Reference

Thursday, May 2, 2024 11:08 PM

1. https://www.rareskills.io/post/quadratic-arithmetic-program
2. https://www.rareskills.io/post/circom-tutorial

Notes

Wednesday, June 5, 2024 10:29 PM

QAP and encrypted polynomial evaluation

As you see these matrices are sparse. If we build ZK on R1CS, it won't be "succinct". The succinctness of zk-SNARK is handled by QAP and encrypted polynomial evaluation. Specifically:

- QAP: Lagrange Interpolation
- Encrypted polynomial evaluation: Schwartz-Zippel Lemma

First, we demonstrate how to "squeeze" a column vector into a polynomial. Let's pick the 2nd column of L are do Lagrange Interpolation:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \rightarrow \begin{matrix} (1,1) \\ (2,0) \\ (3,0) \\ (4,0) \\ (5,1) \\ (6,1) \\ (7,0) \end{matrix}$$

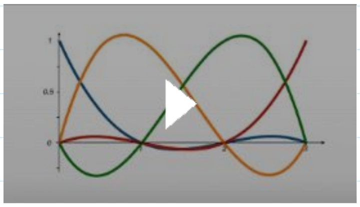
labels

x-coordinate fixed: $\underline{1, 2, 3, 4, 5, 6, 7}$

y-coordinate taken from column vector

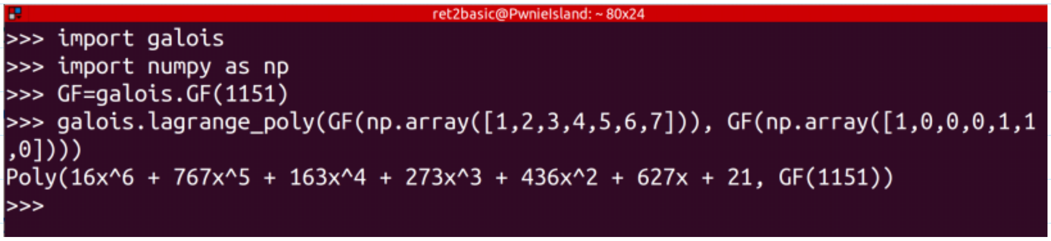
Given n+1 points, Lagrange Interpolation finds a polynomial of degree n that goes through all the points.

Lagrange Interpolation



galois.lagrange_poly() takes two inputs:

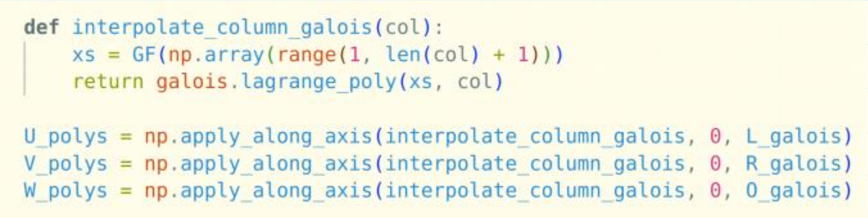
- Input 1: x coordinates as GF array
- Input 2: y coordinates as GF array



ret2basic@PwnieIsland: ~ 80x24

```
>>> import galois
>>> import numpy as np
>>> GF=galois.GF(1151)
>>> galois.lagrange_poly(GF(np.array([1,2,3,4,5,6,7])), GF(np.array([1,0,0,0,1,1
,0])))
Poly(16x^6 + 767x^5 + 163x^4 + 273x^3 + 436x^2 + 627x + 21, GF(1151))
>>>
```

Implementation:



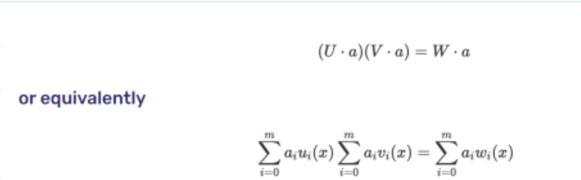
```
def interpolate_column_galois(col):
    xs = GF(np.array(range(1, len(col) + 1)))
    return galois.lagrange_poly(xs, col)

U_polys = np.apply_along_axis(interpolate_column_galois, 0, L_galois)
V_polys = np.apply_along_axis(interpolate_column_galois, 0, R_galois)
W_polys = np.apply_along_axis(interpolate_column_galois, 0, O_galois)
```

https://numpy.org/doc/stable/reference/generated/numpy.apply_along_axis.html

np.apply_along_axis takes 3 inputs:

- Input 1: apply which function
- Input 2: which axis (0 for column and 1 for row)
- Input 3: apply function to which matrix

Building QAP formula



$$(U\cdot a)(V\cdot a) = W\cdot a$$

or equivalently

$$\sum_{i=0}^{m} a_i u_i(x) \sum_{i=0}^{m} a_i v_i(x) = \sum_{i=0}^{m} a_i w_i(x)$$

Recall there R1CS formula was:

$$(L\cdot a) \underset{\uparrow}{\circ} (R\cdot a) = O\cdot a$$

Hadamard product

$\varphi(L) = U$

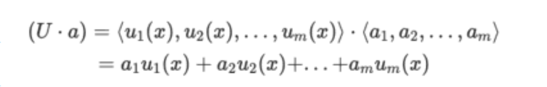
$\varphi(R) = V$

$\varphi(O) = W$

φ homomorphism

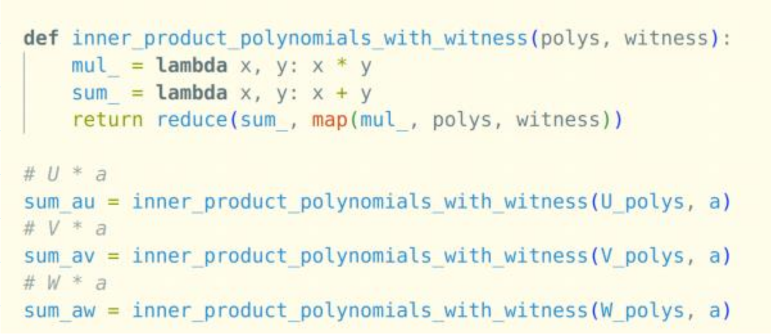
(U * a) stands for inner product:

But this is imbalanced, will explain later



$$(U\cdot a) = \langle u_1(x), u_2(x), \ldots, u_m(x)\rangle \cdot \langle a_1, a_2, \ldots, a_m\rangle$$
$$= a_1u_1(x) + a_2u_2(x) + \ldots + a_mu_m(x)$$

Implementation:



```
def inner_product_polynomials_with_witness(polys, witness):
    mul_ = lambda x, y: x * y
    sum_ = lambda x, y: x + y
    return reduce(sum_, map(mul_, polys, witness))

# U * a
sum_au = inner_product_polynomials_with_witness(U_polys, a)
# V * a
sum_av = inner_product_polynomials_with_witness(V_polys, a)
# W * a
sum_aw = inner_product_polynomials_with_witness(W_polys, a)
```

map:

polys = $[p_1, p_2, \cdots, p_n]$

witness = $[w_1, w_2, \cdots, w_n]$

map_result = $[p_1 * w_1, p_2 * w_2, \cdots, p_n * w_n]$

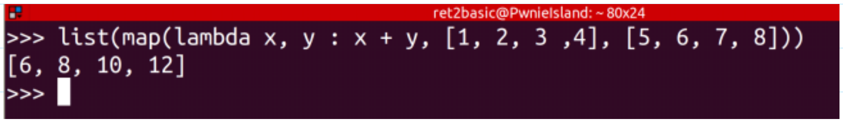
reduce:

reduce_result = $p_1 * w_1 + p_2 * w_2 + \cdots + p_n * w_n$

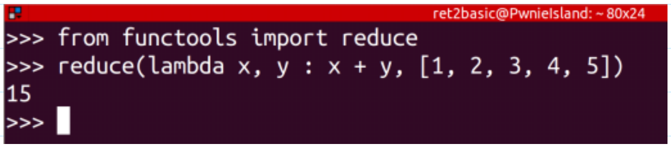
lambda function: inline function with no function name

Map reduce:

- map(): apply a function to each entry of an iterator
- reduce(): "fold" an iterator using a function



```
>>> list(map(lambda x, y : x + y, [1, 2, 3 ,4], [5, 6, 7, 8]))
[6, 8, 10, 12]
>>> 
```



```
>>> from functools import reduce
>>> reduce(lambda x, y : x + y, [1, 2, 3, 4, 5])
15
>>> 
```

Balance out QAP formula

U : array for polys of degree 6

V : array for polys of degree 6

W : array for polys of degree 6

$$(c_1x^6+\cdots)\cdot(c_2x^6+\cdots)$$

$$\overset{\text{degree } 6}{\downarrow} \quad \overset{\text{degree } 6}{\downarrow}$$

$$\Rightarrow \underbrace{(U\cdot a)(V\cdot a)}_{\text{degree } 12} = \underbrace{W\cdot a}_{\text{degree } 6} + \text{some\_poly}$$

$$(c_1c_2x^{12}+\cdots)$$

Why? Because R1CS formula can be viewed in another way:

$$(L\cdot a)\circ(R\cdot a) = O\cdot a + \vec{0} = \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\end{bmatrix}$$

$\nwarrow$ zero vector

$$\varphi(\vec{0}) \neq \vec{0}$$

Let $\varphi(\vec{0})$ be a degree 12 poly to balance QAP equation

$\Rightarrow$ $\text{some\_poly} = h(x)\,\underline{t(x)}$ Lagrange interpolation through $(1,0)$, $(2,0)$, $\vdots$, $(7,0)$

$\uparrow$

$(x-1)(x-2)\cdots(x-7)$: 7 constraints → up to $(x-7)$

Now $t(x)$ contributes degree 7, need $h(x)$ to be degree 5

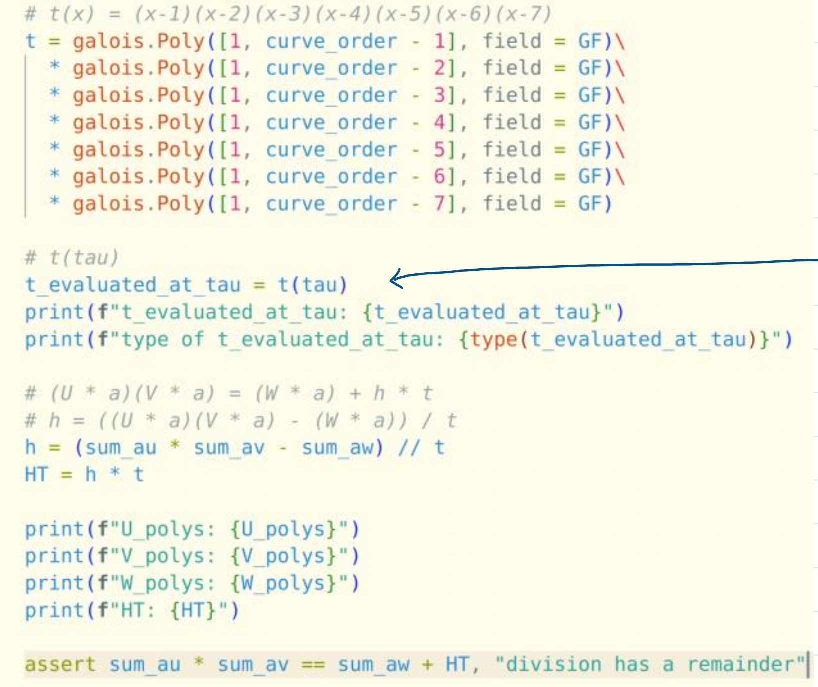
$$(U\cdot a)(V\cdot a) = W\cdot a + h\cdot t$$

$$h\cdot t = (U\cdot a)\cdot(V\cdot a)$$

$$h = \frac{(U\cdot a)\cdot(V\cdot a)}{t}$$

$$h(x) = \frac{(U(x)\cdot a)\cdot(V(x)\cdot a)}{t(x)}$$

Implementation:



```
# t(x) = (x-1)(x-2)(x-3)(x-4)(x-5)(x-6)(x-7)
t = galois.Poly([1, curve_order - 1], field = GF)\
  * galois.Poly([1, curve_order - 2], field = GF)\
  * galois.Poly([1, curve_order - 3], field = GF)\
  * galois.Poly([1, curve_order - 4], field = GF)\
  * galois.Poly([1, curve_order - 5], field = GF)\
  * galois.Poly([1, curve_order - 6], field = GF)\
  * galois.Poly([1, curve_order - 7], field = GF)

# t(tau)
t_evaluated_at_tau = t(tau)
print(f"t_evaluated_at_tau: {t_evaluated_at_tau}")
print(f"type of t_evaluated_at_tau: {type(t_evaluated_at_tau)}")

# (U * a)(V * a) = (W * a) + h * t
# h = ((U * a)(V * a) - (W * a)) / t
h = (sum_au * sum_av - sum_aw) // t
HT = h * t

print(f"U_polys: {U_polys}")
print(f"V_polys: {V_polys}")
print(f"W_polys: {W_polys}")
print(f"HT: {HT}")

assert sum_au * sum_av == sum_aw + HT, "division has a remainder"
```

← will be explained later

Succinctness: encrypted poly at a single point

Now we have QAP equation, but comparing equality of two polynomials is still expensive when there are many constraints. To satisfy the "S" in "SNARK", we only evaluate polynomials at a single point p(tau), where tau is a random value generated by trusted setup.

We claim that comparing equality of two polynomials is (almost) equivalent to evaluating them at a random point and then compare the result. This is supported by Schwartz-Zippel Lemma:

What is...the Schwartz-Zippel lemma?

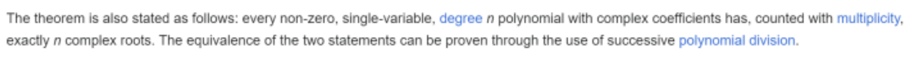
Observation:

Degree d poly(x) over $\mathbb{F}_p$, guess its root r

$$\Pr[\underset{\substack{\uparrow \\ \text{guessed root is correct}}}{\underline{\text{poly}(r) = 0}}] \le \frac{d}{p} \begin{array}{l} \leftarrow \text{\# roots of poly}(x) \\ \leftarrow \text{all possibilities} \end{array}$$

<- From fundamental theorem of algebra. Number of roots "can't do better" than over complex numbers **C**.

https://en.wikipedia.org/wiki/Fundamental_theorem_of_algebra



The theorem is also stated as follows: every non-zero, single-variable, degree *n* polynomial with complex coefficients has, counted with multiplicity, exactly *n* complex roots. The equivalence of the two statements can be proven through the use of successive polynomial division.

When p is huge, the probability of guessing correct root in one shot is close to 0. In other words, poly is zero polynomial with extremely high probability.

An equivalent version:

$$\Pr[\overset{\text{degree } d}{poly_1(r)} - \overset{\text{degree } d}{poly_2(r)} = 0] \le \frac{d}{p}$$

$$\Pr[poly_1(r) = poly_2(r)] \le \frac{d}{p}$$

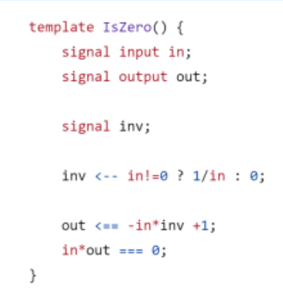
The above is saying, the probability of getting the same result after evaluating two polynomials is close to 0. In other words, poly1 and poly2 are the same polynomial with extremely high probability.

Conclusion: we can evaluate both sides of QAP equation at a random point and compare the result. If the result is the same, we deduce that the polynomials are the same. This is the idea behind "succinctness" in SNARK.

(Random point needs to be generated by trusted setup, will cover that next week)

Circomlib - comparators.circom

https://github.com/iden3/circomlib/blob/master/circuits/comparators.circom

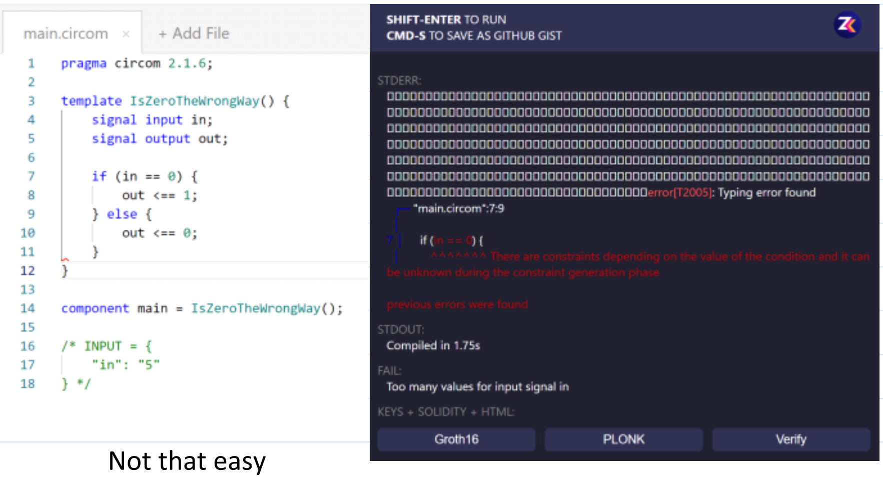


```
template IsZero() {
    signal input in;
    signal output out;

    signal inv;

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

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

Idea: non-zero field element has multiplicative inverse.

<-- assign to signal

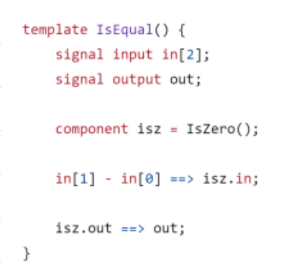
<== assign and add constraint



```
pragma circom 2.1.6;

template IsZeroTheWrongWay() {
    signal input in;
    signal output out;

    if (in == 0) {
        out <== 1;
    } else {
        out <== 0;
    }
}

component main = IsZeroTheWrongWay();

/* INPUT = {
    "in": "5"
} */
```

```
STDERR:
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□error[T2005]: Typing error found
     ┌─ "main.circom":7:9
     │
   7 │     if (in == 0) {
     │         ^^^^^^^ There are constraints depending on the value of the condition and it can
  be unknown during the constraint generation phase

  previous errors were found
STDOUT:
  Compiled in 1.75s
FAIL:
  Too many values for input signal in
KEYS + SOLIDITY + HTML:
```

Not that easy

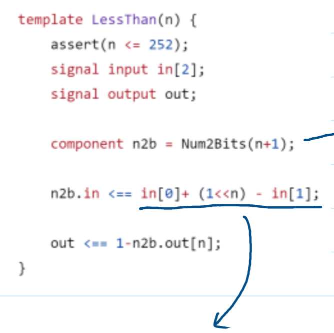


```
template IsEqual() {
    signal input in[2];
    signal output out;

    component isz = IsZero();

    in[1] - in[0] ==> isz.in;

    isz.out ==> out;
}
```

Common pattern: When there are multiple inputs, store them into an array. Usually it is called in[].

component: instantiate another template and "wire" inputs.

Arrow direction can be <== or ==>



```
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];
}
```

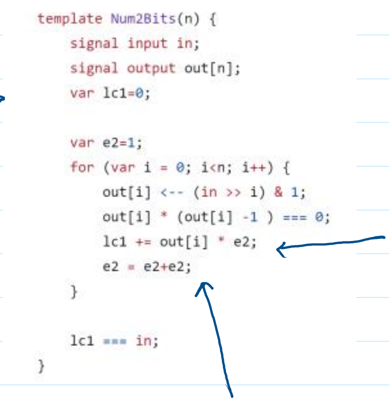
Compare 5 = 0101   n=4

and 7 = 0111

```
  0101
+ 10000  ← 1<< 4
  10101

  10101
-  0111
  01110
```

if MSB is 0, then a<b

if MSB is 1, then a>b



```
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;
}
```

accumulator

e2 = 1, 2, 4, 8, ...

1. 5 = 0 1 0 1   e2=1   lc1=0

& 1

1 → out[0]

lc1 += 1 * 1 → lc1 = 1

e2 = 2

2. 0 0 1 0   e2=2   lc1=1

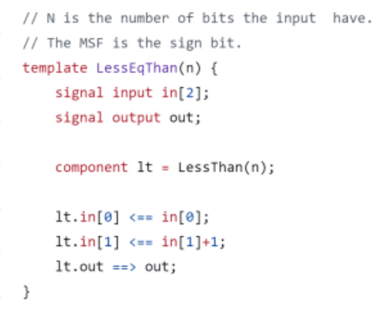
& 1

0 → out[1]

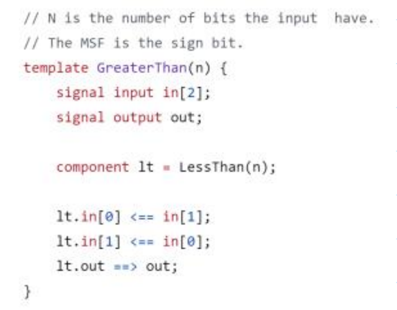
lc1 += 0 * 1 → lc1 = 1

e2 = 4

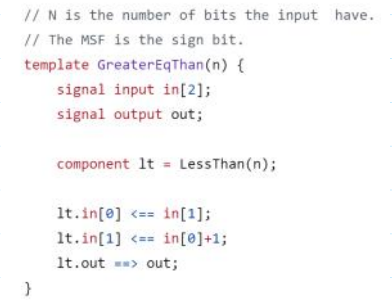
⋮



```
// N is the number of bits the input  have.
// The MSF is the sign bit.
template LessEqThan(n) {
    signal input in[2];
    signal output out;

    component lt = LessThan(n);

    lt.in[0] <== in[0];
    lt.in[1] <== in[1]+1;
    lt.out ==> out;
}
```



```
// N is the number of bits the input  have.
// The MSF is the sign bit.
template GreaterThan(n) {
    signal input in[2];
    signal output out;

    component lt = LessThan(n);

    lt.in[0] <== in[1];
    lt.in[1] <== in[0];
    lt.out ==> out;
}
```



```
// N is the number of bits the input  have.
// The MSF is the sign bit.
template GreaterEqThan(n) {
    signal input in[2];
    signal output out;

    component lt = LessThan(n);

    lt.in[0] <== in[1];
    lt.in[1] <== in[0]+1;
    lt.out ==> out;
}
```