ZK notes

Some keynotes made while learning about ZK, SNARK, STARK and ZK-VM.

Lectures: Youtube Playlist

1 SNARK

SNARK is not necessary ZK and can be quite different from STARK and ZK-VM. But it should be a good starting ground to understand some basic concept.

1.1 Polynomial Commitment Scheme (PCS)

A PCS is a functional commitment for the family $\mathcal{F} \in \mathbb{F}_p^{(\leq d)}[X]$. A prover commits to univariate polynomial f in $\mathbb{F}_p^{(\leq d)}[X]$ and can later prove to the verifier that $v = f(u)$ for public $u, v \in \mathbb{F}_p$.

Some examples PCS (here we focus on KZG'10):

1. Bulletproof (elliptic curves, but verification is $O(d)$)
2. KZG'10 (trusted setup, bilinear), Dory'20 (bilinear)
3. Dark'20 (groups of unknown order)
4. Hash (FRI)

1.2 KZG'10

Set cyclic group $\mathbb{G} = \{0, G, 2 \cdot G, 3 \cdot G, \ldots, (p-1) \cdot G\}$ of order p.

Setup algorithm

1. Sample random $\alpha \in \mathbb{F}_p$.
2. $pp = (H_0 = G, H_1 = \alpha \cdot G, \ldots, H_d = \alpha^d \cdot G) \in \mathbb{G}^{d+1}$.
3. **delete** α (i.e., A trusted setup)

Commitment

In short: $commit(pp, f) \to com_f$, where $com_f = f(\alpha) \cdot G \in \mathbb{G}$.[1]

Remark. *As a result, the committed message is extremely short (an element G) regardless how large our polynomial is.*

But α is deleted during trusted setup, how does prover compute $f(\alpha)$? Observe:

[1] This is not a hiding commitment

$\Rightarrow f(X) = f_0 + f_1 X + \ldots + f_d X^d$

$\Rightarrow f(\alpha) \cdot G = f_0 \cdot G + f_1 \cdot \alpha \cdot G + \ldots + f_d \cdot \alpha^d \cdot G$

$\Rightarrow f(\alpha) \cdot G = f_0 \cdot H_0 + f_1 \cdot H_1 + \ldots + f_d \cdot H_d$

Evaluation

How to prove $f(u) = v$?

First Observe:

1. If $f(u) = v \iff u$ is a root of polynomial $\hat{f}(X) = f(X) - v$.
2. If u is a root of $\hat{f}(X) \iff \hat{f}(X)$ is divisible by $(x - u)$.
3. $f(u) = v \iff \exists q \in \mathbb{F}_p^{(\leq d)}[X]$ s.t. $q(X)(X - u) = f(X) - v$

The prover then computes quotient polynomial $q(X) = (f(X) - v)/(X - u)$ and sends com_q to verifier.

The verifier accepts if $(\alpha - u) \cdot com_q = com_f - v \cdot G$.

LHS:

$\Rightarrow (\alpha - u) \cdot com_q$

$\Rightarrow (\alpha - u) \cdot (q_0 H_0 + q_1 H_1 + \ldots + q_d H_d)$

$\Rightarrow (\alpha - u) \cdot (q_0 G + q_1 \alpha G + \ldots + q_d \alpha^d G)$

$\Rightarrow commit(pp, (X - u)q(X))$

RHS is similar. [2]

Remark. *The verification work only take constant time, regardless of the degree of the polynomial.*

Extension

1. KZG for k-variant polynomial (PST'13)
2. Batch proofs: prove a batch of commitments in a single step.

[2] Important: verifier does not actually need to know about α. The *pairing* is used here to allow verifier to compute $(\alpha - u) * com_q$ with only G and H_1

1.3 A Useful Observation

A Useful and important observation.

For $0 \neq f \in \mathbb{F}_p^{(\leq d)}[X]$. Let r be a random point $r \leftarrow \mathbb{F}_p$, the probability $pr[f(r) = 0] = d/p$.[3]

For large enough p and reasonable d, e.g., $p \approx 2^{256}$ and $d \leq 2^{40}$, d/p is negligible.

Lemma 1. *for $r \leftarrow \mathbb{F}_p$, if $f(r) = 0$, we can conclude f is identically zero w.h.p.*[4]

Further more, with the same settings.

Lemma 2. *Let $f, g \in \mathbb{F}_p^{(\leq d)}[X]$. For $r \leftarrow \mathbb{F}_p$, if $f(r) = g(r)$ then $f = g$.*

$\Rightarrow f(r) - g(r) = 0$

$\Rightarrow$ Let $h = f - g$, from Lemma 1, h is identical zero w.h.p.

$\Rightarrow f = g$, w.h.p

1.4 Zero Test On H

One of the (and the simplest) poly-IOP tasks that the verifier would like the prover to do.

Let $\omega \in \mathbb{F}_p$ be a primitive k-th root of unity (such that $\omega^k = 1$ and $\omega^n \neq 1$ for $n < k$).

Set $H = \{1, \omega, \omega^2, \ldots, \omega^{k-1}\} \in \mathbb{F}_p$.

Let polynomial $f \in \mathbb{F}_p^{(\leq d)}[X]$.

A zero test is a test from verifier to prover to prove that: f **is identically zero on set** H**.**

Lemma 3. *f is zero on H iff $f(X)$ is divisible by $X^k - 1$.*

1. The prover can compute the quotient polynomial $q(X) = f(X)/(X^k - 1)$. and send the commitment of q to the verifier.
2. The verifier then choose random $r \in \mathbb{F}_p$ and ask prover to open $f(X)$ and $q(X)$ at r.
3. The verifier then accepts the test if $f(r) = q(r) \cdot (r^k - 1)$

As mentioned in Lemma 2, two polynomials that agree on a random point r has a high probability that the two polynomials are identical. Therefore, the above implies $f(X) = q(X)(X^k - 1)$. This proves $f(X)$ is indeed divisible by $X^k - 1$, hence from Lemma 3, f is identical on H.

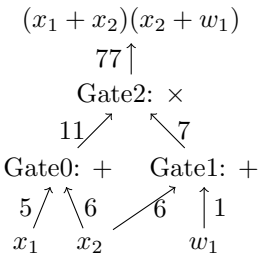
[3] Given f has at most d roots and p elements

[4] Also holds for multivariate polynomial, see SZDL lemma.

1.5 Interpolate Polynomial

Plonk.

1.5.1 Compile a circuit into a computation trace



inputs:	5	6	1
	left	right	out
Gate0	5	6	11
Gate1	6	1	7
Gate2	11	7	77

1.5.2 Encoding the trace as polynomial

$C \leftarrow$: total # of gates

$I \leftarrow |I_x| + |I_w|$: # inputs to circuit

$d \leftarrow 3|C| + |I| = 12$ for our example. (3 since each gate has 3 inputs).

$H \leftarrow \{1, \omega, \ldots, \omega^{(d-1)}\}$

The goal here is to interpolate a polynomial P that encodes the computation trace. To achieve that, we want to

1. let P encodes all inputs, such that $P(\omega^{-j}) =$ inputs # j for all $j = 1, \ldots, |I|$.
2. let P encodes all wires, such that $\forall l = 0, \ldots, |C| - 1$:
   (a) $P(\omega^{3l}) =$ left input of gate # l.
   (b) $P(\omega^{3l+1}) =$ right input of gate # l.
   (c) $P(\omega^{3l+2}) =$ output of gate # l.

This results in 12 constraints for P, which means there exists a P with degree at most 11 that satisfies all the constraints. Prover can then constructs P using Fast Fourier Transform in time $O(d \log d)$, which I don't know how yet.

1.5.3 Prove that encoding is correct

There are four things to prove.

Inputs are correctly encoded.
Both prover and verifier takes input x and interpolate a polynomial $v(X) \in \mathbb{F}_p^{(\leq d)}[X]$ that satisfies $\forall j = 1, \ldots, |I_x| : v(\omega^{-j}) =$input #j.

From the slides, it says construction takes time linear to the size of x, shouldn't it still be using FFT and the time is actually $O(n \log n)$?.

Then prover just proves that $P(y) - v(y) = 0 \quad \forall y \in H_{inp}$ where H_{inp} is all the input points, i.e., $\{\omega^{-1}, \ldots, \omega^{-|I_x|}\}$. This can be done using zero-test.

Gates evaluations are correctly encoded.
Interpolate selector polynomial $S(X) \in \mathbb{F}_p^{(\leq d)}[X]$ such that $\forall l = 0, \ldots, |C| - 1$:

1. $S(\omega^{3l}) = 1$ if gate l is addition
2. $S(\omega^{3l}) = 0$ if gate l is multiplication

Observe $\forall y \in H_{gates} = \{1, \omega^3, \omega^6, \ldots, \omega^{3(|C|-1)}\}$:

$$S(y) \cdot [P(y) + P(\omega y)] + (1 - S(y)) \cdot P(y) \cdot P(\omega y) = P(\omega^2 y)$$

When $S(y) = 1$, which means a gate is addition gate, and $[P(y)+P(\omega y)]$ encodes the two inputs of that gate, which equals to $P(\omega^2 y)$ (where $\omega^2 y$ encodes the output of the circuit). At the same time, since the gate is addition, the right operand $((1 - S(y)) \cdot \ldots)$ must evaluated to zero. The same goes for when $S(y) = 0$, i.e., multiplication gate.

Overall, another zero-test on H_{gates}.

Wirings are encoded correctly.
For example, the input 6 flows to right input of Gate0 and left input of Gate1, we need to prove that does data flows (wiring) are encoded correctly. For our examples, the equivalent constraints are:

$$\begin{cases} P(\omega^{-2}) = P(\omega^1) = P(\omega^3) \\ P(\omega^{-1}) = P(\omega^0) \\ P(\omega^2) = P(\omega^6) \\ P(\omega^3) = P(\omega^4) \end{cases}$$

To do so, define a rotation polynomial $W : H \to H$ such that:

$$\begin{cases} W(\omega^{-2}, \omega^1, \omega^3) = (\omega^3, \omega^{-2}, \omega^1) \\ W(\omega^{-1}, \omega^0) = (\omega^0, \omega^1) \\ \ldots \end{cases}$$

Lemma 4. $\forall y \in H : P(y) = P(W(y)) \Rightarrow$ *wiring constraints are satisfied.*

Since W has degree of d and P has degree of d, the verification can takes quadratic time. The trick here is to use prod-check (another IOP check) to reduce it to linear complexity. Not sure how to yet, another time. :P

Outputs are encoded correctly (is zero).
Just let prover to open P at the output of the final gate.

2 STARK

This follows the tutorial at here

2.1 Extended Euclidean Algorithm

Refresh myself with the Extended Euclidean algorithm...

Extended Euclidean algorithm is an extension of the Euclidean algorithm, in addition to computing the greatest common divisor of two integer a and b, it also gives x and y such that:

$$ax + by = gcd(a, b)$$

Recall the standard Euclidean algorithm in recursive form: $gcd(a, b) = gcd(b, a \mod b)$, stops at $gcd(r, 0)$ and returns r.

The Extended Euclidean Algorithm works the same, but keeps the quotient at each iterations.

$$\begin{aligned} r_0 &= a \quad s_0 = 1 & t_0 &= 0 \\ r_1 &= b \quad s_1 = 0 & t_1 &= 1 \\ &\cdots \\ r_i &= r_{i-2} - q_{i-1} r_{i-1} \\ s_i &= s_{i-2} - q_{i-1} s_{i-1} \\ t_i &= t_{i-2} - q_{i-1} t_{i-1} \end{aligned}$$

The EEA is useful as it defines the inverse of an element under $\mathbb{F}_p$. Give an element $x \in \mathbb{F}_p$, the inverse of x is therefore a:

$\Rightarrow gcd(x, p) = 1$

$\Rightarrow ax + bp \equiv 1 \mod p$

$\Rightarrow ax \equiv 1 \mod p$

2.2 Lagrange Interpolation

Refresh myself with Lagrange Interpolation.

The Lagrange Interpolation returns a polynomial of lowest degree that pass through a set of points D.

As an example, consider three points $(3, 1), (4, 2), (7, -3)$. The algorithm first construct three polynomial such that each polynomial f_i go through $(x_i, 1)$ for the i^{th} point in D and $(x_j, 0)\ \forall j \neq i$. This is extremely easy:

$$\begin{cases} f_1(x) = 1/4(x-4)(x-7) \\ f_2(x) = 1/3(x-3)(x-7) \\ f_3(x) = 1/12(x-3)(x-4) \end{cases}$$

Finally, just scale each f_i so when $x = x_i \Longrightarrow y = y_i$, and let $p = \sum_i f_i$

$$p(X) = f_1(X) + 2 * f_2(X) + -3 * f_3(X)$$

We can also prove the resulting polynomial $p(X)$ is the unique polynomial of degree $(n-1)$ that go through those points, by contridiciton.

Assuming the opposite, let $q(X)$ be another polynomial of degree $n-1$ that statistifes points $d \in D$. Since $q(X) \neq p(X)$, $r(X) = q(X) - p(X)$ is not a zero polynomial and $degree(r) \leq (n-1)$. However, we know for points $d \in D$, $r(X) = q(X) - p(X) = 0$, which means $degree(r) = n$, therefore contridiciton.

2.3 Generator

Not sure yet how generator is applied in SNARK or how it is produced — How to produce a generator of multiplicative group of order prime?

2.4 Polynomial Implementation

Just some notes on the notations here.

leading coefficient is the coefficient of the highest degree. It is confusing as I interpreted it as the first coefficient (it is the last one).

The *zerofier* takes a set of domain D where $\forall x \in D$, produce the result polynomial $f(x) = 0$.

2.5 FRI

Fast Reed-Solomon IOP of Proximity.

FRI is a protocol between a prover and a verifier, which establishes that a given codeword belongs to a polynomial of low degree – low meaning at most p times the length of the codeword.

2.6 Fast Fourier Transform & Polynomials

Consider some simple polynomial algorithms.

- Multiplying two polynomial of coefficient representation. $O(n^2)$.
- Coefficient representation to point-value representation. $O(n^2)$ as you need to evaluate $n+1$ points for a n-degree polynomial.
- Lagrange. $O(n^2)$ as mentioned in previous section.

With Fast Fourier Transform, it is possible to perform above algorithms in $O(n \log n)$.

2.7 Point-value Representation and Polynomial Multiplication

Consider two polynomials of degree of $n-1$:

$$A(X) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$$
$$B(X) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1}$$

Each polynomial can be represented by a list of coefficients of size n, $a = (a_0, a_1, \ldots, a_{n-1})$. The multiplication is represented by $a * b$, which takes $O(n^2)$.

Another way of representing a polynomial is through the idea that a polynomial of degree $n-1$ is uniquely defined by $n+1$ points. So

$$A(X) = \{(x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_{n-1}, A(x_{n-1}))\}$$

Multiplication of two polynomial can then be defined by multiplication of those uniquely defined points, $C(X) = A(X)B(X)$:

$$C(X) = \{(x_0, A(x_0)B(x_0)), (x_1, A(x_1)B(x_1)), \ldots, (x_{n-1}, A(x_{n-1})B(x_{n-1}))\}$$

The only catch here is that $C(X)$ is a polynomial of degree $2n-2$, so we actually need to find more points $(2n-1)$ on A and B in order to uniquely define C. However, the complexity of the multiplication drop from $O(n^2)$ to $O(n)$.

2.7.1 Goals

Obviously if we simply choosing arbitrary $2n$ points on the polynomial then this will take us $O(n)$ time for each point and ends up with $O(n^2)$ complexity again. The remark here is that by choosing a specific set of points with Fast Fourier Transform, we can find the point-value representation in just $O(n \log n)$.

2.7.2 Complex Root of Unity

A number $z \in \mathbb{C}$ is an n^{th} *root of unity if* $z^n = 1$. The pricipal nth root of unity is $\omega_n = e^{\frac{2\pi i}{n}}$. The intution here is that $e^{\frac{2\pi i}{n}}$ is a function that rotates as a cycle (through i) and has radius of 1 (through $e^{2\pi}$), n then defines a radians of the rotation. The magic value ω_n^n then defines each full rotation.

Lemma 5. *For integer $n \geq 0$, $k \geq 0$, $d \geq 0$, $\omega_{dn}^{dk} = \omega_n^k$.*

$$\omega_{dn}^{dk} = (e^{\frac{2\pi i}{dn}})^{dk} = (e^{\frac{2dk\pi i}{dn}}) = (e^{\frac{2k\pi i}{n}}) = \omega_n^k$$

Lemma 6. *If n is even, then squares of the nth complex unity root is equal to the $n/2$ of the $n/2$th complex unity root.*

$$\omega_n^2 = (e^{\frac{4\pi i}{n}}) = (e^{\frac{2\pi i}{n/2}}) = \omega_{\frac{n}{2}}$$

This extends to the kth power of the nth complex unity root:

$$(\omega_n^k)^2 = (\omega_n^{2k}) = \omega_{\frac{n}{2}}^k$$

Lemma 7. *If $n \geq 1$ and k is not divisible by n:*

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

Proof:

$$\sum_{j=0}^{n-1}(\omega_n^k)^j = \frac{1-\omega_n^{kn}}{1-\omega_n^k} = \frac{1-(\omega_n^n)^k}{1-\omega_n^k} = \frac{1-1^k}{1-\omega_n^k} = 0$$

This is true as k is not divisible by n hence $\omega_n^k \neq 1$.

2.7.3 Discrete Fourier Transform

Let $a = (a_0, a_1, \ldots, a_{n-1})$ be a coefficient representation of polynomial A, define $\hat{a} = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1})$ where:

$$\hat{a}_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

The remark is shown by considering value of polynomial A at point ω_n^k.

$$A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj} = \hat{a}_k.$$

Each $A(\omega_n^k)$ still takes $O(n)$ to compute, there are still a lot of missing pieces, wait...

2.7.4 Fast Fourier Transform

FFT is an efficient algorithm of computing the above sequence using a divide-and-conquer approach.

Assuming n is a power of 2, in addition to $A(X)$, we define two other polynomials:

$$A_e(X) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{\frac{n-2}{2}}$$
$$A_o(X) = a_1 + a_3 x^1 + a_5 x^2 \ldots + a_{n-1} x^{\frac{n-2}{2}}$$

We can then represent A by:

$$A(X) = A_e(X^2) + X A_o(X^2)$$

The problem of evaluating A at $(\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1})$ is reduced to:

1. Compute A_e and A_o at $((\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2)$
2. Combine the result

Wait, but A_e and A_0 only have degree $n/2$ and cannot be used to evaluated the full sequence. But consider $d \leq \frac{n}{2}$: [5]

$$(\omega_n^d)^2 = \omega_n^{2d}\omega_n^n = \omega_n^{2d+n} = (\omega_n^{d+\frac{n}{2}})^2$$

Therefore, calculating the first halves of the sequence give us the result in the second halves. This also explains in the earlier section why only providing n points is enough to calculate polynomial C of degree $2n-2$.

[5] The slides specifically refers to halving lemma 6, but I don't see how the lemma is necessary here to get the conclusion.

2.7.5 Inverse of FFT

Is it needed? TBD...

2.7.6 Q

It seems like as long as ω_n statistifes that only $\omega_n^n = 1$ we can get the same conclusion, not sure why the primitive root is used here (as an example?). As well as some other lemmas... E.g., I don't get why halving lemma is necessary to get the final conclusion, it just uses basic exponent rules.

3 Finite Fields

Basic concept of finite field, for future references.

3.1 Groups

A group is a set of elements $G = \{a, b, c, \ldots\}$ and an operator $\oplus$ such that

1. Closure: $\forall a, b \in G : a \oplus b \in G$
2. Associative $\forall a, b, c \in G : (a \oplus b) \oplus c = a \oplus (b \oplus c) \in G$
3. Identity: $\exists 0 \in G, \forall a \in G : a \oplus 0 = a$
4. Inverse: $\forall a \in G, \exists b \in G : a \oplus b = 0$, simply denote $b = -a$.
5. Permutation: $\forall a \in G : a \oplus G = \{a \oplus b \mid b \in G\} = G$, in another word, this gives us a different permutation of G.

In addition, if $\forall a, b \in G : a \oplus b = b \oplus a$, G is called *abelian*.

Note: Often, the operation is called multiplication, presented by $\times$, and identity is written as 1, inverse is written with a^{-1}.

3.2 Cyclic Groups

A *finite cyclic group* is a finite group G with a *generator* element g such that $G = \{g, g \oplus g, g \oplus g \oplus g, \ldots\}$

Since g generates G, this must includes the identity element 0. Therefore, we must have $ig = 0$ for some i. Let k be the smallest i such that $kg = 0$, we must have $ng \neq 0$ for $1 \leq n \leq k - 1$. For each element ng with $1 \leq n \leq k - 1$, we have:

$$kg \neq 0$$

$$\implies kg + ng \neq ng$$

$$\implies (k + n)g \neq ng$$

Therefore, all elements in G are different.

3.3 Subgroups

A subgroup S of a group G is a subset of G such that $\forall a, b \in S : a \oplus b \in S$. Therefore, the subgroup must contain the identity element in G and all the inverse of each element in S. A subgroup is also a group. If G is abelian, then the subgroup is abelian. The inverse is not necessary true.

3.4 TBD

4 Elliptic Curve

Slowly connecting elliptic curve, pairing and KZG commitments. It is hard to plot elliptic curve in latex, so figure would be omitted here. I might add figure in later editing. First part followed from here.

4.1 Group Law for Elliptic Curve

We can define a group over elliptic curve.

- the elements of the group are the points of an elliptic curve
- the identity element is the point at infinity 0
- the inverse of a point is the one symmetric about the x-axis
- given three aligned points, P, Q, R, their sum is $P + Q + R = 0$.

Immediately, we have $P + (Q + R) = (P + Q) + R = R + (P + Q) = 0$, both associative and commutative rules are satisfied, hence an *abelian* group.

4.2 Geometric Addition

Given this is an abelian group, we have $P + Q = -R$. So we can draw a line across P, Q, R, the symmetric point of R gives us the result $-R$. Consider some corner cases:

1. What if $P = 0$ or $Q = 0$? Given 0 is defined as identity element, we have $0 + P = P$.
2. What if $P = -Q$? I.e., the line is vertical and only pass two points. Given the definition of inverse, we have $P + Q = P + (-P) = 0$.
3. What if $P = Q$? Imagine a point Q' move toward P, as $Q' \approx P$, we have a line that is tangent to the curve, hence we say $P + P = -R$, where R is the point that intersect with the tangent line.
4. What if $P \neq Q$ but there is no third point and the line is not vertical? This is similar to the previous point, where line defined by P, Q is tangent to the curve. Assuming P is the tangent point, we then have $P + P = -Q$.

4.3 Algebraic Addition

This is entirely by me, not sure if my conclusion is right (audit it please), the details are omitted in the origin post.

Consider two points $P = (x_P, y_P), Q = (x_Q, y_Q)$, how to find the third point $R = (x_R, y_R)$ such that three are points aligned (assuming exists).

We know line pass three points have the form:

$$y = y_Q + m(x - x_Q)$$

or:

$$y = y_P + m(x - x_P)$$

Hence $y_R = y_Q + m(x_R - x_Q)$. Next finding x_R. Given curve of form $y^2 = x^3 + Ax + B$, we have:

$$x^3 + Ax + B - (m(x - x_Q) + y_Q)^2 = 0$$

for x taking x_Q, x_P and x_R. Expanding it:

$$\implies x^3 + Ax + B - (mx - mx_Q + y_Q)(mx - mx_Q + y_Q)$$

$$\implies x^3 + Ax + B - m^2x^2 + m^2xx_Q - mxy_Q + m^2xx_Q - m^2x_Q^2 + mx_Qy_Q - mxy_Q + mx_Qy_Q - y_Q^2$$

$$\implies x^3+(-m^2)x^2+(A+2m^2x_Q-2my_Q)x+(B-m^2x_Q^2+mx_Qy_Q-mx_Qy_Q-y_Q^2)$$

Looks complicated, but from *Vieta's formulas*, we have the sum of roots:

$$\sum_{r_i} = -\frac{a_{n-1}}{a_n}$$

where a_n is the n^{th} coefficient. So $x_Q + x_P + x_R = m^2$, and we have

$$x_R = m^2 - x_Q - x_P$$

with slop m

$$m = \frac{y_P - y_Q}{x_P - x_Q}$$

Note, this also works on tangency point ($Q = P$). It's just that the slop m needs to taken by first derivatives:

$$m = \frac{3x_P^2 + a}{2y_P}$$

4.4 Scalar Multiplication

We define scalar multiplication nP:

$$nP = P + P + P \dots P$$

There exists simple logarithm algorithm for computing nP.

4.5 Elliptic Curve over $\mathbb{F}_p$

When define over $\mathbb{F}_p$, we have a set of points:

$$\{(x, y) \in \mathbb{F}_p^2 \mid y^2 \equiv x^3 + ax + b \mod p\} \cup \{0\}$$

The *order* of a group is the cardinality of the set of points. This can be done using the *Schoof's Algorithm*.

The core of the discrete logarithm is that: given P and Q, it is "hard" to find k such that $P = Qk$.

4.5.1 Point addition and algebraic sum

The definition is almost the same, except we define three points are aligned if there exists line $y \equiv ax + b \mod p$ that pass through all points. If you can imagine, this line would wrap around the panels.

- $Q + 0 = 0 + Q$.
- Given $Q = (x_Q, y_Q)$, $-Q = (x_Q, -y_Q \mod p)$
- $P + (-P) = 0$

The equations for calculating the third points given two points is exactly the same, except taking the module of p.

- $x_R = m^2 - x_P - x_Q \mod p$
- $y_R = y_P + m(x_R - x_P) \mod p$
- $m = (y_P - y_Q)(x_P - x_Q)^{-1} \mod p$ if $P \neq Q \mod p$
- otherwise, $m = (3x_P^2 + a)(2y_P)^{-1} \mod p$

4.5.2 Cyclic Subgroup

Taking a point P from the curve, the set of multiplicative of P is closed and forms a cyclic subgroup. P is also called a *generator* or a *base*. The order of such group (I.e., the order of P) is defined by the smallest positive integer n such that $nP = 0$, so brute force takes at least $O(n)$. However, from *Lagrange theorem*, which state that the order of subgroup must be a divisor of the parent group. Therefore, we can improve the algorithm:

1. Calculate order of the curve N by Schoof's algorithm.
2. Find all divisor of N
3. For all divisor, try nP, the smallest n such that $nP = 0$ is the order of P.

4.5.3 Finding base point

In some applications, we want to find a group of higher order. So instead of starting from a base point P, we would start by finding all divisor of N, taking a high divisor (n) to be the subgroup order, and work in reverse to find the generator of that subgroup.

Note Lagrange theorem implies $h = N/n$ is always an integer (since n must be a divisor). The number h is called *cofactor*.

Consider for any point P on the curve, we must have $NP = 0$, hence $n(hP) = 0$. Now suppose n is a prime number:

1. Calculate order N of the elliptic curve.
2. Choose a n we want to use as our subgroup order, but n has to be a prime number.
3. Compute cofactor h by N/n.
4. Choose a random point P on the curve.
5. Compute $G = hP$, if $G = 0$, go back to step 4, otherwise, we have found the generator.

The reason of why n must be prime is because: if it is not, P could have an order that is actually a divisor of n.

This is interesting, I wonder whether special curve needs to be crafted so that finding P would not be so hard (a group with prime order or order with only few divisor?).

4.5.4 Encryption with ECDH

Since encryption is not our main concern, I'll just use an example to quickly finish off this chapter.

A *private key* is d, which is a random integer taken from the divisor of the subgroup order N. A *public key* is the point $H = dG$, where G is the base of the subgroup.

1. Alice and Bob exchange their public key $H_A = d_A G$ and $H_B = d_B G$ (G can be shared, public information).
2. Alice calculate $S_A = d_A H_B$ and Bob calculates $S_B = d_B H_A$, you can see $S_A = d_A(d_B)G = d_B(d_A)G = S_B$

5 Connecting Pairing With KZG

This section still missing tons of detail

1. Yet another curve, but THE curve for your KZG!

2. KZG polynomial commitments
3. Exploring Elliptic Curve Pairings
4. BLS12-381 For The Rest Of Us

5.1 Pairing

A pairing is a bilinear map:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$$

where

- $\mathbb{G}_1$ and $\mathbb{G}_2$ are groups of order r defined over elliptic curves with generator G_1 and G_2.
- $\mathbb{G}_T$ is also a group over a *finite field extension*.

In particular, $\mathbb{G}_1$ is usually defined over curve $E(\mathbb{F}_p)$, and $\mathbb{G}_2$ and $\mathbb{G}_T$ are defined over $E(\mathbb{F}_{p^k})$ TODO: Example with BLS12-381.

The function e has the following property:

1. $e(r \times m, n) = r \times e(m, n)$
2. $e(m, r \times n) = e(m, r) \times r$

Almost certain it is wrong:
For paring purpose on elliptic curve over finite field, we can conclude :

$e(r \times P, d \times Q)$

$\implies r \times e(P, d \times Q)$

$\implies r \times e(P, Q) \times d$

$\implies r \times d \times e(P, Q)$

$\implies e(P, Q)^{rd}$ something is missing here

Many references omit what exactly is the function e here. Looking at some code examples and the post blow, I believe it is using either *Tate Paring* or *Weil Paring*
https://crypto.stanford.edu/pbc/notes/ep/

5.2 KZG commitment

Let $p(x)$ be the polynomial we want to commit

1. setup secret τ, sample randomly from $\mathbb{F}_p$.
2. Prove and verifier share G_1, G_2, $pk = \tau^i G_1$ and $vk = \tau G_2$

3. Verifier receive commitment $cm_p = \sum_{i=0} p_i \cdot \tau^i G_1$ for $i \in \{1, \ldots, m\}$
4. Verifier send requests $open(p, y, z)$: to prove $p(z) = y$.
5. Prover compute $q(x) = \frac{p(z)-y}{x-z}$ and send commitment cm_q.
6. Verifier compute $P_z = zG_2$ and $P_y = yG_1$ and check $e(cm_q, vk - P_z) = e(cm_p - P_y, G_2)$.

For convince, we write $cm_p = \sum_{i=0} p_i \cdot \tau^i G_1 = p(\tau)$, same goes for cm_q.
To see the correctness:

$$\begin{aligned} e(cm_q, vk - P_z) &= e(cm_p - P_y, G_2) \\ e(\sum_{i=0} q_i \cdot \tau^i G_1, \tau G_2 - zG_2) &= e(cm_p = \sum_{i=0} p_i \cdot \tau^i G_1 - yG_1, G_2) \\ e(G_1, G_2)^{q(\tau)(\tau - z)} &= e(G_1, G_2)^{p(\tau)-y} \\ \mathbb{X}_{\mathbb{G}_T}^{q(\tau)(\tau-z)} &= \mathbb{X}_{\mathbb{G}_T}^{q(\tau)-y} \end{aligned}$$