iden3

iden3.io

@identhree

OKIMS

circom and snarkJS

https://github.com/iden3/circom

https://github.com/iden3/snarkjs

@jbaylina

What’s a zero non interactive knowledge proof

Given a circuit and its input, the prover can run the circuit and generate a proof.

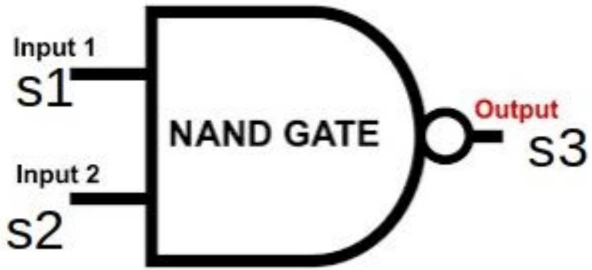
Given the proof and the public input/output, I can proof to a verifier that who generated that proof knew the private input and executed the circuit.

This proof does not reveal anything about the private input

Simple Circuit

Truth table

s1	s2	s3
0	0	1
0	1	1
1	0	1
1	1	0



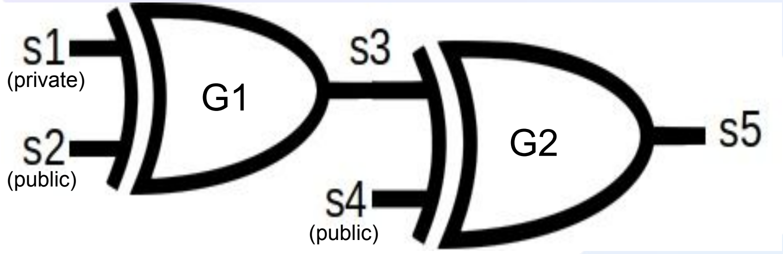
Constraints system

$$s_3 = 1 - s_1 s_2$$

Circom circuit

```
template NAND() {
    signal input s1;
    signal input s2;
    signal output s3;

    s3 <== 1 - s1*s2;
}

component main = NAND()
```

Composite Circuit



Constraints system

$$\begin{cases} s_3 = s_1 + s_2 - 2s_1s_2 \\ s_5 = s_3 + s_4 - 2s_3s_4 \end{cases}$$

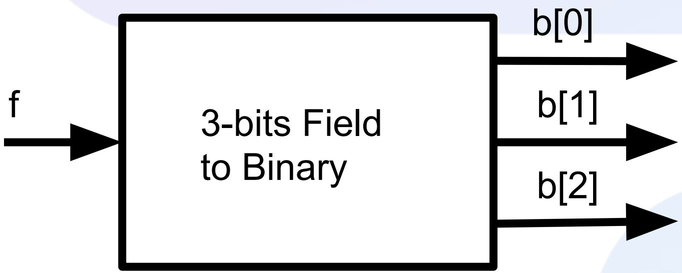
nand.circom

```
template NAND() {
    signal input a;
    signal input b;
    signal output out;

    out <== 1 - a*b;
}
```

composite.circom

```
include "nand.circom"

template Composite() {
    signal private input s1;
    signal input s2;
    signal input s3;
    signal output s5;
    signal s4

    component G1 = NAND();
    component G2 = NAND();

    s1 ==> G1.a;
    s2 ==> G1.b;
    G1.out ==> G2.a;
    s3 ==> G2.b;
    G2.out ==> s5;
}

component main =
Composite();
```

Touching metal



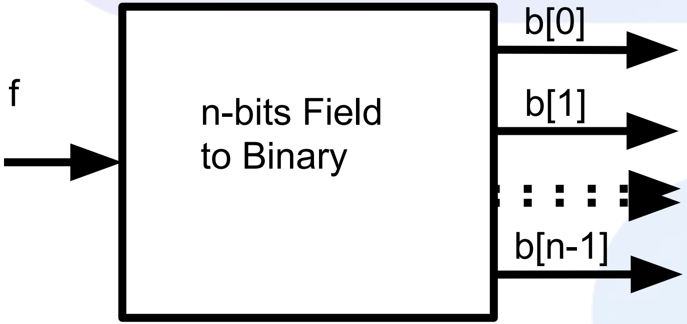

```
template Num2Bits3() {
    signal input f;
    signal output b[3];

    for (i=0; i<3; i++) {
        b[i] <-- (f>>i) & 1;
        b[i] * (b[i]-1) === 0;
    }
    b[0] + 2*b[1] + 4*b[2] ===
f;
}
```

Constraints system

$$
\begin{cases}
b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 & = & f \\
b_0 \cdot (b_0 - 1) & = & 0 \\
b_1 \cdot (b_1 - 1) & = & 0 \\
b_2 \cdot (b_2 - 1) & = & 0
\end{cases}
$$

```
==>    -->
       ===

<==    <--
       ===
```

Parametric templates

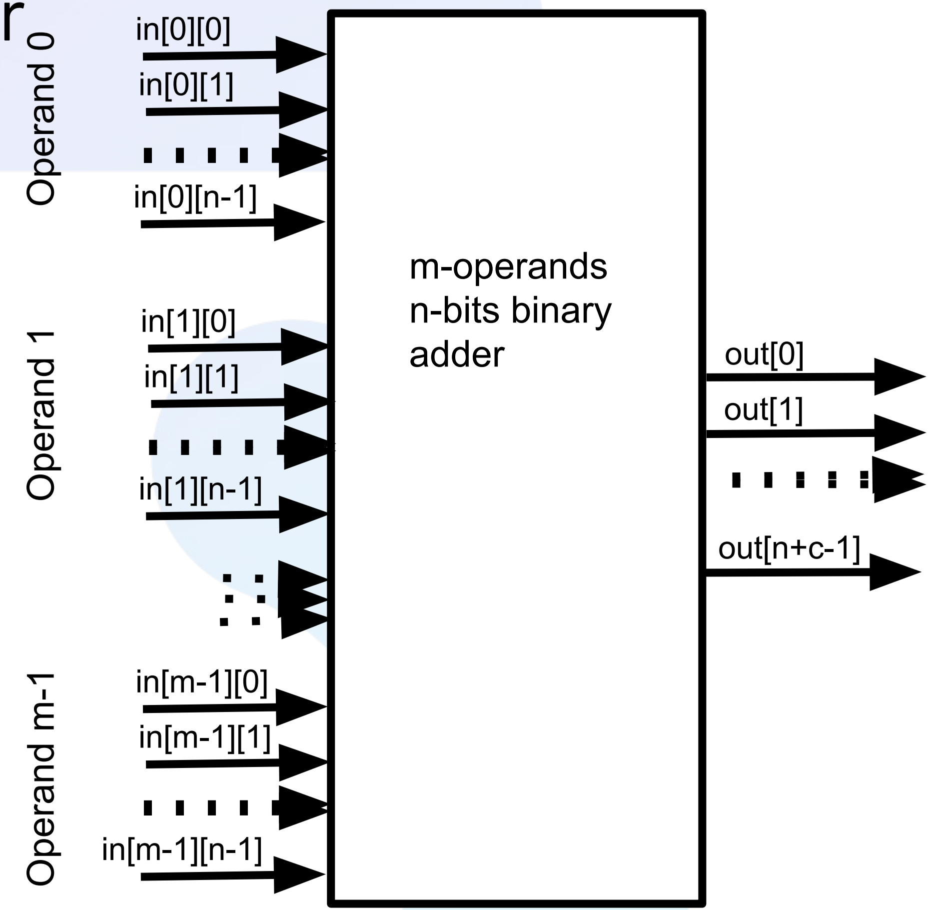


Constraints system

$$
\begin{cases}
b_0 \cdot 2^0 + b_1 \cdot 2^1 + \quad \ldots \quad + b_{n-1} \cdot 2^{n-1} = f \\
b_0 \cdot (b_0 - 1) = 0 \\
b_1 \cdot (b_1 - 1) = 0 \\
\ldots \\
b_{n-1} \cdot (b_{n-1} - 1) = 0
\end{cases}
$$

```
template Num2Bits(n) {
    signal input f;
    signal output b[n];
    var lc1=0;

    for (var i = 0; i<n; i++) {
        b[i] <-- (f >> i) & 1;
        b[i] * (b[i] - 1) === 0;
        lc1 += b[i] * 2**i;
    }

    lc1 === f;
}

component main = Num2Bits(253);
```

Binary adder



Constraints system for generic binary adder

$$
\begin{cases}
\quad in_{0,0} \cdot 2^0 + in_{0,1} \cdot 2^1 + \quad \dots \quad + \quad in_{0,n-1} \cdot 2^{n-1} \quad + \\
+ \quad in_{1,0} \cdot 2^0 + in_{1,1} \cdot 2^1 + \quad \dots \quad + \quad in_{1,n-1} \cdot 2^{n-1} \quad + \\
\quad \dots \qquad\qquad\qquad\qquad\qquad\qquad \dots \\
+ \quad in_{m-1,0} \cdot 2^0 + in_{m-1,1} \cdot 2^1 + \quad \dots \quad + \quad in_{m-1,n-1} \cdot 2^{n-1} \quad = \\
= \quad out_0 \cdot 2^0 + out_1 \cdot 2^1 + \quad \dots \quad + \quad out_{n+c-1} \cdot 2^{n+c-1} \\
\\
out_0 \cdot (out_0 - 1) = 0 \\
out_1 \cdot (out_1 - 1) = 0 \\
\dots \\
out_{n+c-1} \cdot (out_{n+c-1} - 1) = 0
\end{cases}
$$

```
function nbits(a) {
    var n = 1;
    var r = 0;
    while (n-1<a) {
        r++;
        n *= 2;
    }
    return r;
}

template BinSum(n, m) {
    var nout = nbits((2**n -1)*m);
    signal input in[m][n];
    signal output out[nout];

    var lin = 0;
    var lout = 0;

    var k;
    var j;

    for (k=0; k<n; k++) {
        for (j=0; j<m; j++) {
            lin += in[j][k] * 2**k;
        }
    }

    for (k=0; k<nout; k++) {
        out[k] <-- (lin >> k) & 1;

        // Ensure out is binary
        out[k] * (out[k] - 1) === 0;

        lout += out[k] * 2**k;
    }

    // Ensure the sum;
    lin === lout;
}

component main = BinSum(64,2)
```

Baby Jub

The Order of the field is the order of the bn128 curve.

Field operations matches with the field of the circuit.

The curve is safe.

$$E_{E,a,d}: ax^2 + y^2 = 1 + dx^2y^2$$

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

a=1
d=9706598848417545097372247223557719406784115219466060233080913168975159366771

```
template BabyAdd() {
    signal input x1;
    signal input y1;
    signal input x2;
    signal input y2;
    signal output xout;
    signal output yout;

    signal beta;
    signal gamma;
    signal delta;
    signal tau;

    var a = 168700;
    var d = 168696;

    beta <== x1*y2;
    gamma <== y1*x2;
    delta <== (-a*x1+y1)*(x2 + y2);
    tau <== beta * gamma;

    xout <-- (beta + gamma) / (1+ d*tau);
    (1+ d*tau) * xout === (beta + gamma);

    yout <-- (delta + a*beta - gamma) / (1-d*tau);
    (1-d*tau)*yout === (delta + a*beta - gamma);
}
```

CircomLib - https://github.com/iden3/circomlib

- Binary to Field and Field to Binary converters (With strict option)
- BabyJub
    - Addition / Constant scalar multiplication / Variable scalar multiplication
    - Edwards and Montgomery conversion
    - Point compression/decompression
- EdDSA
- Pedersen commitments
- MiMC7 Hash
- Sparse merkle trees processors to add/update/remove elements.
- Sparse merkle tree verifiers to verify inclusion and exclusion.
- Comparators
- Logical operators like adders and Binary Gates.
- SHA256 Hash function.
- ... and more

snarkJS

- Independent implementation of zkSnarks protocol.
- Supports classical pinocchio format and Groth16
- Purely written in javascript
- Works together with circom
- Browser compatible.
- Fully Open source.
- Npm library with a javascript API and a command line interface.

Basic Tutorial

```
# npm install -g circom
# npm install -g snarkjs
```

Create a working directory:

```
# mkdir factor
# cd factor
```

Create a basic circuit:

```
# echo "
template Multiplier() {
    signal private input a;
    signal private input b;
    signal output c;

    c <== a*b;
}

component main = Multiplier();
" > multiplier.cicom
```

Compile the circuit

```
# circom multiplier.cicom -o multiplier.json
```

Snarkjs help

```
# snarkjs --help
```

Print Info on the circuit

```
# snarkjs info -c multiplier.json
```

Print raw constraints

```
# snarkjs printconstraints -c multiplier.json
```

Setup: Once per circuit and generally done in a multiparty ceremony

```
# snarkjs setup -c multiplier.json --pk proving_key.json --vk
verification_key.json --protocol groth
```

Calculate the witness. (Run the circuit)

```
# echo "{\"a\": 3, \"b\": 11}" >in.json
# snarkjs calculatewitness -c multiplier.json -i in.json -w witness.json
```

Generate the proof

```
# snarkjs proof -w witness.json --pk proving_key.json --pub pub.json --proof=proof.json
```

Verify the proof

```
# snarkjs verify --vk verification_key.json --proof proof.json --pub pub.json
```

Generate solidity smart contract to verify the proof

```
# snarkjs generateverifier --vk verification_key.json --verifier verifier.sol
```

You can use remix to deploy this contract in any Ethereum testnet or main net

Generate the prepared call parameters of a proof to be verified with the smart contract

```
# snarkjs generatecall --proof proof.json --pub pub.json
```

Current status of the project

- Reimplementing snarkjs in pure WebAssemby
- Optimizing witness generation in Circom.
- Working in a GPU cluster to generate 2^28 constraints proofs.

Thank you

www.iden3.io

@identhree

Jordi Baylina

@jbaylina