Veridise.

Abstracting ZK Circuits with Graphs

Daniel Dominguez
Security Engineer, Veridise

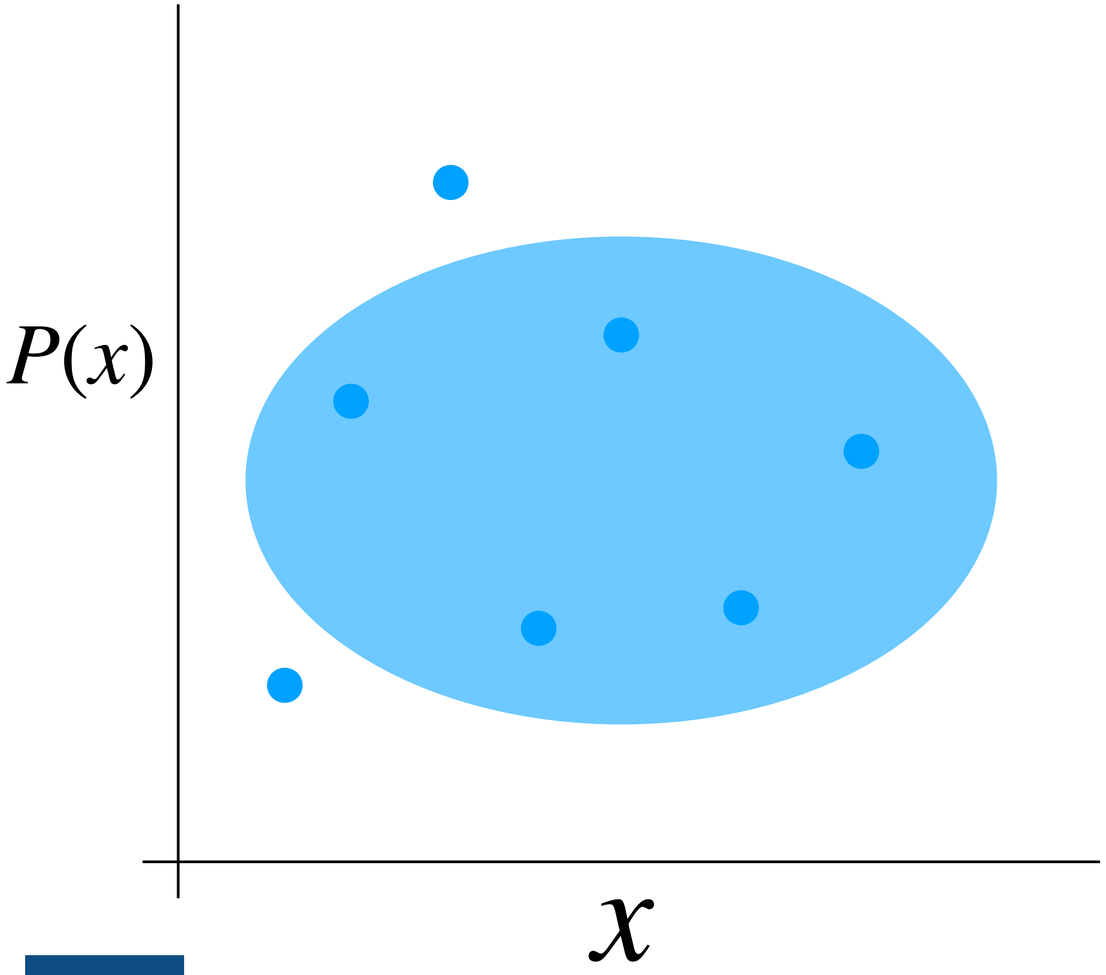
Pre-requisites

- Please send an onboard request for SaaS
- The instructions are posted in Discord
- My colleagues will handle them during the lecture
- We will use it at the end of the lecture and you will need it for the quiz!
- Day 1’s quiz is still running!

Recap

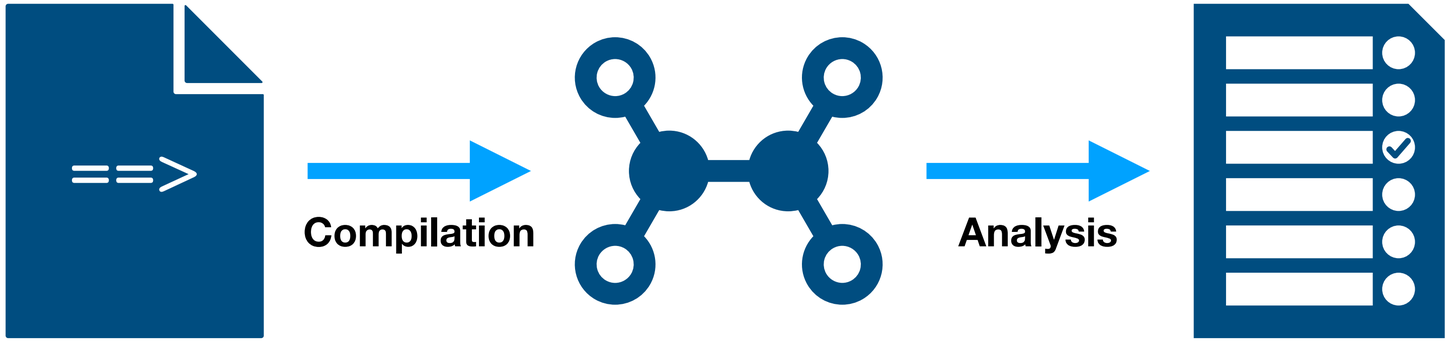
- We are working with circom
- A language for building zk circuits that have two parts
  - The witness generator generates a proof
  - The proving system checks the proof
- One party generates the proof while the other checks it

What is static analysis?



- Unit testing the circuit may not be enough
- To solve some of the issues we use Static Analysis
  - It analyses the circuit without running it
  - May catch issues that can't be seen by running the circuit

Static analysis architecture

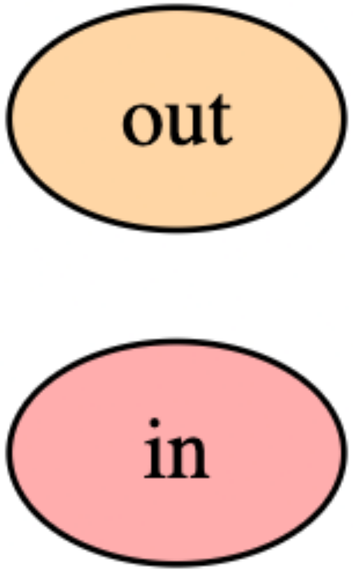


We are going to take a look at this
to understand how zkVanguard works

Veridise.

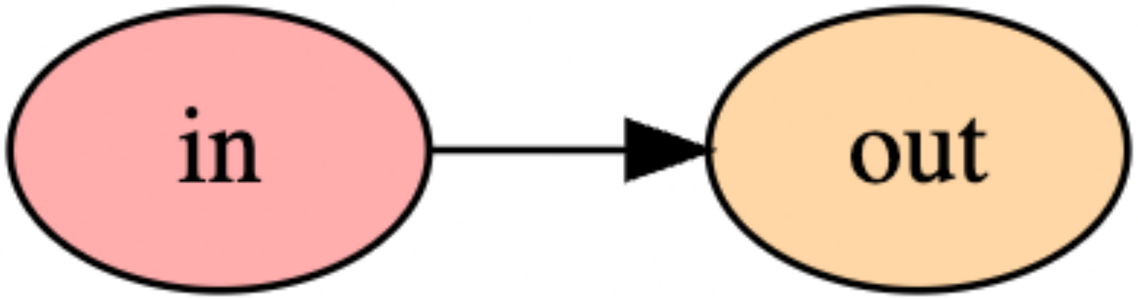
Elements of a circom circuit

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

    out <-- in;
    out === in;
}
```



Elements of a circom circuit

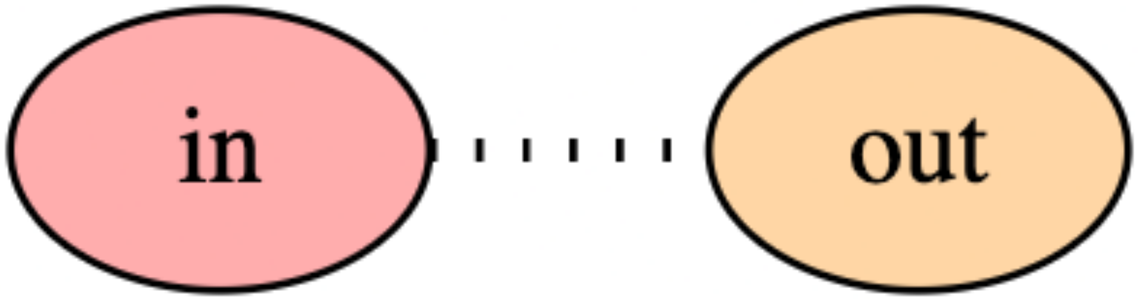
```
template comp() {
    signal input in;
    signal output out;

    out <-- in;
    out === in;
}
```



Veridise.

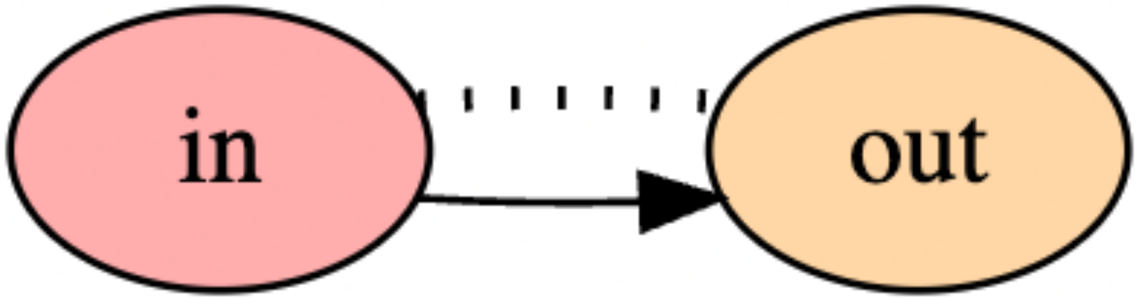
Elements of a circom circuit

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

    out <-- in;
    out === in;
}
```



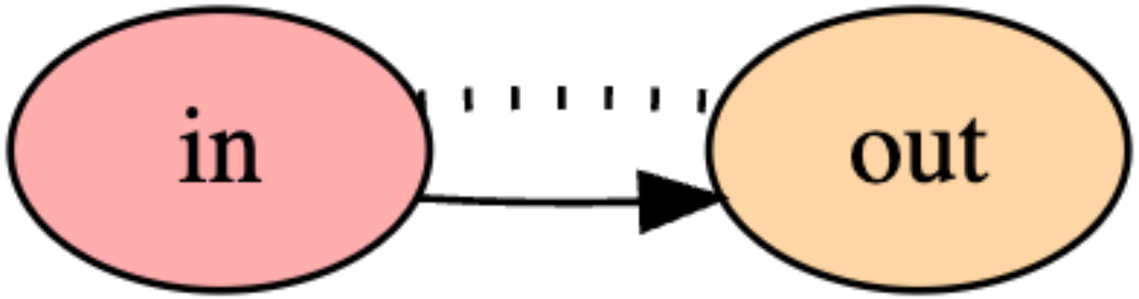
Elements of a circom circuit

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

    out <-- in;
    out === in;
}
```



Elements of a circom circuit

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

    out <== in;
}
```



This graph is the Constraint-Dataflow graph (CDG)

Circom bugs

```
template adder() {
    signal input b1;
    signal input b2;
    signal input carry;

    signal output v;
    signal output c_out;

    v <-- (b1 + b2 + carry) % 2;
    v * (v - 1) === 0;
    c_out <-- (b1 + b2 + carry) \ 2;
    c_out * (c_out - 1) === 0;
}
```

$$\{b1 \mapsto 1, b2 \mapsto 0, carry \mapsto 0, v \mapsto 1, c_{out} \mapsto 0\}$$

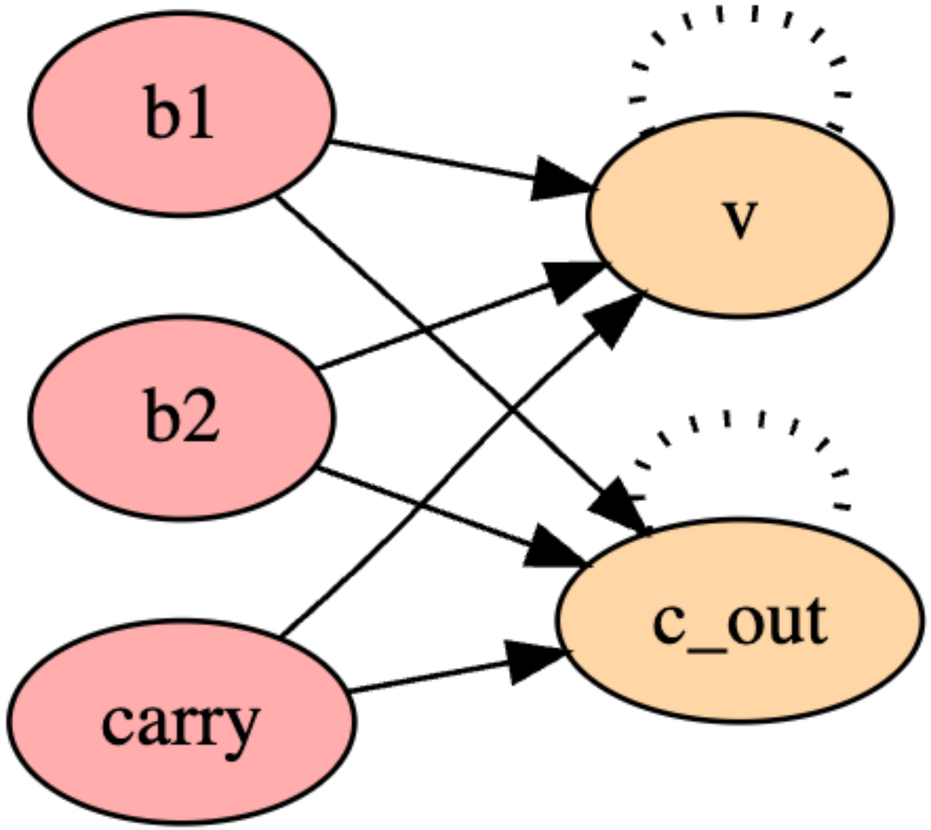
$$1 + 0 = 1$$

$$\{b1 \mapsto 1, b2 \mapsto 0, carry \mapsto 0, v \mapsto 1, c_{out} \mapsto 1\}$$

$$1 + 0 \neq 3$$

Circom bugs

```
template adder() {
    signal input b1;
    signal input b2;
    signal input carry;

    signal output v;
    signal output c_out;

    v <-- (b1 + b2 + carry) % 2;
    v * (v - 1) === 0;
    c_out <-- (b1 + b2 + carry) \ 2;
    c_out * (c_out - 1) === 0;
}
```

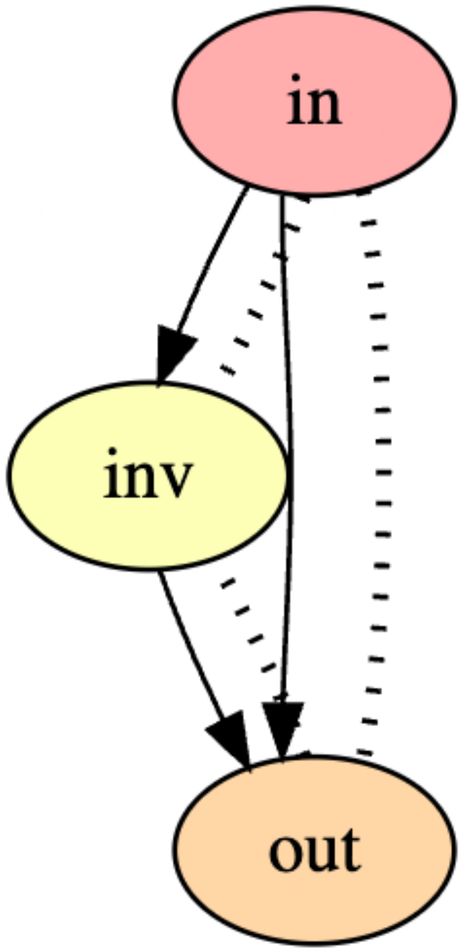


Veridise.

CDG recap

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

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

    out <== -in * inv + 1;
    in * out === 0;
}
```

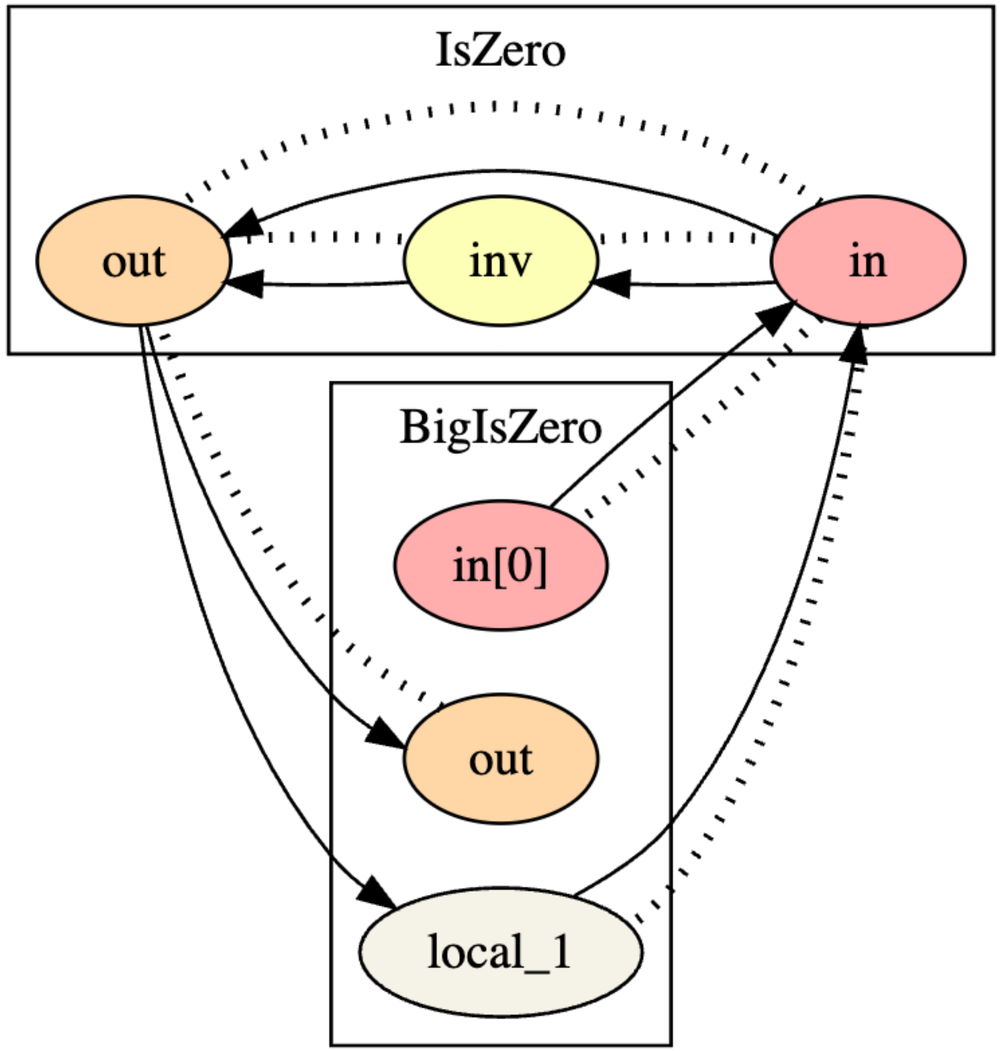
Insufficient for reasoning about, for example:
- **Proper usage of subcomponents**
- **Checking is the circuit is deterministic**
- **Division by zero issues**

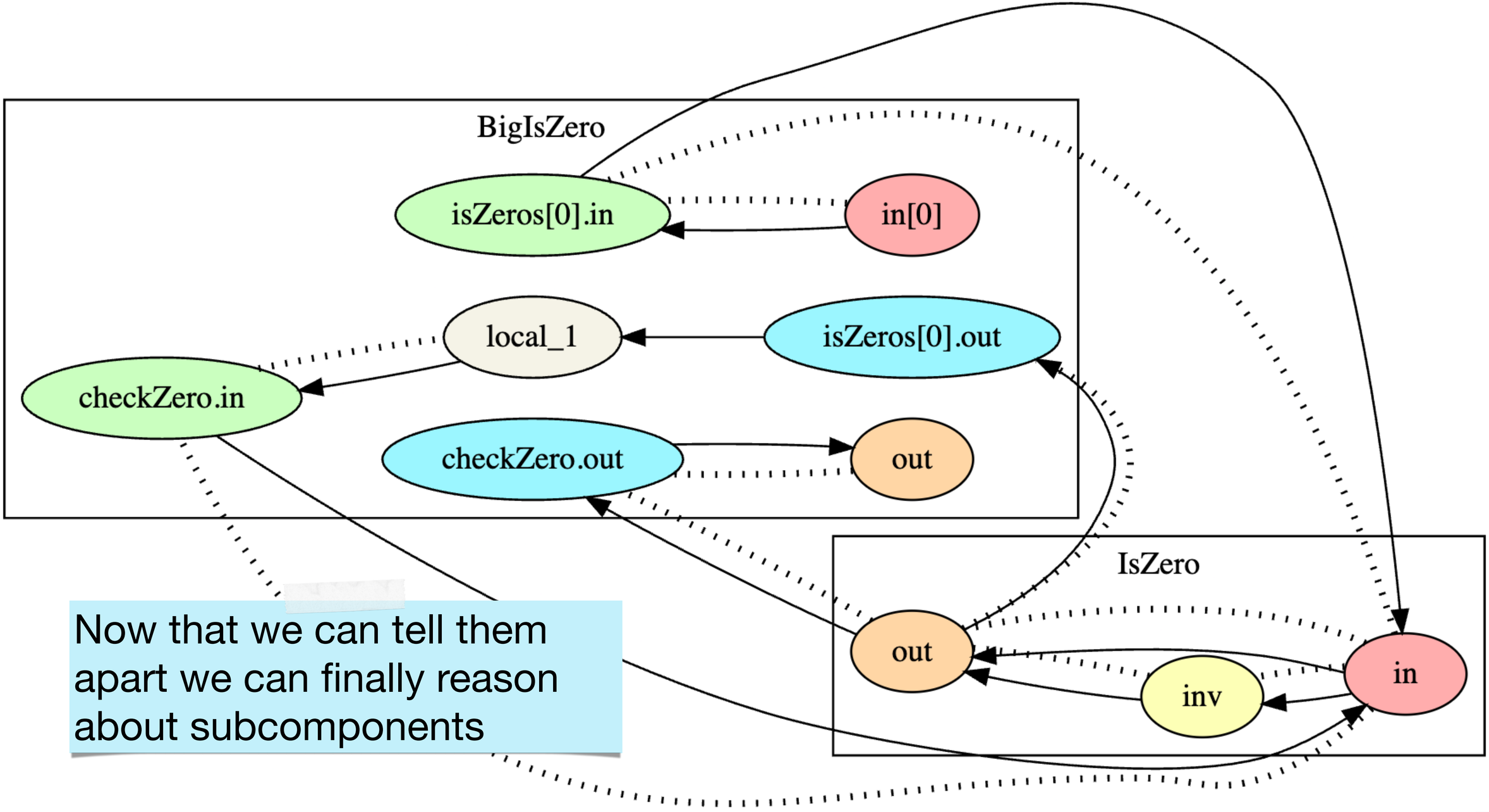


Veridise.

Subcomponents in the CDG

```
template BigIsZero(k) {
    signal input in[k];
    signal output out;

    component isZeros[k];
    var total = k;
    for(var i = 0; i < k; i++) {
        isZeros[i] = IsZero();
        isZeros[i].in <== in[i];
        total -= isZeros[i].out;
    }
    component checkZero = IsZero();
    checkZero.in <== total;
    out <== checkZero.out;
}
```





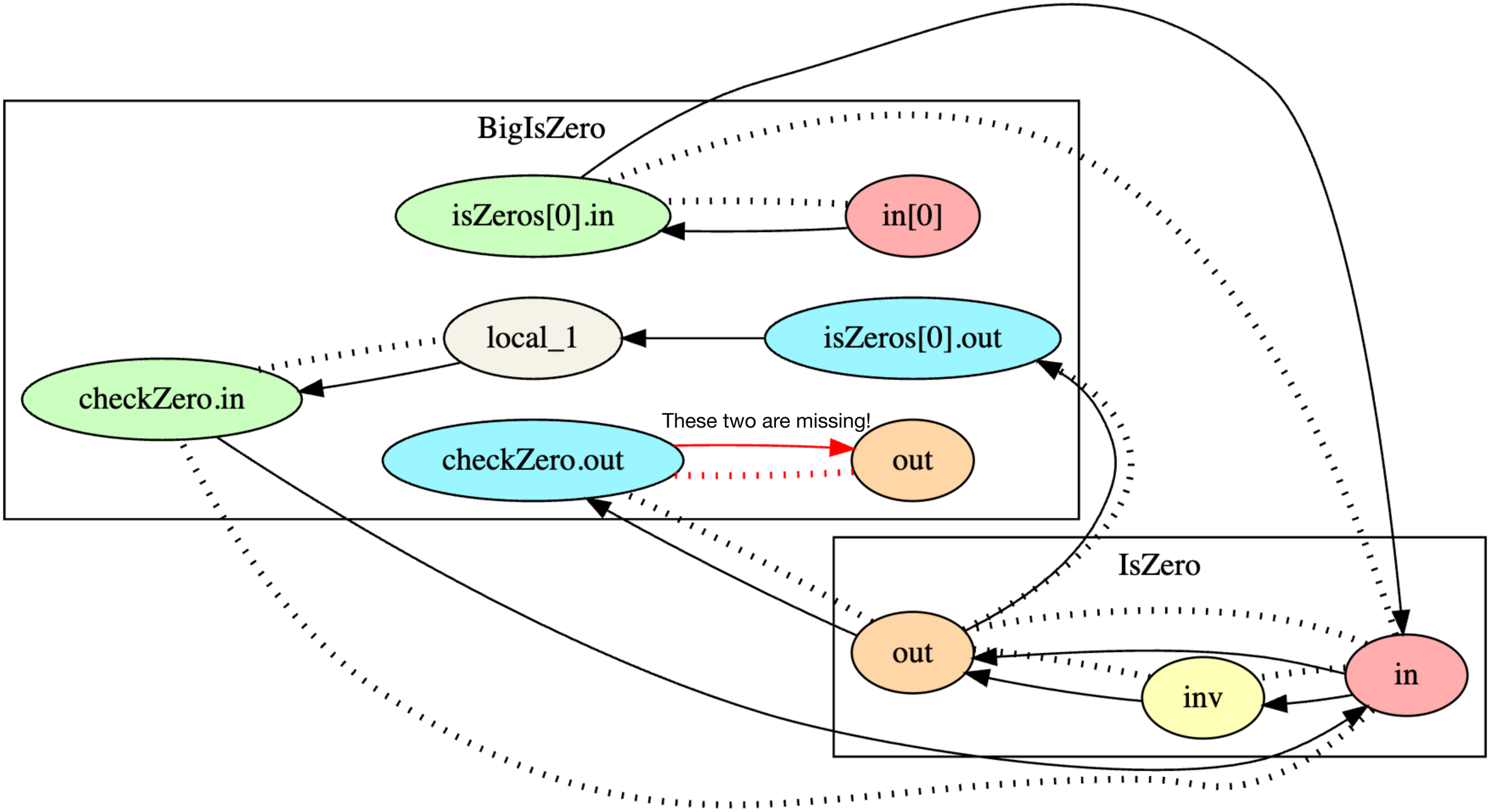
Now that we can tell them apart we can finally reason about subcomponents

Veridise.

Subcomponents in the CDG

```
template BigIsZero(k) {
    signal input in[k];
    signal output out;

    component isZeros[k];
    var total = k;
    for(var i = 0; i < k; i++) {
        isZeros[i] = IsZero();
        isZeros[i].in <== in[i];
        total -= isZeros[i].out;
    }
    component checkZero = IsZero();
    checkZero.in <== total;
//     out <== checkZero.out;
}
```

- One kind of issue circom code could have is that the output of the subcomponent is never used
- In some cases the output of the subcomponent is not constrained internally and it's part of the contract to constraint it
- If we don't we would be under-constraining the circuit


BigIsZero
isZeros[0].in
in[0]
local_1
isZeros[0].out
checkZero.in
These two are missing!
checkZero.out
out
IsZero
out
in
inv

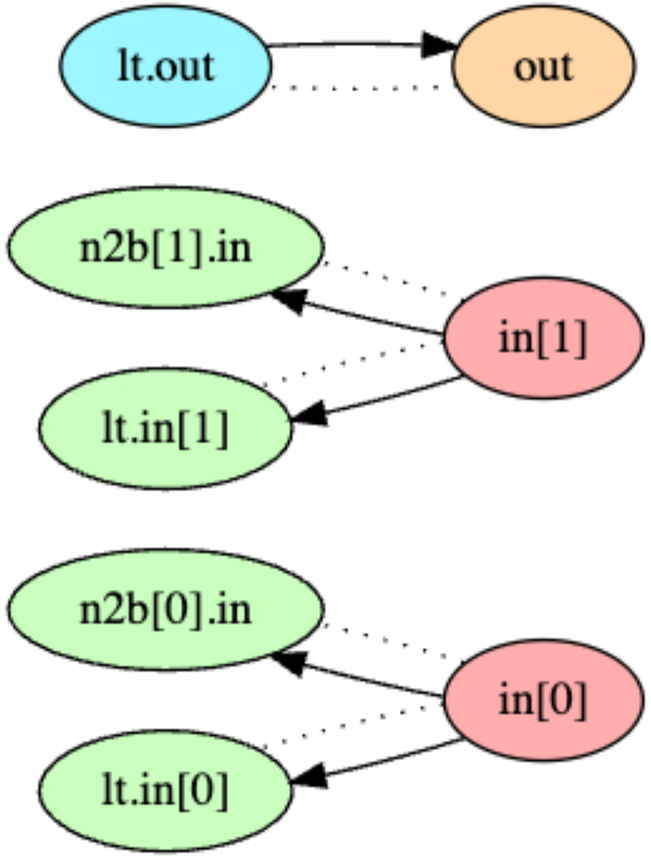
Subcomponents in the CDG

```
template LessThan(n) {
    assert(n <= 252);
    signal input in[2];
    signal output out;

    component n2b = Num2Bits(n+1);

    n2b.in <== in[0]+ (1<<n) - in[1];

    out <== 1-n2b.out[n];
}
```

- Some templates depend on the inputs being of a particular bit width
- Both inputs of LessThan are expected to be of less than 253 bits and both the same
- The caller of LessThan needs to ensure that
  - In circom the way is with Num2Bits

Subcomponents in the CDG

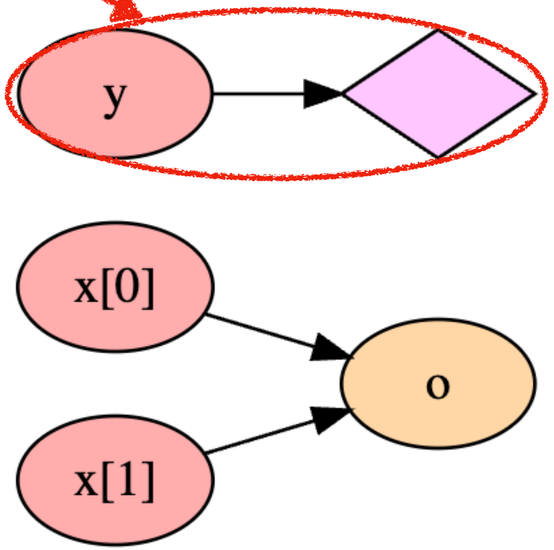
```
template LessThan(n) {
    signal input in[2];
    signal output out;

    component n2b[2];
    n2b[0] = Num2Bits(8);
    n2b[0].in <== in[0];

    n2b[1] = Num2Bits(8);
    n2b[1].in <== in[1];

    component lt = LessThan(8);
    lt.in[0] <== in[0];
    lt.in[1] <== in[1];

    out <== lt.out;
}
```



Veridise.

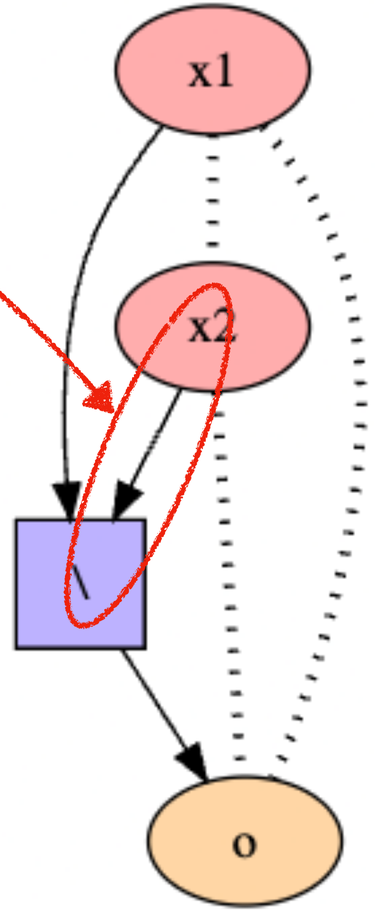
Nondeterministic witness

```
template mux() {
    signal input x[2];
    signal input y;
    signal output o;

    o <-- x[y % 2 == 0 ? 1 : 0];
}
```



Division by zero

```
template div() {
    signal input x1;
    signal input x2;
    signal output o;
    o <-- x1 / x2;
    o * x2 === x1;
}
```



Veridise.

Generating CDGs

Now we are gonna see how to generate this graphs in SaaS