Reference

Wednesday, June 26, 2024 10:12 AM

1. https://alinush.github.io/2022/12/31/pairings-or-bilinear-maps.html

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.
We solve 3 problems in this session:

1. QAP: Lagrange Interpolation
2. Encrypted polynomial evaluation: Schwartz-Zippel Lemma
3. Homomorphism between R1CS and QAP

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

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

x-coordinate fixed: 1, 2, 3, 4, 5, 6, 7 (labels)
y-coordinate taken from column vector

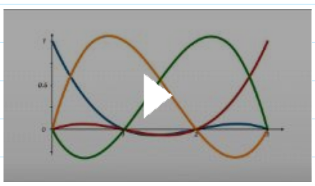
Side note: Factor Theorem
http://abstract.ups.edu/aata/poly-section-division-algorithm.html

Corollary 17.8. *Let F be a field. An element $\alpha \in F$ is a zero of $p(x) \in F[x]$ if and only if $x - \alpha$ is a factor of $p(x)$ in $F[x]$.*

Proof.

Given n+1 points, **Lagrange Interpolation** finds a polynomial of degree n that goes through all the points. Recall that we are working with finite field so poly ring will be F[x].

Lagrange Interpolation



Idea: say we interpolate $(0, k_0)$, $(1, k_1)$, $(2, k_2)$ and $(3, k_3)$

let $f_0(0) = k_0$, $f_0(1) = f_0(2) = f_0(3) = 0 \Rightarrow f_0(x) = (x-1)(x-2)(x-3) \cdot \frac{k_0}{-6}$

$f_1(1) = k_1$, $f_1(0) = f_1(2) = f_1(3) = 0 \Rightarrow f_1(x) = x(x-2)(x-3) \cdot \frac{k_1}{2}$

$f_2(2) = k_2$, $f_2(0) = f_2(1) = f_2(3) = 0 \Rightarrow f_2(x) = x(x-1)(x-3) \cdot \frac{k_2}{-2}$

$f_3(3) = k_3$, $f_3(0) = f_3(1) = f_3(2) = 0 \Rightarrow f_3(x) = x(x-1)(x-2) \cdot \frac{k_3}{6}$

Result:

$f(x) = f_0(x) + f_1(x) + f_2(x) + f_3(x)$

$x=0: f(0) = k_0 + 0 + 0 + 0 = k_0$

$x=1: f(1) = 0 + k_1 + 0 + 0 = k_1$

$x=2: f(2) = 0 + 0 + k_2 + 0 = k_2$

$x=3: f(3) = 0 + 0 + 0 + k_3 = k_3$

Lagrange Interpolation finds the **lowest degree** poly f(x) that interpolates all the given points. For the sake of contradiction, suppose that there exists a poly f'(x) of degree <= n that interpolates same set of points. Since f(x) and f'(x) are equal at the given set of points, f(x) - f'(x) = 0 at those points. That means poly (f-f')(x) has n+1 zeroes. However, (f-f')(x) has degree <= n, so it has at most n zeroes OR it is the zero polynomial. The only possibility is that (f-f')(x) is the zero polynomial, thus f(x) = f'(x), a contradiction, since f(x) has non-zero degree-n term.

http://abstract.ups.edu/aata/poly-section-division-algorithm.html

Corollary 17.9. *Let F be a field. A nonzero polynomial $p(x)$ of degree n in $F[x]$ can have at most n distinct zeros in F.*

Proof.

Note that the deg(f'(x)) = n case also proves the **uniqueness** of f(x).

galois.lagrange_poly() takes two inputs:

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

Implementation:

```
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))
>>>
```

```python
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 that R1CS formula was:

$$(L \cdot a) \circ (R \cdot a) = O \cdot a$$

↑ Hadamard product

$f(L) = U$
$f(R) = V$
$f(O) = W$

f homomorphism

↑ will explain later as well

But this is imbalanced, will explain later

(U * a) stands for inner product:

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

Implementation:

```python
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

```
ret2basic@PwnIsland: ~ 80x24
>>> list(map(lambda x, y : x + y, [1, 2, 3 ,4], [5, 6, 7, 8]))
[6, 8, 10, 12]
>>>
```

```
ret2basic@PwnIsland: ~ 80x24
>>> 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

$(c_1 x^6 + \cdots) \cdot (c_2 x^6 + \cdots)$

degree 6 degree 6

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

$(c_1 c_2 x^{12} + \cdots)$

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

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

zero vector

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

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

$\Rightarrow$ some_poly $= h(x)\, t(x)$ Lagrange interpolation through $(1,0)$, $(2,0)$, $\vdots$, $(7,0)$

$t(x) = (x-1)(x-2)\cdots(x-7)$: 7 constraints $\rightarrow$ up to $(x-7)$

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

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

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

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

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

Side note: this step is slow, can be optimized

Implementation:

```python
# 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

Idea behind QAP:

1. Operations in R1CS (addition and Hadamard product) form a ring when viewed as a set of column vectors (why this is the case will be explained later)
2. Polynomials under addition and multiplication are rings
3. There exists an easily computable homomorphism from R1CS to polynomials

https://www.rareskills.io/post/quadratic-arithmetic-program

Theorem: there exists a Ring Homomorphism from column vectors of dimension n with real number elements to polynomials with real coefficients.
Proof: I'm going to trigger the mathematicians by not putting one here. Let's move on.

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

Ring homomorphism

17 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

Algebraic structure → Ring theory
Ring theory
Basic concepts [show]
Commutative algebra [show]
Noncommutative algebra [show]
V · T · E

In mathematics, a **ring homomorphism** is a structure-preserving function between two rings. More explicitly, if R and S are rings, then a ring homomorphism is a function $f : R \to S$ that preserves addition, multiplication and multiplicative identity; that is,[1][2][3][4][5]

$$f(a+b) = f(a) + f(b),$$
$$f(ab) = f(a)f(b),$$
$$f(1_R) = 1_S,$$

for all a, b in R.

These conditions imply that additive inverses and the additive identity are preserved too.

If in addition f is a bijection, then its inverse f^{-1} is also a ring homomorphism. In this case, f is called a **ring isomorphism**, and the rings R and S are called *isomorphic*. From the standpoint of ring theory, isomorphic rings have exactly the same properties.

If R and S are rngs, then the corresponding notion is that of a **rng homomorphism**,[a] defined as above except without the third condition $f(1_R) = 1_S$. A rng homomorphism between (unital) rings need not be a ring homomorphism.

The composition of two ring homomorphisms is a ring homomorphism. It follows that the rings forms a category with ring homomorphisms as morphisms (see Category of rings). In particular, one obtains the notions of ring endomorphism, ring isomorphism, and ring automorphism.

domain

Let R be the ring of column vectors with entries in $\mathbb{R}$ of dimension n

S be the ring of $\mathbb{R}[x] \to$ polys with real coefficients

codomain

Write $r \in R$ as $(r_0, r_1, \dots, r_n)$

$s \in S$ as $s_0 + s_1 x + s_2 x^2 + \cdots + s_{n-1} x^{n-1}$ $\Rightarrow$ f is lagrange interpolation

① $f(r+r') = f((r_0+r_0', r_1+r_1', \cdots, r_n+r_n'))$

$= \text{Lagrange\_interpolate}((0, r_0+r_0'), (1, r_1+r_1'), \cdots, (n, r_n+r_n'))$

$= \text{Lagrange\_interpolate}((0, r_0), (1, r_1), \cdots, (n, r_n)) + \text{Lagrange\_interpolate}((0, r_0'), (1, r_1'), \cdots (n, r_n'))$

$= f(r) + f(r') \in R$

```
ret2basic@PwnieIsland: ~ 79x22
>>> import galois
>>> import numpy as np
>>> GF = galois.GF(1151)
>>> galois.lagrange_poly(GF(np.array([1,2,3,4])), GF(np.array([1,5,3,2])))
Poly(193x^3 + 1141x^2 + 985x + 1135, GF(1151))
>>> galois.lagrange_poly(GF(np.array([1,2,3,4])), GF(np.array([2,1,6,8])))
Poly(574x^3 + 12x^2 + 549x + 18, GF(1151))
>>> galois.lagrange_poly(GF(np.array([1,2,3,4])), GF(np.array([3,6,9,10])))
Poly(767x^3 + 2x^2 + 383x + 2, GF(1151))
>>>
```

② $f(r \cdot r') = f((r_0 \cdot r_0', r_1 \cdot r_1', \cdots r_n \cdot r_n'))$

$= \text{Lagrange\_interpolate}((0, r_0 \cdot r_0'), (1, r_1 \cdot r_1'), \cdots, (n, r_n \cdot r_n'))$

$\neq \text{Lagrange\_interpolate}((0, r_0), (1, r_1), \cdots (n, r_n))$

$\cdot \text{Lagrange\_interpolate}((0, r_0'), (1, r_1'), \cdots, (n, r_n'))$

```
>>> a = galois.lagrange_poly(GF(np.array([1,2,3,4])), GF(np.array([1,5,3,2])))
>>> b = galois.lagrange_poly(GF(np.array([1,2,3,4])), GF(np.array([2,1,6,8])))
>>> c = galois.lagrange_poly(GF(np.array([1,2,3,4])), GF(np.array([2,5,18,16]))
)
>>> a * b
Poly(286x^6 + 29x^5 + 194x^4 + 620x^3 + 574x^2 + 889x + 863, GF(1151))
>>> c
Poly(955x^3 + 30x^2 + 134x + 34, GF(1151))
>>>
```

③ $f(1_R) = f((1, 1, \cdots, 1))$

$= \text{Lagrange\_interpolate}((0,1), (1,1), \cdots, (n,1))$

$= 1_S$

```
ret2basic@PwnieIsland: ~ 79x22
>>> galois.lagrange_poly(GF(np.array([1,2,3,4])), GF(np.array([1,1,1,1])))
Poly(1, GF(1151))
>>>
```

Succinctness: evaluate 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:

Observation:

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

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

$\longrightarrow$ This result was discussed in Lagrange Interpolation section.

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:

$$\Pr[\overset{\text{degree } d}{\text{poly}_1(r)} - \overset{\text{degree } d}{\text{poly}_2(r)} = 0] \le \frac{d}{p}$$

$$\Pr[\text{poly}_1(r) = \text{poly}_2(r)] \le \frac{d}{p}$$

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 in the next session)

Pairing (as black box)

G1 point = (x-coordinate, y-coordinate) -> 2 coordinates
G2 point = 4 coordinates, since the underlying curve involves complex number
Pairing = "multiplication" between G1 and G2 points

Math behind pairing will be covered in session 4. At this moment we use pairing as a black box.

```
ret2basic@PwnieIsland: ~ 80x24
Python 3.10.12 (main, Mar 22 2024, 16:50:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from py_ecc.bn128 import G1, G2, pairing
>>> G1
(1, 2)
>>> G2
((10857046999023057135944570762232829481370756359578518086990519993285655852781,
 11559732032986387107991004021392285783925812861821192530917403151452391805634),
 (8495653923123431417604973247489272438418190587263600148770280649306958101930,
4082367875863433681332203403145435568316851327593401208105741076214120093531))
>>> pairing(G2, G1)
(18443897754565973717256850119554731228214108935025491924036055734000366132575,
10734401203193558706037776473742910696504851986739882094082017010340198538454, 5
985796159921227033560968606339653189163760772067273492369082490994528765680, 409
3294155816392700623820137842432921872230622290337094591654151434545306688, 64212
13701608332327661814934949550440743213855288837916688684268790701034340, 45274498
4994760135703704417895294248992648707165389643560281487233409862539 1, 3758435817
7662881888045612538386700307629707643666725947842474470678680880 68, 180591685461
481526718570263727117243793197783067920111467846650809870641646 12, 1465660657393
650174345763304104802465661222730147308480562739074887261728098 4, 17918828665069
491344039743589118342552553375221610735811112289083834142789347, 194554243435768
8643088984977336739794645744907352845509721094683900014769837 2, 7484542354754424
633621663080190936924481536615300815203692506276894207018007)
>>>
```

ret2basic@PwnieIsland: ~ 80x24

```
>>> from py_ecc.bn128 import G1, G2, pairing, add, multiply
>>> multiply(G1, 2)
(1368015179489954701390400359078579693043519447331113978918064868415326638035, 9
918110051302171585080402603319702774565515993150576347155970296011118125764)
>>> add(G1, G1) == multiply(G1, 2)
True
>>>
```

Elliptic curve (over finite field) addition is "partial homomorphic encryption" under addition:

$$3G + 4G = (G+G+G) + (G+G+G+G) = 7G$$

also : $3+4=7 \Rightarrow 7G$

But it is not "partial homomorphic encryption" under multiplication:

$$3G \cdot 4G = ?$$

Definitely not $12G$

Pairing (denoted as e()) acts as "partial homomorphic encryption" under multiplication:

e(aG1, bG2) = e(cG1, G2) iff a * b = c

e.g. : $e(3G_1, 4G_2) = e(12G_1, G_2)$ since $3 \cdot 4 = 12$

ret2basic@PwnieIsland: ~ 79x22

```
>>> from py_ecc.bn128 import G1, G2, add, multiply, pairing
>>> term1 = multiply(G2, 4)
>>> term2 = multiply(G1, 3)
>>> term3 = G2
>>> term4 = multiply(G1, 12)
>>> pairing(term1, term2) == pairing(term3, term4)
True
>>>
```

In Groth16, verifier receives a proof containing some G1 and G2 points on bn128 curve, and the only thing he does is computing this pairing.

It is ok to understand pairing this way if you don't care about the math behind it.