

circom

Saravanan Vijayakumaran

Department of Electrical Engineering
Indian Institute of Technology Bombay

June 8, 2024

circom

- circom = circuit compiler
- A toolchain for expressing statements that can be proved in zero-knowledge
- Uses Groth16 as the proving system
 - <https://eprint.iacr.org/2016/260>
- Proofs can be verified in an Ethereum smart contract
 - Gas costs $\approx 181,000 + 6,150 \times k$ where k is the number of public inputs
 - For gas price 20 gwei/gas and \$3000/ETH, it costs \$12 to verify a proof with 3 public inputs
- Used by Tornado Cash, Dark Forest

Proving Statements using SNARKs

- SNARK = Succinct Non-interactive Arguments of Knowledge
 - Protocols that enable verifiable computation
 - Succinct = Proofs are smaller than size of statement
 - Non-interactive = A single message from prover to verifier
 - Argument = Soundness only guaranteed for PPT provers
 - Knowledge = Prover knows a witness (secret information)
- zkSNARK = Zero-Knowledge SNARK
- To prove statements using SNARKs, they have to be expressed as **arithmetic circuits**
 - Circuit variables are prime field elements
 - Only addition and multiplication operators are available
- Rank-1 Constraint System (R1CS) is one method for arithmetizing statements

Prime Fields

- $\mathbb{F}_p = \{0, 1, 2, \dots, p-1\}$ where p is prime
- $+$ and $*$ defined on \mathbb{F}_p as

$$x + y = x + y \bmod p,$$

$$x * y = xy \bmod p.$$

- \mathbb{F}_5

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

*	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

- Subtraction is addition with additive inverse, $x - y = x + (p - y)$
- Division is multiplication by multiplicative inverse

$$\frac{x}{y} = x * y^{-1}$$

Rank-1 Constraint Systems

- Statement is represented using quadratic constraints of the form

$$\left(u_0 + \sum_{i=1}^n a_i u_i\right) \cdot \left(v_0 + \sum_{i=1}^n a_i v_i\right) = \left(w_0 + \sum_{i=1}^n a_i w_i\right)$$

- The u_i, v_i, w_i values are determined by the statement
- The a_i 's are **witness** values specific to the instance
- Why rank 1?

$$\begin{aligned} & \left(u_0 + \sum_{i=1}^n a_i u_i\right) \cdot \left(v_0 + \sum_{i=1}^n a_i v_i\right) = \langle \mathbf{u}, (\mathbf{1}, \mathbf{a}) \rangle \cdot \langle \mathbf{v}, (\mathbf{1}, \mathbf{a}) \rangle \\ & = \underbrace{\begin{bmatrix} 1 & \mathbf{a} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{bmatrix}}_M \begin{bmatrix} v_0 & v_1 & \cdots & v_n \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{a}^T \end{bmatrix} \end{aligned}$$

- The matrix M has rank one

Boolean Gates in R1CS

- AND and OR Gates

- If $a \in \mathbb{F}_p = \{0, 1, \dots, p-1\}$ satisfies $a(1-a) = 0$, then $a \in \{0, 1\}$
- Given $a_1(1-a_1) = 0, a_2(1-a_2) = 0$
 - $a_3 = a_1 \wedge a_2$ is expressed as

$$a_1 a_2 = a_3$$

- $a_3 = a_1 \vee a_2$ is expressed as

$$(1 - a_1) \cdot (1 - a_2) = 1 - a_3$$

- XOR Gate

- Given $a_1(1-a_1) = 0, a_2(1-a_2) = 0$, we can express $a_3 = a_1 \oplus a_2$ as

$$(a_1 + a_1) \cdot a_2 = a_1 + a_2 - a_3.$$

- If $a_2 = 0$, then $a_3 = a_1$
- If $a_2 = 1$, then $a_3 = 1 - a_1$

- NOT Gate

- Given $a_1(1-a_1) = 0$, we can express $a_2 = \neg a_1$ as

$$(1 - a_1) \cdot 1 = a_2.$$

Signals in circom

- Example circuit

```
pragma circom 2.1.6;

template Multiplier2(){
  //Declaration of signals
  signal input in1;
  signal input in2;
  signal tmp;
  tmp <== in1 * in2;
  signal output out <== tmp * in2;
}

component main {public [in1, in2]} = Multiplier2();
```

- **Signals:** Field elements that appear in an arithmetic circuit
- A signal is immutable; once assigned it cannot change
- A circuit is made up of subcircuits (components)
- In a component, signals can be inputs, outputs, or neither
- Input signals are private by default
- List of public signals are declared in the main component

The <== operator

- Recall the R1CS constraint structure

$$\left(u_0 + \sum_{i=1}^n a_i u_i\right) \cdot \left(v_0 + \sum_{i=1}^n a_i v_i\right) = \left(w_0 + \sum_{i=1}^n a_i w_i\right)$$

- The === operator constrains a linear combination to equal a product of two linear combinations

```
a*(a-1) === 0;
```

- The <== operator is a combination of an assignment operator <-- and the === operator

```
out <-- a*b;  
out === a*b;  
// The line below is equivalent to the above statements  
out <== a*b;
```

- Sometimes the <-- and === operators cannot be combined

```
a <-- b/c;  
a*c === b;
```


The <== operator

- Only quadratic constraints are allowed while using the <== operator

```
a*(a-1) == 0;
```

- Recall the example circuit

```
pragma circom 2.1.6;

template Multiplier2(){
    signal input in1;
    signal input in2;
    signal tmp;
    tmp <== in1 * in2;
    signal output out <== tmp * in2;
}

component main {public [in1, in2]} = Multiplier2();
```

- Replacing the calculation of `out` with the following fails

```
signal output out <== in1 * in2 * in2;
```

zkREPL

- A web-based IDE for circom at <https://zkrepl.dev/>
- Circuit inputs are provided in JSON format as a comment
- Example circuit

```
pragma circom 2.1.6;

template Multiplier2(){
  signal input in1;
  signal input in2;
  signal tmp;
  tmp <== in1 * in2;
  signal output out <== tmp * in2;
}

component main {public [in1, in2]} = Multiplier2();

/* INPUT = {
  "in1": "5",
  "in2": "8"
} */
```

- Note that there is no trailing comma after the last key-value pair in INPUT

zkREPL Examples



Example: AND Gate

- Given $a(1 - a) = 0, b(1 - b) = 0$, that AND of a and b is ab
- Circom circuit

```
pragma circom 2.1.6;

template And() {
    signal input a;
    signal input b;
    signal output out;

    a * (1-a) === 0;
    b * (1-b) === 0;
    out <== a*b;
}

component main {public [a, b]} = And();

/* INPUT = {
    "a": "1",
    "b": "0"
} */
```

Example: OR Gate

- Given $a(1 - a) = 0, b(1 - b) = 0$, that OR of a and b is $1 - (1 - a)(1 - b)$
- Circom circuit

```
pragma circom 2.1.6;

template Or(){
    signal input a;
    signal input b;
    signal output out;

    a * (1-a) === 0;
    b * (1-b) === 0;
    out <== 1 - (1-a) * (1-b);
}

component main {public [a, b]} = Or();

/* INPUT = {
    "a": "0",
    "b": "0"
} */
```

Arrays of Signals and Components

- Signals can be organized in arrays

```
signal input in[3];  
signal output out[2];  
signal intermediate[4];
```

- Components (subcircuits) can also be organized as arrays

```
template fun(N){  
    signal output out;  
    out <== N;  
}  
  
template all(N){  
    component c[N];  
    for(var i = 0; i < N; i++){  
        c[i] = fun(i);  
    }  
}  
  
component main = all(5);
```

- **Aside:** *var* keyword denotes mutable variables that hold non-signal data

Example: Multiplexer

- Multiplexer circuit

```
template MultiMux(n) {  
    signal input c[n][2]; // Inputs  
    signal input s;       // Selector  
    signal output out[n];  
  
    s * (s-1) === 0;  
  
    for (var i=0; i<n; i++) {  
        out[i] <== (c[i][1] - c[i][0])*s + c[i][0];  
    }  
}  
  
component main = MultiMux(3);
```

- If $s=0$, then $out[i] <== c[i][0]$
- If $s=1$, then $out[i] <== c[i][1]$

Example: Zero Equality Check

- Suppose we want to check that an input is zero

```
template IsZero() {  
    signal input in;  
    signal output out;  
  
    signal inv;  
  
    inv <-- in!=0 ? 1/in : 0;  
  
    out <== -in*inv +1;  
    in*out === 0;  
}
```

- The value of `inv` is non-deterministic advice
- If `in` is zero, then `out <== 1`
- If `in` is non-zero, then `out` must be zero

Example: Bit Decomposition

- Suppose we want to decompose a signal in the range $\{0, 1, 2, \dots, 2^n - 1\}$ into n bits

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

    var e2=1;
    for (var i = 0; i<n; i++) {
        out[i] <-- (in >> i) & 1;
        out[i] * (out[i] -1 ) === 0;
        lc1 += out[i] * e2;
        e2 = e2+e2;
    }

    lc1 === in;
}
```

- The value of `out[i]` is derived from `in`
- `out[i]` is constrained to be a bit
- `e2` contains powers of 2
- The final constraint `lc1 === in` will be satisfied if `in` fits in n bits

Example: Comparator

- Given two inputs, suppose we want to check that the first is less than the second as non-negative integers
- Assume that the two inputs both fit in n bits
- LessThan circuit

```
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];
}
```

- The default field used in circom can accommodate 253-bit integers; hence the $n \leq 252$ constraint
- out is 1 if and only if $in[0] < in[1]$

Variable Length Subarray Puzzle

- Published by Axiom to select contributors to their open source program
- Puzzle specification

```
// start, end lie in [0, 1000)
// The first values of out are the values at
// indices [start, end) of in.
// The remainder of out is 0-padded
template VarSubarray() {
    signal input in[1000];
    signal input start;
    signal input end;

    // Fill in your solution here

    signal output out[1000];
}
```

An $O(N^2)$ VLS Puzzle Solution

```
template VarSubarray(N) {
    signal input in[N];
    signal input start;
    signal input end;

    signal output out[N];
    component isequal[N];
    component lessthan[N];

    for (var i = 0; i < N; i++) {
        isequal[i] = IsEqual();
        isequal[i].in[0] <== start;
        isequal[i].in[1] <== i;

        lessthan[i] = LessThan(10); // 10 bits for [0, 1000)
        lessthan[i].in[0] <== i;
        lessthan[i].in[1] <== end - start;
    }

    // Continued on next slide
}
```

An $O(N^2)$ VLS Puzzle Solution

```
signal running_sum[N][N];

for (var i = 0; i < N; i++) {
  for (var j = i; j < N; j++)
  {
    if (j == i) {
      running_sum[i][j] <== in[j]*isequal[0].out;
    }
    else {
      running_sum[i][j] <== running_sum[i][j-1] +
                             in[j]*isequal[j-i].out;
    }
  }
  out[i] <== running_sum[i][N-1]*lessthan[i].out;
}

component main { public [ in ] } = VarSubarray(1000);
```

- This solution requires 514,500 R1CS constraints
- We will see a much better solution that requires only 22,020 constraints
 - Published by Ayush Shukla <https://x.com/0xa9a>

An $O(N \log N)$ VLS Puzzle Solution (1/3)

```
template VarShiftLeft(n, nBits) {
  signal input in[n]; signal input shift; signal output out[n];
  signal tmp[nBits][n];

  component n2b = Num2Bits(nBits);
  n2b.in <== shift;

  for (var j = 0; j < nBits; j++) {
    for (var i = 0; i < n; i++) {
      var off = (i + (1 << j)) % n; // offset
      // Shift left by 2^j indices if bit is 1
      if (j == 0) {
        tmp[j][i] <== n2b.out[j]*(in[off] - in[i]) + in[i];
      } else {
        tmp[j][i] <== n2b.out[j]*(tmp[j-1][off] - tmp[j-1][i])
          + tmp[j-1][i];
      }
    }
  }
  // Return last row
  for (var i = 0; i < n; i++) {
    out[i] <== tmp[nBits - 1][i];
  }
}
```

An $\mathcal{O}(N \log N)$ VLS Puzzle Solution (2/3)

```
template VarSubarrayFromZeroIndex(n, nBits) {
    signal input in[n];
    signal input end;

    signal output out[n];

    component lt[n];
    for (var i = 0; i < n; i++) {
        lt[i] = LessThan(nBits);
        lt[i].in[0] <== i;
        lt[i].in[1] <== end;

        out[i] <== lt[i].out * in[i];
    }
}
```

An $O(N \log N)$ VLS Puzzle Solution (3/3)

```
template VarSubarray(n, nBits) {
    signal input in[n];
    signal input start;
    signal input end;

    signal output out[n];

    component lt = LessThan(nBits);
    lt.in[0] <== start;
    lt.in[1] <== end;
    lt.out == 1;

    // Shift left "start" times
    component shiftLeft = VarShiftLeft(n, nBits);
    shiftLeft.in <== in;
    shiftLeft.shift <== start;

    // Take first (end - start) indices
    component subarrayFromZeroIndex = VarSubarrayFromZeroIndex(n,
        nBits);
    subarrayFromZeroIndex.in <== shiftLeft.out;
    subarrayFromZeroIndex.end <== end - start;

    out <== subarrayFromZeroIndex.out;
}
```


References

- **circom** <https://docs.circom.io/>
- **circom repo** <https://github.com/iden3/circom>
- **Pairing gas costs** <https://eips.ethereum.org/EIPS/eip-1108>
- **Groth16 gas costs** <https://hackmd.io/@nebra-one/ByoMB8Zf6>
- **zkrepl** <https://zkrepl.dev/>
- **zkrepl examples**
<https://zkrepl.dev/?gist=8f878e394bcf2bbdc42c06f186a6410a>
- **Tornado Cash circuits** <https://github.com/tornadocash/tornado-core/tree/master/circuits>
- **Dark Forest circuits** <https://github.com/darkforest-eth/circuits>
- **circomlib circuits** <https://github.com/iden3/circomlib>
- **Variable Length Subarray $\mathcal{O}(N \log N)$ Puzzle Solution**
<https://demo.hedgedoc.org/s/Le0R3xUhB>