ZERO KNOWLEDGE PROOF

WEB3零知識證明 高效密碼運算算法

陸晨博士 教授

香港國際家族辦公室總會HKIFOA特邀顧問
紐約大學庫朗數學研究所應用數學博士
哈佛大學商業心理學BPSY特聘教授
香港大學中國商學院客座教授
北京大學深圳研究院智能科技研討班首席科學家
深圳前海金融創新促進會專委會首席科學家

陆晨博士 教授

香港國際區塊鏈&金融科技總會特邀顧問
香港國際家族辦公室總會特邀顧問
北大深圳研究院智慧科技研討班首席科學家
前海產業智庫首席國際金融學家
深圳市人工智能產業協會金融委員會首席風險官
香港青年專業精英人士促進會榮譽總顧問
深圳私募基金協會證券合規風控專業委員會委員
哈佛大學商業心理學BPSY特聘教授
上海交大高級金融學院EMBA特邀教授
北大光華管理學院客座教授
中國人大商學院EMBA特聘導師
前平安磐海資本首席風險官

纽约大学数学博士，CFA，FRM，PRM; 曾任普华永道，德勤（香港）市场风险、流动性风险、利率风险，交易对手信用风险总监，协助香港金管局HKMA处理2008年香港金融衍生品风险事件（Accumulator和MiniBond）并接受香港主流媒体和美国Bloomerg的采访报导

中国平安 保险·银行·投资 平安证券
pwc
VE RI TAS HARVARD
清華大學 Tsinghua University
MACQUARIE UNIVERSITY
光华管理学院 Guanghua School of Management
SAIF Shanghai Advanced Institute of Finance
香港大學 THE UNIVERSITY OF HONG KONG

課程安排

零知識證明高效密碼運算算法

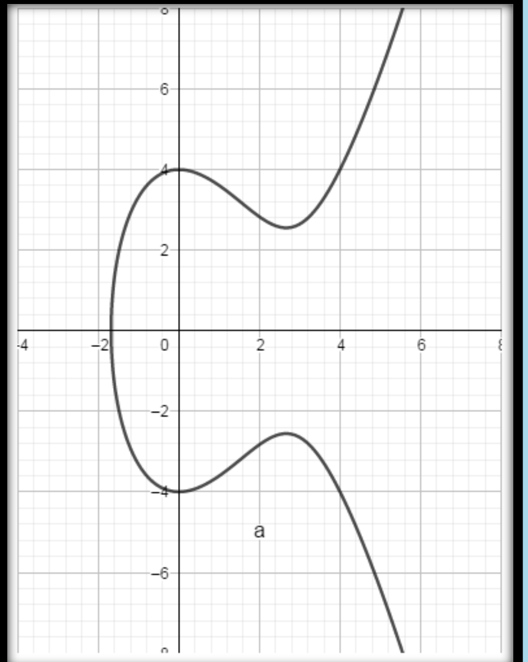
- 快速橢圓曲綫翻倍和加法算法和乘法算法
- NTT 數論變換算法
- MSM 多標量乘法

神奇的橢圓曲綫的發展

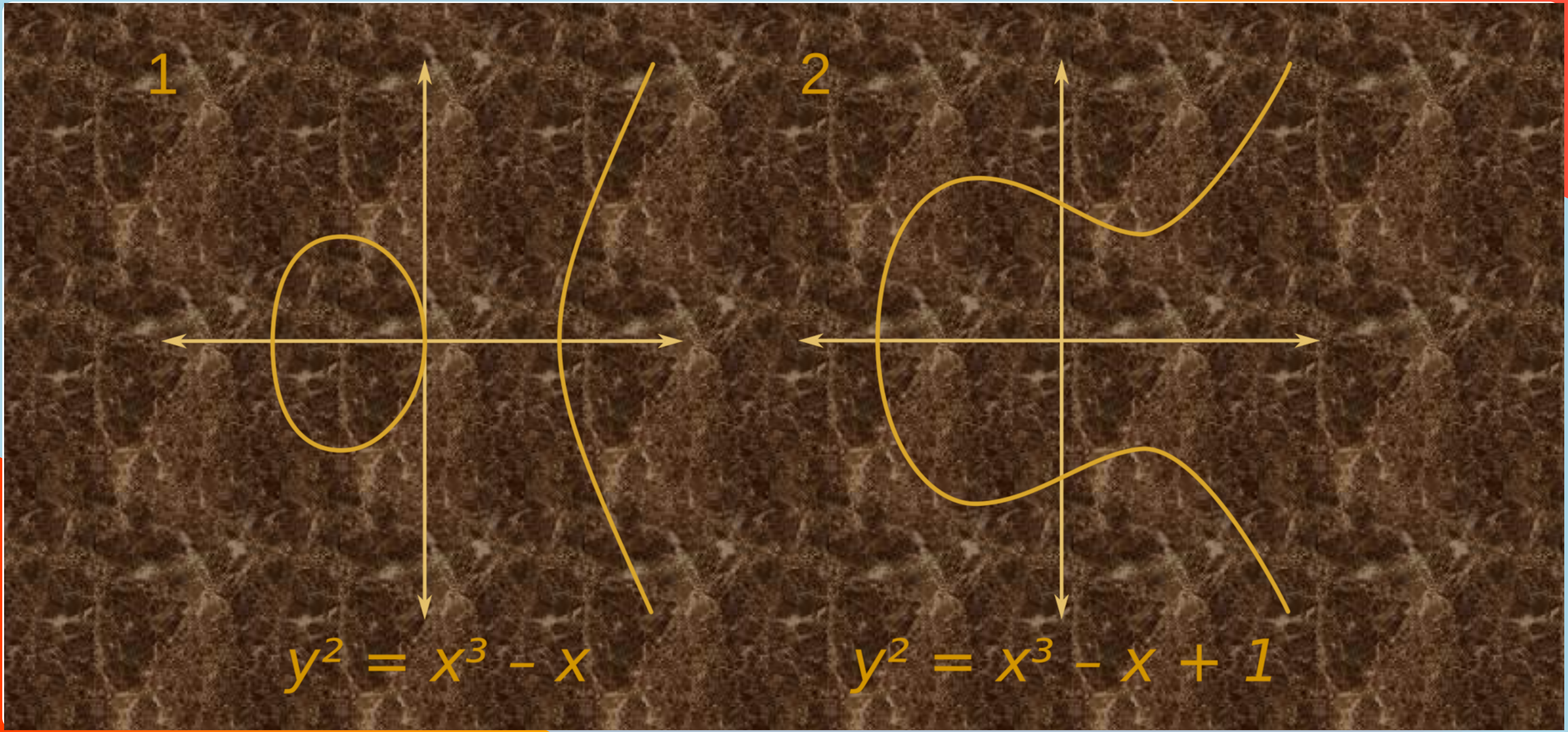
- 椭圆曲线方程来源于椭圆积分，后者来最初来源于计算椭圆周长的问题，有一段时间的历史了，在欧拉时期就开始研究
- 对于椭圆曲线上的点和O点组成的集合，以及集合上定义的二元加法运算，构成一个**Abel**群。单位元是O点，P(x,y)的逆元是P(x,-y)，封闭性，结合性以及交换性也是显然满足的

密码学中普遍采用的是有限域上的椭圆曲线，也即是变元和系数均在有限域中取值的椭圆曲线。使用模素数p的有限域Zp，将模运算引入到椭圆曲线算术中，变量和系数从集合0,1,2,...,p-1中取值而非是在实数上取值

有限域上的椭圆曲线的点和加法运算构成一个有限交换群S



橢圓曲綫大观园

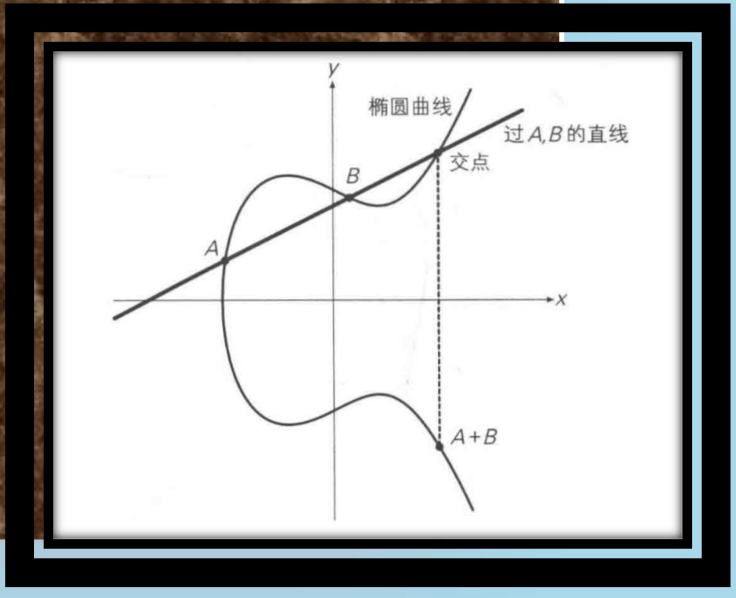


快速橢圓曲綫翻倍和加法算法

- *p a prime #*
- *g a group member of cyclic order p =>* $g*p=1_G$
- *we can think about g as a point on the elliptic curve*
- **We like to calculate [n]g**
- Classical way to compute this

  (((g + g) + g) + g) + g …….
- Another way: repeatedly doubling $2^{\wedge}kg$

  $P_0 = g; P_1 = 2P_0; P_j = 2P_{j-1}$ 。。。。。。



快速橢圓曲綫翻倍和加法算法

$Q = g$

For $i = 1$ to t

$Q = 2Q$

if $e_i = 1$ $R = R + Q$

Return R (ng)

This is the so-called Squared and Plus operation

DAA ②

First, we write out the binary expansion of n,

$\lambda - 1 = t$

$$n = e_0 + e_1 \cdot 2 + e_2 \cdot 2^2 + \cdots + e_{\lambda-1} 2^{\lambda-1}$$

so $e_0, \dots, e_{\lambda-1}$ is the binary representation of n

output accumulator

op identity, base pt, pt at ∞, $(0,1,0)$

Set our double accumulator $Q = P$, $R = \begin{cases} O & e_0 = 0 \\ P & e_0 = 1 \end{cases}$

快速橢圓曲綫乘法群算法

For a multiplicative group (any abelian group, not even cyclic) G, $g \in G$,

if $n = \sum e_i 2^i$

compute $g^n = \prod (g^{2^i})^{e_i}$

in $\le \log_2 n$ squarings

and $\le \log_2 n$ multiplications

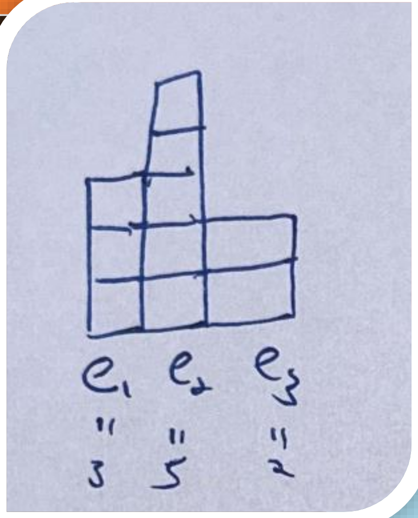
If n is a random #, it will be expected half of the bits are ones and half of the bits are zeroes. So, $\log_2 n$ squares and ½ $\log_2 n$ additions

橢圓曲綫乘法快速高效算法

- Start with n = 3

$G = g_1^{e_1} g_2^{e_2} g_3^{e_3}$ G a group

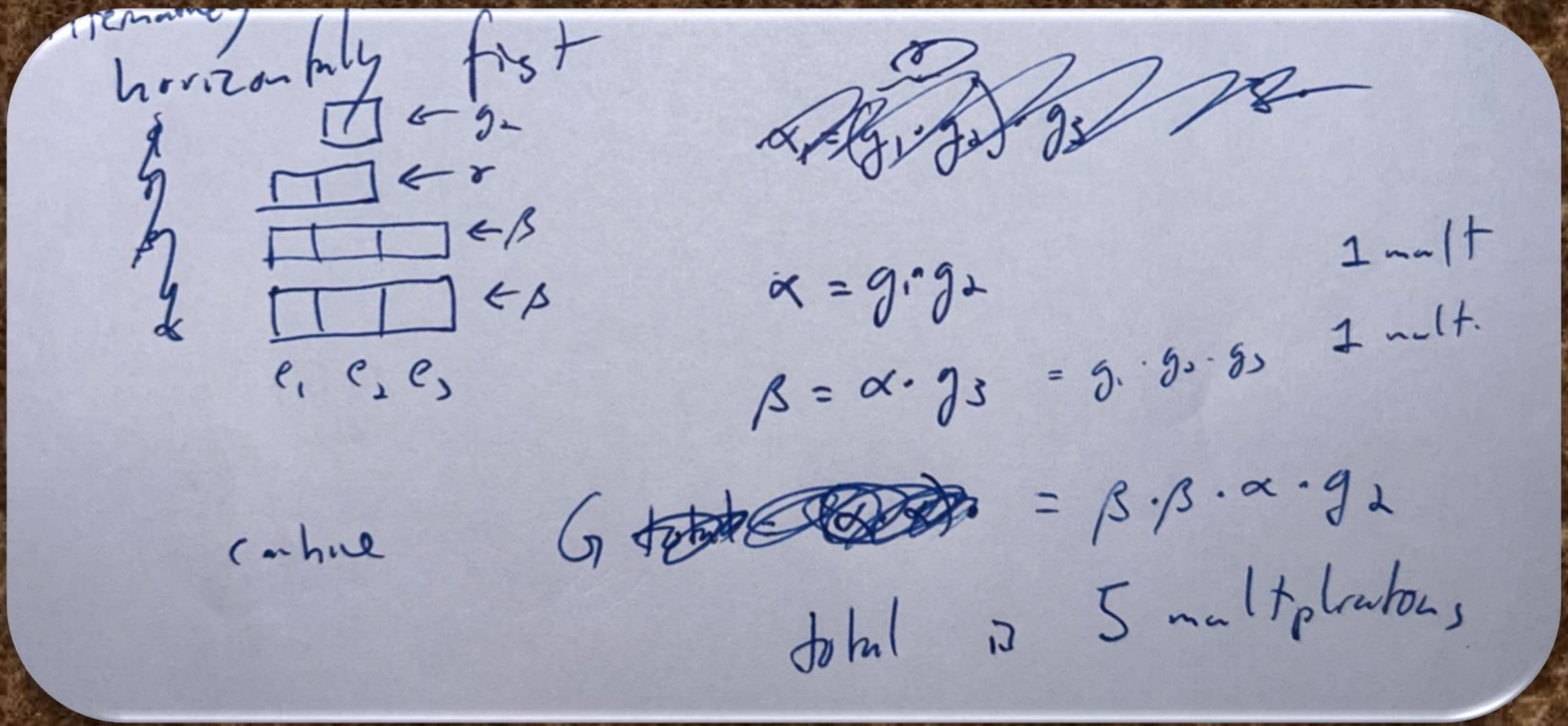
- If $e_i = \{0, 1\}$ then **G** is the multi-product



- $G = g_1 * g_1 * g_1\ g_2 * g_2 * g_2\ g_2 * g_2 * g_3 * g_3$
- Model the operation cost as either the multiplication or squaring
- We calculate this example, **2 + 3 + 1 + 2 = 8**

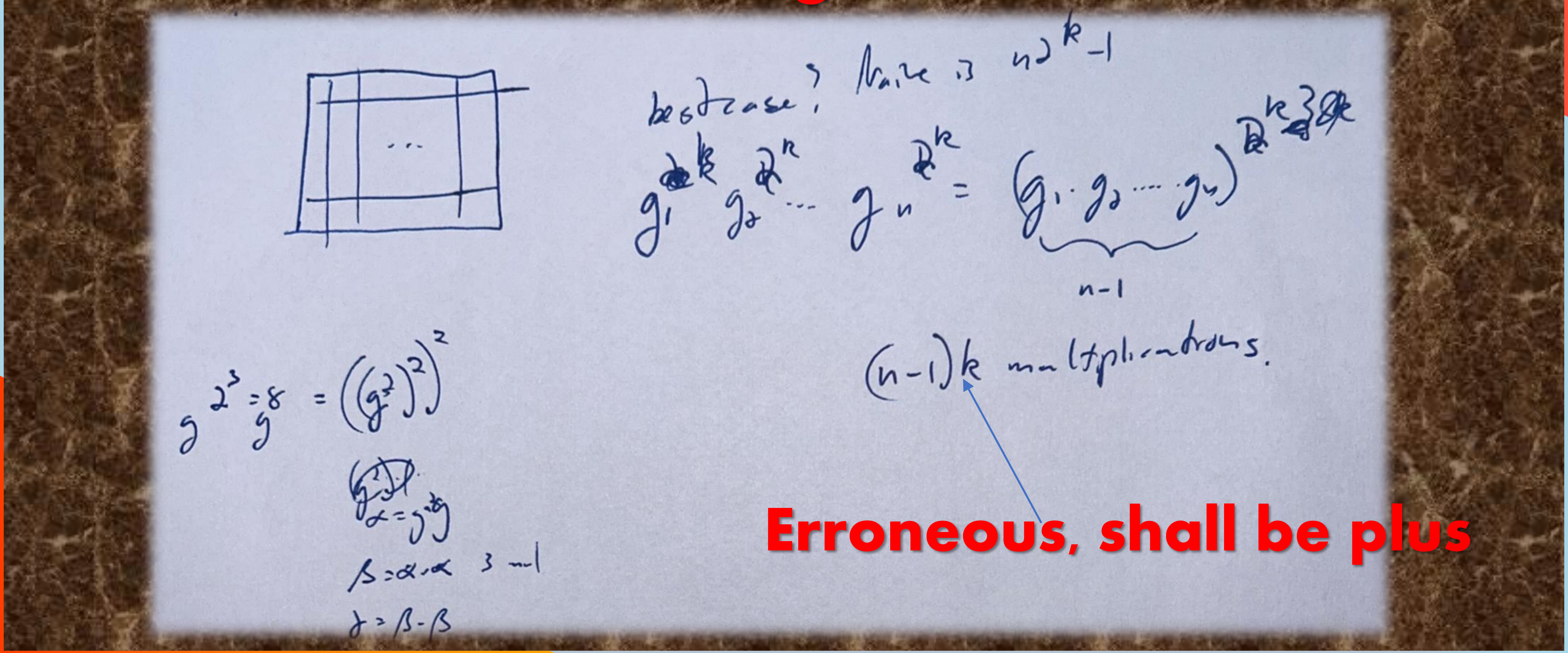
橢圓曲綫乘法快速高效算法

- **Another way, Abelian sum method or Lebesgue integration method**



橢圓曲綫乘法快速高效算法

- Now we move to the general situation



Erroneous, shall be plus

椭圆曲线乘法快速高效算法

Now we'll use some information about the group, Assume G is a cyclic group of order p, where p is a $\approx \lambda$-bit prime, 254 bit so $\lambda = 256$,

$$G = \prod_{i=0}^{N-1} g_i^{e_i} \quad \text{msm}$$

Assume $\lambda \geq N$ and decompose λ into s legs, $\lambda =$ s.t, say $s = 4$ legs and $t = 64$ of size t each w/ $N = 16$. (turns out $s \approx \sqrt{\frac{\lambda}{N}}$ and $t \approx \sqrt{\lambda N}$ is optimal and convenient)

橢圓曲綫乘法快速高效算法

Let $e_i = \sum_{\ell=0}^{\lambda-1} \overbrace{e_{i,\ell}}^{\in\{0,1\}} 2^{\ell}$ so the $e_{i,\ell}$ are the binary digits of e_i lowest-order to highest. little-endian

splitting λ into legs

$$e_i = \sum_{j=0}^{s-1} \sum_{k=0}^{t-1} 2^{j+sk} \underbrace{e_{i,j+sk}}_{0 \text{ or } 1}$$

Then $g_i^{e_i} = \prod_{\ell=0}^{\lambda-1} g_i^{2^{\ell} e_{i,\ell}} \underset{\lambda=st}{=} \prod_{j=0}^{s-1} \prod_{k=0}^{t-1} g_i^{2^{j+sk} \underbrace{e_{i,j+sk}}_{0 \text{ or } 1}}$

橢圓曲綫乘法快速高效算法

so $$G = \prod_{i=0}^{N-1} g_i^{e_i} = \prod_{i=0}^{N-1} \left(\prod_{j=0}^{s-1} \prod_{k=0}^{t-1} g_i^{2^{j+sk} e_{i,j+sk}} \right)$$

and rearranging the product,

$$G = \prod_{k=0}^{t-1} \Bigg(\underbrace{\prod_{i=0}^{N-1} \prod_{j=0}^{s-1} g_i^{2^j e_{i,j+sk}}}_{:= G_k'} \Bigg)^{2^{sk}}$$

repeated squaring

橢圓曲綫乘法快速高效算法

- So, now we only need to figure out how to handle those multiplication products inside the parentheses G_k'
- Learning from the early part, we would like to utilize the squared terms such as

$$g_i, g_i^2, g_i^4, \ldots g_i^{2^j}, \ldots g_i^{2^{s-1}}$$

- The total cost is N*S = N * $\sqrt{\lambda/N} = \sqrt{\lambda N}$

橢圓曲綫乘法快速高效算法

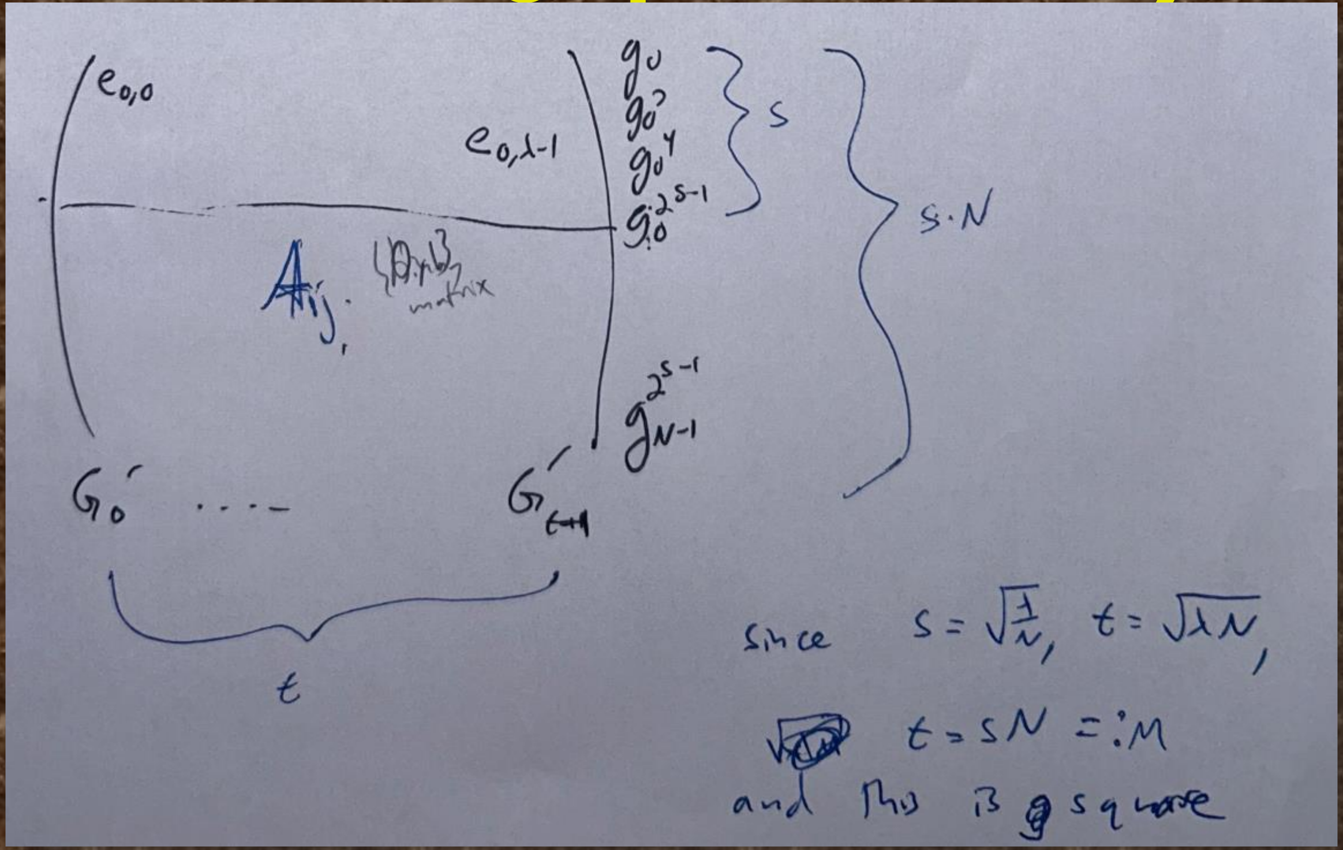
- For such inputs, each inner product term

$$\prod_{i=0}^{N-1} \prod_{j=0}^{S-1} g_i^{\alpha^j e_{i,j+sk}}$$

will be a multiplication of $g_i^{\alpha^j}$

橢圓曲綫乘法快速高效算法

- We will form the following sparse binary square matrix



橢圓曲綫乘法快速高效算法

The first matrix is as follows

$$\begin{pmatrix}
e_{0,0} & e_{0,s} & e_{0,2s} & & e_{0,(t-3)s} & e_{0,(t-2)s} & e_{0,(t-1)s} \\
e_{0,1} & e_{0,s+1} & e_{0,2s+1} & \cdots & e_{0,(t-3)s+1} & e_{0,(t-2)s+1} & e_{0,(t-1)s+1} \\
e_{0,2} & e_{0,s+2} & e_{0,2s+2} & & e_{0,(t-3)s+2} & e_{0,(t-2)s+2} & e_{0,(t-1)s+2} \\
 & \vdots & & \ddots & & \vdots & \\
e_{0,s-3} & e_{0,2s-3} & e_{0,3s-3} & & e_{0,(t-2)s-3} & e_{0,(t-1)s-3} & e_{0,ts-3} \\
e_{0,s-2} & e_{0,2s-2} & e_{0,3s-2} & \cdots & e_{0,(t-2)s-2} & e_{0,(t-1)s-2} & e_{0,ts-2} \\
e_{0,s-1} & e_{0,2s-1} & e_{0,3s-1} & & e_{0,(t-2)s-1} & e_{0,(t-1)s-1} & e_{0,ts-1}
\end{pmatrix}$$

橢圓曲綫乘法快速高效算法

The k-th block matrix is as follows

$$
\begin{pmatrix}
e_{k,0} & e_{k,s} & e_{k,2s} & & e_{k,(t-3)s} & e_{k,(t-2)s} & e_{k,(t-1)s} \\
e_{k,1} & e_{k,s+1} & e_{k,2s+1} & \cdots & e_{k,(t-3)s+1} & e_{k,(t-2)s+1} & e_{k,(t-1)s+1} \\
e_{k,2} & e_{k,s+2} & e_{k,2s+2} & & e_{k,(t-3)s+2} & e_{k,(t-2)s+2} & e_{k,(t-1)s+2} \\
 & \vdots & & \ddots & & \vdots & \\
e_{k,s-3} & e_{k,2s-3} & e_{k,3s-3} & & e_{k,(t-2)s-3} & e_{k,(t-1)s-3} & e_{k,ts-3} \\
e_{k,s-2} & e_{k,2s-2} & e_{k,3s-2} & \cdots & e_{k,(t-2)s-2} & e_{k,(t-1)s-2} & e_{k,ts-2} \\
e_{k,s-1} & e_{k,2s-1} & e_{k,3s-1} & & e_{k,(t-2)s-1} & e_{k,(t-1)s-1} & e_{k,ts-1}
\end{pmatrix}
$$

$$G_k' = \prod_{i=0}^{N-1}\prod_{j=0}^{s-1} g_i^{2^j e_{i,j+sk}}$$

橢圓曲綫乘法快速高效算法

- The expression $\prod_{j=0}^{s-1} g_i^{2^j e_{i,j+sk}}$ is obtained by

橢圓曲綫乘法快速高效算法

- A simple strategy for evaluating these expressions is to pick a partition of the vertical vector $S_0, \dots S_{M/b-1}$ of at most b elements
- For each S_i, we compute all possible multiplications, denoted as T_i
- For example, for b = 3, then, $S_0 = \{ h_0, h_1, h_2 \}$

$T_0 = \{ h_0 , h_1 , h_2 , h_0h_1, h_0h_2 , h_1h_2 , h_0h_1h_2 \}$

橢圓曲綫乘法快速高效算法

- G_k' is a product of at most one element from each T_i

Analysis: each S_i has b elts, 2^b gp ops to compute all possible multiproducts / contributions.

M/b sets, so this precompute is $\frac{2^b M}{b}$ gp ops.

Given the T_i, Each H_i needs at most one elt from each set,

so $\frac{M}{b}$ gp ops

~~M His be~~

There are M of the H_is, so $\frac{M^2}{b}$ given the T_i

橢圓曲綫乘法快速高效算法

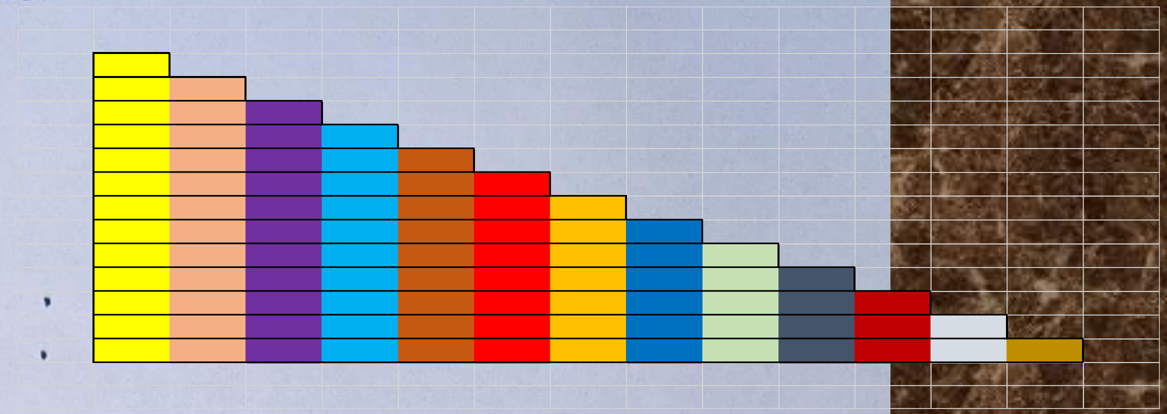
Finally we recombine inputs

$$G = \prod_{k=0}^{t-1} G_k'^{\,2^{sk}}$$

using $st = \lambda$ squarings:

square G'_{t-1} s times,

multiply by G'_{t-2}, square that s times, ...



橢圓曲綫乘法快速高效算法

$M = \sqrt{\lambda N} \quad : \quad t = sN$

$$\lambda \quad + \quad M \quad + \quad 2^b \frac{M}{b} \quad + \quad \frac{M^2}{b}$$

squarings in $\prod_{k=0}^{t-1} G_k^{2^{sk}}$

multiplications in

precompute contributions T_i

compute $H_k = G_k$ given contributions T_i

橢圓曲綫乘法快速高效算法

- For one of the optimal choices of b, we have

$$b = \log M - \log\log M,$$

- Plug this choice of b into the above calculation

$$\lambda + M + \frac{M^2}{(\log M - \log\log M)(\log M)} + \frac{M^2}{\log M - \log\log M}$$

$$= \lambda + (1+o(1))\frac{M^2}{\log M}$$

椭圆曲线乘法快速高效算法

- We put everything together to obtain the final optimal cost result

$M = \sqrt{\lambda N}$ so cost is

$$\lambda + (1+o(1))\frac{2\lambda N}{\log \lambda N}$$

why doesn't it depend on the degree bound?

λ is the degree bound

Maybe if actual exponents are $e \ll p$, there is a better way.

FFT&DFT高效算法

- We are goanna deal with polynomials

The analogy here (can be made precise!) is that
a polynomial can be given in several forms
as well, including

1. as coefficients   expanded $3 + 4x + 7x^2 + 5x^3 + \dots$

   or

   $3x_1x_2x_{10} + 17x_3x_7x_{30} + \dots$

or 2. evaluated at points (need degree + 1 points for
a univariate polynomial, # of monomials of degree $\leq n$ o/w
w/ caveats

FFT&DFT高效算法

Thm if $x_1,\dots,x_p \in \mathbb{F}^m$ distinct points in $\mathbb{F}^m$, TFAE

(1) given $y_1,\dots,y_p \in \mathbb{F}$

$\exists!\, f \in F[\vec{x}]$ of deg $\le n$

s.t. $f(x_i) = y_i \quad \forall\, 1 \le i \le p$

(2) Sample $p \times p$ matrix

$M = \left(x_i^{I_j} \right)$ is invertible

Vandermonde w/ $m=1$.

FFT&DFT高效算法

$$
\begin{array}{rl}
& a_0 + a_1 x + a_2 x^2 + a_3 x^3 = f \\
+ & \underline{b_0 + b_1 x + b_2 x^2 + b_3 x^3} = g \\
& (a_0+b_0) + (a_1+b_1)x + (a_2+b_2)x^2 + (a_3+b_3)x^3
\end{array}
$$

deg + 1 additions

<u>In evaluation form</u>

$(f+g)(1) = f(1) + g(1)$

$(f+g)(2) = f(2) + g(2)$ enough to determine f+g in eval form

$(f+g)(3) = f(3) + g(3)$

$(f+g)(4) = f(4) + g(4)$ ← ~~deg + deg~~ degree + 1

FFT&DFT高效算法

Polynomial multiplication

- ZK-snark constructions use polynomial multiplications
- The polynomials involved are univariate and high degree.

Let's say we have coeff-form poly f, g & want coeff-form-poly fg

$\deg f = 2$

$$f(x) = a_0 + a_1 x + a_2 x^2$$

$$g(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3$$

naively we need $(\deg f + 1)(\deg g + 1)$ (field) multiplications

$$f \cdot g = (a_0 + a_1 x + a_2 x^2)(b_0 + b_1 x + b_2 x^2 + b_3 x^3)$$

FFT&DFT高效算法

$$f(x) = a_0 + a_1 x + a_2 x^2$$

$$g(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3$$

naively we need $(\deg f + 1)(\deg g + 1)$ multiplications (field)

$$f \cdot g = (a_0 + a_1 x + a_2 x^2)(b_0 + b_1 x + b_2 x^2 + b_3 x^3)$$

$$= a_0 b_0 + (a_0 b_1 + a_1 b_0)x + (a_0 b_2 + a_1 b_1 + a_2 b_0)$$

$$(f \cdot g)_\ell = \sum_{j+k=\ell} a_j b_k = \sum_{j=0}^{\ell} a_j b_{\ell-j}$$

"$O(n^2)$"

But $(f \cdot g)(x) := f(x) \cdot g(x)$ (the definition above is fact)

FFT&DFT高效算法

But $(f \cdot g)(x) := f(x) \cdot g(x)$ (the definition above in fact)

so in evaluation form, we only need deg + 1 multiplications " ~~nearly linear time~~ " $O(n)$

so the idea is to multiply polynomials in coefficient form, change them to evaluation form, multiply, change back.

This is good if the changes are cheap, FFT makes it $O(n \log n)$

FFT&DFT高效算法

- N-th unit roots $\omega^n = 1$
- Prime n-th unit root, if for any natural # q < n $\omega^q \mathrel{!=} 1$

So the idea a is that the coefficient ring of the polynomials (here $\mathbb{F}_p$, a field) should contain certain roots of unity, w

we p use, instead of evaluations $f(1), f(2) \ldots,$ evaluations $f(w^1), f(w^2), \ldots,$ and this makes interpolating f from its evaluations efficient. Thus eval form $\to$ coeff form will be efficient.

FFT&DFT高效算法

NTT ③

Ex $\quad x^{8-1}=1$ so every nonzero elt is a root of unity $(x^8=x)$.

An $n\in\mathbb{Z}_{>0}$ is a primitive root of unity if $x^n=1$ but $x^m\neq 1$ for any $m<n$.

If a is an n^{th} pr.o.u,

$\mathbb{F}$ has all n roots of unity $1, a, a^2, \dots a^{n-1}$.

$\mathbb{F}_q$ contains n^{th} primitive root of

FFT&DFT高效算法

So if $f = \sum_{j=0}^{n-1} f_j x^j$ is a polynomial of n-dim

coeff vector $f_0, \dots, f_{n-1}$,

we have the map "evaluate at powers of ω"

$DFT_\omega : \mathbb{F}_q^n \to \mathbb{F}_q^n$

$f \mapsto (f(1), f(\omega), f(\omega^2), \dots, f(\omega^{n-1}))$

this is the "discrete fourier transform" is an $\mathbb{F}$-linear map

to convolution

FFT&DFT高效算法

We have to adjust the notion of multiplication,
basically the same but degree is mod n

$$f = f_0 + f_1 x + \cdots + f_{n-1} x^{n-1}$$

$$g = g_0 + g_1 x + \cdots + g_{n-1} x^{n-1}$$

convolved

$$h = f *_n g = \sum_{l=0}^{n-1} h_l x^l$$

$0 \le l \le n$

$$\text{w/ } h_l = \sum_{j+k \equiv l \bmod n} f_j g_k = \sum_{j=0}^{n-1} f_j g_{l-j}$$

FFT&DFT高效算法

$$\underbrace{2x^6+3x^5+x^4}_{\text{upper}} + \underbrace{3x^3+3x^2+x+1}_{\text{lower}} \quad \bmod x^4-1$$

$$= (2x^2+3x+1)(x^4-1) + \text{lower} + \underbrace{(2x^2+3x+1)}_{\text{add back}}$$

$$= \underbrace{3x^3+3x^2+x+1}_{\text{lower}} + \underbrace{(2x^2+3x+1)}_{\text{upper}} \quad \bmod (x^4-1)$$

$$= 3x^3+5x^2+4x+2 \quad \bmod (x^4-1)$$

FFT&DFT高效算法

$*_n$ is cyclic convolution.

Equivalent to polynomial multiplication in $\mathbb{F}_q[x] / \langle x^n - 1 \rangle$

$$f * g \equiv fg \mod x^n - 1$$

If $\deg(fg) < n$, then $f * g = fg \mod (x^n - 1)$

implies $fg = f * g$

so need 2^kth root funct

w/ $2^{k-1} < 2n \leq 2^k$

FFT&DFT高效算法

So we want to find a $kn \log n$ and show DFT & inverse is $O(n \log n)$

(poly mult will be $18 n \log n + O(n)$)

Vandermonde matrix

$$V_\omega = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^{2(n-1)} \\ \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} \cdots & \omega^{(n-1)^2} \end{pmatrix} = (\omega^{jk})$$

$0 \le j, k < n$

is the matrix of the linear multipoint evaluation map coeff → evals at $1, \omega, \ldots$

FFT&DFT高效算法

V_ω is invertible, $(V_\omega)^{-1} = \frac{1}{n} V_{\omega^{-1}}$ so everything works both ways

But applying V_ω can be done faster than taking n, n-size dot products.

FFT&DFT高效算法

The idea is as follows

n even (usually $n=2^k$ for some k)

w a primitive n^{th} root of unity

f deg $< n$

To evaluate f at $1, w, w^2, \dots, w^{n-1}$,

we divide f by $x^{n/2}-1$

and $x^{n/2}+1$ with remainder:

obtaining quotients

FFT&DFT高效算法

To evaluate f at $1, \omega, \omega^2, \ldots, \omega^{n-1}$,

we divide f by $x^{n/2} - 1$

and $x^{n/2} + 1$ with remainder:

obtaining quotients + remainders

write $f = q_0(x^{n/2} - 1) + r_0 = q_1(x^{n/2} + 1) + r_1$

w/ $q_0, r_0, q_1, r_1 \in \mathbb{F}[x]$, $\deg < \frac{n}{2}$

we only need the remainders, and (Ex),

- we can get r_0 by adding the upper $\frac{n}{2}$ coeff of f to the lower $\frac{n}{2}$ coeff

FFT&DFT高效算法

NTT 12

Then plug in a power of w

$$f(w^{2l}) = q_0(w^{2l})(w^{2l}-1) + r_0(w^{2l}) = r_0(w^{2l})$$

and $$f(w^{2l+1}) = q_1(w^{2l+1})(w^{nl}w^{n/2}+1) + r_1(w^{2l+1}) = r_1(w^{2l+1})$$

for $0 \le l < \frac{n}{2}$. (note $w^{nl} = 1$

$w^{n/2} = -1$

since

$0 = w^n - 1 = (w^{n/2}-1)(w^{n/2}+1)$)

Erroneous

FFT&DFT高效算法

So we have all the eval pts

split into even & odd powers of deg ≤ n f

$$f(\omega^{2\ell}) = r_0(\omega^{2\ell})$$

$$f(\omega^{2\ell+1}) = r_1(\omega^{2\ell+1}) = r_1(\omega(\omega^{2\ell}))$$

but, r_0, r_1 have degree $\leq \frac{n}{2}$

- ω^2 is a primitive $(\frac{n}{2})^{th}$ root of unity

So we recurse.

FFT&DFT高效算法

FFT Alg

Input: $n = 2^k$ $\quad f = \sum_{j=0}^{n-1} f_j x^j$

$\omega, \omega^2, \ldots, \omega^{n-1}$ powers of a primitive nth root of unity

Output: $DFT_\omega(f) = (f(1), f(\omega), \ldots, f(\omega^{n-1})) \in \mathbb{F}_q^n$

1. If $n = 1$ return f_0
2. $r_0 \leftarrow \sum_{0 \le j < \frac{n}{2}} (f_j + f_{j+\frac{n}{2}}) x^j$

$r_1^* \leftarrow \sum_{0 \le j < \frac{n}{2}} (f_j - f_{j+\frac{n}{2}}) \omega^j x^j$ $\quad r_1^* = r_1(\omega x)$

3. recurse to evaluate r_0 and r_1^* at powers of ω^2
4. return
   $r_0(1), r_1^*(1), r_0(\omega^2), r_1^*(\omega^2), \ldots, r_0(\omega^{n-2}), r_1^*(\omega^{n-2})$

$n \log n$ additions in $\mathbb{F}$

$\frac{n}{2} \log n$ multiplications by powers of ω

Total $\frac{3}{2} n \log$ field ops.

Prof.Dr.Chern Lu, CFA FRM PRM
luchern@yahoo.com

Thank You