Reference

Thursday, May 2, 2024 11:08 PM

1. https://www.rareskills.io/post/encrypted-polynomial-evaluation
2. https://www.rareskills.io/post/elliptic-curve-qap
3. https://www.rareskills.io/post/groth16

Trusted Setup (powers of tau)

Recall:
-> Problem statement
-> R1CS (L, R, O matrices)
-> Lagrange Interpolation (U, V, W array of polys)
-> Evaluate QAP at a random point (guaranteed by Schwartz-Zippel)

Need trusted setup
But why ?

Suppose random point tau is publicly known, QAP formula becomes:

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

Prover can come up with fake numbers that still satisfy the equality, therefore forge fake proof and cheat verifier.

Q: How to generate a fake proof if tau is leaked

A: If prover knows tau, he only needs to do Lagrange interpolation for that specific point instead of all points. That will result in different polys but the forged proof will still go through verifier step successfully.

Trusted setup computes and publishes "encryptions" of tau, called powers of tau:

$$(\tau^0 G_1, \tau^1 G_1, \tau^2 G_1, \tau^3 G_1, \cdots, \tau^6 G_1)$$
$$(\tau^0 G_2, \tau^1 G_2, \tau^2 G_2, \tau^3 G_2, \cdots, \tau^6 G_2)$$

Prover computes evaluation of polys in QAP according to powers of tau. For example, U * a can be computed as:

$[\,]_1$ means G_1 point
(either G_1 or G_2)
$[\,]_1$ $[\,]_2$

$$[A]_1 = [\underbrace{(U\cdot a)}_{\text{inner product, still poly}}(\tau)]_1 = [(c_n x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0)(\tau)]_1$$

$$= [c_n\tau^n + c_{n-1}\tau^{n-1} + \cdots + c_1\tau + c_0]_1$$

EC points addition $= [c_n\tau^n]_1 + [c_{n-1}\tau^{n-1}]_1 + \cdots + [c_1\tau]_1 + [c_0]_1$

EC point scalar multiplication $= c_n(\tau^n G_1) + c_{n-1}(\tau^{n-1}G_1) + \cdots + c_1(\tau G_1) + c_0(G_1)$

Come from trusted setup

$= (\cdots, \cdots)$

Compute $[B]_2 = [(V\cdot a)(\tau)]_2 = ((\cdots,\cdots),(\cdots,\cdots))$

$[C']_1 = [(W\cdot a)(\tau)]_1 = (\cdots,\cdots)$

$[HT]_1 = [(h(x)t(x))(\tau)]_1 = (\cdots,\cdots)$

not that easy, will discuss later

$[C]_1 = [C']_1 + [HT]_1 = (\cdots,\cdots)$

Verifier verifies if:

pretty bad at this moment

Verifier verifies if:

pretty bad at this moment

$$\text{pairing}([A]_1, [B]_2) = \text{pairing}([C]_1, G_2)$$

G_1 point G_2 point G_1 point

Issue: evaluating h(x)t(x)

When we evaluate h(tau)t(tau), we are "multiplying" two G1 points, which will introduce a pairing -> unwanted

(But why not compute h(x) * t(x) first and evaluate ht(tau)?
I think h(x) * t(x) computation is unwanted too because it is expensive)

Solution: embed t(tau) in another powers of tau

$$(\tau^0 t(\tau)G_1, \tau^1 t(\tau) G_1, \tau^2 t(\tau) G_1, \cdots, \tau^5 t(\tau) G_1)$$ → 3rd powers of tau generated by trusted setup

Then the computation of h(tau)t(tau) by prover becomes:

say $t(x) = x-1$ $\tau = 5$

$h(x) = 2x+1$

$h(\tau)t(\tau) = 11 \cdot 4 = 44$

$(h(x) \cdot t(\tau))(\tau) = (2x+1) \cdot 4$

$= 8x+4$

$= 40+4 = 44$

$$[h(\tau)t(\tau)]_1 = [ht(\tau)]_1$$
$$= [(h(x) \cdot t(\tau))(\tau)]_1$$
$$= [((h_0 + h_1 x + h_2 x^2 + \cdots + h_n x^n) \cdot t(\tau))(\tau)]_1$$
$$= [h_0 t(\tau) + h_1 t(\tau) x + h_2 t(\tau) x^2 + \cdots + h_n t(\tau) x^n)(\tau)]_1$$
$$= [h_0 t(\tau) + h_1 \tau t(\tau) + h_2 \tau^2 t(\tau) + \cdots + h_n \tau^n t(\tau)]_1$$

EC addition $$= [h_0 t(\tau)]_1 + [h_1 \tau t(\tau)]_1 + [h_2 \tau^2 t(\tau)]_1 + \cdots + [h_n \tau^n t(\tau)]_1$$

EC scalar multiplication $$= h_0(\underline{t(\tau) G_1}) + h_1(\underline{\tau t(\tau) G_1}) + h_2(\underline{\tau^2 t(\tau) G_1}) + \cdots + h_n(\underline{\tau^n t(\tau) G_1})$$

Come from trusted setup

Q: What if I evaluate h(tau)t(tau) directly without doing the extra powers of tau

A: Prover does not know tau, so he can't compute h(tau)t(tau) on his own without the help from trusted setup. h(tau) needs an encrypted evaluation, and t(tau) needs another one. That will result in a "multiplication" of two G1 points, which is undesired.

Implementation of powers of tau

```
# polynomial degree is 6

# Powers of tau for A
def generate_powers_of_tau_G1(tau):
    return [multiply(G1, int(tau ** i)) for i in range(t.degree)] # up to tau**6
```

```
# Powers of tau for B
def generate_powers_of_tau_G2(tau):
    return [multiply(G2, int(tau ** i)) for i in range(t.degree)] # up to tau**6
```

```
# Powers of tau for h(tau)t(tau)
def generate_powers_of_tau_HT(tau):
    before_delta_inverse = [multiply(G1, int(tau ** i * t_evaluated_at_tau)) for i in range(t.degree - 1)] # up to tau**5
    return [multiply(entry, int(delta_inverse)) for entry in before_delta_inverse]
```

Python list comprehension
https://realpython.com/list-comprehension-python/

Syntax:
new_list = [expression for member in iterable if conditional]

Optional

Example:

```
ret2basic@PwnieIsland: ~ 80x24
ret2basic@PwnieIsland:~$ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> new_list = [10*x for x in range(5) if x % 2 == 1]
>>> print(new_list)
[10, 30]
>>> new_list2 = [10*x for x in range(5)]
>>> print(new_list2)
[0, 10, 20, 30, 40]
>>>
```

Implementation for encrypted poly evaluation:

```python
def inner_product(ec_points, coeffs):
    return reduce(add, (multiply(point, int(coeff)) for point, coeff in zip(ec_points, coeffs)), Z1)

def encrypted_evaluation_G1(poly):
    powers_of_tau = generate_powers_of_tau_G1(tau)
    evaluate_on_ec = inner_product(powers_of_tau, poly.coeffs[::-1])

    return evaluate_on_ec

def encrypted_evaluation_G2(poly):
    powers_of_tau = generate_powers_of_tau_G2(tau)
    evaluate_on_ec = inner_product(powers_of_tau, poly.coeffs[::-1])

    return evaluate_on_ec

def encrypted_evaluation_HT(poly):
    powers_of_tau = generate_powers_of_tau_HT(tau)
    evaluate_on_ec = inner_product(powers_of_tau, poly.coeffs[::-1])

    return evaluate_on_ec
```

Python generator comprehension

https://www.pythonlikeyoumeanit.com/Module2_EssentialsOfPython/Generators_and_Comprehensions.html

Introducing Generators

Now we introduce an important type of object called a **generator**, which allows us to generate arbitrarily-many items in a series, without having to store them all in memory at once.

Definition:

A **generator** is a special kind of iterator, which stores the instructions for how to *generate* each of its members, in order, along with its current state of iterations. It generates each member, one at a time, only as it is requested via iteration.

Recall that a list readily stores all of its members; you can access any of its contents via indexing. A generator, on the other hand, *does not store any items*. Instead, it stores the instructions for generating each of its members, and stores its iteration state; this means that the generator will know if it has generated its second member, and will thus generate its third member the next time it is iterated on.

The whole point of this is that you can use a generator to produce a long sequence of items, without having to store them all in memory.

Example:

Q: Why the 3rd powers of tau has 1 less term?

A: Because the length equals to deg(h(x)). Do the math: x + x = unknown + (x+1) => unknown = x - 1.
My derivation:

ret2basic.eth 06/20/2024 10:36 PM
I think I kind of know why
In the encryption evaluation of (h(x) * t(tau))(tau), you are doing degree(h(x)) + 1 operations (starting from index 0) (edited)
So the question boils down to finding degree of h(x)
On LHS of QAP, you are multiplying two degree 6 polys (in our case), so the result is a degree 12 poly (at most)
On RHS of QAP, the W * a term should have degree 6 too, which can be omitted since it is added into h(x)t(x), which has a much higher degree
h(x)t(x) is supposed to have degree 12 in order to match LHS. Here t(x) is publicly known, it is t(x) = (x-1)(x-2)...(x-7), since there are 7 constraints. That means degree of h(x) is 5
5 = 6 - 1, so always 1 term less

ret2basic.eth 06/20/2024 10:47 PM
If you do this degree derivation in algebra, it will be x + x = unknown + (x+1) => unknown = x - 1
Recall that multiplication of polys means addition in their degrees

ret2basic@PwnieIsland: ~/Desktop/Groth16 80x24

```
>>> even_gen = (i for i in range(100) if i%2 == 0)
>>> print(even_gen)
<generator object <genexpr> at 0x7c16a5fddc40>
>>> next(even_gen)
0
>>> next(even_gen)
2
>>> next(even_gen)
4
>>> next(even_gen)
6
>>>
```

Python zip() -> return the cartesian product of two iterators

Example:

ret2basic@PwnieIsland: ~/Desktop/Groth16 80x24

```
>>> list1 = ["a", "b", "c"]
>>> list2 = [123, 456, 789]
>>> zip(list1, list2)
<zip object at 0x7c16a5530400>
>>> print(list(_))
[('a', 123), ('b', 456), ('c', 789)]
>>>
```

py_ecc.bn128.Z1 -> point at infinity over FQ