text{ where } a \in \mathbb{F} = \mathbb{Z}_p\end{aligned}$$

$$\begin{aligned}i^{th} \text{ Lagrange polynomial} &:= \mathcal{L}_i(X) \\ &:= \prod_{j \in [0,n), j \neq i} \frac{X - \omega^j}{\omega^i - \omega^j}\end{aligned}$$

Groth16 - Trusted Setup

1. Sample Random Field Elements

We need 5 random field elements to generate the Proving key and the Verification Key:

1. t
2. α
3. β
4. γ
5. δ

Groth16 - Trusted Setup

2. Compute Proving Key

$$\text{proving key} \leftarrow \Bigg(\underbrace{[\alpha]_1, [\beta]_1, [\beta]_2, [\delta]_1, [\delta]_2}_{\text{Com(random scalars)}} \quad \underbrace{([u_j(\tau)]_1, [v_j(\tau)]_1)_{j\in[0,m]}, ([v_j(\tau)]_2)_{j\in[0,m]}}_{\text{Com}(u_j),\ \text{Com}(v_j)} \quad \underbrace{\left([\frac{\mathcal{L}_i(\tau^i)(\tau^n-1)}{\delta}]_1\right)_{i\in[0,n-2]}}_{Com(\text{srs for } h(\tau)t(\tau))} \quad \underbrace{\left([\frac{\beta u_j(\tau)+\alpha v_j(\tau)+w_j(\tau)}{\delta}]_1\right)_{j\in[\ell+1,m]}}_{\text{Com(private witness)}} \Bigg)$$

l public wires + $(m-l)$ private wires = Total m wires

Groth16 - Trusted Setup

3. Compute Verification Key

$$\text{verification key} \leftarrow \left(\underbrace{[\alpha]_1, [\beta]_2, [\gamma]_2, [\delta]_2,}_{\text{Com(random scalars)}} \underbrace{\left(\left[\frac{\beta u_j(\tau) + \alpha v_j(\tau) + w_j(\tau)}{\gamma} \right]_1 \right)_{j \in [0, \ell]}}_{\text{public witness}} \right)$$

- contains information about the public portion of the witness

Groth16 - Trusted Setup

4. Discard Random Field Elements

- The 5 random field elements are not needed anymore and must be discarded to protect the integrity of the ZKP and prevent malicious actors from creating false proof that will be verified correctly
- As such, the random field elements are called *"toxic waste."*

Groth16 - Prove

$$[A]_1 \leftarrow [\alpha]_1 + r[\delta]_1 + \sum_{j=0}^{m} a_j [u_j(\tau)]_1$$

$$[B]_2 \leftarrow [\beta]_2 + s[\delta]_2 + \sum_{j=0}^{m} a_j [v_j(\tau)]_2$$

$$[C]_1 \leftarrow \sum_{j=\ell+1}^{m} a_j \left[\frac{\beta u_j(\tau) + \alpha v_j(\tau) + \omega_j(\tau)}{\delta} \right]_1 + \sum_{i=0}^{n-2} h(\omega^i) \left[\frac{\mathcal{L}_i(\tau)(\tau^n - 1)}{\delta} \right]_1 + s[A]_1 + r[B]_1 - rs[\delta]_1$$

where $h(X) = \sum_{i=0}^{n-2} h(\omega^i)\mathcal{L}_i(\tau)$ is the quotient polynomial

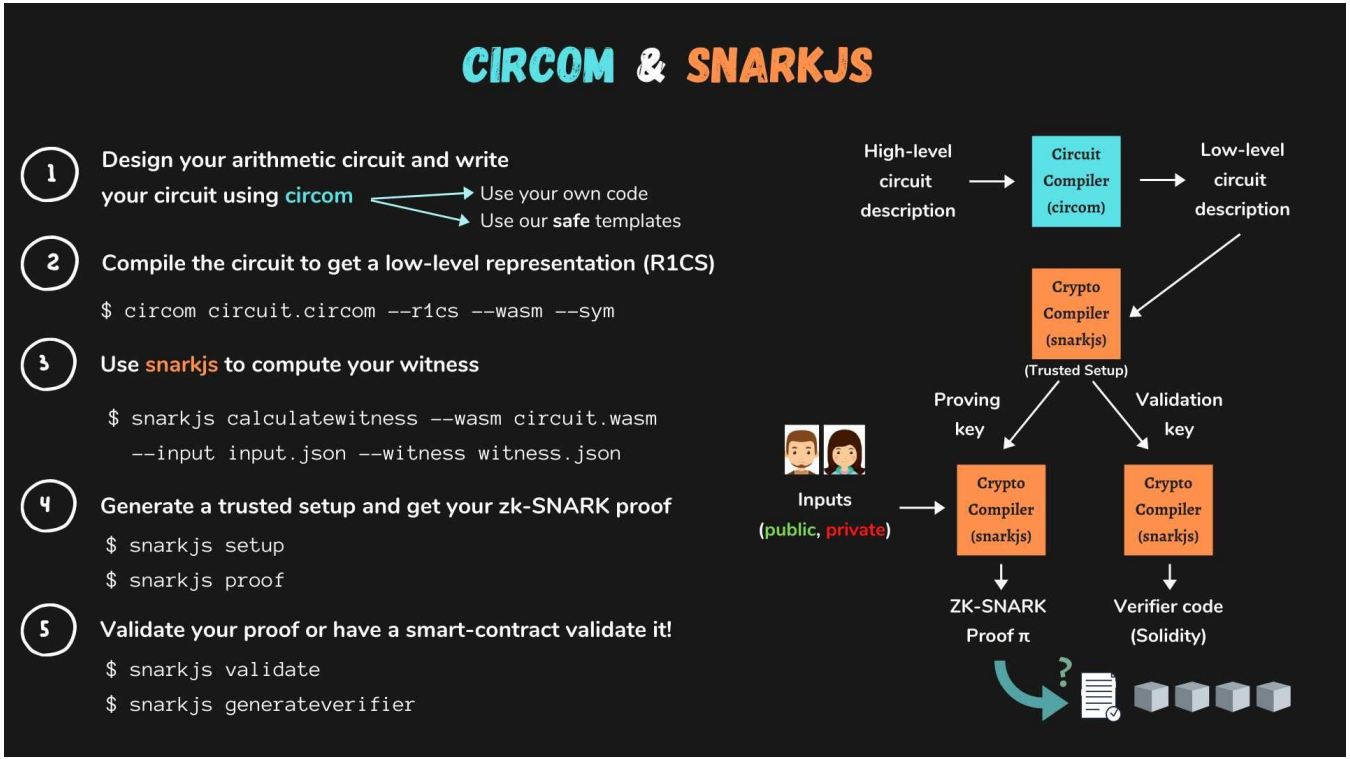
$$\text{proof } \pi \leftarrow ([A]_1, [B]_2, [C]_1) \in \mathbb{G}_1 \times G_2 \times G_1$$

Groth16 - Verify

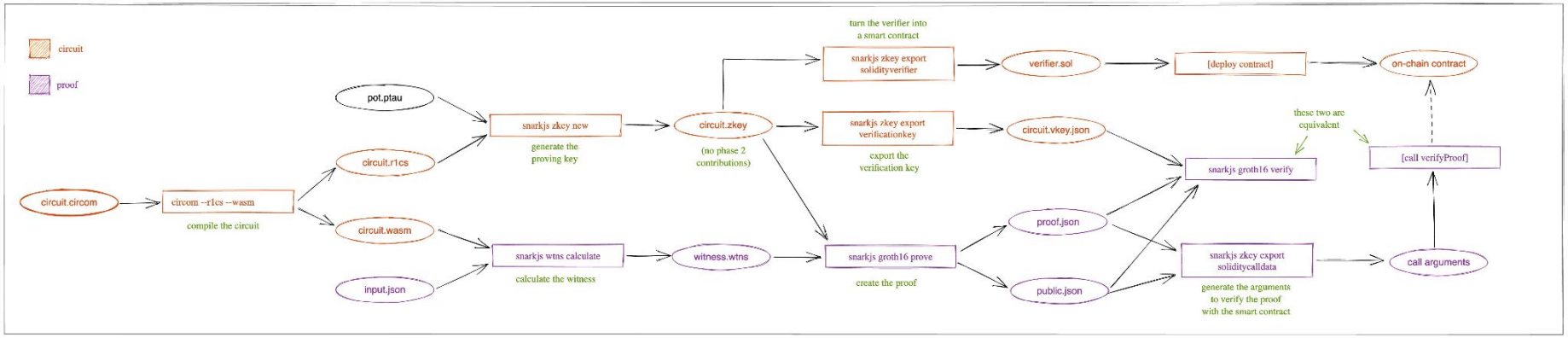
$$e([A]_1, [B]_2) \stackrel{?}{=} e([\alpha]_1, [\beta]_2) + e\left(\sum_{j=0}^{l} a_j \left[\frac{\beta u_j(\tau) + \alpha v_j(\tau) + \omega_j(\tau)}{\gamma}\right]_1, [\gamma]_2\right) + e([C]_1, [\delta]_2)$$

Break

Circom - Architecture



Circom - Architecture (Detailed)



Hands-on: zkREPL

- https://zkrepl.dev
- run: `shift` + `enter`
- save to Github Gist: `Ctrl + s`
- circuit inputs: comments with `/* */`
- signal values derived from inputs: tooltips

Circom Language

- Variables
  - Mutable
  - 2 basic variable types:
    - *field elements* where is a large prime
    - arrays that hold many field elements
- Signals:
  - Immutable
  - 3 types of signals:
    - `input`
    - `output`
    - `intermediate`
- Constraint generation: `===`
- Assigning a value to a signal: `<--`
- Constraint generation + assigning a value to a signal: `<==`
- Reference: `Signals - Circom 2 Documentation`

Circom: Circuit vs Program

- circuit has no memory
- circuit has no loop

Circom Example: isZero()

if `in == 0:`
- `inv = 0`
- `in * inv = 0`
- `out == 1`

if `in != 0:`
- `inv = 1/x`
  - `1/in` is a non-quadratic constraint, can only use `<--` and not `<==`
  - `inv` is unconstrained, can be any value
- `in * inv = 1`
- `out = 0`

```
template IsZero() {
    signal input in;
    signal output out;

    signal inv;

    inv <-- in!=0 ? 1/in : 0;
    out <== 1 - in*inv;

    in*out === 0;
}
```

Circom Example: isZero() Problem

- it is not possible to constrain `inv` directly
- as `inv` is not constrained so one can force `inv` be 0, but this would break the `isZero()` function!

				out
Valid?	in	inv	in * inv	1 - in * inv
yes	x	1/x	1	0
no	*x*	*0* (?)	*0*	*1* (wrong)
yes	0	0	0	1
yes	0	y	0	1

Circom Example: isZero() Problem

- the constraint `in * out === 0` prevent this from happening

				`out`	Constraint (must == 0)
Valid?	`in`	`inv`	`in * inv`	`1 - in * inv`	`in * out`
yes	`x`	`1/x`	`1`	`0`	`0`
no	*`x`*	*`0`*	*`0`*	*`1`*	*`x`*
yes	`0`	`0`	`0`	`1`	`0`
yes	`0`	`y`	`0`	`1`	`0`

Circom: Exercises

Try to do the exercises (don’t peak at the solution!)

- ZKRepl Questions
- ZKRepl Solutions

Other Ideas:

- range check

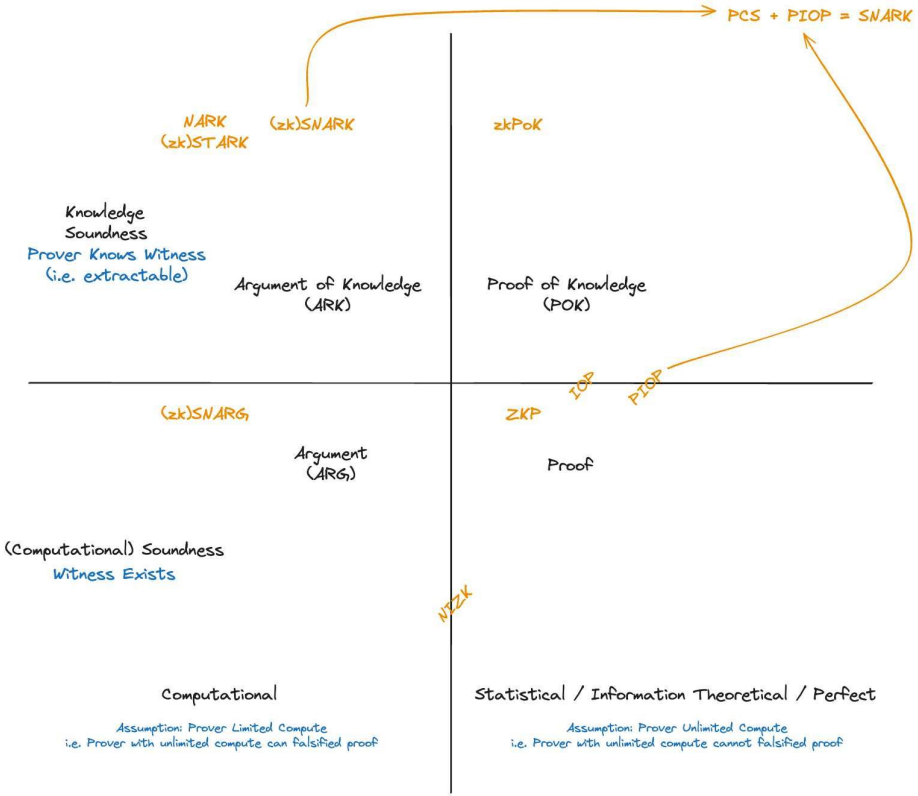
Resources & Reference

- On the Size of Pairing-Based Non-interactive Arguments, by Jens Groth, 2016
- zk-SNARKs: A Gentle Introduction - Anca Nitulescu
- Groth16 - Alin Tomescu
- An overview of the Groth16 proof system
- The Hidden Little Secret in SnarkJS - Kobi Gurkan
- Groth16 Explained - RareSkills

Thank you!

1 min Anonymous Feedback Form

Appendix - Computational vs Statistical



Source: Nicolas Mohnblatt