Programming ZKPs

Here is the general overview to get from an idea to ZKP.

$$\boxed{\textbf{Concept}} \xrightarrow{\text{Coding}} \boxed{\textbf{Program}} \xrightarrow{\text{Compiler}} \boxed{\begin{array}{c}\textbf{R1CS}\\ \textbf{Arithemtic circuit}\end{array}} \xrightarrow{\text{Setup}} \boxed{\textbf{Params}} \xrightarrow{\text{Prove}} \boxed{\textbf{ZKP}}$$

Arithemtic Circuits

Arithmetic circuit is concrete instance of a predicate φ that the prove tries to prove over inputs x, w.

Arithemtic circuits perform over a prime field where

- p a large prime
- $\mathbb{Z}_p$ integers that $\bmod p$ - prime field
- Operations that are performed on the field are: $+, \times, = \pmod 5$
- e.g. $\mathbb{Z}_5$
  - $4 + 5 = 9 \bmod 5 = 4$
  - $4 \times 4 = 16 \bmod 5 = 1$

One way of viewing the Arithemtic Circuits is as systems of field equations over a prime field:
- $w_0 \times w_0 \times w_0 = x$
- $w_1 \times w_1 = x$

Rank 1 Contraint Systems (R1CS)

The most common for ZKP ACs

Representation:
- x: field elements $x_1, ..., x_l$
- w: field elements $w_1, ..., w_{m-l-1}$
- φ: n contraints (equations) of form
  - $\alpha \times \beta = \gamma$
  - where α, β, γ are **affine** (linear with an optional constant added) combinations of variables

Examples:

- $w_2 \times (w_3 - w_2 - 1) = x_1$
  - $\alpha = w_2$
    $\beta = (w_3 - w_2 - 1)$(it's affine)
    $\gamma = x_1$
- $w_2 \times w_2 = w_2$
- $\cancel{w_2 \times w_2 \times w_2 = x_1}$ We have three variables multiplied so this is not an equation of the acceptable format. Instead, we can transform it into 2 equations by introducing another variable:
  - $w_2 \times w_2 = w_4$
  - $w_4 \times w_2 = x_1$

We can also represent R1CS in the matrix form where:

- x: vector of ℓ field element
- w: vector of $m - \ell - 1$ field elements
- φ: matrices $A, B, C \in \mathbb{Z}_p^{n \times m}$ s.t.
  - $z = (1 \parallel x \parallel w) \in \mathbb{Z}_p^{n \times m}$, $\parallel$ - means concatenation
  - which hold when $Az \circ Bz = Cz$, $\circ$ - element-wise product.

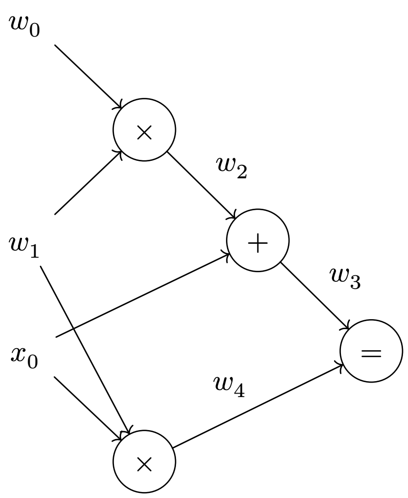
An example of element wise product:

$$A \circ B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \circ \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a \times e & b \times f \\ c \times g & d \times h \end{pmatrix}$$

When taking an inner product of Az, every row of A define an affine combination of variables x and w. So, every row in A, B and C define a single rank 1 contraint.

Example of writing an AC as R1CS

Given the following circuit.



We can transform it to R1CS using the following procedure:

1. Introduce intermediate witness (w) variables
2. Rewrite equations
   - $w_0 \times w_1 = w_2$
   - $w_3 = w_2 + x_0$ (β is *1*, therefore omitted)
   - $w_1 \times x_0 = w_4$
   - $w_3 = w_4$

HDLs for R1CS

As an HDL (hardware description language) we are going to use Circom.

In HDL objects are:

- Wires
- Gates
- Circuits/Subcircuits

Actions are:

- Connect Wires
- Create sub-circuits
- cannot call functions or mutate variables

Circom is an HDL for R1CS:

- Wires: R1CS vars
- Gates: R1CS contraints

It sets values to vars and creates R1CS contraints.

Circom

Let's looks at the basic example:

```
template Multiply() {
  signal input x; // signal is a wire
  signal input y;
  signal output z;

  z <-- x * y // set signal value
  z === x * y // creates a contraint, must rank-1
  // OR z <== x * y
}

component main {public [x]} = Multiply();
```

=== creates a contraint, must rank-1, one side must be linear, the other side must be quadratic

- `template` is a subcircuit.
- `public [x]` describes that `x` is public input in the instance of the template.

Circom Metaprogramming

Circom has following metaprogramming features:

- template args
- Signal arrays
- Vars
  - Mutable
  - Not signals
  - Evaluated at compile-time
- Loops
- If statements
- Array access

```
template RepeatedSquaring(n) {
  signal input x;
  singal output y;

  signal xs[n+1];
  xs[0] <== x;

  for (var i = 0; i < n; i++) {
    xs[i+1] <== xs[i] * xs[i];
  }
  y <== xs[n]
}
component main {public [x]} = RepeatedSquaring(1000);
```

Circom witness Computation and Sub-circuits

Witness computation is more general than R1CS

- `<--` is more general than `===`, you can put any value since it justs sets the value, it doesn't create a constraint.

```
template NonZero() {
  signal input in;
  signal inverse;
  inveser <-- 1 / in; // not R1CS
  1 === in * signal' // is R1CS, creates constraint
}
```

`components` hold sub-circuits

- Accesses input/outputs with dot notation

```
template Main() {
  signal input a; signal input b;
  component nz = NonZero();
  nz.in <== a;
  0 === a * b;
}
```