

zk-SNARK从理论到实践

0. zk-SNARK历史

1. 密码学基础

1. 群/环/域/有限域

2. 椭圆曲线计算逻辑

1. 实数域椭圆曲线

2. 有限域椭圆曲线/循环子群

3. 寻找生成元

3. 基于多项式构建零知识证明

4. 各种椭圆曲线总结

5. Groth16算法介绍

1. Groth16算法介绍和推导

2. 什么是Simulation?

6. 深入理解Groth16的计算

1. R1CS和QAP电路

2. Setup/Prove/Verify计算过程

7. libsnark电路搭建实战

1. libsnark源代码分析

2. 如何搭建一个简单的Merkle树电路?

3. 如何看懂libsnark生成的log?

4. 其他开发工具介绍 (bellman/ZoKrates/Circom)

8. zk-SNARK应用介绍

1. Zcash

2. Filecoin (PoREP/PoST)

3. Dex 3.0 (Loopring)

4. Coda (递归证明)

5. Mixer

9. 其他

1. zk-SNARK Trusted Setup

2. zkSTARK, BulletProof

3. DIZK

4. 零知识证明加速 - GPU(cuda)

欢迎关注微信公众号：星想法(sparkbyte)。一起学习，零知识证明等区块链技术。



0 - zk-SNARK历史

论文的命名规则：xxxYY。xxx是作者首字母的组合（只有一个作者，就用该作者的名字），YY是年份后两个数字。Groth16，就是Groth，在2016年发表的文章。

1. 起源：**GMR85**，提出了交互式证明的概念，在交互式证明的模式中，验证者V可以向证明者P提随机问题。V可以验证超出它计算能力以外的问题。GMR85这篇文章，还提出了所谓的“零知识证明概念”，即在交互过程中，P的私密信息不会泄露给V。
2. zkP的可行性：**GMW91**，证明了任何一个NP问题都有对应的零知识证明算法。
3. 效率的考虑：效率包括：a). 交互轮数 b). 证明时间 c). 验证时间及证明大小 d). 是否可以转变成非交互式。
4. 高效的方案：对于一些特定问题，有一些非常高效的零知识证明算法。比如：离散对数问题，离散对数问题的组合，就可以用Sigma协议来解决。
5. Sigma协议：Sigma协议是一种设计思路，基于它可以实现Pedersen commitment的打开证明、多个离散对数的AND关系、离散对数问题的OR。但是，这种方法依然不够通用。换句话说，它的可编程性还是不够。网络上关于Sigma协议的资料很多，最广为人知的就是Ivan Damgård所写的《On Sigma-Protocols》一文。除此之外，还有Benny Pinkas在第九届Bar-Ilan密码学寒期学校上的PPT。
6. zk-SNARKs：从2013年第一代zk-SNARKs技术提出以来，功能更加强大的零知识证明算法涌现，他们的突出特点就是，可编程性极大的提高，可以实现复杂的布尔电路（如SHA256电路、BLAKE2电路），复杂的累加器（merkle tree）。除了可编程性的大幅提升之外，第一代zk-SNARKs技术（以**PGHR13**为代表）和第二代zk-SNARKs技术（以**Groth16**为代表）的突出特点还包括：proof size 很小，证明验证时间短的优点。但是，zk-SNARKs技术也有一个比较大的问题，即它们需要提前生成可信参数，而且针对每个计算问题，需要不同的可信参数。这极大地限制了zk-SNARKs技术的发展。
7. Universal zk-SNARKs 甚至是 Transparent zk-SNARKs：自2018年底起，有数个研究团队在解决zk-SNARKs需要提前生成可信参数（trusted setup）。**Sonic**算法是第一个Universal zk-SNARKs算法，universal的意思是，可信参数只需要生成一次。

Q - 有理数

Z - 整数（包括0）

Z^* - 正整数

N - 非负整数

N^* - 正整数

R - 实数

1 - 密码学基础

1. 群

讲“群”，先讲讲“代数结构”。代数结构是指具有一个及以上运算的非空集合。群是一种只有一个运算的、比较简单的代数结构。所谓的群，是一个集合（有限或者无限）和一个二元操作组成。这个集合满足如下的四个性质：

1. 封闭性：如果 $a, b \in G$ ，则 $ab \in G$ 。
2. 结合律：如果 $a, b, c \in G$ ，则 $(ab)c = a(bc)$ 。

3. 单位元: 集合中存在一个元素 I , 保证 $aI = Ia = a$, 对所有的 $a \in G$ 都成立。

4. 逆元: 对每个集合的元素 $a \in G$, 存在对应的 $b = a^{-1}$, 保证 $ab = ba = I$ 。

有两类特殊的群: 阿贝尔群 (Abelian群) 和 循环群 (Cyclic群)。阿贝尔群满足额外的一个性质:

交换律: 如果 $a, b \in G$, 则 $ab = ba$ 。所以, 阿贝尔群又称为交换群。

循环群中存在一个元素 g (生成元, generator), 群中的所有元素都可以由的生成元幂计算获得。循环群都是阿贝尔群, 但不是所有的阿贝尔群都是循环群。

如果一个群的非空子集合, 在同样的运算下, 也构成一个群, 则该子集合是一个子群。

2. 环

所谓的环, 是一个集合和两个二元操作组成 (加法和乘法)。这个集合满足如下的性质:

1. 加法结合律: 如果 $a, b, c \in S$, 则 $(a + b) + c = a + (b + c)$ 。

2. 加法交换律: 如果 $a, b \in S$, 则 $a + b = b + a$ 。

3. 加法单位元: 存在 $0 \in S$, 保证 $a \in S, 0 + a = a + 0 = a$ 。

4. 加法逆元: 如果 $a \in S$, 则存在 $-a \in S, a + (-a) = (-a) + a = 0$ 。

5. 乘法结合律: 如果 $a, b, c \in S$, 则 $(a * b) * c = a * (b * c)$ 。

6. 分配律: 如果 $a, b, c \in S$, 则 $a * (b + c) = (a * b) + (a * c), (b + c) * a = (b * a) + (c * a)$ 。

可以看出, 环在加法操作下是个阿贝尔群。

2.1 特征数

设 $(R, +, *)$ 是一个环, 若存在自然数 n , 满足 $\forall a \in R$, 存在 $na = 0$, 则这个最小自然数称为 R 的特征数。也就是说, 特征数指的是使得若干个1相加等于0的最少的1的个数。如果对一个环来说, n 不存在, 则称特征数为0。一个显然的结果: 特征数必须是0或者素数。如果特征数不为素数, 则特征数可以分解成 $x \cdot y$ (x, y 都不为0), 也就是 $(x \cdot y)a = x \cdot (ya) = 0$, $ya = 0$ 。从而和 n 是最小的自然数矛盾。

2.2 子环

环的一个非空子集, 如果在加法和乘法上依然是个环, 那么就称这个环是原来的环的子环。对于有理数域 Q , 整数环就是它的一个子环。对于整数环, 所有偶数依然在加法、乘法下构成一个环 (因为任何两个偶数通过加、减、乘得到的还是偶数, 对于加、减、乘是封闭的, 所以依然是一个环), 偶数环是整数环的一个子环。对于n阶实数矩阵环, 其所有的非对角线上的值全为0的n阶矩阵在矩阵加法、矩阵乘法上也构成了原矩阵环的一个子环, 对于a、b两个矩阵, 如果非对角线上为0, 那么无论加法、减法还是乘法, 得到的结果非对角线上都为0。

2.3 理想

理想(ideal)是一种特殊的子环, 在子环的基础上, 理想还要满足如下条件:

如果B是A的一个理想, 那么对于任何 $a \in A, b \in B$, 存在 $ab \in B$ (左理想), 并且 $ba \in B$ (右理想)。注意环中乘法不一定可交换, 所以 ab 和 ba 不同。

很明显, 每个环至少有两个理想: 一个理想是单个0元所组成的环, 因为任何一个元与0元的乘都为0元; 另一个是这个环本身。这两个理想, 称为“平凡理想”(trivial ideal)。

对于整数环, 所有偶数组成的子环是一个理想, 因为任何整数和偶数的乘积还是偶数。

设 R 是一个环, S 是 R 的一个非空子集, R 的所有包含 S 的理想的交是 R 中包含 S 的最小理想, 称之为由 S 生成的理想, 记为 (S) 。如果 S 是个有限集, $(S) = (a_1, a_2, \dots, a_n)$ 。如果 S 中只有一个元素, $(S) = (a)$, 称为主理想。主理想的计算公式对于不同的环不同。对于含有单位元的交换环 R , $(a) = \{ra \mid r \in R\}$ 。举个例子, Z 是含有单位元1的交换环, $(2) = \{\dots -4, -2, 0, 2, 4, \dots\}$ 。

2.4 商环

环 $(R/I, +, \cdot)$ 称为环 R 关于理想 I 的商环 Q , 记为 R/I 。先介绍两个概念: 分划和类。分划是指, 一个非空子集的集合, 并满足所有元素有且只有其中一个非空子集上。比如 $\{1, 2, 3, 4\}$ 可以有如下的分划:

$\{\{1, 2\}, \{3, 4\}\}$

$\{\{1\}, \{2, 3, 4\}\}$

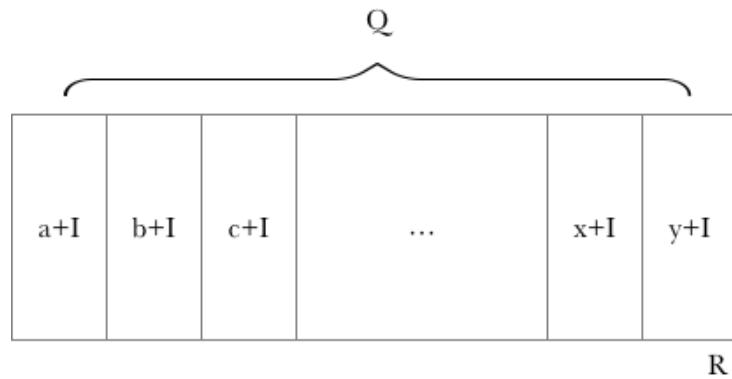
$\{\{1\}, \{2\}, \{3\}, \{4\}\}$

分划中的任意一个非空子集, 称为类。定义商环 Q :

1) 商环 Q 是 R 的一个分划

2) 商环 Q 中的元素 x, y , 满足 $x - y \in I$ 。

也就是说, 商环 Q 是以 I 为“界”的切割后的子环的集合。



Z 是整数环， (n) 是 Z 的理想，商环 $Z/(n)$ 就是模 n 的剩余类环 Z_n 。很显然， Z_n 的特征数为 n 。

3. 除法代数

除法代数，是一个集合和两个二元操作组成（加法和乘法）。这个集合满足如下的性质：

1. 结合律：加法和乘法
2. 交换律：加法
3. 单位元：加法和乘法
4. 逆元：加法和乘法（除零）
5. 结合律

很显然，除法代数是个环。注意，除法代数的乘法不具有交换律。

4. 域和有限域

域，是一个集合。这个集合满足加法和乘法的结合律，交换律，分配律，单位元以及逆元五个性质。有限域是集合中元素有限的域，又称为伽罗瓦域（Galois域）。它是伽罗瓦在18世纪30年代研究代数方程根式求解问题时提出的。可以看出，如果域上不要求乘法的交换律，就是除法代数。在域上存在逆元，也就是说域上支持除法运算。环不是域，举个例子，矩阵环因为不支持乘法的交换律，所以不是域。

域中的所有非零元素的集合是关于乘法的阿贝尔群。**有限域的乘法群是循环群**。有限域中的所有非零元素的集合的每个有限子群都是循环群。 R 是域， Z 是环。

4.1 素域

设 F 是域，则由 F 的单位元 e 生成的 F 的最小子域 P 称为素域。如果 F 的特征数为 $p = 0$ 时， $P \cong Q$ 。如果 F 的特征数为 $p \neq 0$ 时， $P \cong Z/(p)$ 。

4.2 域的特征数

域的特征数只能是0（不存在）或者素数。

2 - 椭圆曲线(Elliptic Curves)

1. 定义

椭圆曲线的数学定义可以查看Wolfram MathWorld：<http://mathworld.wolfram.com/EllipticCurve.html>。不是密码学或者数学专业的小伙伴，看的是一头雾水。便于工程理解，椭圆曲线是一系列满足下方程的点：

$$y^2 = x^3 + ax + b$$

并且 $4a^3 + 27b^2 \neq 0$ 。该方程称为椭圆曲线的Weierstrass方程。

如下是 $b=1$, a 从2到-3的椭圆曲线：



从方程可以看出，椭圆曲线是关于x坐标对称的曲线。除了坐标系上曲线的点，椭圆曲线额外定义一个点（无穷远处），记为 **0**。

也就是说，椭圆曲线是由如下的点组成：

$$\{(x, y) \in \mathbb{R}^2 \mid y^2 = x^3 + ax + b, 4a^3 + 27b^2 \neq 0\} \cup \{0\}$$

2. 基于椭圆曲线的群定义

在椭圆曲线的基础上，可以定义一个加法群：

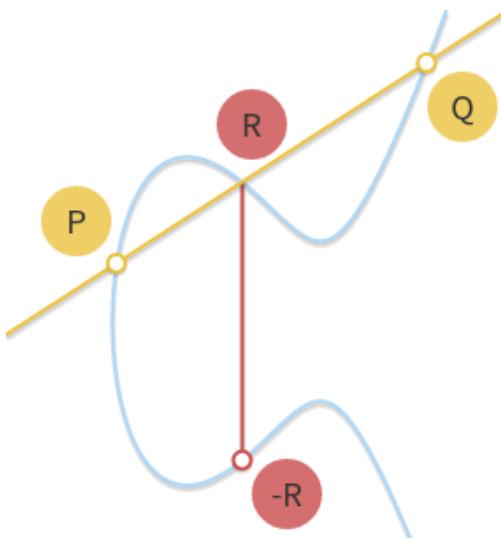
- 所有椭圆曲线上的点，就是这个群里的元素
- 单位元就是0
- 点P的逆元是点P相对x坐标的对称点
- 加法定义如下：在椭圆曲线上，和一条直线相交的3个点P,Q以及R，三点相加满足 $P + Q + R = 0$ 。也就是说，椭圆曲线上的两点相加的结果，还在椭圆曲线上。

结合群的定义，可以证明定义的这个加法群，就是阿贝尔群。

1. 封闭性：因为椭圆曲线上的点相加，还是椭圆曲线上的点。
2. 结合律： $P + (Q + R) = (P + Q) + R = 0$
3. 单位元：单位元是0
4. 逆元：一个椭圆曲线上的点P的逆元，是相对x坐标的对称点
5. 交换律： $P + Q = Q + P$

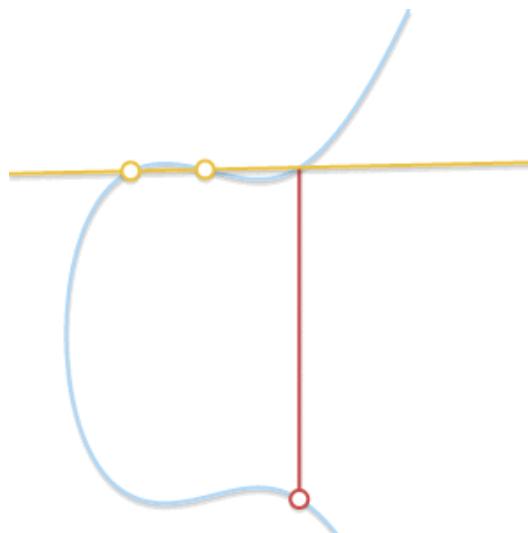
3. 椭圆曲线加法计算

因为 $P + Q + R = 0$ ，也就是说 $P + Q = -R$ 。计算 $P + Q$ 的方法就比较直观了：连接P和Q划一条线，该线和椭圆曲线交的另外一个点为R。 $P + Q$ 的结果就是R的逆。

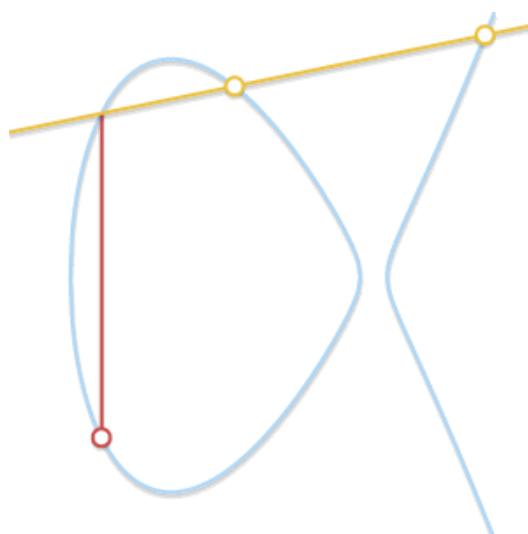


考虑几种特殊情况，对加法计算进行“修正”：

- $P = 0$ 或者 $Q = 0$: 因为定义 0 为无穷远处，不能基于无穷远处划线。但是因为定义了 0 为单位元，所以 $P + 0 = P$ 以及 $0 + Q = Q$ 。
- $P = -Q$: 因为两个点是对称的，所以基于这两个点划的线垂直于 x 轴，不再相交于其他点。 $P + Q = -Q + Q = 0$ 。
- $P = Q$: 如果 P 和 Q 是同一个点的话，那存在多条线穿过这“两个”点。如果把 Q 看作是无限接近 P 的过程，可以看出，穿过 P 和 Q 的是椭圆曲线在 P 点的切线。如果切线和椭圆曲线相交的点为 R，则 $P + P + R = 0$ ， $P + P = 2P = -R$ 。



- $P \neq Q$ ，并且不存在第三个点相交：这种情况和上一种情况有点类似，也就是说，P/Q 的连线是椭圆曲线的切线。如果 P 点是切点， $P + P + Q = 0$ 。也就是说， $P + Q = -P$ 。



4. 加法计算推导

加法的定义是完备的。针对最普通的情况，就是在椭圆曲线上一条直线能穿过三个点，分别是 P, Q, R 。

$P = (x_P, y_P), Q = (x_Q, y_Q), R = (x_R, y_R)$ 。这条直线有个斜率：

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

可以推导出：

$$\begin{aligned} x_R &= m^2 - x_P - x_Q \\ y_R &= y_P + m(x_R - x_P) \end{aligned}$$

或者

$$y_R = y_Q + m(x_R - x_Q)$$

当然，如果P/Q是同一个点的话，斜率的计算公式不同。

5. 标量乘法 (Scalar Multiplication)

在加法的基础上，定义了标量乘法，同一个点相加多次：

$$nP = \underbrace{P + P + \cdots + P}_{n \text{ times}}$$

计算标量乘法，最简单的方法是一个个P点相加。如果n是k位的话，算法复杂度是： $O(2^k)$ 。

有个快速的计算方法：double后相加。假设n=151，二进制表示为：10010111₂。

还是用n=151举个例子：

$$\begin{aligned} 151 &= 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 2^7 + 2^4 + 2^2 + 2^1 + 2^0 \end{aligned}$$

"Double"主要是依次获得某个位对应的变量的结果。如果该位是1，就加到最后的结果中。这种算法的复杂度是： $O(k)$ 。

6. 对数问题

已知n和P， $Q = nP$ 的计算比较容易。但是，在Q和P已知的情况下，求解n非常困难，没有多项式时间求解算法。

有限域上的椭圆曲线

上面介绍的是基于实数的椭圆曲线的点，可以构造一个群。考虑特征数为p的有限域，p为素数。该有限域是由模p的结果组成，记 F_p 。因为有限域中的元素都有逆元，也就是 $x \cdot x^{-1} = 1$ ，则 $x/y = x \cdot y^{-1}$ 。

1. 扩展欧几里得定理

给予二整数 a 与 b，必存在有整数 x 与 y 使得 $ax + by = \gcd(a, b)$ 。gcd(a,b)是最大公约数。

2. 模p运算下的乘法逆

假设元素a，在模p运算下，有逆元x。满足， $a \cdot x = 1 \pmod{p}$ 。也就是说， $a \cdot x + p \cdot y = 1 = a \cdot x + p \cdot y = \gcd(a, p)$ 。

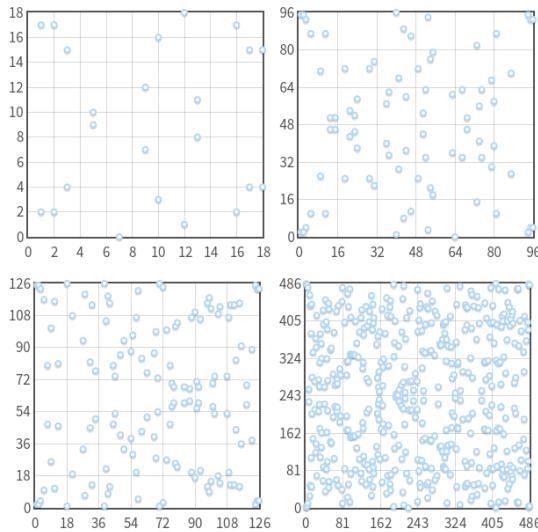
通过扩展欧几里得定理，可以求得x和y。x就是a的乘法逆。

3. 在 F_p 上定义椭圆曲线

在 F_q 上椭圆曲线定义如下：

$$\begin{aligned} \{(x, y) \in (\mathbb{F}_p)^2 \mid y^2 &\equiv x^3 + ax + b \pmod{p}, \\ 4a^3 + 27b^2 &\not\equiv 0 \pmod{p}\} \cup \{0\} \end{aligned}$$

定义和实数上的定义类似。如下是 $y^2 \equiv x^3 - 7x + 10 \pmod{p}$ ，p分别是19, 97, 127, 487对应的椭圆曲线的点。



椭圆曲线是关于 $y = p/2$ 对称，因为

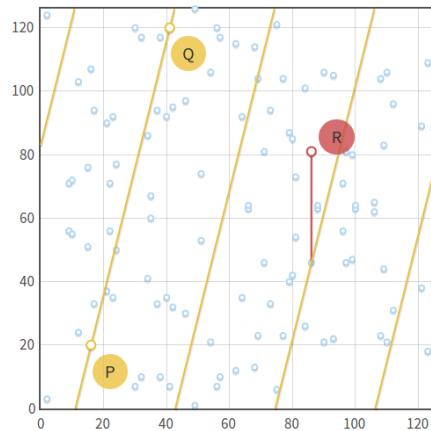
$$(p/2 + \delta)^2 = p^2/4 + p \cdot \delta + \delta^2$$

$$(p/2 - \delta)^2 = p^2/4 - p \cdot \delta + \delta^2$$

在模p的情况下，这两个等式相等。

4. 点加

和实数上椭圆曲线的点加类似，定义在一条“线”上的三点相加等于0： $P + Q + R = 0$ 。在 F_p 有限域上，一条直线定义为： $ax + by + c \equiv 0 \pmod{p}$ 。



上图是 $y^2 \equiv x^3 - x + 3 \pmod{127}$ 的椭圆曲线，其中 $P = (16, 20)$, $Q = (41, 120)$ 。图中的黄色的一系列的斜线是 $y \equiv 4x + 83 \pmod{127}$ 的直线。R就在其中一条斜线上，-R就是图中标出的R的对称点，也就是P+Q的结果。

点加性质：

- $Q + 0 = 0 + Q = Q$
- $-Q = (x_Q, -y_Q \pmod{p})$ ，也就是说，-Q是横坐标相同但纵坐标相反的点，也就是，相对 $p/2$ 对称的点。
- $P + (-P) = 0$

5. 点加计算

假设三个点在一条线上， $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$, $R = (x_R, y_R)$ 。如果P和Q不是同一个点：

$$m = (y_P - y_Q)(x_P - x_Q)^{-1} \pmod{p}$$

从而，推导出：

$$\begin{aligned} x_R &= (m^2 - x_P - x_Q) \pmod{p} \\ y_R &= [y_P + m(x_R - x_P)] \pmod{p} \\ &= [y_Q + m(x_R - x_Q)] \pmod{p} \end{aligned}$$

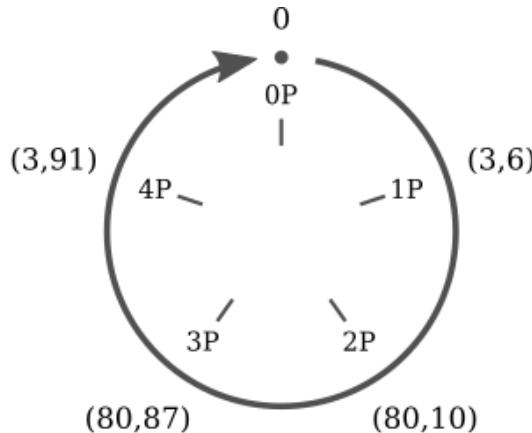
其他条件下的推导，涉及的公式比较多。有兴趣的小伙伴可以自行推导。

6. 在有限群上的椭圆曲线有多少点？

椭圆曲线上的点的个数，称为“阶”。如果枚举 $0 \sim p-1$ ，查看点的个数，不太现实，因为 p 是一个非常大的质数。Schoof算法能在多项式时间确定椭圆曲线阶：https://en.wikipedia.org/wiki/Schoof%27s_algorithm。

7. 标量乘法

和实数域上一样，可以使用double后相加的方法计算。在有限域上，有额外的特性，举个例子：



已知 $y^2 \equiv x^3 + 2x + 3 \pmod{97}$ 以及点 $P = (3, 6)$ 。点P的标量乘法的结果是循环的，只有五个点。

$$0P = 0$$

$$1P = (3, 6)$$

$$2P = (80, 10)$$

$$3P = (80, 87)$$

$$4P = (3, 91)$$

$$5P = 0$$

$$6P = (3, 6)$$

$$7P = (80, 10)$$

$$8P = (80, 87)$$

$$9P = (3, 91)$$

...

$$nP = (n\%6)P$$

很容易看出，在有限域上的椭圆曲线上一个点标量乘法的结果，组成一个在加法操作下的循环子群。在子群中的点，所有的加法的结果都还在子群中。而且，存在一个点，幂次（加法操作）能生成子群中的所有点。这样的点，称为“生成元”。

绕了一大圈，在有限域上的椭圆曲线上，存在很多个循环子群。子群是基于加法操作。

8. 循环子群的阶

Schoof能确定整个基于有限域上的椭圆曲线上的点的个数，但是不能确定循环子群的个数。拉格朗日定理指出，对于任何有限群 G ， G 的每个子群 H 的阶次（元素数）都会被 G 的阶次整除。

[https://en.wikipedia.org/wiki/Lagrange%27s_theorem_\(group_theory\)](https://en.wikipedia.org/wiki/Lagrange%27s_theorem_(group_theory))

该定理给寻找循环子群的阶 n ，提供了一个思路：

1/ 利用Schoof算法，计算出整个椭圆曲线的阶

2/ 找出其所有的约数

3/ 找出最小的约数 n ，满足 $nP = 0$

9. 寻找生成元

通常使用椭圆曲线算法，先选择曲线，计算椭圆曲线的阶，然后在这条曲线上找到最大的子群。找子群，就是寻找子群对应的生成元。

假设椭圆曲线的阶为 N ，子群的阶为 n ，由拉格朗日定理， $h = N/n$ 。

又因为椭圆曲线的阶为 N ， P 为椭圆曲线上的随机的点，存在 $NP = 0$ 。也就是说 $n(hP) = 0$ 。

则 $G = hP$ 为子群的生成元。

10. 离散对数问题

已知两个在子群上的点 P 和 Q , 求解 $Q = kP$ 是非常难的问题。目前该问题没有多项式时间求解算法。

11. 同态

如果子群的阶为 r , 则 $Q = (k \% r)P$ 。

- 同态加法: $(k_1 \% r)P + (k_2 \% r)P = ((k_1 + k_2) \% r)P$

12.

我们在零知识证明中涉及的椭圆曲线, 都是建立在有限域上的椭圆曲线(离散的点)。在该椭圆曲线的基础上, 能构造基于“加法”的循环子群。

3 - 基于多项式构建零知识证明

推荐大家阅读这篇文章: **Why and How zk-SNARK Works: Definitive Explanation**。链接在引用中。

协议0 - 直观版本

随机抽查, 随机提供样本, 对照结果。这种协议, 需要通过设定随机抽查的次数, 确定安全系数。

协议1 - 从多项式的系数证明开始

最简单的d阶多项式, 可以随机选择一个点 x , 让prover通过多项式计算生成 y , verifier可以查看 y 是否正确。不同的多项式, 最多有 d 个相交的点(相等的点)。

如果 x/y 的取值范围很大(设为 n), 在不知道原始多项式的情况下, 能正确给出证明的概率比较低: d/n 。多项式, 能在一次交互, 就能获取比较好的安全系数(如果 n 远大于 d 的话, 将近100%)。比版本0, 优秀不少。

这种协议还是比较简单和原始。证明建立在双方还是相互信任的基础上。本质上, prover并没有证明他/她知道多项式。证明本身并不能推出他/她知道一个确定的多项式。知道一个值, 并不代表知道一个多项式。而且, 如果多项式的值范围不大的话, 证明者可以随机选择一个值撞概率。

协议2 - 基于多项式因式分解改进

假设一个多项式可以分解成: $p(x) = t(x)h(x)$ 。 $t(x)$ 是目标多项式, 是由 $p(x)$ 的部分解组成的多项式:

$t(x) = (x - x_0)(x - x_1) \dots (x - x_{d-1})$ 。也就是说, 证明者需要证明的一部分, 证明者知道一个多项式的部分解。从验证者的角度看, 既然你知道一个多项式的部分解, 那你知道的多项式一定能整除 $t(x)$ 。在随机挑选 x 的情况下, 你都能给出正确的证明, 即可认定你知道这个满足条件的多项式。

- 证明者, 随机生成 r , 并计算出 $t(r)$, 将 r 发送给证明者
- 证明者, 计算 $h(x) = p(x)/t(x)$, 并给出 $p(r)$ 以及 $h(r)$
- 证明者验证, 是否 $p(r) = t(r)h(r)$

该版本要求证明者, 知道一个能整除 $t(x)$ 的多项式。但比较容易发现, 该协议存在如下的问题:

1/ 证明者, 可以自己计算出 $t(r)$, 随机生成 $h(r)$, 并构造出 $p(r) = t(r)h(r)$

2/ 即使不像1, 证明者不考虑多项式, 直接通过结果伪造证明。证明者, 因为知道随机数 r , 证明者可以使用任何一个多项式, 保证存在相同点 $(r, t(r)h(r))$ 。这样, 证明的 $p(r)$ 和 $h(r)$ 虽然和正确的证明是一样的, 但是, 证明者却不需要知道正确的多项式。

3/ 证明者, 可以自己构造多项式, 只要满足 $h(x) = p(x)/t(x)$ 即可。

版本3 - 引入同态加密

协议2中的问题1/2, 都是因为证明者知道随机数 r 以及 $t(r)$ 。引入同态加密解决。数据是加密的, 在加密数据的基础上可以进行计算, 能和原始数据计算具有同样的“形态”。

在模运算基础上的幂次计算, 是同态加密的一种方式, 表示为: $E(v) = g^v \pmod{n}$, 其中 v 是需要加密的数据。在同态加密的基础上, 可以定义运算:

加法: 加密结果相乘, 加密结果相乘等于原始数据相加后的加密结果

乘法: 幂次运算, 一个加密结果进行幂次计算等于两个原始数据乘法的加密结果(注意不是同态乘法)

除法: 相对复杂, 不深究

不要被加法（相乘），乘法（幂次）迷糊了。你可以认为，加密结果加法是通过加密结果的乘法实现的。

注意，乘法不支持两个加密结果乘法。也就是说，只支持一个原始数据和一个同态加密结果进行乘法操作。

有了同态加密，一个多项式的加密结果，可以通过多项式的各个项的加密结果计算。

比如说，一个多项式 $p(x) = x^3 - 3x^2 + 2x$ 。 $p(x)$ 的加密结果可以通过如下的方式计算：

$$E(X^3)^1 \cdot E(x^2)^{-3} \cdot E(x)^2 = (g^{x^3})^1 \cdot (g^{x^2})^{-3} \cdot (g^x)^2 = g^{x^3 - 3x^2 + 2x}$$

显而易见，一个多项式的值，可以通过各项的加密结果乘上各项的系数，计算出多项式的加密结果。

- 验证者，随机挑选 s ，计算出多项式各项的加密结果 $E(s^0), E(s^1), \dots, E(s^d)$ ，并将这些值发给证明者。同时计算出目标多项式的加密值 $t(s)$
- 证明者先计算出 $h(x) = p(x)/t(x)$ ，并通过多项式加密计算的方法计算出 $E(p(s))$ 和 $E(h(s))$
- 验证者验证： $E(p(s)) = E(h(s)) * t(s)$ 是否成立

这个协议，验证者没有暴露随机值 s ，证明者只能通过多项式计算出 h ，从而计算出证明。这个协议，也有问题，证明者有可能只用验证者提供的各项加密结果中的几项来构造证明。并且，聪明的证明者可以自己通过多项式的各项的加密结果计算出 $t(s)$ 。在知道 $t(s)$ 的情况下，要构造一个证明非常容易，只需要满足幂次计算即可。

版本4 - KEA (Knowledge-of-Exponent Assumption)

假设 Alice 想获得 a 的幂次结果，具体的幂次不关心。为了限制幂次结果的提供者 Bob 只在 a 上进行幂次操作，Alice 提供 a 以及 a^α 。Bob 选择一个幂次 c ，计算 $b = a^c \bmod n$ 以及 $b' = (a')^c \bmod n$ ，并发送给 Alice。Alice 通过检查 $b^\alpha = b'$ 确定是否 b 是在 a 基础上的幂次计算。

(a, a^α) 就是“ α 对”。通过一个“ α 对”，数据进行同样的幂次操作。

在上述同态加密的情况下，幂次计算，是同态加密后的乘法操作。举个例子，有个简单的多项式 $f(x) = c \cdot x$ ，验证者为了保证证明者在随机选择的 s 的加密结果上都“乘以” c ，提供 $(g^s, g^{\alpha \cdot s})$ 。证明者，通过提供 $((g^s)^c, (g^{\alpha \cdot s})^c)$ ，证明计算是对同一个加密结果进行了乘法操作。

因为同态加法成立，该方法可以从简单的多项式可以扩展到一半多项式。

- 验证者提供 $g^{s^0}, g^{s^1}, \dots, g^{s^d}$ 以及 $g^{\alpha \cdot s^0}, g^{\alpha \cdot s^1}, \dots, g^{\alpha \cdot s^d}$
- 证明者分别计算两组值：
 - $g^p = g^{p(s)} = (g^{s^0})^{c_0} \cdot (g^{s^1})^{c_1} \cdots \cdot (g^{s^d})^{c_d}$
 - $g^{p'} = g^{\alpha p(s)} = (g^{\alpha s^0})^{c_0} \cdot (g^{\alpha s^1})^{c_1} \cdots \cdot (g^{\alpha s^d})^{c_d}$
- 验证者验证： $(g^p)^\alpha = g^{p'}$

从证明可以推导出： $g^{p'} = g^{c_0 \alpha s^0 + c_1 \alpha s^1 + \dots + c_d \alpha s^d} = g^{\alpha(c_0 s^0 + c_1 s^1 + \dots + c_d s^d)}$ 。

该协议很好的限制证明者必须在多项式的计算下，每个系数都正确“计算”。但是，这个协议并没有保护证明者的知识：验证者可以从证明信息中暴力反推多项式系数（毕竟现实场景下系数的组合还不算多）。

版本5 - ZK (零知识)

为了保护证明信息，也就是保护 $c_0, c_1 \dots c_d$ 信息。还是利用“偏移”，在 g^p 和 $g^{p'}$ 都“乘”上一个随机系数 δ 。即使验证者能枚举出 $(\delta c_0, \delta c_1, \delta c_2 \dots \delta c_d)$ ，也非常难获取 (c_0, c_1, \dots, c_d) 。

版本6 - 无交互

之前所有的协议都是交互式的。交互式协议有些问题：

- 验证者可以和证明者串通，伪造证明
- 验证者，自己生成证明
- 在整个交互过程中，验证者必须保存 α 和 $t(s)$ 。可能会造成其他攻击的可能。

配对函数和双线性映射

到目前为止，验证者需要验证如下的两个等式：

$$g^p = (g^h)^{t(s)}$$

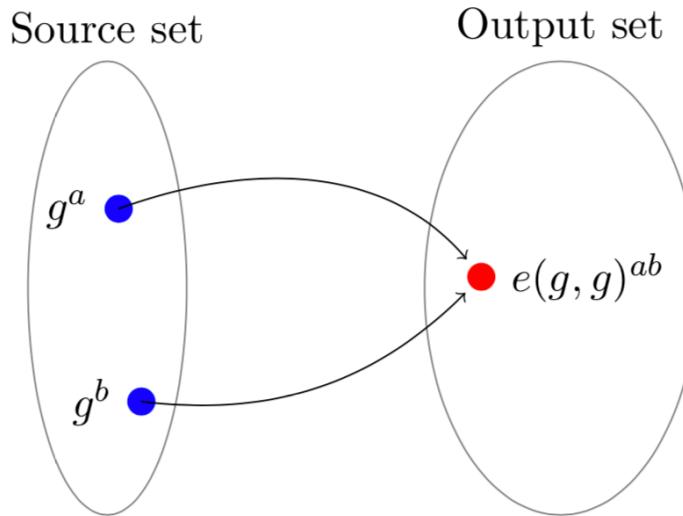
$$(g^p)^\alpha = g^{p'}$$

假设，验证者不存储 α 和 $t(s)$ ，而存储对应的加密数据。这样在验证的时候， g^h 和 $g^{t(s)}$ 两个加密结果“相乘”， g^p 和 g^α “相乘”。从同态加密的定义，我们发现两个加密结果不能相乘。于是引入了新的计算特性：双线性映射。

$$e(g^a, g^b) = e(g, g)^{ab}$$

双线性映射的核心性质，可以表达成如下的等式：

$$e(g^a, g^b) = e(g^b, g^a) = e(g^{ab}, g^1) = e(g^1, g^{ab}) = e(g^1, g^a)^b = e(g^1, g^1)^{ab}$$



值得一提的是， e 配对函数也具有加法同态：

$$e(g^a, g^b) \cdot e(g^c, g^d) = e(g^1, g^1)^{ab} \cdot e(g^1, g^1)^{cd} = e(g^1, g^1)^{ab+cd}$$

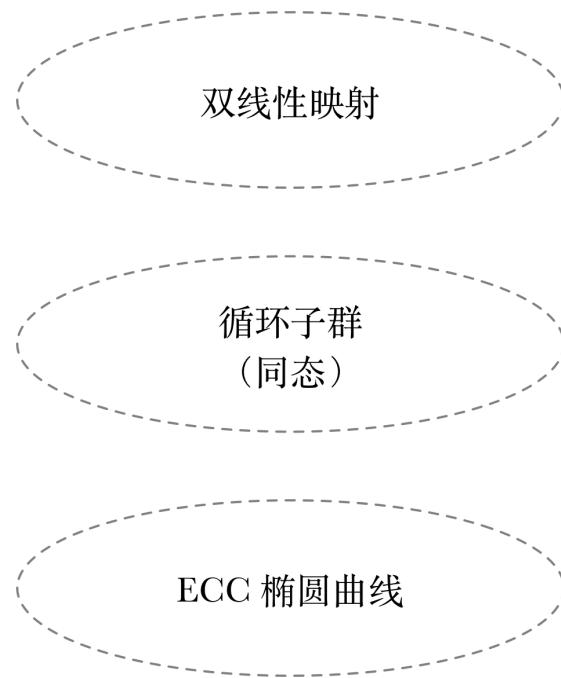
到此，一个zk-SNARK的协议样子就出来了：

- Setup
 - 随机产生 s 和 α
 - 计算生成 g^α 以及 $g^{s^i}, g^{\alpha s^i} i \in d$ ，其中 $(g^{s^i}, g^{\alpha s^i} i \in d)$ 为生成密钥， $(g^\alpha, g^{t(s)})$ 为验证密钥
- Prove
 - 计算 $h(x) = p(x)/t(x)$
 - 利用 g^{s^i} 计算 $g^{p(s)}$ 和 $g^{h(s)}$
 - 利用 $g^{\alpha s^i}$ 计算 $g^{\alpha p(s)}$
 - 随机生成 δ
 - 生成证明 $\pi = (g^{\delta p(s)}, g^{\delta h(s)}, g^{\delta \alpha p(s)})$
- Verification
 - 假设证明 $\pi = (g^p, g^h, g^{p'})$
 - 检查等式 $e(g^{p'}, g) = e(g^p, g^\alpha)$
 - 检查等式 $e(g^p, g) = e(g^{t(s)}, g^h)$

总结：

1. 协议都是建立在多项式基础上。
2. 多项式分解，提供了一个能证明证明者知道一个满足一定条件多项式的方法。为了能让这种方法工作需要其他一些条件：
 1. 同态加密：随机数以及对应多项式的各个项都能加密的同时，还能进行乘法和加法的计算。同态加密的作用是防止证明者能反推随机数，直接伪造证明。同时还能在加密数据的基础上提供证明者需要的计算。
 2. " α 对"保证证明者的计算都是在加密数据乘法和加法的基础。也就是说，证明者的计算是基于多项式的。
 3. 双线性映射，能让两个加密数据映射成原始数据乘积的加密结果。
3. zk的实现反倒简单。在证明数据上进行一个偏移。保证证明者的原始数据不泄漏。
4. Trusted Setup生成初始的参数。

注意，虽然文章中采用的是幂次模运算作为同态加密的示例，其实椭圆曲线子群点运算也是同态加密的一种方式。



扩展到一般计算

如果把一个算子的左右两个输入和输出，都看作多项式的话：

$$l(x) \text{ operator } r(x) = o(x)$$

$$\text{也就是说, } l(x) \text{ operator } r(x) - o(x) = 0$$

可以把 $l(x) \text{ operator } r(x) - o(x)$ 看成一个多项式，类似上述的 $p(x)$ 。通过之前的协议，可以证明证明者知道一个多项式，但是，并没有证明这个多项式的组成方式（不一定具有 $l(x) \text{ operator } r(x) - o(x)$ 的形式）。

通用版本0 - 支持乘法算子

如果这个算子是乘法的话，并且如果随机数为 s 的话， $l(s) \cdot r(s) - o(s) = t(s)h(s)$ 。也就是说，验证者除了需要验证 l , r 以及 o 是多项式外，还需要在加密空间验证等式成立。因为在加密空间做“减”法相对麻烦，需要算逆，可以将等式稍稍变形：

$$l(s) \cdot r(s) = t(s)h(s) + o(s)$$

这样，加密空间就能用配对函数进行验证：

$$e(g^{l(s)}, g^{r(s)}) = e(g^{t(s)}, g^{h(s)}) \cdot e(g^{o(s)}, g)$$

$$e(g, g)^{l(s)r(s)} = e(g, g)^{t(s)h(s)} \cdot e(g, g)^{o(s)}$$

$$e(g, g)^{l(s)r(s)} = e(g, g)^{t(s)h(s)+o(s)}$$

这样，在之前的协议基础上，可以扩展为：

- Prove

- 计算 $h(x) = \frac{l(x)r(x)-o(x)}{t(x)}$
- 利用 g^{s^i} 计算 $g^{l(s)}, g^{r(s)}, g^{o(s)}$ 和 $g^{h(s)}$
- 利用 $g^{\alpha s^i}$ 计算 $g^{\alpha l(s)}, g^{\alpha r(s)}$ 和 $g^{\alpha o(s)}$
- 生成证明 $\pi = (g^{l(s)}, g^{r(s)}, g^{o(s)}, g^{h(s)}, g^{\alpha l(s)}, g^{\alpha r(s)}, g^{\alpha o(s)})$

- Verification

- 假设证明 $\pi = (g^l, g^r, g^o, g^h, g^{l'}, g^{r'}, g^{o'})$
- 检查等式 $e(g^{l'}, g) = e(g^l, g^\alpha)$
- 检查等式 $e(g^{r'}, g) = e(g^r, g^\alpha)$
- 检查等式 $e(g^{o'}, g) = e(g^o, g^\alpha)$
- 检查等式 $e(g^l, g^r) = e(g^{t(s)}, g^h) \cdot e(g^o, g)$

多个乘法，可以采用中间变量，分解成多个乘法操作。比如 $a * b * c$ ，可以分解成：

$$a * b = r_1$$

$$r_1 * c = r_2$$

多个乘法表达式，同样可以表达成 $l(x) \cdot r(x) = o(x)$ 。和前一个例子不同的是，需要选择两个 x ，保证等式成立。可以通过差值计算出 $l(x), r(x)$ 以及 $o(x)$ 。通过通用版本0的协议，可以进行证明（扩展 $t(x)$ ）。

通用版本1 - 多项式 α 对

之前讲的 α 对，都是针对多项式中的某一个项。整个多项式，也可以实现 α 对，保证某个多项式乘以一个系数。

- Setup
 - 随机产生 s 和 α
 - 计算生成 g^α 以及 $g^{l(s)}$ 和 $g^{\alpha l(s)}$
- Prove
 - 如果多项式的系数为 v
 - 计算生成 $(g^{vl(s)}, g^{\alpha vl(s)})$
- Verification
 - 假设证明 $\pi = (g^l, g^L)$
 - 检查等式 $e(g^L, g) = e(g^l, g^\alpha)$

这个协议有个问题，在多项式上做任何偏移，也能让验证等式成立。

$$g^{vl(s)} \cdot g^{v'} = g^{vl(s)+v'}$$

$$g^{\alpha vl(s)} \cdot (g^\alpha)^{v'} = g^{\alpha(vl(s)+v')}$$

$$e(g^{\alpha(vl(s)+v')}, g) = e(g^{vl(s)+v'}, g^\alpha)$$

在多个计算情况下，某个算子的输入可能是由多个变量组成：

$$a \times b = r_1$$

$$a \times c = r_2$$

$$d \times c = r_3$$

可以用如下的多项式来表达左输入：

$$L(x) = al_a(x) + dl_d(x)$$

$l_a(x)$ 在 $x=1, 2$ 的时候为1， $x=2$ 的时候为0。

$l_d(x)$ 在 $x=1, 2$ 的时候为0， $x=2$ 的时候为1。

α 对，不仅对单个多项式有限制作用，对多项式的组合同样有用：

- Setup
 - 随机产生 s 和 α
 - 计算生成 g^α 以及 $g^{l_a(s)}, g^{l_d(s)}$ 和 $g^{\alpha l_a(s)}, g^{\alpha l_d(s)}$
- Prove
 - 计算生成 $g^{L(s)} = g^{al_a(x)+dl_d(x)} = (g^{l_a(s)})^a \cdot (g^{l_d(s)})^d$
 - 计算生成 $g^{\alpha L(s)} = g^{\alpha al_a(x)+\alpha dl_d(x)} = (g^{\alpha l_a(s)})^a \cdot (g^{\alpha l_d(s)})^d$
- Verification
 - 假设证明 $\pi = (g^L, g^{L'})$
 - 检查等式 $e(g^{L'}, g) = e(g^L, g^\alpha)$

也就是说， α 对能保证一种计算结构，加密数据乘上系数的结构。加密数据本身具有加法同态，所以，可以从多项式的一项，扩展为多项式，再扩展为多个多项式。

通用版本2 - 通用计算

计算表达也进一步延伸，每个算子的输入和输出都可以是多个变量的累积。

$$\sum_{i=1}^n c_{l,i} \cdot v_i \times \sum_{i=1}^n c_{r,i} \cdot v_i = \sum_{i=1}^n c_{o,i} \cdot v_i$$

其中， v_i 是各个变量（包括临时变量）。

假设存在d个计算表达式，变量个数是n，相应的系数是： $C_{L,i,j}, C_{R,i,j}, C_{O,i,j}$ $i \in 1...n, j \in 1...d$

- Setup

- 通过插值，计算出 $l_i(x), r_i(x), o_i(x)$, $i \in 1...n$
- 随机产生s和 α
- 计算生成 g^α, g^{s^k} $k \in 1...d$ 以及 $g^{l_i(s)}, g^{r_i(s)}, g^{o_i(s)}$ 和 $g^{\alpha l_i(s)}, g^{\alpha r_i(s)}, g^{\alpha o_i(s)}$

- Prove

- 计算 $h(x) = \frac{L(x) \times R(x) - O(x)}{t(x)}$, 其中 $L(x) = \sum_{i=1}^n v_i \cdot l_i(x)$, $R(x)/O(x)$ 类似
- 计算生成 $g^{L(s)} = \prod_{i=1}^n (g^{l_i(s)})^{v_i}, g^{R(s)} = \prod_{i=1}^n (g^{r_i(s)})^{v_i}, g^{O(s)} = \prod_{i=1}^n (g^{o_i(s)})^{v_i}$
- 计算生成 $g^{\alpha L(s)} = \prod_{i=1}^n (g^{\alpha l_i(s)})^{v_i}, g^{\alpha R(s)} = \prod_{i=1}^n (g^{\alpha r_i(s)})^{v_i}, g^{\alpha O(s)} = \prod_{i=1}^n (g^{\alpha o_i(s)})^{v_i}$
- 计算生成 $g^{h(s)}$

- Verification

- 假设证明 $\pi = (g^L, g^R, g^O, g^{L'}, g^{R'}, g^{O'}, g^h)$
- 检查等式 $e(g^{L'}, g) = e(g^L, g^\alpha), e(g^{R'}, g) = e(g^R, g^\alpha), e(g^{O'}, g) = e(g^O, g^\alpha)$
- 检查等式 $e(g^L, g^R) = e(g^t, g^h) \cdot e(g^O, g)$

通用版本3 - 固定输入和输出

上述的协议因为对l, r, 以及o使用的是同一个 α 值，存在如下的问题：

1. L(x)的计算采用R(x)/O(x)中的多项式： $L'(s) = o_1(s) + r_1(s) + r_5(s) + \dots$
2. 输入/输出，换位： $O(s) \times R(s) = L(s)$ （证明者，提供证明时，只是交换O/L的位置，同样能通过验证）
3. 重用同样的输入： $L(s) \times L(s) = O(s)$ （证明者，提供证明时，用L(s)代替R(s)，同样能通过验证）

解决上述问题的简单的做法，分别对L(x)/R(x)/O(x)使用不同的 α 对。

通用版本4 - 限定变量

细心一点会发现，之前的协议并没有限制证明者使用“一致”的变量的取值。也就是说，证明者，可以在计算L(x), R(x), O(x)的时候，使用不同的 v_i 。

如何限制证明者使用同样的变量值？还是采用“ α 对”的思想，将多项式“累积”在一起，先用统一一个偏移进行限制。

$$e(g^{v_{L,i} \cdot l_i(s)} \cdot g^{v_{R,i} \cdot r_i(s)} \cdot g^{v_{O,i} \cdot o_i(s)}, g^\beta) = e(g^{v_{\beta,i} \cdot \beta(l_i(s) + r_i(s) + o_i(s))}, g)$$

如果 $v_{L,i} = v_{R,i} = v_{O,i} = v_{\beta,i}$ 的话，显然表达式成立。但是，在一些场景下，比如证明者知道 $L(x)=R(x)$ 的情况下，不需要变量相等，等式也能成立。

$$(V_{L,i} \cdot l_i(s) + V_{R,i} \cdot r_i(s) + V_{O,i} \cdot o_i(s)) \cdot \beta = V_{\beta,i} \cdot \beta \cdot (l_i(s) + r_i(s) + o_i(s))$$

假设 $w = l(s) = r(s)$, 并且 $y = o(s)$:

$$\beta(v_L w + v_R w + v_O y) = v_\beta \cdot \beta(w + w + y)$$

证明者，知道这些信息，要让等式成立，可以让 $v_\beta = v_O, v_L = 2v_O - v_R$ 。也就是存在可能性，不同的变量值，验证等式依然成立。

进一步优化验证等式，让每个变量采用不同的偏移，记为 β 对：

- Setup

- 额外随机生成 $\beta_l, \beta_r, \beta_o$
- 额外计算 $\{g^{\beta_l l_i(s) + \beta_r r_i(s) + \beta_o o_i(s)}\}_{i \in \{1...n\}}$

- Prove

- 额外计算 $g^{Z(s)} = \sum_{i=1}^n g^{z_i(s)} = g^{\beta_l L(s) + \beta_r R(s) + \beta_o O(s)}$

- Verification

- 验证等式： $e(g^L, g^{\beta_l}) \cdot e(g^R, g^{\beta_r}) \cdot e(g^O, g^{\beta_o}) = e(g^Z, g)$

因为：

$$L \cdot \beta_l + R \cdot \beta_r + O \cdot \beta_o = \sum_{i=0}^n v_i l_i(s) \beta_l + v_i r_i(s) \beta_r + v_i o_i(s) \beta_o$$

$$= \sum_{i=0}^n v_i (l_i(s) \beta_l + r_i(s) \beta_r + o_i(s) \beta_o)$$

既然，L/R/O采用通用的变量值v，则L+R+O，可以看成合并的一个多项式。为了避免证明者利用L/R/O的关系，伪造证明，使用多个 β 对。

也就是说， α 对限制计算是按照指定的多项式乘加， β 对限制计算采用同样的变量值。

通用版本4 - 不可变形

上述的协议还存在两种变形问题：

1/ 变量的多项式变形： α 对限制了计算是按照指定的多项式乘加。比如说，L是多个 $l_i(x)$ 乘加。但是因为证明者知道 $g^{l_i(x)}$ 和 g^α ，可以从验证者提供的信息构造 $g^{l_i(x)+1}$ 。

2/ 变量取值变形： β 对虽然限制了计算需要是 $l_i(x) + r_i(x) + o_i(x)$ 的形式，但并没有限制在证明数据上同时偏移。

这些变形的问题，因为证明者完全知道 β 的加密结果。解决的办法是：引入 γ ，进一步隐藏 β 的加密结果。

- Setup
 - 额外随机生成 $\beta_l, \beta_r, \beta_o, \gamma$
 - 额外设置验证密钥： $g^{\beta_l\gamma}, g^{\beta_r\gamma}, g^{\beta_o\gamma}, g^\gamma$
- Prove
 - 额外计算 $g^{Z(s)} = \sum_{i=1}^n g^{z_i(s)} = g^{\beta_l\gamma L(s) + \beta_r\gamma R(s) + \beta_o\gamma O(s)}$
- Verification
 - 验证等式： $e(g^L, g^{\beta_l\gamma}) \cdot e(g^R, g^{\beta_r\gamma}) \cdot e(g^O, g^{\beta_o\gamma}) = e(g^Z, g^\gamma)$

通用版本4 - 优化配对计算个数

上一个协议需要4个配对函数的计算。皮诺曹协议（Pinocchio protocol）通过对L/R/O，使用不同的生成元，减少了配对函数的计算。

- Setup
 - 随机生成 $\alpha_l, \alpha_r, \alpha_o, \beta, \gamma, \rho_l, \rho_r$ ，并设置 $\rho_o = \rho_l \cdot \rho_r$
 - 生成三个生成元： $g_l = g^{\rho_l}, g_r = g^{\rho_r}, g_o = g^{\rho_o}$
 - 设置证明密钥： $\{g^{s^k}\}_{k \in d}, \{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}, g_l^{\alpha_l l_i(s)}, g_r^{\alpha_r r_i(s)}, g_o^{\alpha_o o_i(s)}, g_l^{\beta l_i(s)}, g_r^{\beta r_i(s)}, g_o^{\beta o_i(s)}\}$
 - 设置验证密钥： $g_o^{t(s)}, g^{a_l}, g^{a_r}, g^{a_o}, g^{\beta\gamma}, g^\gamma$
- Prove
 - 额外计算 $g^{Z(s)} = \prod_{i=1}^n (g_l^{\beta l_i(s)} \cdot g_r^{\beta r_i(s)} \cdot g_o^{\beta o_i(s)})^{v_i}$
- Verification
 - 验证是否采用指定的多项式： $e(g_l^{L'}, g) = e(g_l^L, g^{\alpha_l})$ ，同样检查R和O
 - 验证是否采用一致的变量值： $e(g_l^L \cdot g_r^R \cdot g_o^O, g^{\beta\gamma}) = e(g^Z, g^\gamma)$
 - 验证计算是否成立： $e(g_l^L, g_r^R) = e(g_o^t, g^h) \cdot e(g_o^O, g)$

$$e(g_l^L, g_r^R) = e(g, g)^{\rho_l \rho_r LR} = e(g_o^t, g^h) \cdot e(g_o^O, g) = e(g, g)^{\rho_l \rho_r th + \rho_l \rho_r O}$$

4 - 各种椭圆曲线总结

1/ bn128/bn254/bn256 - 以太坊预编译智能合约

2/ bls12-381 - Zcash, Filecoin

3/ mnt4/mnt6 - Coda

5 - Groth16算法介绍

<https://mp.weixin.qq.com/s/SguBb5vyAm2Vzht7WK zug>

术语介绍

线性 (Linear) 函数 - 假设函数f满足两个条件: 1. $f(x + y) = f(x) + f(y)$ 2. $f(\alpha x) = \alpha f(x)$, 则称函数f为线性函数。

Affine 函数 - 假设函数g, 能找到一个线性函数f, 满足 $g(x) = f(x) + b$, 则称函数g为Affine函数。也就是, Affine函数是由一个线性函数和偏移构成。

Trapdoor函数 - 假设一个Trapdoor函数f, $x \rightarrow f(x)$ 很容易, 但是 $f(x) \rightarrow x$ 非常难。但是, 如果提供一个secret, $f(x) \rightarrow x$ 也非常容易。

Jens Groth是谁?

Groth是英国伦敦UCL大学的计算机系的教授。伦敦大学学院 (*University College London*) , 简称UCL, 建校于1826年, 位于英国伦敦, 是一所世界著名的顶尖高等学府, 为享有顶级声誉的综合研究型大学, 伦敦大学联盟创始院校, 英国金三角名校, 与剑桥大学、牛津大学、帝国理工、伦敦政经学院并称G5超级精英大学。

<http://www0.cs.ucl.ac.uk/staff/j.groth/>

Groth从2009年开始, 每年发表一篇或者多篇密码学或者零知识证明的文章, 所以你经常会听到Groth09, Groth10等等算法。

NILP

Groth16的论文先引出NILP (non-interactive linear proofs) 的定义:

1. $(\sigma, \tau) \leftarrow Setup(R)$: 设置过程, 生成 $\sigma \in F^m, \tau \in F^n$ 。
2. $\pi \leftarrow Prove(R, \sigma, \phi, \omega)$: 证明过程, 证明过程又分成两步: a. 生成线性关系 $\Pi \leftarrow ProofMatrix(R, \phi, \omega)$, 其中ProofMatrix是个多项式算法。b. 生成证明: $\pi = \Pi\sigma$ 。
3. $0/1 \leftarrow Vfy(R, \sigma, \phi, \pi)$: 验证过程, 验证者使用 (R, ϕ) 生成电路t, 并验证 $t(\sigma, \pi)$ 是否成立。

在NILP定义的基础上, Groth16进一步定义了split NILP, 也就是说, CRS分成两部分 (σ_1, σ_2) , 证明者提交的证明也分成两部分 (π_1, π_2) 。

总的来说, 核心在"Linear"上, 证明者生成的证明和CRS成线性关系。

QAP的NILP

QAP的定义为"Relation": $R = (F, aux, \ell, \{u_i(X), v_i(X), w_i(X)\}_{i=0}^m, t(X))$ 。也就是说, statements为 $(a_1, \dots, a_\ell) \in F^\ell$, witness为 $(a_{l+1}, \dots, a_m) \in F^{m-\ell}$, 并且 $a_0 = 1$ 的情况下, 满足如下的等式:

$$\sum_{i=0}^m a_i u_i(X) \cdot \sum_{i=0}^m a_i v_i(X) = \sum_{i=0}^m a_i w_i(X) + h(X)t(X)$$

$t(X)$ 的阶为n。

1. 设置过程: 随机选取 $\alpha, \beta, \gamma, \delta, x \leftarrow F^*$, 生成 σ, τ 。

$$\tau = (\alpha, \beta, \gamma, \delta, x)$$

$$\sigma = (\alpha, \beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1}, \{\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\}_{i=0}^\ell, \{\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta}\}_{i=\ell+1}^m, \{\frac{x^i t(x)}{\delta}\}_{i=0}^{n-2})$$

2. 证明过程: 随机选择两个参数r和s, 计算 $\pi = \Pi\sigma = (A, B, C)$

$$A = \alpha + \sum_{i=0}^m a_i u_i(x) + r\delta$$

$$B = \beta + \sum_{i=0}^m a_i v_i(x) + s\delta$$

$$C = \frac{\sum_{i=\ell+1}^m a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + h(x)t(x)}{\delta} + As + rB - rs\delta$$

3. 验证过程:

验证过程, 计算如下的等式是否成立:

$$A \cdot B = \alpha \cdot \beta + \frac{\sum_{i=0}^\ell a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x))}{\gamma} \cdot \gamma + C \cdot \delta$$

注意, 设置过程中的x是一个值, 不是代表多项式。在理解证明/验证过程的时候, 必须要明确, A/B/C的计算是和CRS中的参数成线性关系 (NILP的定义)。在明确这一点的基础上, 可以看出 α 和 β 的参数能保证A/B/C的计算采用统一的 a_0, a_1, \dots, a_m 参数。因为 $A \cdot B$ 会包含 $\sum_{i=0}^\ell a_i (\beta u_i(x) + \alpha v_i(x))$ 子项, 要保证 $A \cdot B$ 和C相等, 必须采用统一的 a_0, a_1, \dots, a_m 参数。参数r和s增加随机因子, 保证零知识 (验证者无法从证明中获取有用信息)。参数 γ 和 δ 保证了验证等式的最后两个乘积独立于 α 和 β 的参数。

完备性证明 (Completeness): 完备性证明, 也就是验证等式成立。

$$\begin{aligned}
A \cdot B &= (\alpha + \sum_{i=0}^m a_i u_i(x) + r\delta) \cdot (\beta + \sum_{i=0}^m a_i v_i(x) + s\delta) \\
&= \alpha \cdot \beta + \alpha \cdot \sum_{i=0}^m a_i v_i(x) + \alpha s\delta + \beta \cdot \sum_{i=0}^m a_i u_i(x) + \sum_{i=0}^m a_i u_i(x) \cdot \sum_{i=0}^m a_i v_i(x) + s\delta \cdot \sum_{i=0}^m a_i u_i(x) \\
&\quad + r\delta \beta + r\delta \sum_{i=0}^m a_i v_i(x) + rs\delta^2 \\
&= \alpha \cdot \beta + \sum_{i=0}^\ell a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + \sum_{i=\ell+1}^m a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + h(t) \cdot t(x) \\
&\quad + \alpha s\delta + s\delta \cdot \sum_{i=0}^m a_i u_i(x) + rs\delta^2 \\
&\quad + r\delta \beta + r\delta \sum_{i=0}^m a_i v_i(x) + rs\delta^2 \\
&\quad - rs\delta^2 \\
&= \alpha \cdot \beta + \sum_{i=0}^\ell a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + \sum_{i=\ell+1}^m a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + h(t) \cdot t(x) \\
&\quad + As\delta + rB\delta - rs\delta^2 \\
&= \alpha \cdot \beta + \frac{\sum_{i=0}^\ell a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x))}{\gamma} \cdot \gamma + C \cdot \delta
\end{aligned}$$

可靠性证明 (Soundness):

Groth16算法证明的是**statistical knowledge soundness**，假设证明者提供的证明和CRS成线性关系。也就是说，证明A可以用如下的表达式表达(A和CRS的各个参数成线性关系)：

$$A = A_\alpha \alpha + A_\beta \beta + A_\gamma \gamma + A_\delta \delta + A(x) + \sum_{i=0}^{\ell} A_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} + \sum_{i=\ell+1}^m A_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} + A_h(x) \frac{t(x)}{\delta}$$

同理，B/C都可以写成类似的表达：

$$B = B_\alpha \alpha + B_\beta \beta + B_\gamma \gamma + B_\delta \delta + B(x) + \sum_{i=0}^{\ell} B_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} + \sum_{i=\ell+1}^m B_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} + B_h(x) \frac{t(x)}{\delta}$$

$$C = C_\alpha \alpha + C_\beta \beta + C_\gamma \gamma + C_\delta \delta + C(x) + \sum_{i=0}^{\ell} C_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} + \sum_{i=\ell+1}^m C_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} + C_h(x) \frac{t(x)}{\delta}$$

从Schwartz-Zippel 定理，我们可以把A/B/C看作是 $\alpha, \beta, \gamma, \delta, x$ 的多项式。观察

$A \cdot B = \alpha \cdot \beta + \frac{\sum_{i=0}^{\ell} a_i(\beta u_i(x) + \alpha v_i(x) + w_i(x))}{\gamma} \cdot \gamma + C \cdot \delta$ 这个验证等式，发现一些变量的限制条件：

1) $A_\alpha B_\alpha \alpha^2 = 0$ (等式的右边没有 α^2 因子)

不失一般性，可以假设 $B_\alpha = 0$ 。

$$2) A_\alpha B_\beta + A_\beta B_\alpha = A_\alpha B_\beta = 1 \quad (\text{等式右边} \alpha\beta = 1)$$

不失一般性，可以假设 $A_\alpha = B_\beta = 1$ 。

3) $A_\beta B_\beta = A_\beta = 0$ (等式的右边没有 β^2 因子)

也就是 $A_\beta = 0$ 。

在上述三个约束下，A/B的表达式变成：

$$A = \alpha + A_\gamma \gamma + A_\delta \delta + A(x) + \sum_{i=0}^{\ell} A_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} + \sum_{i=\ell+1}^m A_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} + A_h(x) \frac{t(x)}{\delta}$$

$$B = \beta + B_\gamma \gamma + B_\delta \delta + B(x) + \sum_{i=0}^{\ell} B_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} + \sum_{i=\ell+1}^m B_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} + B_h(x) \frac{t(x)}{\delta}$$

4) 等式的右边没有 $\frac{1}{\delta^2}$

$$(\sum_{i=\ell+1}^m A_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + A_h(x)t(x))(\sum_{i=\ell+1}^m B_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + A_h(x)t(x)) = 0$$

不失一般性, $\sum_{i=\ell+1}^m A_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + A_h(x)t(x) = 0$

5) 等式的右边没有 $\frac{1}{\gamma^2}$

$$(\sum_{i=0}^{\ell} A_i(\beta u_i(x) + \alpha v_i(x) + w_i(x))) (\sum_{i=0}^{\ell} B_i(\beta u_i(x) + \alpha v_i(x) + w_i(x))) = 0$$

不失一般性， $\sum_{i=0}^{\ell} A_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) = 0$ 。

6) 等式的右边没有 $\frac{\alpha}{\gamma}, \frac{\alpha}{\delta}$

$$\alpha \frac{\sum_{i=0}^{\ell} B_i(\beta u_i(x) + \alpha v_i(x) + w_i(x))}{\gamma} = 0$$

$$\alpha \frac{\sum_{i=\ell+1}^m B_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + B_h(x)t(x)}{\delta} = 0$$

所以， $\sum_{i=0}^{\ell} B_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) = 0, \sum_{i=\ell+1}^m B_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + B_h(x)t(x) = 0$ 。

7) 等式的右边没有 $\beta\gamma$ 和 $\alpha\gamma$

$$A_{\gamma}\beta\gamma = 0, B_{\gamma}\alpha\gamma = 0$$

所以， $A_{\gamma} = 0, B_{\gamma} = 0$ 。

在上述七个约束下，A/B的表达式变成：

$$A = \alpha + A_{\delta}\delta + A(x)$$

$$B = \beta + B_{\delta}\delta + B(x)$$

再看验证的等式：

$$A \cdot B = \alpha \cdot \beta + \frac{\sum_{i=0}^{\ell} a_i(\beta u_i(x) + \alpha v_i(x) + w_i(x))}{\gamma} \cdot \gamma + C \cdot \delta$$

$$= \alpha \cdot \beta + \sum_{i=0}^{\ell} a_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + C \cdot \delta$$

$$\text{观察 } C \cdot \delta, \text{ 因为不存在 } \frac{\delta}{\gamma}, \text{ 所以, } \sum_{i=0}^{\ell} C_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} = 0.$$

$$\text{也就是说, } C = C_{\alpha}\alpha + C_{\beta}\beta + C_{\gamma}\gamma + C_{\delta}\delta + C(x) + \sum_{i=\ell+1}^m C_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} + C_h(x) \frac{t(x)}{\delta}.$$

代入验证等式，所以可以推导出：

$$\alpha B(x) = \sum_{i=0}^{\ell} a_i \alpha v_i(x) + \sum_{i=\ell+1}^m C_i \alpha v_i(x),$$

$$\beta A(x) = \sum_{i=0}^{\ell} a_i \beta u_i(x) + \sum_{i=\ell+1}^m C_i \beta u_i(x)$$

如果，假设，对于 $i = \ell + 1, \dots, m$, $a_i = C_i$, 则

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

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

代入A/B，可以获取以下等式：

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

在证明和CRS线性关系下，所有能使验证等式成立的情况下， $(a_{\ell+1}, \dots, a_m) = (C_{\ell+1}, \dots, C_m)$ 等式必须成立。也就是说，能提供正确证明的，肯定知道witness。

QAP的基于配对函数的NIZK QAP的定义为"Relation": $R = (p, G_1, G_2, G_T, e, g, h, \ell, \{u_i(X), v_i(X), w_i(X)\}_{i=0}^m, t(X))$ 。也就是说，在一个域 Z_p 中，statements为 $(a_1, \dots, a_{\ell}) \in Z_p^{\ell}$, witness为 $(a_{\ell+1}, \dots, a_m) \in Z_p^{m-\ell}$, 并且 $a_0 = 1$ 的情况下，满足如下的等式 ($t(X)$ 的阶为n) :

$$\sum_{i=0}^m a_i u_i(X) \cdot \sum_{i=0}^m a_i v_i(X) = \sum_{i=0}^m a_i w_i(X) + h(X)t(X)$$

也就是说，三个有限群 G_1, G_2, G_T ，对应的生成元分别是 $g, h, e(g, h)$ 。为了方便起见，也为了和论文的表达方式一致， G_1 有限群的计算用 $[y]_1 = g^y$ 表示， G_2 有限群的计算用 $[y]_2 = h^y$ 表示。

1. 设置过程：随机选取 $\alpha, \beta, \gamma, \delta, x \leftarrow Z_p^*$, 生成 σ, τ 。

$$\tau = (\alpha, \beta, \gamma, \delta, x)$$

$$\sigma = ([\sigma_1]_1, [\sigma_2]_2)$$

$$\sigma_1 = (\alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, \{\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\}_{i=0}^\ell, \{\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta}\}_{i=\ell+1}^m, \{\frac{x^i t(x)}{\delta}\}_{i=0}^{n-2})$$

$$\sigma_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1})$$

2. 证明过程：随机选择两个参数 r 和 s ，计算 $\pi = \Pi\sigma = ([A]_1, [C]_1, [B]_2)$

$$A = \alpha + \sum_{i=0}^m a_i u_i(x) + r\delta$$

$$B = \beta + \sum_{i=0}^m a_i v_i(x) + s\delta$$

$$C = \frac{\sum_{i=\ell+1}^m a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + h(x)t(x)}{\delta} + As + rB - rs\delta$$

3. 验证过程：验证如下的等式是否成立。

$$[A]_1 \cdot [B]_2 = [\alpha]_1 \cdot [\beta]_2 + \sum_{i=0}^\ell a_i [\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}]_1 \cdot [\gamma]_2 + [C]_1 \cdot [\delta]_2$$

很容易发现，验证过程的等式也可以用4个配对函数表示：

$$e([A]_1, [B]_2) = e([\alpha]_1, [\beta]_2) \cdot e(\sum_{i=0}^\ell a_i [\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}]_1, [\gamma]_2) \cdot e([C]_1, [\delta]_2)$$

证明过程和QAP的NILP的证明过程类似，不再详细展开。

证明元素的最小个数

论文指出zk-SNARK的最少的证明元素是2个。上述的证明方式是需要提供3个证明元素（A/B/C）。论文进一步说明，如果将电路进行一定方式的改造，使用同样的理论，可以降低证明元素为2个，但是，电路的大小会变的很大。

总结：Groth16算法是Jens Groth在2016年发表的算法。该算法的优点是提供的证明元素个数少（只需要3个），验证等式简单，保证完整性和多项式计算能力下的可靠性。Groth16算法论文同时指出，zk-SNARK算法需要的最少的证明元素为2个。目前Groth16算法已经被ZCash, Filecoin等项目使用。

有关Simulation

$\pi \leftarrow \text{Sim}(R, \tau, a_1, \dots, a_\ell)$: Pick $A, B \leftarrow \mathbb{Z}_p$ and compute a simulated proof $\pi = ([A]_1, [C]_1, [B]_2)$ with

$$C = \frac{AB - \alpha\beta - \sum_{i=0}^\ell a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x))}{\delta}.$$

通过Simulation，也就是通过trapdoor和statement，能生成同样的证明。特别注意，通过Simulation生成证明，并不需要witness的信息。也就是说，理论上，存在使用和“witness”完全无关的信息能生成同样的证明。从而证明了，证明本身没有透露任何和“witness”相关的信息。也就是，完美零知识。

6 - 深入理解Groth16算法计算

1. 电路描述

所有的电路描述有个专业的术语：Relation（变量和变量的关系描述）。描述Relation的语言很多：R1CS, QAP, tinyRAM, bacs等等。目前开发，电路一般采用R1CS语言描述。R1CS相对来说，非常直观。A*B=C（A/B/C分别是输入变量的线性组合）。但是，要应用Groth16算法，需要将R1CS描述的电路，转化为QAP描述。两种电路描述语言的转化，称为Reduction。

1.1 R1CS描述

给定M'个变量（第一个变量约定为恒量1），以及N'个约束，所有的R1CS描述可以表示如下：

M' - Variables
 N' - Constraints

$$U * V = W$$

	a_0	a_1	a_2	\dots	$a_{M'}$		a_0	a_1	a_2	\dots	$a_{M'}$		a_0	a_1	a_2	\dots	$a_{M'}$
		U						V						W			
N'		$U_{00} \ U_{01} \ U_{02} \ \dots \ U_{0M'}$					$V_{00} \ V_{01} \ V_{02} \ \dots \ V_{0M'}$						$W_{00} \ W_{01} \ W_{02} \ \dots \ W_{0M'}$				
		$U_{10} \ U_{11} \ U_{12} \ \dots \ U_{1M'}$					$V_{10} \ V_{11} \ V_{12} \ \dots \ V_{1M'}$						$W_{10} \ W_{11} \ W_{12} \ \dots \ W_{1M'}$				
		\dots					\dots						\dots				
		$U_{N'-1,0} \ U_{N'-1,1} \ U_{N'-1,2} \ \dots \ U_{N'-1,M'}$					$V_{N'-1,0} \ V_{N'-1,1} \ V_{N'-1,2} \ \dots \ V_{N'-1,M'}$						$W_{N'-1,0} \ W_{N'-1,1} \ W_{N'-1,2} \ \dots \ W_{N'-1,M'}$				
		$\underbrace{\quad\quad\quad}_{M'+1}$					$\underbrace{\quad\quad\quad}_{M'+1}$						$\underbrace{\quad\quad\quad}_{M'+1}$				

每一行是一个约束。举例，第一行的约束表示的是： $(\sum_{i=0}^{M'} a_i u_{0i}) * (\sum_{i=0}^{M'} a_i v_{0i}) = (\sum_{i=0}^{M'} a_i w_{0i})$

1.2 QAP转化 介绍具体的转化之前，先介绍一个简单的术语，拉格朗日插值以及拉格朗日basis。

给定一系列的x和y的对应关系，通过拉格朗日插值的方式，可以确定多项式： $p(x) = y_0 l_0(x) + y_1 l_1(x) + \dots + y_n l_n(x)$

其中 $l_0(x), l_1(x), \dots, l_n(x)$ 就称为拉格朗日basis，计算公式如下：

$$l_j(x) := \prod_{i=0, i \neq j}^n \frac{x-x_i}{x_j-x_i} = \left(\frac{x-x_0}{x_j-x_0} \right) \left(\frac{x-x_1}{x_j-x_1} \right) \dots \left(\frac{x-x_{j-1}}{x_j-x_{j-1}} \right) \left(\frac{x-x_{j+1}}{x_j-x_{j+1}} \right) \dots \left(\frac{x-x_{n-1}}{x_j-x_{n-1}} \right) \left(\frac{x-x_n}{x_j-x_n} \right)$$

简单的说，在给定一系列的x/y的对应关系后，可以通过拉格朗日插值表示成多项式。在R1CS的表达方式下，U/V/W多项式很自然用拉格朗日basis表示，并不是以多项式的系数表示。在R1CS转化为QAP之前，必须对现有约束进行增强，增加 $a_i * 0 = 0$ 的约束。增加这些约束的原因是为了保证转化后的QAP的各个多项式不线性依赖。

	$U_0(x)$	$U_1(x)$	$U_2(x)$	\dots	$U_{M'}(x)$		$V_0(x)$	$V_1(x)$	$V_2(x)$	\dots	$V_{M'}(x)$		$W_0(x)$	$W_1(x)$	$W_2(x)$	\dots	$W_{M'}(x)$
$N = N' + M' + 1$		$U_{00} \ U_{01} \ U_{02}$	\dots	\dots	$U_{0M'}$		$V_{00} \ V_{01} \ V_{02}$	\dots	\dots	$V_{0M'}$			$W_{00} \ W_{01} \ W_{02}$	\dots	\dots	$W_{0M'}$	
		$U_{10} \ U_{11} \ U_{12}$	\dots	\dots	$U_{1M'}$		$V_{10} \ V_{11} \ V_{12}$	\dots	\dots	$V_{1M'}$			$W_{10} \ W_{11} \ W_{12}$	\dots	\dots	$W_{1M'}$	
		\dots					\dots						\dots				
		$U_{N'-1,0} \ U_{N'-1,1} \ U_{N'-1,2}$	\dots	\dots	$U_{N'-1,M'}$		$V_{N'-1,0} \ V_{N'-1,1} \ V_{N'-1,2}$	\dots	\dots	$V_{N'-1,M'}$			$W_{N'-1,0} \ W_{N'-1,1} \ W_{N'-1,2}$	\dots	\dots	$W_{N'-1,M'}$	
		$\underbrace{\quad\quad\quad}_{M'+1}$					$\underbrace{\quad\quad\quad}_{M'+1}$						$\underbrace{\quad\quad\quad}_{M'+1}$				

1.3 domain选择 针对每个变量，已经知道 N 个 y 值。如何选择这些 y 值，对应的 x 值？这个就是domain的选择。选择domain，主要考虑两个计算性能：1) 拉格朗日插值 2) FFT和iFFT。libff提供几种domain：1) Basic Radix-2 2) Extended Radix-2 3) Step Radix-2 4) Arithmetic Sequence 5) Geometric Sequence

选择哪一种domain和输入个数（M）有关。为了配合特定domain的计算，domain的阶（M）会稍稍变大。

确定了domain，也就确定了domain上的一组元素s:

	$U_0(x)$	$U_1(x)$	$U_2(x)$	$U_{M'}(x)$	$V_0(x)$	$V_1(x)$	$V_2(x)$	$V_{M'}(x)$	$W_0(x)$	$W_1(x)$	$W_2(x)$	$W_{M'}(x)$
S_0	U_{00}	U_{01}	U_{02}	\dots	$U_{0M'}$	V_{00}	V_{01}	V_{02}	\dots	$V_{0M'}$		
S_1	U_{10}	U_{11}	U_{12}	\dots	$U_{1M'}$	V_{10}	V_{11}	V_{12}	\dots	$V_{1M'}$		
\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots
$S_{N'-1}$	$U_{N'-1,0}$	$U_{N'-1,1}$	$U_{N'-1,2}$	\dots	$U_{N'-1,M'}$	$V_{N'-1,0}$	$V_{N'-1,1}$	$V_{N'-1,2}$	\dots	$V_{N'-1,M'}$		
	1	1	\dots					0			0	
S_{N-1}			\dots									

2. **Setup** 计算 随机生成 $\alpha, \beta, \gamma, \delta, x \in F_r$ 。注意这里的 x 和上一节中的 x 含义不同，不要混淆。

2.1 拉格朗日插值

已知 x 的情况下，通过 1.2 的公式，先通过 domain 计算拉格朗日 basis。再乘上系数，可以获得 $u_i(x), v_i(x), w_i(x)$ 。这些多项式的阶是 M 。

2.2 计算 x^i 和 $t(x)$

x^i 的计算相对简单，注意幂次计算都是在 F_r 的计算。在 domain 确定后，多项式 t 也确定，从而可以计算出 $t(x)$ 。

2.3 生成 Pk/Vk

按照如下的公式，计算 Pk/Vk。 σ_1 是 G1 上的点， σ_2 是 G2 上的点。

$$\sigma_1 = \left(\alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right\}_{i=0}^{\ell} \right) \quad \sigma_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1})$$

其中， $(\alpha, \{\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\}_{i=0}^{\ell})_1$ $(\beta, \gamma, \delta)_2$ 是 Vk。其他部分是 Pk。可以看出，V_k 的大小取决于公共输入的变量个数，相对来说数量比较小。Pk 的数据量大小和所有的变量个数相关。计算过程，主要由 scalarMul 组成。

3. **Prove** 计算 在 domain 选择后， $U^*V=W$ ，可以变换为如下的多项式方程：

$$\sum_{i=0}^m a_i u_i(X) \cdot \sum_{i=0}^m a_i v_i(X) = \sum_{i=0}^m a_i w_i(X) + h(X)t(X) \quad \text{3.1 } u_i(X), v_i(X), w_i(X) \text{ 多项式系数 通过 iFFT,}$$

在已知 domain 上元素 s 和值对应关系，可以计算出多项式系数。

3.2 $u_i(X), v_i(X), w_i(X)$ 在 coset 的值 已知多项式系数，通过 FFT，计算出多项式在 coset 的值。注意，元素 s 以及对应的 coset 是特殊设计的，便于 FFT/iFFT 的计算，和 domain 的选择有关系。

3.3 $h(X)$ 在 coset 的值 $h(X)$ 多项式的计算公式如下： $h(X) = \frac{\sum_{i=0}^m a_i u_i(X) \cdot \sum_{i=0}^m a_i v_i(X) - \sum_{i=0}^m a_i w_i(X)}{t(X)}$ 代入 3.1/3.2，直接计算出 $h(X)$ 在 coset 的值。

3.4 计算 $h(X)$ 多项式系数 通过 iFFT，获取 $h(X)$ 的多项式系数，阶为 $N-2$ 。

3.5 生成证明 随机选择 $r, s \in F_r$, 在已知 $u_i(x), v_i(x), w_i(x), h(x)$ 的情况下, 通过如下的公式计算证明 A, B, C:

$$A = \alpha + \sum_{i=0}^m a_i u_i(x) + r\delta \quad B = \beta + \sum_{i=0}^m a_i v_i(x) + s\delta$$

其中, A需要计算在

$$C = \frac{\sum_{i=\ell+1}^m a_i (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + h(x)t(x)}{\delta} + As + Br - rs\delta.$$

G1上的点, B需要计算在G1/G2上的点, C需要计算G1上的点。C中的 $\frac{h(x)t(x)}{\delta}$ 计算如下: $\frac{h(x)t(x)}{\delta} = \sum_{i=0}^{N-2} \frac{h_i x^i t(x)}{\delta}$ 很显

然, 生成证明的计算量主要由四个Multiexp组成 (A-1, B-1, C-2), 和变量个数以及约束的个数有关。在一个大型电路中, 生成证明的时间比较长 (秒级, 甚至分钟级)。

4. Verify计算

在已知证明以及V_k的情况下, 通过配对 (pairing) 函数, 很容易计算如下的等式是否成立。计算在毫秒级。

$$[A]_1 \cdot [B]_2 = [\alpha]_1 \cdot [\beta]_2 + \sum_{i=0}^{\ell} a_i \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1 \cdot [\gamma]_2 + [C]_1 \cdot [\delta]_2.$$

总结:

Groth16算法的主要计算量由两部分组成: FFT/iFFT以及MultiExp。在生成证明时, 需要4次iFFT以及三次FFT计算。Setup计算和生成证明时, 需要大量的MultiExp。Verify计算量相对较小。

7 - libsnark电路搭建实战

libsnark源代码分析

<https://mp.weixin.qq.com/s/UHqpfI6ImVwa4HtsiksgJA>

libsnark源代码, 建议想深入零知识证明的小伙伴都读一读。Bellman库主要围绕Groth16算法, libsnark给出了SNARK相关算法的全貌, 各种Relation, Language, Proof System。为了更好的生成R1CS电路, libsnark抽象出protoboard和gadget, 方便开发者快速搭建电路。

本文中使用的libsnark源代码的最后一个commit如下:

```
commit 477c9df07b280e42369f82f89c08416319e24ae
Author: Madars Virza <madars@mit.edu>
Date:   Tue Jun 18 18:43:12 2019 -0400

Document that we also implement the Groth16 proof system.
```

1. 源代码目录

源代码在libsnark目录下:

CMakeLists.txt	gadgetlib1	knowledge_commitment	relations
common	gadgetlib2	reductions	zk_proof_systems

common - 定义和实现了一些通用的数据结构, 例如默克尔树, 稀疏向量等等。

relations - relation描述了“约束”关系。除了我们通常说的R1CS外, 还有很多其他约束的描述语言。

reductions - 各种不同描述语言之间的转化。

knowledge_commit - 在multiexp的基础上, 引入pair的概念, 两个基点一个系数, 计算结果称为一个pair。

zk_proof_systems - 零知识证明中的各种证明系统 (包括Groth16, GM17等等)。

gadgetlib1/gadgetlib2 - 为了更方便的构建R1CS, libsnark抽象出一层gadget。已有的gadget, 可以方便地整合搭建出新的电路。

2. Relation

需要零知识证明的问题都是NP问题。NP问题中有一类问题NPC (NP-complete) 问题。所有的NP问题都可以转化为一个NPC问题。只要有一个NPC问题能多项式时间内解决，所有的NP问题都能多项式时间内解决。描述一个NPC问题，有多种方式。描述NPC问题的方式，称为“language”。Relation指的是一个NPC问题和该问题的解的关系。

libsrank库总结了几种描述语言：

- constraint satisfaction problem类
 - **R1CS** - Rank-1 Constraint System
 - **USCS** - Unitary-Square Constraint System
- circuit satisfaction problem类
 - **BACS** - Bilinear Arithmetic Circuit Satisfiability
 - **TBCS** - Two-input Boolean Circuit Satisfiability
- ram computation类

RAM是Random Access Machine的缩写。libsrank总结了两种RAM计算框架：

- **tinyRAM**
- **fooRAM**
- arithmetic program类
 - **QAP** - Quadratic Arithmetic Program (GGPR13)
 - **SQP** - Square Arithmetic Program (GM17)
 - **SSP** - Square Span Program (DFGK14)

先介绍实现各种语言中需要的“variable” (variable.hpp/variable.tcc)，再详细介绍R1CS以及QAP语言。

2.1 variable

```
template<typename FieldT>
class variable {
public:
    var_index_t index;
    ...
};
```

variable的定义非常简单，描述一个variable，只需要记录一个variable对应的标号就行了。比如对应编号为index的variable，表示的是 $x_{\{index\}}$ 变量。

2.2 linear_term

linear_term描述了一个线性组合中的一项。线性组合中的一项由变量以及对应的系数组成：

```
template<typename FieldT>
class linear_term {
public:
    var_index_t index;
    FieldT coeff;
    ...
};
```

2.3 linear_combination linear_combination描述了一个完整的线性组合。一个linear combination由多个linear term组成：

```
template<typename FieldT>
class linear_combination {
public:
    std::vector<linear_term<FieldT>> terms;
    ...
};
```

2.4 R1CS

R1CS定义在constraint_satisfaction_problems/r1cs/r1cs.hpp。R1CS约束就是满足以下形式的一个表达式：

$\langle A, X \rangle * \langle B, X \rangle = \langle C, X \rangle$

X是所有变量组合的向量，A/B/C是和X等长的向量。 $\langle \rangle$ 代表的是点乘。一个R1CS系统由多个R1CS约束组成。

R1CS约束定义为：

```
template<typename FieldT>
class r1cs_constraint {
public:
    linear_combination<FieldT> a, b, c;
    ...
};
```

一个R1CS约束，可以由a/b/c三个linear_combination表示。一个R1CS系统中的所有变量的赋值，又分成两部分：primary input 和auxiliary input。primary就是"statement"，auxiliary就是"witness"。

```
template<typename FieldT>
using r1cs_primary_input = std::vector<FieldT>;
template<typename FieldT>
using r1cs_auxiliary_input = std::vector<FieldT>;
```

一个R1CS系统，包括多个R1CS约束。当然，每个约束的向量的长度是固定的 (primary input size + auxiliary input size + 1) 。

```
template<typename FieldT>
class r1cs_constraint_system {
public:
    size_t primary_input_size;
    size_t auxiliary_input_size;
    std::vector<r1cs_constraint<FieldT>> constraints;
    ...
}
```

2.5 QAP QAP定义在arithmetic_programs/qap/qap.hpp。libsrank采用的QAP的公式是： $A^*B - C = H^*Z$ 。

```
template<typename FieldT>
class qap_instance {
private:
    size_t num_variables_;
    size_t degree_;
    size_t num_inputs_;
public:
    std::shared_ptr<libfqfft::evaluation_domain<FieldT>> domain;
    std::vector<std::map<size_t, FieldT>> A_in_Lagrange_basis;
    std::vector<std::map<size_t, FieldT>> B_in_Lagrange_basis;
    std::vector<std::map<size_t, FieldT>> C_in_Lagrange_basis;
}
```

num_variables_表示QAP电路的变量的个数。num_inputs_表示QAP电路的"statement"对应变量的个数。degree_表示A/B/C中每个多项式的阶的个数（和电路的门的个数相关）。

domain是计算傅立叶变换/反傅立叶变换的引擎，由libfqfft库实现。

何为Lagrange basis?

Given a table of points:

x	x_0	x_1	x_2	x_3	x_4	\cdots	x_n
y	y_0	y_1	y_2	y_3	y_4	\cdots	y_n

there is a **unique** polynomial $p(x) = \sum_{j=0}^n y_j \ell_j(x) = y_0 \ell_0(x) + y_1 \ell_1(x) + \cdots + y_{n-1} \ell_{n-1}(x) + y_n \ell_n(x)$, where

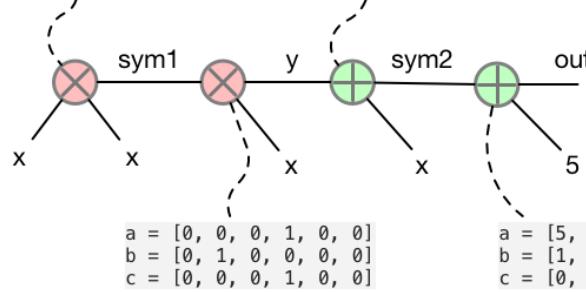
$$\ell_j(x) := \prod_{i=0, i \neq j}^n \frac{x - x_i}{x_j - x_i} = \left(\frac{x - x_0}{x_j - x_0} \right) \left(\frac{x - x_1}{x_j - x_1} \right) \cdots \left(\frac{x - x_{j-1}}{x_j - x_{j-1}} \right) \left(\frac{x - x_{j+1}}{x_j - x_{j+1}} \right) \cdots \left(\frac{x - x_{n-1}}{x_j - x_{n-1}} \right) \left(\frac{x - x_n}{x_j - x_n} \right)$$

A polynomial written in this form uses a **Lagrange basis**: $\{\ell_0, \ell_1, \ell_2, \ell_3, \dots, \ell_{n-1}, \ell_n\}$

给定一系列的x和y的对应关系，通过拉格朗日插值的方式，可以确定多项式： $p(x) = y_{0l_0}(x) + y_{1l_1}(x) + \dots + y_{nl_n}(x)$ 其中 $l_0(x), l_1(x), \dots l_n(x)$ 就称为拉格朗日basis。

A_in_Lagrange_basis/B_in_Lagrange_basis/C_in_Lagrange_basis把一个电路中每个变量不同门的值整理在一起。举个例子，如下

$$\begin{array}{ll} a = [0, 1, 0, 0, 0, 0] & a = [0, 1, 0, 0, 1, 0] \\ b = [0, 1, 0, 0, 0, 0] & b = [1, 0, 0, 0, 0, 0] \\ c = [0, 0, 0, 1, 0, 0] & c = [0, 0, 0, 0, 0, 1] \end{array}$$



是 x^3+x+5 的电路对应的R1CS的约束：

该

电路对应的A_in_Lagrange_basis/B_in_Lagrange_basis/C_in_Lagrange_basis为：

A	B	C
$\begin{bmatrix} 0, 1, 0, 0, 0, 0 \\ 0, 0, 1, 0, 0, 0 \\ 0, 1, 0, 1, 0, 0 \\ 5, 0, 0, 0, 0, 1 \end{bmatrix}$	$\begin{bmatrix} 0, 1, 0, 0, 0, 0 \\ 0, 1, 0, 0, 0, 0 \\ 1, 0, 0, 0, 0, 0 \\ 1, 0, 0, 0, 0, 0 \end{bmatrix}$	$\begin{bmatrix} 0, 0, 1, 0, 0, 0 \\ 0, 0, 0, 1, 0, 0 \\ 0, 0, 0, 0, 0, 1 \\ 0, 0, 1, 0, 0, 0 \end{bmatrix}$

qap_instance描述的是一个QAP电路，A/B/C对应的多项式表达式（虽然是用Lagrange basis表示）。A/B/C多项式在一个点上的结果，用qap_instance_evaluation表示：

```
template<typename FieldT>
class qap_instance_evaluation {
private:
    size_t num_variables_;
    size_t degree_;
    size_t num_inputs_;
public:
    std::shared_ptr<libfqfft::evaluation_domain<FieldT>> domain;
    FieldT t;
    std::vector<FieldT> At, Bt, Ct, Ht;
    FieldT Zt;
    ...
}
```

qap_instance_evaluation，记录了在t点上，A/B/C/H以及Z对应的值。

一个QAP电路，对应的primary/auxiliary，称为witness，定义为：

```

template<typename FieldT>
class qap_witness {
private:
    size_t num_variables_;
    size_t degree_;
    size_t num_inputs_;
public:
    FieldT d1, d2, d3;
    std::vector<FieldT> coefficients_for_ABCs;
    std::vector<FieldT> coefficients_for_H;
    ...
}

```

coefficients_for_ABCs就是witness。为了计算的方便，同时给出了对应的H多项式的系数。在给定一个qap_instance_evaluation和一个qap_witness的前提下，可以通过is_satisfied函数确定，是否witness合理：

```

template<typename FieldT>
bool qap_instance_evaluation<FieldT>::is_satisfied(const qap_witness<FieldT> &witness) const
{
    ...
    ans_A = ans_A + libff::inner_product<FieldT>(this->At.begin() + 1,
                                                    this->At.begin() + 1 + this-
    >num_variables(), witness.coefficients_for_ABCs.begin(), witness.coefficients_for_ABCs.begin() + this-
    >num_variables());
    ans_B = ans_B + libff::inner_product<FieldT>(this->Bt.begin() + 1,
                                                    this->Bt.begin() + 1 + this-
    >num_variables(), witness.coefficients_for_ABCs.begin(), witness.coefficients_for_ABCs.begin() + this-
    >num_variables());
    ans_C = ans_C + libff::inner_product<FieldT>(this->Ct.begin() + 1,
                                                    this->Ct.begin() + 1 + this-
    >num_variables(), witness.coefficients_for_ABCs.begin(), witness.coefficients_for_ABCs.begin() + this-
    >num_variables());
    ans_H = ans_H + libff::inner_product<FieldT>(this->Ht.begin(),
                                                    this->Ht.begin() + this->degree() + 1,
                                                    witness.coefficients_for_H.begin(),
                                                    witness.coefficients_for_H.begin() + this-
    >degree() + 1);

    if (ans_A * ans_B - ans_C != ans_H * this->Zt)
    {
        return false;
    }
    ...
}

```

检查一个witness是否正确，就是计算 $wA \cdot wB - wC = wZ$ 是否相等。

3. Reduction

Reduction实现了不同描述语言之间的转化。libsrank给出了如下一系列的转化实现：

- bacs -> r1cs
- r1cs -> qap
- r1cs -> sap
- ram -> r1cs
- tbcs -> uscs
- uscs -> ssp

以r1cs->qap为例，梳理一下Reduction的逻辑。从R1CS到QAP的转化逻辑在reductions/r1cs_to_qap/目录中，定义了三个函数：

3.1 r1cs_to_qap_instance_map

r1cs_to_qap_instance_map函数实现了从一个R1CS实例到QAP instance的转化。转化过程比较简单，就是将系数重新整理的过程（可以查看2.5中的QAP的描述）。

3.2 r1cs_to_qap_instance_map_with_evaluation

r1cs_to_qap_instance_map_with_evaluation函数实现了从一个R1CS实例到qap_instance_evaluation的转化。给定t，计算A/B/C以及H/Z。

3.3 r1cs_to_qap_witness_map

r1cs_to_qap_witness_map函数实现了从一个R1CS实例到qap_witness的转化。

```
template<typename FieldT>
qap_witness<FieldT> r1cs_to_qap_witness_map(const r1cs_constraint_system<FieldT> &cs,
                                                const r1cs_primary_input<FieldT> &primary_input,
                                                const r1cs_auxiliary_input<FieldT> &auxiliary_input,
                                                const FieldT &d1,
                                                const FieldT &d2,
                                                const FieldT &d3)
```

在给定primary/auxiliary input的基础上，计算出witness（A/B/C以及H的系数）。d1/d2/d3在计算H系数提供扩展能力。Groth16计算的时候，d1/d2/d3取值都为0。给定primary/auxiliary input，A/B/C的系数比较简单，就是primary/auxiliary input的简单拼接后的结果。

```
r1cs_variable_assignment<FieldT> full_variable_assignment = primary_input;
full_variable_assignment.insert(full_variable_assignment.end(), auxiliary_input.begin(),
auxiliary_input.end());
```

H多项式系数的计算相对复杂一些，涉及到傅立叶变换/反傅立叶变换。H多项式的计算公式计算如下： $H(z) := (A(z)*B(z)-C(z))/Z(z)$

其中， $A(z) := A_0(z) + w_1 A_1(z) + \dots + w_m A_m(z) + d1 * Z(z)$, $B(z) := B_0(z) + w_1 B_1(z) + \dots + w_m B_m(z) + d2 * Z(z)$, $C(z) := C_0(z) + w_1 C_1(z) + \dots + w_m C_m(z) + d3 * Z(z)$, $Z(z) = (z-\sigma_1)(z-\sigma_2)\dots(z-\sigma_n)$ (m是QAP电路中变量的个数，n是QAP电路门的个数)。特别强调的是，A(z)/B(z)/C(z)是多个多项式的和，并不是每个变量对应的多项式。

1/ 确定A和B的多项式（通过反傅立叶变换）

```
for (size_t i = 0; i < cs.num_constraints(); ++i)
{
    aA[i] += cs.constraints[i].a.evaluate(full_variable_assignment);
    aB[i] += cs.constraints[i].b.evaluate(full_variable_assignment);
}
domain->iFFT(aA);
domain->iFFT(aB);
```

2/ 计算A和B，对应FieldT::multiplicative_generator的计算结果

```
domain->coSetFFT(aA, FieldT::multiplicative_generator);
domain->coSetFFT(aB, FieldT::multiplicative_generator);
```

3/ 确定C的多项式（通过反傅立叶变换）

```
for (size_t i = 0; i < cs.num_constraints(); ++i)
{
    aC[i] += cs.constraints[i].c.evaluate(full_variable_assignment);
}
domain->iFFT(aC);
```

4/ 计算C，对应FieldT::multiplicative_generator的计算结果

```
domain->cosetFFT(aC, FieldT::multiplicative_generator);
```

5/ 计算H, 对应FieldT::multiplicative_generator的计算结果

```
for (size_t i = 0; i < domain->m; ++i)
{
    H_tmp[i] = aA[i]*aB[i];
}
for (size_t i = 0; i < domain->m; ++i)
{
    H_tmp[i] = (H_tmp[i]-aC[i]);
}
domain->divide_by_Z_on_coset(H_tmp);
```

6/ 计算H多项式的系数 (反傅立叶变换)

```
domain->icosetFFT(H_tmp, FieldT::multiplicative_generator);
```

4. ZK Proof System

libsrank提供了四种证明系统：

- **pcd** (Proof-Carrying Data)
- **ppzkadrank** (PreProcessing Zero-Knowledge Succinct Non-interactive ARgument of Knowledge Over Authenticated Data)
- **ppzksnark** (PreProcessing Zero-Knowledge Succinct Non-interactive ARgument of Knowledge)
- **zksnark** (Zero-Knowledge Succinct Non-interactive ARgument of Knowledge)

著名的Groth16算法，属于ppzksnark。因为Groth16在证明之前，需要预处理生成CRS。ppzksnark证明系统又可以细分成好几种：

ibacs_ppzksnark r1cs_gg_ppzksnark r1cs_ppzksnark r1cs_se_ppzksnark ram_ppzksnark tbcs_ppzksnark uscs_ppzksnark

其中：

r1cs_gg_ppzksnark, gg代表General Group, 就是Groth16算法。

r1cs_se_ppzksnark, se代表Simulation Extractable, 就是GM17算法。

r1cs_ppzksnark, 就是PGHR13算法。

以Groth16算法 (r1cs_gg_ppzksnark) 为例，梳理一下相关的逻辑。Groth16算法的相关逻辑在 zk_proof_systems/ppzksnark/r1cs_gg_ppzksnark目录中。

4.1 r1cs_gg_ppzksnark_proving_key

r1cs_gg_ppzksnark_proving_key记录了CRS中在prove过程需要的信息。

```
template<typename ppT>
class r1cs_gg_ppzksnark_proving_key {
public:
    libff::G1<ppT> alpha_g1;
    libff::G1<ppT> beta_g1;
    libff::G2<ppT> beta_g2;
    libff::G1<ppT> delta_g1;
    libff::G2<ppT> delta_g2;

    libff::G1_vector<ppT> A_query;
    knowledge_commitment_vector<libff::G2<ppT>, libff::G1<ppT> > B_query;
    libff::G1_vector<ppT> H_query;
    libff::G1_vector<ppT> L_query;
```

```

r1cs_gg_ppzksnark_constraint_system<ppT> constraint_system;
...
}

```

A_query是A(t)以G1生成元的一系列计算结果。

B_query是B(t)以G1/G2生成元的一系列计算结果。

H_query是如下的计算以G1位生成元的一系列计算结果：

$H(t) \cdot Z(t) / \delta$

L_query是如下的计算在G1位生成元的一系列计算结果：

$(\beta * A(t) + \alpha * B(t) + C(t)) / \delta$

也就是说，`r1cs_gg_ppzksnark_proving_key`给出了Groth16算法中CRS的如下信息（用蓝色圈出）

$(\sigma, \tau) \leftarrow \text{Setup}(R)$: Pick $\alpha, \beta, \gamma, \delta, x \leftarrow \mathbb{Z}_p^*$. Define $\tau = (\alpha, \beta, \gamma, \delta, x)$ and compute $\sigma = ([\sigma_1]_1, [\sigma_2]_2)$, where

$$\sigma_1 = \left(\begin{array}{l} \alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right\}_{i=0}^{\ell} \\ \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} \right\}_{i=\ell+1}^m, \left\{ \frac{x^i t(x)}{\delta} \right\}_{i=0}^{n-2} \end{array} \right) \quad \sigma_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1}).$$

`r1cs_gg_ppzksnark_constraint_system`定义在

`zk_proof_systems/ppzksnark/r1cs_gg_ppzksnark/r1cs_gg_ppzksnark_params.hpp`文件中。

```

template<typename ppT>
using r1cs_gg_ppzksnark_constraint_system = r1cs_constraint_system<libff::Fr<ppT>>;

```

也就是说，`r1cs_gg_ppzksnark_constraint_system`就是`r1cs_constraint_system`。

4.2 r1cs_gg_ppzksnark_verification_key

`r1cs_gg_ppzksnark_verification_key`记录了CRS中在verify过程需要的信息。

```

template<typename ppT>
class r1cs_gg_ppzksnark_verification_key {
public:
    libff::GT<ppT> alpha_g1_beta_g2;
    libff::G2<ppT> gamma_g2;
    libff::G2<ppT> delta_g2;

    accumulation_vector<libff::G1<ppT>> gamma_ABC_g1;
    ...
}

```

也就是说，`r1cs_gg_ppzksnark_verification_key`给出了Groth16算法中CRS的如下信息（用蓝色圈出）

$(\sigma, \tau) \leftarrow \text{Setup}(R)$: Pick $\alpha, \beta, \gamma, \delta, x \leftarrow \mathbb{Z}_p^*$. Define $\tau = (\alpha, \beta, \gamma, \delta, x)$ and compute $\sigma = ([\sigma_1]_1, [\sigma_2]_2)$, where

$$\sigma_1 = \left(\begin{array}{l} \alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right\}_{i=0}^{\ell} \\ \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} \right\}_{i=\ell+1}^m, \left\{ \frac{x^i t(x)}{\delta} \right\}_{i=0}^{n-2} \end{array} \right) \quad \sigma_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1}).$$

4.3 r1cs_gg_ppzksnark_processed_verification_key

r1cs_gg_ppzksnark_processed_verification_key和r1cs_gg_ppzksnark_verification_key类似。“processed”意味着verification key会做进一步处理，验证的过程会更快。

```
template<typename ppT>
class r1cs_gg_ppzksnark_processed_verification_key {
public:
    libff::GT<ppT> vk_alpha_g1_beta_g2;
    libff::G2_precomp<ppT> vk_gamma_g2_precomp;
    libff::G2_precomp<ppT> vk_delta_g2_precomp;

    accumulation_vector<libff::G1<ppT>> gamma_ABC_g1;
    ...
}
```

4.4 r1cs_gg_ppzksnark_keypair

r1cs_gg_ppzksnark_keypair包括两部分：r1cs_gg_ppzksnark_proving_key和r1cs_gg_ppzksnark_verification_key。

```
template<typename ppT>
class r1cs_gg_ppzksnark_keypair {
public:
    r1cs_gg_ppzksnark_proving_key<ppT> pk;
    r1cs_gg_ppzksnark_verification_key<ppT> vk;
    ...
}
```

4.5 r1cs_gg_ppzksnark_proof

众所周知，Groth16的算法的证明包括A/B/C三个结果。

```
template<typename ppT>
class r1cs_gg_ppzksnark_proof {
public:
    libff::G1<ppT> g_A;
    libff::G2<ppT> g_B;
    libff::G1<ppT> g_C;
    ...
}
```

4.6 r1cs_gg_ppzksnark_generator 给定一个r1cs_constraint_system的基础上，r1cs_gg_ppzksnark_generator能生成r1cs_gg_ppzksnark_keypair，也就是生成CRS信息。

```
template<typename ppT>
r1cs_gg_ppzksnark_keypair<ppT> r1cs_gg_ppzksnark_generator(const
r1cs_gg_ppzksnark_constraint_system<ppT> &cs);
```

4.7 r1cs_gg_ppzksnark_prover

给定了proving key以及primary/auxiliary input，计算证明的A/B/C结果。

```
template<typename ppT>
r1cs_gg_ppzksnark_proof<ppT> r1cs_gg_ppzksnark_prover(const r1cs_gg_ppzksnark_proving_key<ppT> &pk,
                                                               const r1cs_gg_ppzksnark_primary_input<ppT>
&primary_input,
                                                               const r1cs_gg_ppzksnark_auxiliary_input<ppT>
&auxiliary_input);
```

已知proving key的情况下，计算A/B/C的结果，相当简单：

```
/* A = alpha + sum_i(a_i*A_i(t)) + r*delta */
libff::G1<ppT> g1_A = pk.alpha_g1 + evaluation_At + r * pk.delta_g1;
/* B = beta + sum_i(a_i*B_i(t)) + s*delta */
libff::G1<ppT> g1_B = pk.beta_g1 + evaluation_Bt.h + s * pk.delta_g1;
libff::G2<ppT> g2_B = pk.beta_g2 + evaluation_Bt.g + s * pk.delta_g2;
/* C = sum_i(a_i*((beta*A_i(t) + alpha*B_i(t) + C_i(t)) + H(t)*Z(t))/delta) + A*s + r*b - r*s*delta */
*/
libff::G1<ppT> g1_C = evaluation_Ht + evaluation_Lt + s * g1_A + r * g1_B - (r * s) * pk.delta_g1;
```

4.8 r1cs_gg_ppzksnark_verifier

总共提供了四种验证函数，分成两类：processed/non-processed 和 weak/strong IC。processed/non-processed是指验证的key是否processed? weak/strong IC指的是，是否input consistency? Primary Input的大小和QAP的statement的大小相等，称为strong IC。Primary Input的大小小于QAP的statement的大小，称为weak IC。

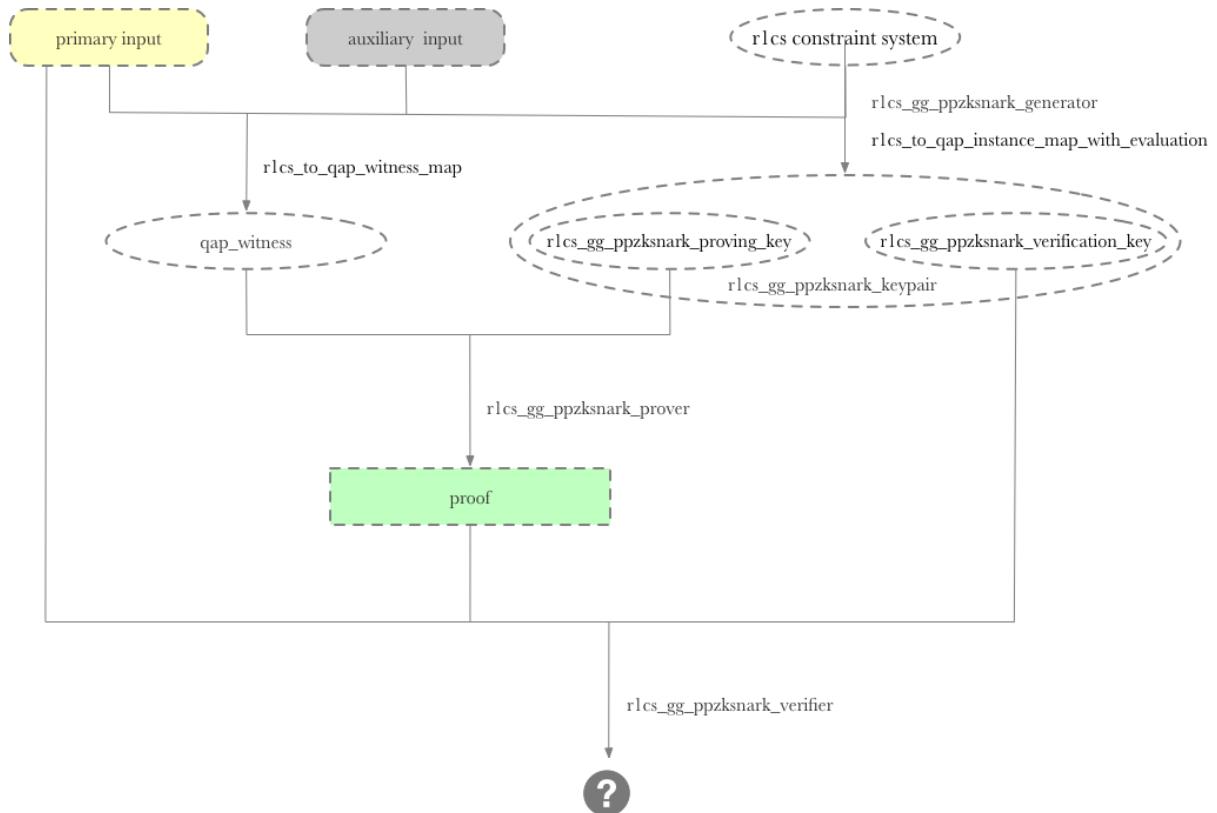
以r1cs_gg_ppzksnark_verifier_strong_IC为例，在给定verification key/primary input的基础上，可以验证proof是否正确。

```
template<typename ppT>
bool r1cs_gg_ppzksnark_verifier_strong_IC(const r1cs_gg_ppzksnark_verification_key<ppT> &vk, const
r1cs_gg_ppzksnark_primary_input<ppT> &primary_input, const r1cs_gg_ppzksnark_proof<ppT> &proof);
```

$0/1 \leftarrow \text{Vfy}(R, \sigma, a_1, \dots, a_\ell, \pi)$: Parse $\pi = ([A]_1, [C]_1, [B]_2) \in \mathbb{G}_1^2 \times \mathbb{G}_2$. Accept the proof if and only if

$$[A]_1 \cdot [B]_2 = [\alpha]_1 \cdot [\beta]_2 + \sum_{i=0}^{\ell} a_i \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1 \cdot [\gamma]_2 + [C]_1 \cdot [\delta]_2.$$

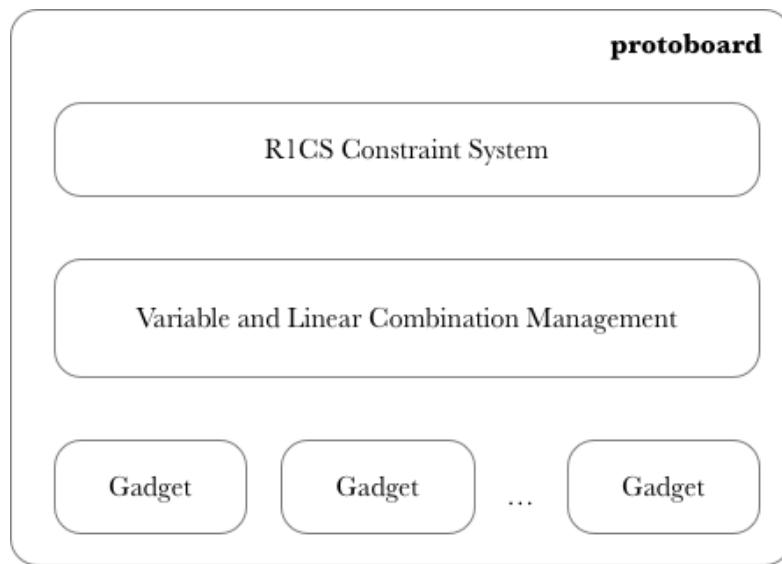
总的来说，Groth16的证明生成和验证的逻辑如下图：



可以看出，使用ZKSNARK(Groth16)证明，需要先创建一个r1cs_constraint_system。libsнark设计了Gadget的框架，方便搭建r1cs_constraint_system。

5. Gadget

libsrank提供了两套Gadget库：gadgetlib1和gadgetlib2。libsrank中很多示例是基于gadgetlib1搭建。gadgetlib1也提供了更丰富的gadget。本文也主要讲解gadgetlib1的逻辑。gadgetlib1的相关代码在libsrank/gadgetlib1目录中。



5.1 protoboard

protoboard是r1cs_constraint_system之上的一层封装。通过一个个的Gadget，向r1cs_constraint_system添加约束。为了让不同的Gadget之间采用统一的Variable以及Lc，protoboard通过“next_free_var”以及“next_free_lc”维护所有Gadget创建的Variable以及Lc。

```
template<typename FieldT>
class protoboard {

    ...
    FieldT constant_term;

    r1cs_variable_assignment<FieldT> values;
    var_index_t next_free_var;
    lc_index_t next_free_lc;
    std::vector<FieldT> lc_values;

    r1cs_constraint_system<FieldT> constraint_system;
    ...
}
```

5.2 pb_variable

libsrank提供了在pb_variable, pb_variable_array, pb_linear_combination和pb_linear_combination_array四个类。这四个类都是variable, linear_combination的封装，为了支持protoboard的管理。

5.3 gadget

gadget的定义非常的简单：

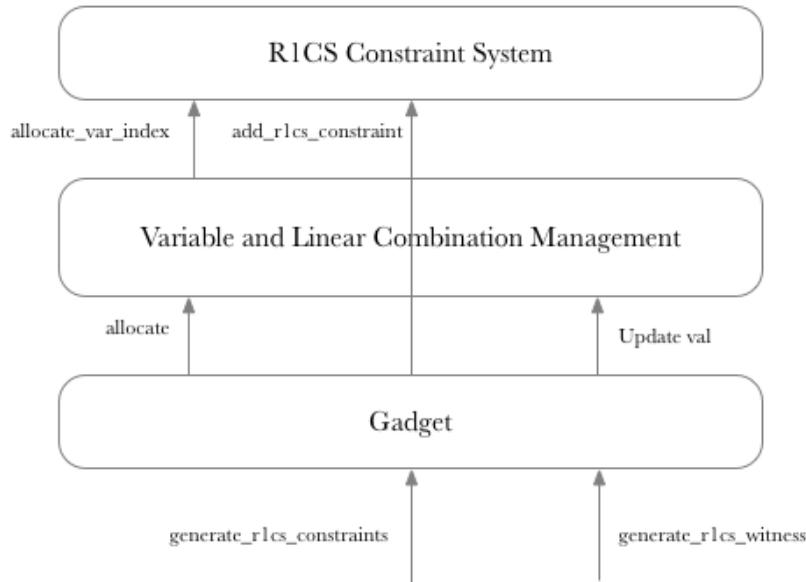
```

template<typename FieldT>
class gadget {
protected:
    protoboard<FieldT> &pb;
    const std::string annotation_prefix;
public:
    gadget(protoboard<FieldT> &pb, const std::string &annotation_prefix = "");
};

```

每一个具体的Gadget逻辑上需要做如下一些事情：

- 1/ 申请Variable或者Lc (allocate)
- 2/ 添加Gadget逻辑相关的约束 (generate_r1cs_constraints)
- 3/ 生成相关的Witness (generate_r1cs_witness)



5.4 example

在gadgetlib1/gadgets目录下有很多Gadget的实现：椭圆曲线计算，各种hash算法，merkle树的计算，配对函数等等。本文以basic_gadget中的两数比较（comparison gadget）为例，说明Gadget的基本逻辑。

comparison_gadget的定义在gadgetlib1/gadgets/basic_gadgets.hpp：

```

comparison_gadget(protoboard<FieldT>& pb,
                  const size_t n,
                  const pb_linear_combination<FieldT> &A,
                  const pb_linear_combination<FieldT> &B,
                  const pb_variable<FieldT> &less,

                  const pb_variable<FieldT> &less_or_eq,

                  const std::string &annotation_prefix = "") :

    gadget<FieldT>(pb, annotation_prefix), n(n), A(A), B(B), less(less), less_or_eq(less_or_eq)
{
    alpha.allocate(pb, n, FMT(this->annotation_prefix, "alpha"));

    alpha.emplace_back(less_or_eq); // alpha[n] is less_or_eq
    alpha_packed.allocate(pb, FMT(this->annotation_prefix, "alpha_packed"));
    not_all_zeros.allocate(pb, FMT(this->annotation_prefix, "not_all_zeros"));
    pack_alpha.reset(new packing_gadget<FieldT>(pb, alpha, alpha_packed,
FMT(this->annotation_prefix, "pack_alpha")));
}

```

```

    all_zeros_test.reset(new disjunction_gadget<FieldT>(pb,
pb_variable_array<FieldT>(alpha.begin(), alpha.begin() + n), not_all_zeros, FMT(this->annotation_prefix, " all_zeros_test")));
};

}

```

comparison_gadget的构造函数比较清晰：在给定两个n位的数A和B，输出两个变量：less和less_or_eq（A是否小于B？）。构造函数，主要是创建其他变量以及Gadget。alpha是长度为n+1的变量数组，其中alpha[n]是less or eq。alpha是 $2^n + B - A$ 的结果的位的表示。alpha_packed是一个变量，alpha对应的值。也就是说，alpha_packed等于 $2^n + B - A$ 。not_all_zeros是一个变量，表示 $B - A$ 的结果是否全是0。pack_alpha是packing_gadget，将n+1位的alpha打包，计算结果存储在alpha_packed变量中。packing_gadget就是将位表示的数据，变成数值表示。all_zeros_test是disjunction_gadget，确定alpha的n个变量中是否全0。comparison_gadget的generate_r1cs_constraints函数负责添加约束。

```

template<typename FieldT>
void comparison_gadget<FieldT>::generate_r1cs_constraints()
{
    generate_boolean_r1cs_constraint<FieldT>(this->pb, not_all_zeros, FMT(this->annotation_prefix,
" not_all_zeros"));
    pack_alpha->generate_r1cs_constraints(true);

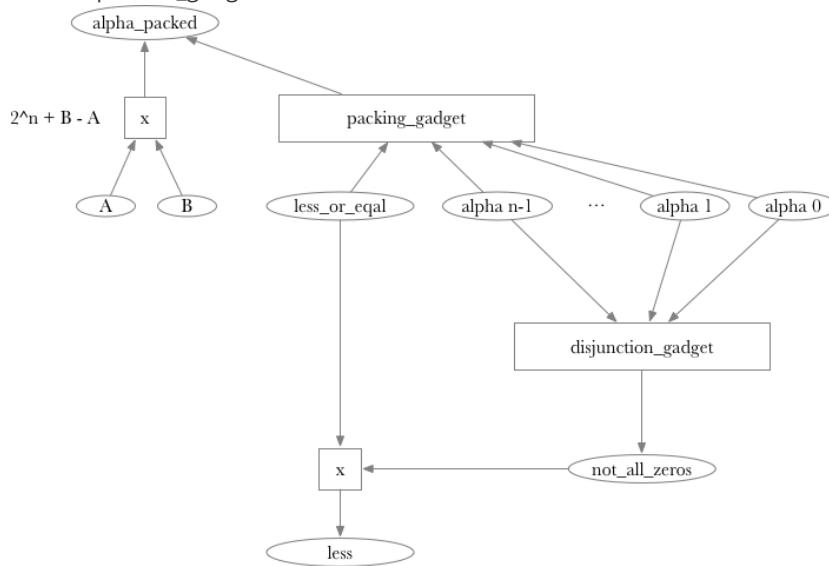
    this->pb.add_r1cs_constraint(r1cs_constraint<FieldT>(1, (FieldT(2)^n) + B - A, alpha_packed),
FMT(this->annotation_prefix, " main_constraint"));
    all_zeros_test->generate_r1cs_constraints();

    this->pb.add_r1cs_constraint(r1cs_constraint<FieldT>(less_or_eq, not_all_zeros, less),
FMT(this->annotation_prefix, " less"));
}

```

a. 对not_all_zeros，添加boolean约束（该变量只能是0或者1） b. pack_alpha->generate_r1cs_constraints(true) 约束alpha对应的数值等于alpha_packed。c. $1 * (2^n + B - A) = \text{alpha_packed}$ d. 确定not_all_zeros变量的值和alpha中n个变量中是否为0的结果一致 e. less_or_eq * not_all_zeros = less

整个comparison_gadget的电路逻辑如下图所示：



comparison_gadget的设计思想是：

如果 $B - A > 0$, 则 $2^n + B - A > 2^n$, $\text{less_or_eq} = 1$, $\text{not_all_zeros} = 1$

如果 $B - A = 0$, 则 $2^n + B - A = 2^n$, $\text{less_or_eq} = 1$, $\text{not_all_zeros} = 0$

如果 $B - A < 0$, 则 $2^n + B - A$ 在 $\{0, 1, \dots, 2^n - 1\}$ 范围内, $\text{less_or_eq} = 0$, $\text{not_all_zeros} = 1$

也就是说, $\text{less_or_eq} * \text{not_all_zeros} = \text{less}$ 。

简单的说，两个数的“大小”比较，是通过 $2^n + B - A$ 的计算结果的相应的一些“符号”位相乘确定。

comparison_gadget的**generate_r1cs_witness**函数生成电路的witness。comparison_gadget的**test_comparison_gadget**函数是comparison gadget的测试函数，相对比较容易理解，小伙伴可以自行查看源码。

讲完libsnnark，不得不提一提：ethsnarks。ethsnarks在libsnnark的基础上，实现了更多的gadget，同时提供了多种语言实现，方便开发调试，相关的源代码导读可以查看：

https://mp.weixin.qq.com/s/4Pe_ULo2bdqWlkCo3VzV1A

除了libsnnark外，还有使用rust语言实现的bellman，相关的源代码导读可以查看：

<https://mp.weixin.qq.com/s/NvX11tNSEpV1DR-3PwpIAQ>

还有个有意思的项目，ZoKrates在libsnnark和bellman的基础上做了一层封装和解析，具体的实现可以查看：

<https://mp.weixin.qq.com/s/08d-5xzM5zNfmtnlSkH0kA>

libsnnark示例

https://github.com/StarLI-Trapdoor/libsnnark_sample.git

8. zk-SNARK应用介绍

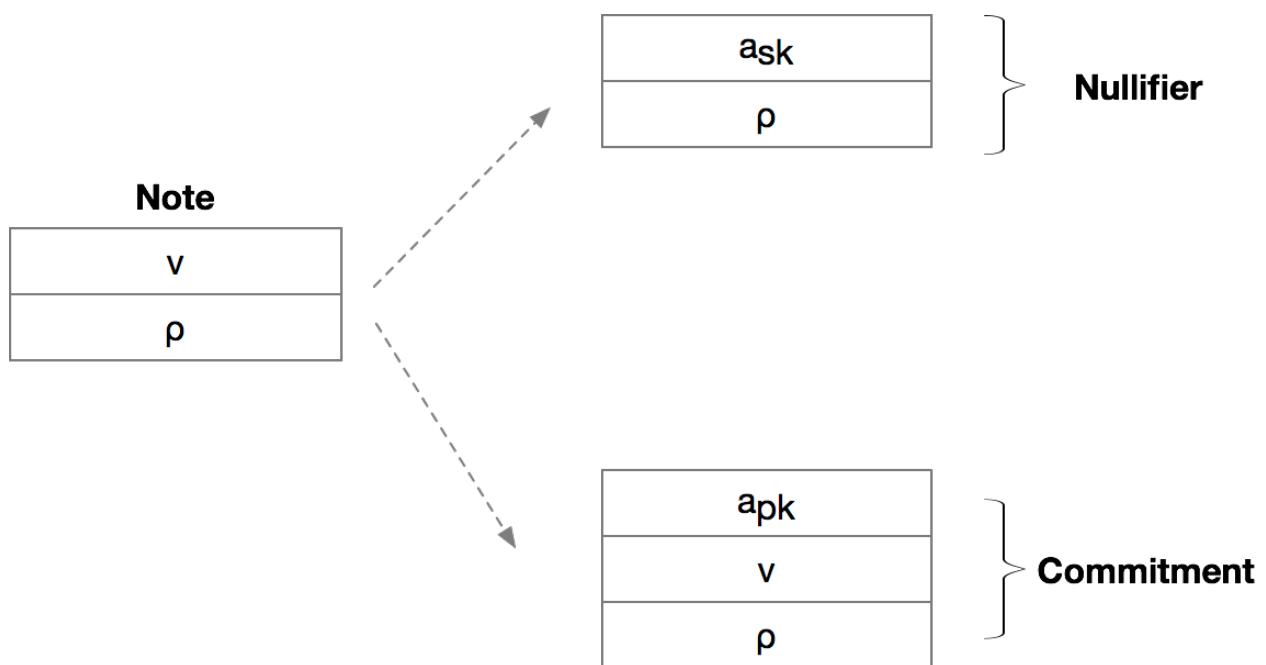
1. Zcash

<https://mp.weixin.qq.com/s/2QjL6UEihZtAM2job7g1tA>

<https://mp.weixin.qq.com/s/WUyVbPH8K1Er7ORU8LING>

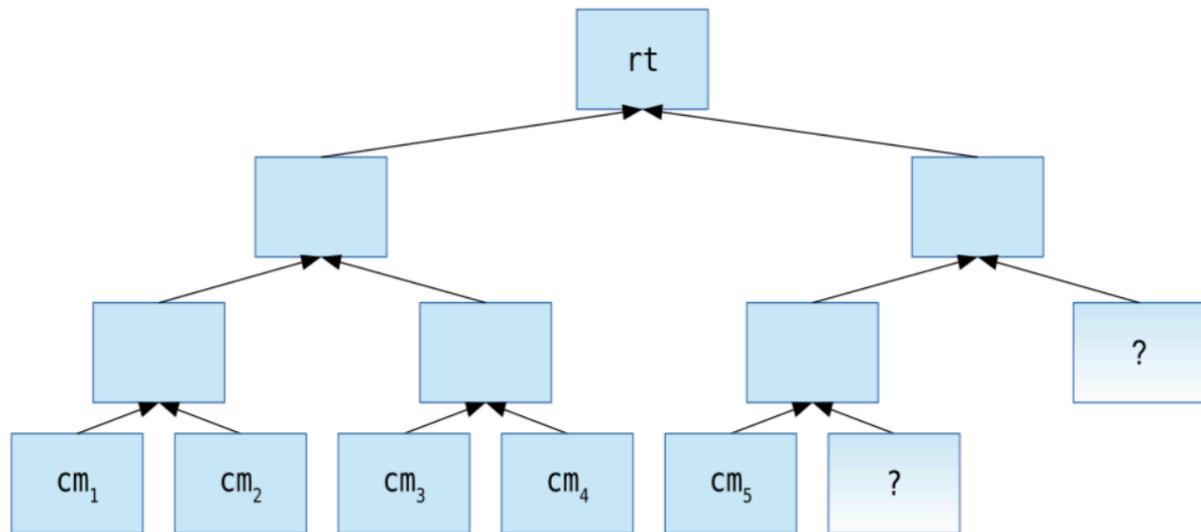
<https://mp.weixin.qq.com/s/7-k7tSiduJOfmByNM0blVA>

Zcash项目，大家都知道是“隐私交易”。Zcash代表了zkSNARK的一个应用方向：隐私。隐私有不同的程度，ZCash的隐私交易指的是隐藏交易的发送方，接收方以及交易金额。Zcash已经经历过三个版本：Overwinter, Sprout, Sapling。这三个版本本质上都没有太大的变化，只是支持的功能更多，生成证明更快，体验更好。



一笔转账用Note来表示，包括转账的金额 v 和一个随机数。 $Note$ 有两个外在的表现形式：一个是Commitment，一个是Nullifier。Commitment和Nullifier都是通过不同的Hash函数生成。Commitment代表一次金额转入，Nullifier代表一次消费。注意，对于一个Note，Commitment和Nullifier都是唯一的。因为Commitment和Nullifier是Hash的结果，即使这两个数据公开，其他人也无法推断出Commitment和Nullifier之间存在联系。也就是说，提供一个Commitment，能说明进行了一笔转账（具体信息其他人未知）。能提供对应的Nullifier，就能消费。

区块链，作为一个隐私转账平台，将所有的Commitment (cm) ，组成一个Merkle树：



某个用户需要消费某个cm，必须向区块链提供零知识证明：

1/ 他知道一个Note，并能生成一个cm，而且这个cm在以rt为树根的Merkle树上

2/ 用同样的Note信息，能生成一个nullifier，而且这个nullifier之前没有生成过。

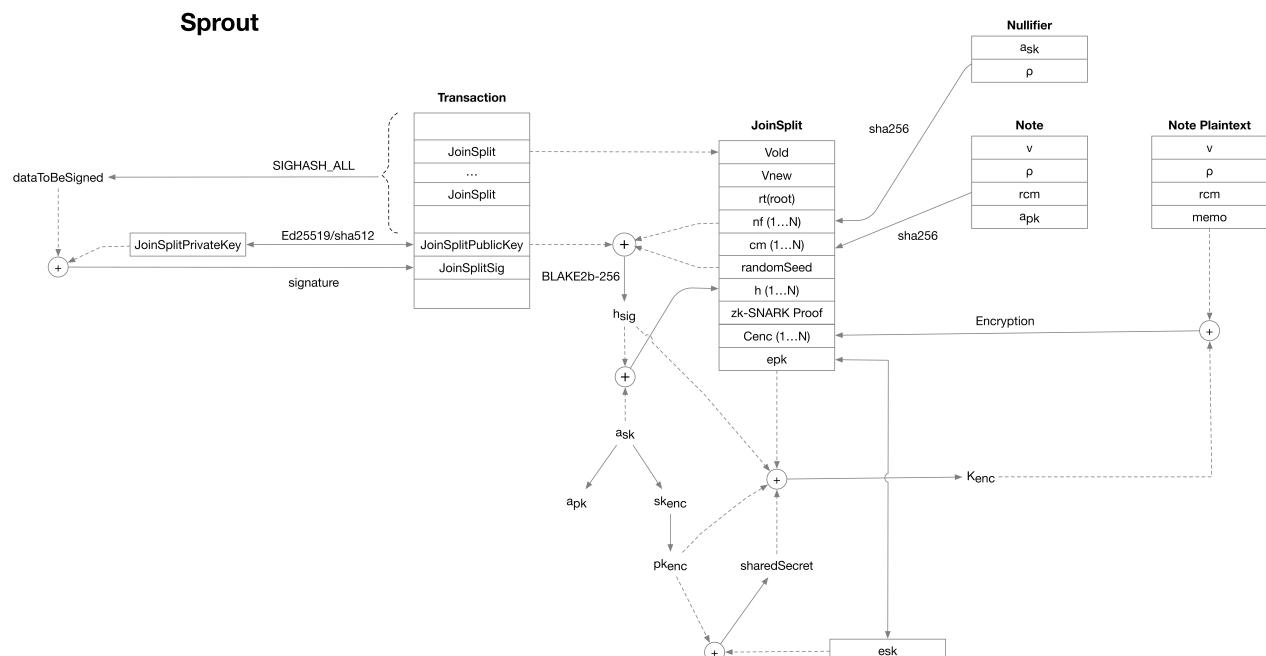
以上只是最简单的概括Zcash零知识证明的大体思路，ZCash的设计非常复杂和严谨，有很多细节。顺便说一句，理解ZCash，只需要看ZCash的白皮书protocol.pdf。理解了白皮书的设计，看源代码非常直接。

这就是零知识证明的一个重要的运用方向 - 隐私，现实中还有很多应用类似技术的项目：EYBlockchain(EY, 安永)，Quorum(JPMorgan)。

看ZCash的白皮书需要一点耐心，144页的白皮书形式化太多，通篇就只有一张图（地址和Key生成关系图）。本文画图总结了Sprout和Sapling的Transaction的数据结构。

经过Sprout和Sapling两次升级，目前ZCash中Transaction中集成了三种交易：1/ 透明交易 2/ JoinSplit (Sprout) 3/ Spend/Output (Sapling)。

Sprout



Sprout使用JoinSplit结构表示一笔交易。JoinSplit中的Void和Vnew实现了隐私和透明交易的交易金额的平衡。rt是Note commit形成merkle树的树根。nf和cm分别是Nullifier和Note的commitment（在Sprout都是使用的sha256算法）。Note, Note Plaintext, 以及Nullifier相对直白。

1.1 JoinSplitSig

JoinSplitSig对整个Transaction数据使用私钥进行签名，保证Transaction的数据不被篡改。签名的数据要被验证，必须提供“公钥”。在ZCash的框架中，隐私考虑，转账双方的“公钥”都不能公开。为了能提供签名，就只能重新生成临时的“公钥”/“私钥”对（JoinSplitPublicKey, JoinSplitPrivateKey）。用JoinSplitPrivateKey对整个Transaction的“SIGHASH_ALL”的结果进行签名，生成JoinSplitSig。

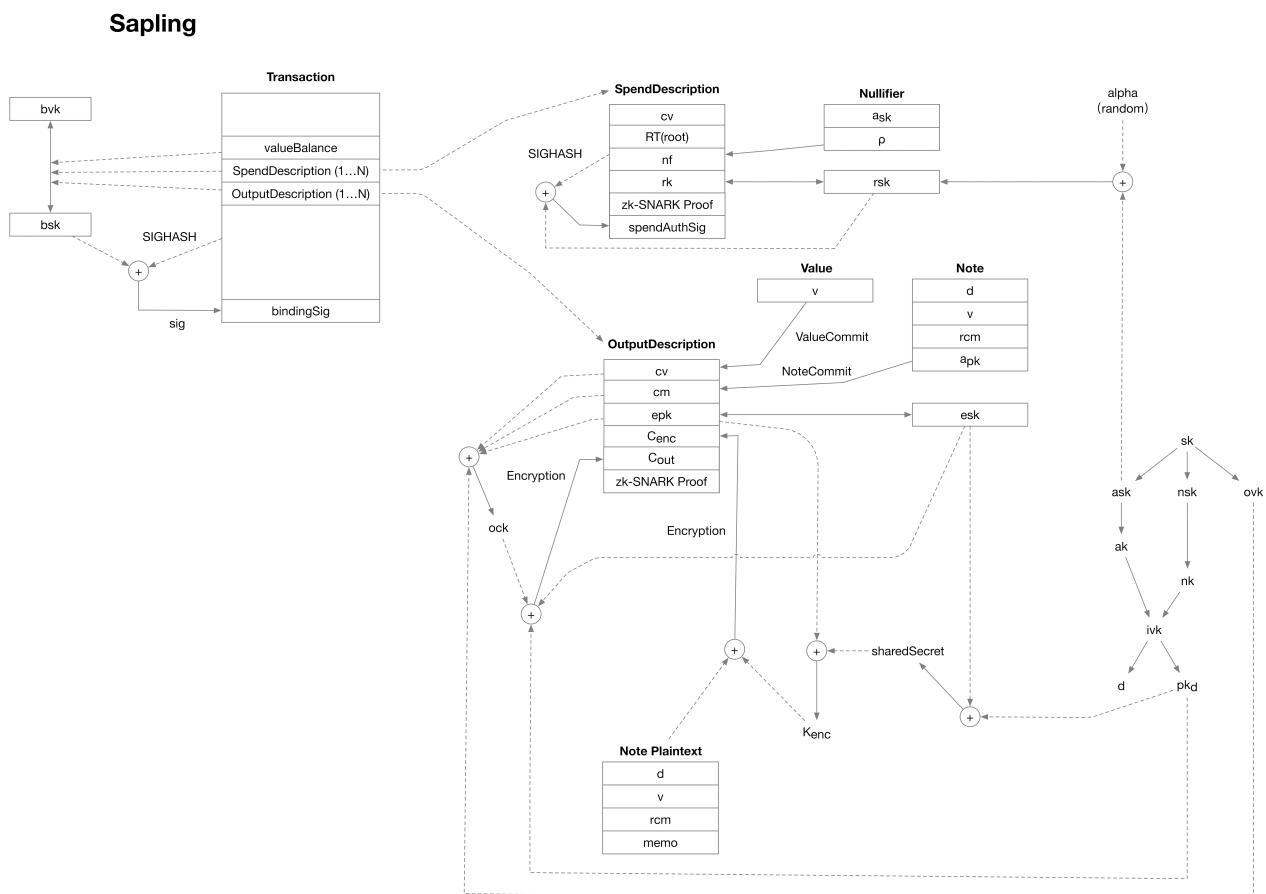
1.2 hsig 和 h

hsig是一个比较有意思的设计。试想，如果只有JoinSplitSig机制，虽然保证了Transaction数据的完整性，但并没有保证签名本身不能变。完全可以在Transaction其他数据不变的情况下，重新生成JoinSplitPublicKey，从而生成新的JoinSplitSig。hsig就是为了解决这个问题。hsig“绑定”所有的nf的数据和当前使用的JoinSplitPublicKey。并且，使用每个nf中对应的“私钥”，对hsig进行hash计算，生成h。也就是说，每个nf对应的私钥都“授权”使用当前的JoinSplitPublicKey。这样，JoinSplitPublicKey就不能随意修改，要做改动，必须知道每个nf对应的“私钥”。

1.4 Cenc

Sprout使用的是“In-band secret distribution”。简单的说，需要传输给转账对方的信息（Note plaintext），加密后存储在链上。采用这种方式，转账对方不需要实时在线，任何时候都能同步链上数据确认交易。和JoinSplitSig一样的思想，转账对方的信息不能直接作为加密密钥。先随机生成epk/esk，再和pkenc结合，生成加密密钥。

Sapling



Sapling是一个比较大的升级，零知识证明的性能提升了十几倍。Sapling不用JoinSplit结构表示交易，而是用SpendDescription和OutputDescription直接表示“花费”和“支出”。一个比较重要的设计是：valueBalance, SpendDescription中的cv以及OutputDescription中的cv都是value的同态commit。所谓的同态commit，就是value的计算后的commit和commit再计算的结果相等。

2.1 spendAuthSig

SpendDescription中的spendAuthSig是对整个SpendDescription进行签名。和Sprout签名的思想类似。先随机出rsk和rk密钥对，再使用rsk进行签名，同时把rk放在SpendDescription中。

2.2 Cenc和Cout

Sapling同样使用的是"In-band secret distribution"。Cenc是对Note Plaintext进行加密的结果。和Sprout类似，加密的密钥由esk和pkd生成。Sapling比Sprout设计了更多的密钥“权限”。众多密钥中，有个ovk (outgoing viewing key)，也就是拥有ovk，可以查看outgoing的交易。原理很简单，就是用ovk将esk和pkd加密，生成Cout。

2.3 bindingSig

bindingSig也是整个Transaction数据的签名。签名使用的公钥/私钥 (bvk/bsk) 是通过cv以及生成cv时采用随机数生成。因为同态commit的算法保证**bvk=bsk*R** (R是生成元)，所以，bsk和bvk存在公钥/私钥关系。bingdingSig就是用bsk对整个Transaction签名的结果。

2. Filecoin

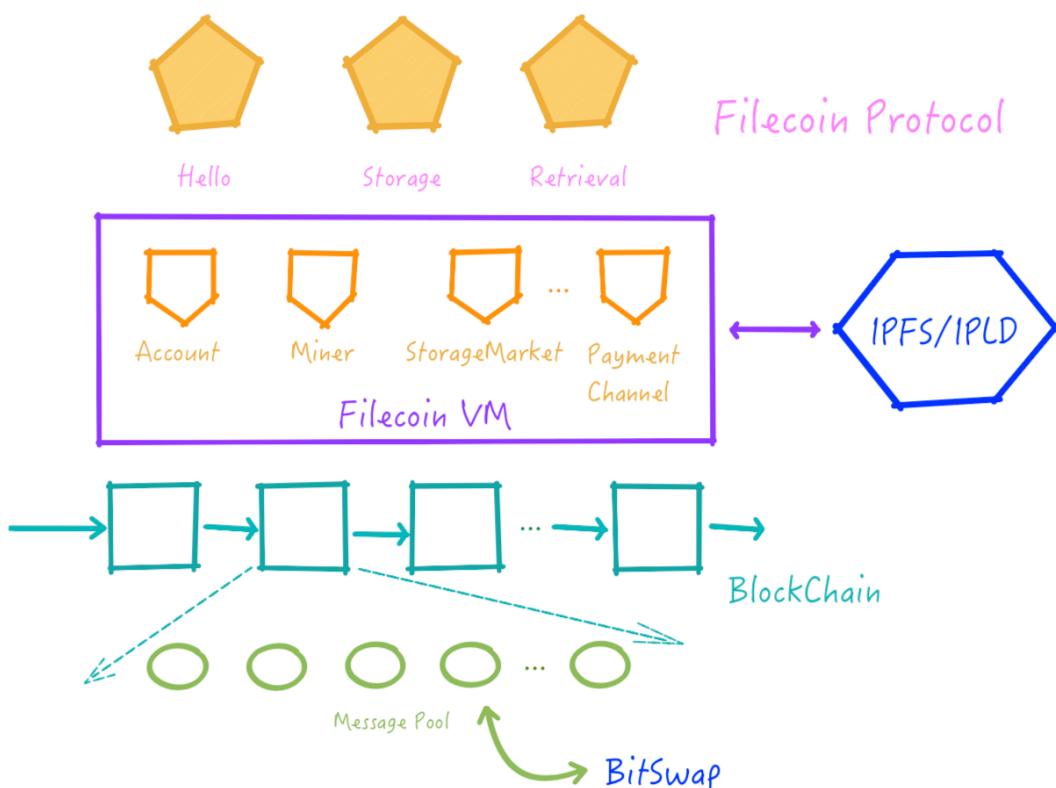
<https://mp.weixin.qq.com/s/xMVN3LbCDux6zsDth9BOQg>

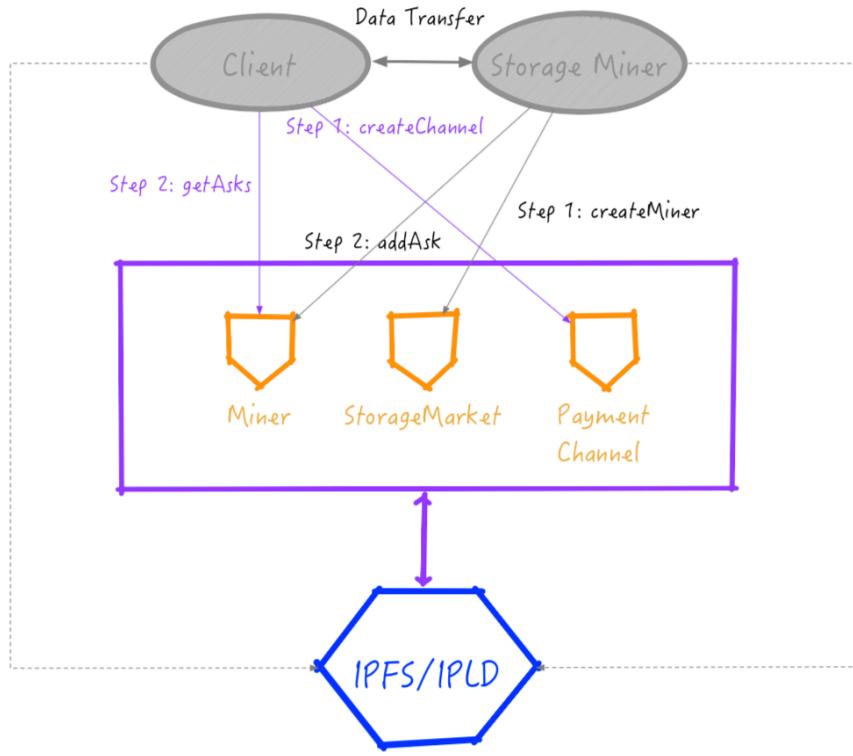
https://mp.weixin.qq.com/s/Sd6Y0gSX6HB4BFRKPV_0dQ

Filecoin是存储行业比较热门的项目。Filecoin想搭建一个去中心化的存储交易平台。去中心化的存储，有个核心问题，怎么证明存储提供方，真实有效的存储了指定的数据。

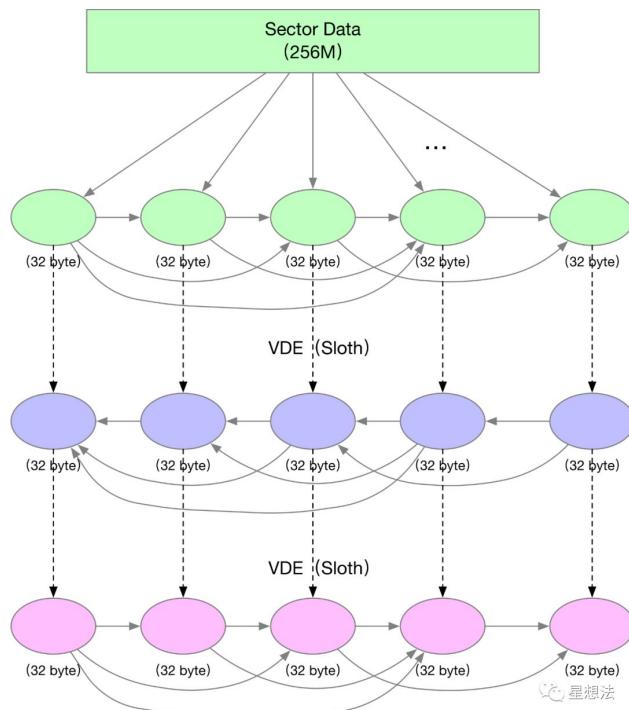
Filecoin是通过PoREP以及PoST算法实现的（其中就包括零知识证明）。

PoREP是数据存储证明算法（证明用户数据被正确的处理）。PoRep算法的全称是ZigZag-DRG-PoRep。





Sector中未Seal的原始数据首先依次分成一个个小数据，每个小数据32个字节。这些小数据之间按照DRG (Depth Robust Graph) 建立连接关系。按照每个小数据的依赖关系，通过VDE (Verifiable Delay Encode) 函数，计算出下一层的所有小数据。整个PoRep的计算过程分为若干层（目前代码设置为4层），仔细观察每一层的DRG关系的箭头方向，上一层向右，下一层就向左，因此得名ZigZag (Z字型)。



每一层的输入称为d (data)，每一层的VDE的结果称为r (replica)。对每一层的输入，建构默克尔树，树根为comm_d，整个树的数据结构称为tree_d。对每一层的输出，建构默克尔树，树根为comm_r，整个树的数据结构称为tree_r。

简单的说，PoREP通过零知识证明，证明每一层的数据都经过VDE计算生成，并提供最后结果的Merkle树的树根。

在数据处理并存储后，每隔一段时间，需要提交存在性证明，也就是PoST算法。PoST算法的基本思想，随机挑选一个Merkle树的叶子节点位置，需要提供出一条Merkle路径的零知识证明。

这个零知识证明的第二个应用方向：链上数据压缩。PoREP算法，通过零知识证明证明数据已经正确处理，并提供了处理后数据形成的Merkle树的树根。PoST，每隔一段时间，随机抽选一个叶子数据，需要存储提供者在一定的时间内提供从该叶子数据到Merkle树根的路径证明。如果，处理完的数据没有保存在一个可靠的存储上，无法在合理的时间内重建整个Merkle树，也就无法提供证明。

3. DEX3.0 (Loopring)

从2017年，路印从“环路撮合”的最初设计，经过了1.0, 2.0以及3.0的三个大的版本的协议升级。1.0/2.0，相对来说，受限以太坊本身性能的限制，交易流程复杂，体验和中心化交易所相比，有较大的差距。路印协议3.0，通过零知识证明技术（ZKP），在zk Rollup的基础上，结合DEX的业务场景开发设计，兼顾去中心化和交易性能。

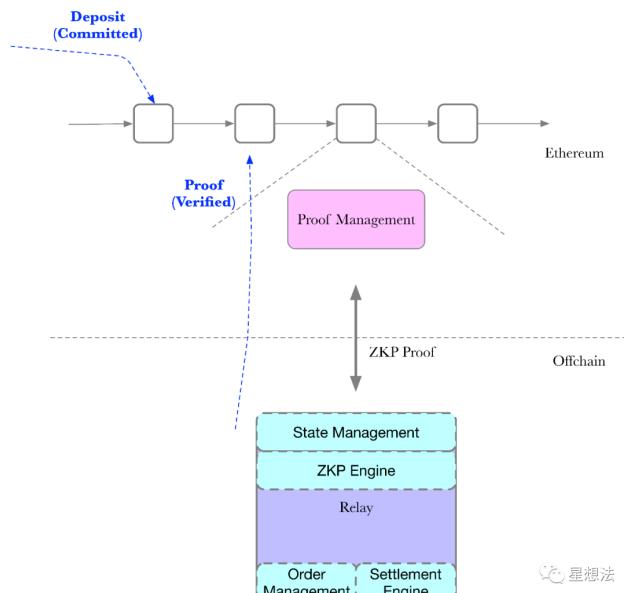
相对1.0, 2.0来说，路印协议3.0采用零知识证明（ZKP）技术，将所有的撮合逻辑都在链下完成。每一笔撮合（Settlement）都会生成证明并提交到链上，证明链下的撮合正确无误。

路印协议采用和以太一致的“账户”模型，所有的账户的“状态”（余额）都记录在链下。

所有和状态相关的操作，都是在链下更改，提交Proof到链上记录。因为存在链上链下的状态同步，账户的任何操作有三个状态：

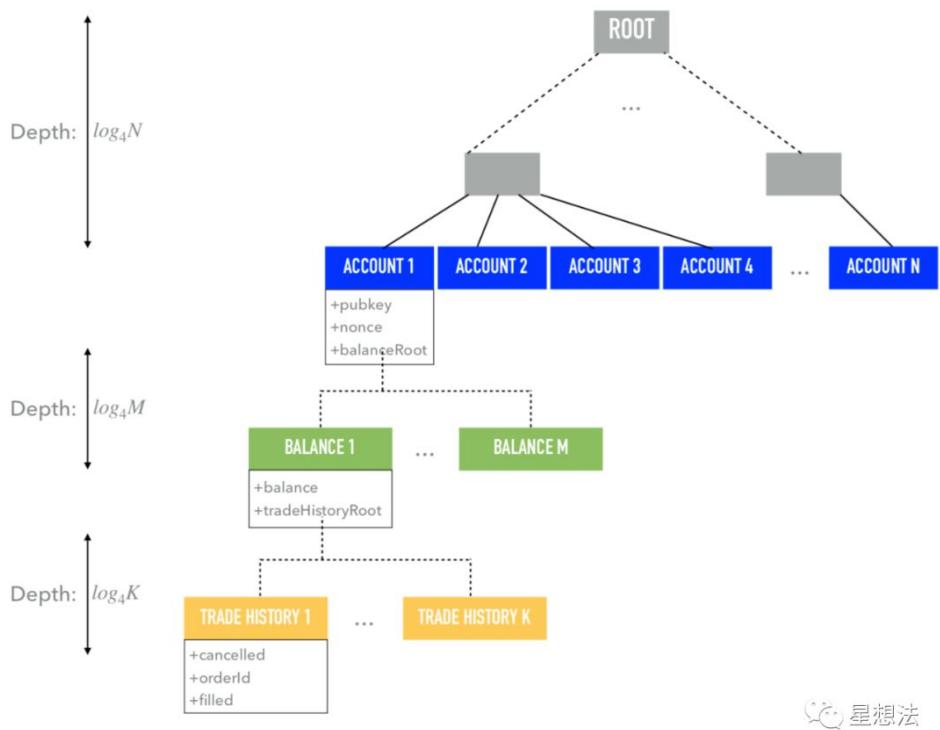
1/ **Committed** (操作已经提交) 2/ **Verified** (该操作已经提供了相应的Proof) 3/ **Finalized** (之前的所有的操作都已经提交正确的Proof)

以用户Deposit“充值”的操作为例：



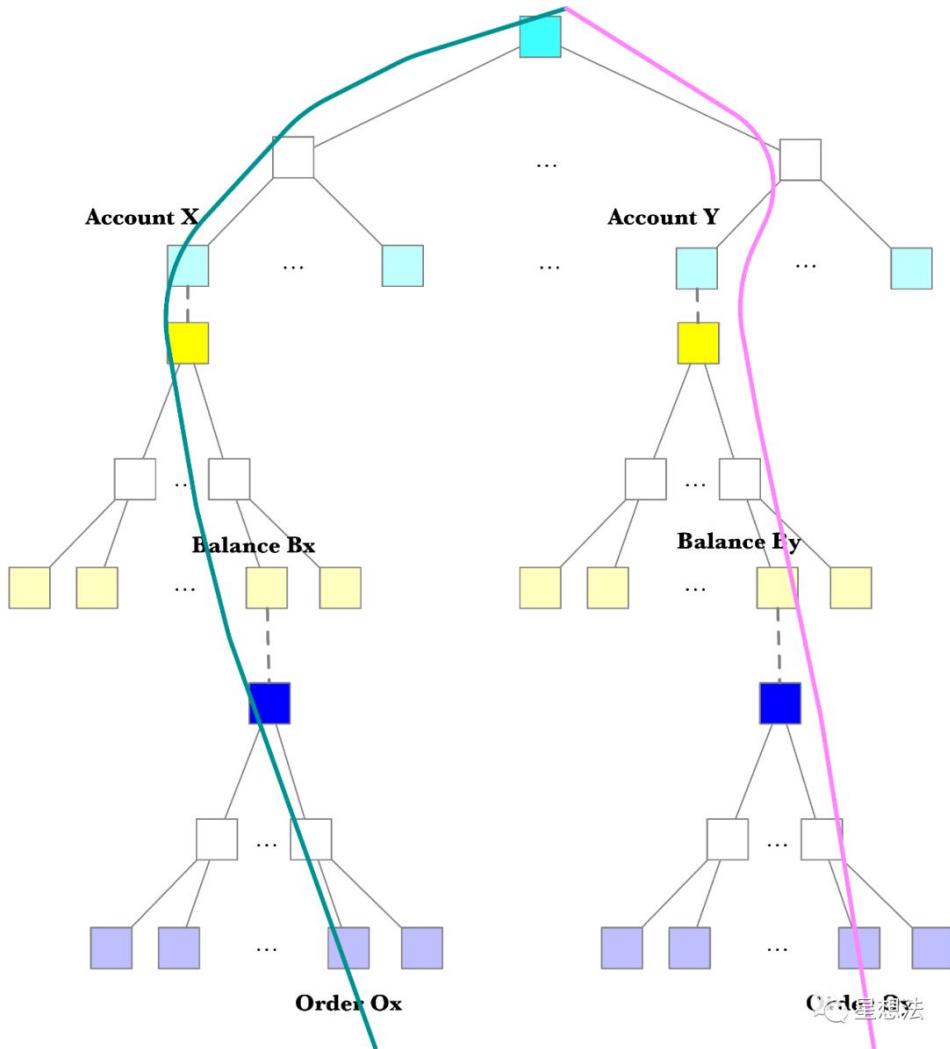
用户转账到路印协议的智能合约，转账在链上确认（链上完成充值）。该操作的状态就是“Committed”。链下的Relay，监测到“Committed”的状态后，更改链下的状态，生成Proof，并将证明提交到链上，此时该“充值”操作的状态为“Verified” - 链下也已经完成充值。如果之前的所有操作都是Verified，那该操作的状态就是Finalized（也就是这个状态是确定的，不会被篡改的）。

链下的状态用三层的四叉Merkle树来表示：



QQ 星想法

路印3.0，采用的是zkSNARK的Groth16算法提供零知识证明。针对每种操作，Relay都会提供对应的ZKP证明电路。以链下撮合为例，相应的电路证明的逻辑如下：



假设Account X链下转账给Account Y。ZKP证明电路，包括：

- 1/ TradeHistory中Order Ox的变化导致TraderHistory的树根的变化
- 2/ TradeHistory中Order Oy的变化导致TraderHistory的树根的变化
- 3/ Balance Bx变化导致Balance的树根的变化
- 4/ Balance By变化导致Balance的树根的变化
- 5/ 两个账户的Balance的变化一致
- 6/ Account X和Account Y账户的变化导致的Account树根的变化

这是零知识证明的第三个方向：扩展性。在足够多的交易的情况下，路印3.0的TPS在目前的以太坊上达到了350。在君士坦丁堡升级后，TPS能达到1400。每笔交易平均下来的费用大约在1美分。

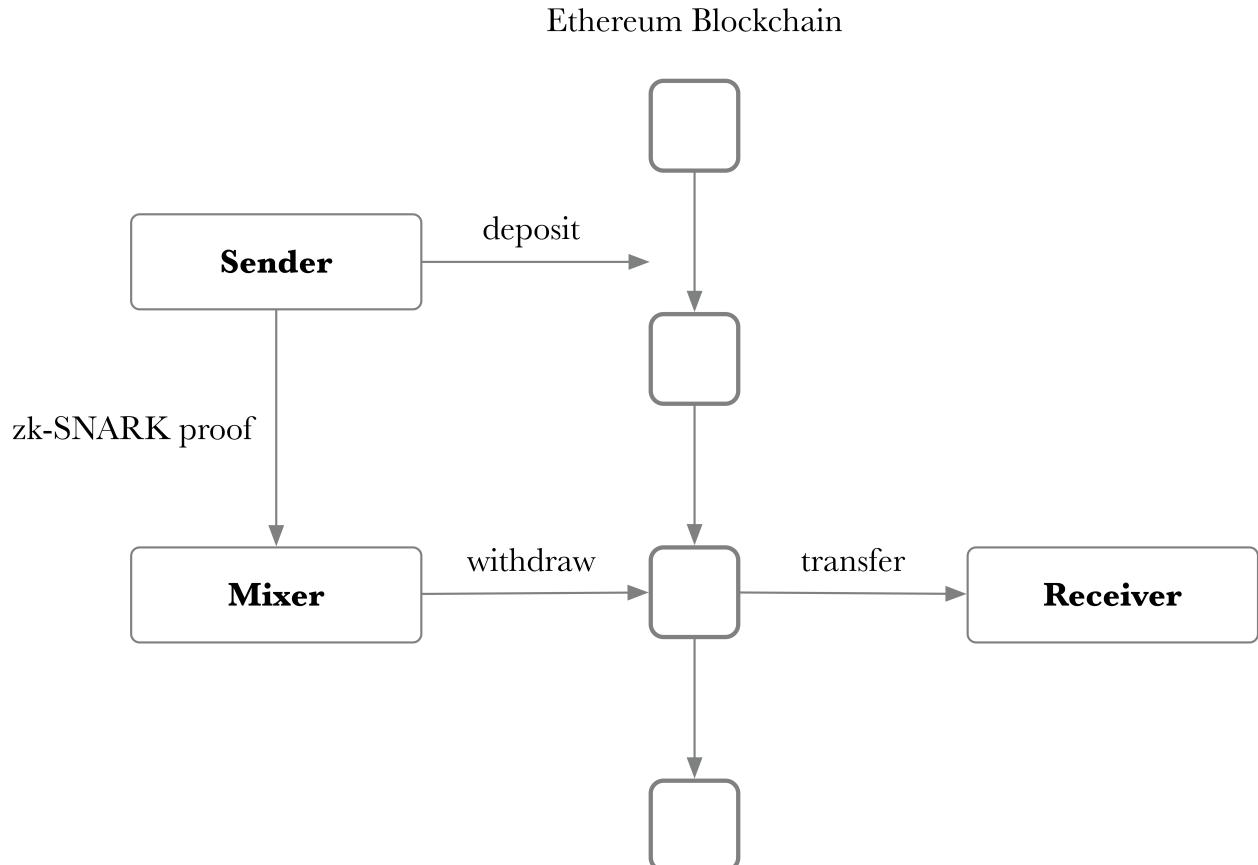
	Loopring 1.x and 2.x	Loopring 3.0 (w/ Data Availability)	Loopring 3.0 (w/o Data Availability)
trades / second	2	350	6900
trades / second, Post Istanbul	2	1400	10500
cost / trade	13¢	1¢	0.08¢ <small>星想法</small>

4. Mixer

除了通过公链或者侧链实现交易的发送方/接收方以及金额隐藏外，Mixer，江湖人称“混币”，是在已有公链上实现交易的发送方的隐藏（匿名）。Mixer，就是将一些账户的资金“混”在一起，由公开的第三方代替发送方发起转账。这个第三方，被称为Mixer或者Relayer。

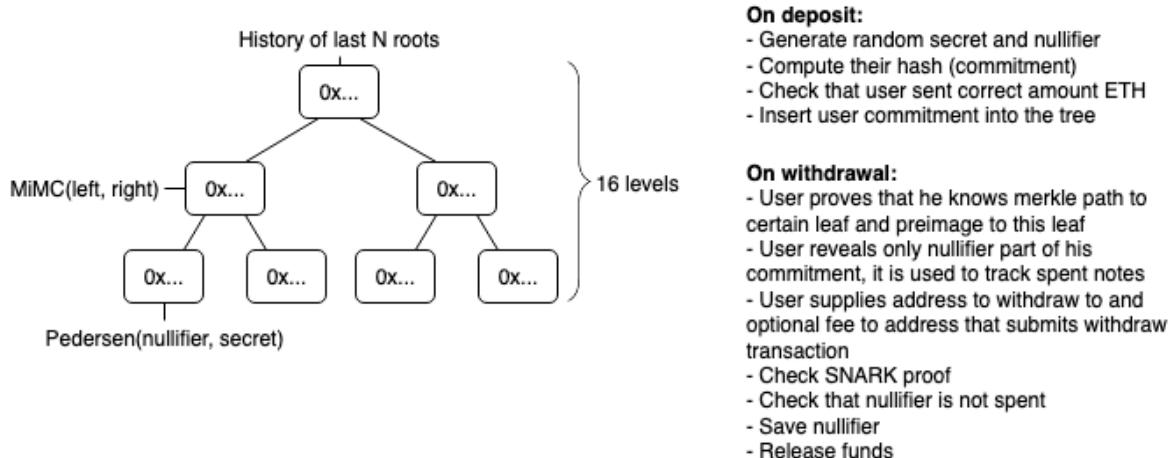
目前的混币项目：MicroMix，Tornado，Hopper。

Tornado Mixer的源代码地址：<https://github.com/peppersec/tornado-mixer>。Tornado Mixer的核心逻辑在contracts/Mixer.sol文件中：一个是deposit函数，一个是withdraw函数。Tonado Mixer的框架如下图：



大体逻辑和MicroMix类似，发送方（Sender）首先向智能合约转账（固定金额），并在智能合约上创建commitment。接下来，发送方（Sender）将零知识证明发送给Mixer，Mixer确认证明后，通过withdraw函数向接收方转帐。

所有的Commitment在智能合约中组织成一个Merkle树：



叶子节点的计算采用Pedersen Hash算法，中间节点采用MiMC Hash算法。整个树高为16。也就是说，Tornado Mixer一个智能合约，支持 2^{16} 次转账。

Commitment Merkle树高为16。Deposit函数大约消耗88.8w的GAS，Withdraw函数大约消耗69.2w的GAS。证明电路的Constraint为22617。生成一次证明的时间大约为6.1秒。

5. Coda

9 - 其他

1. zk-SNARK trusted setup

2. zk-STARK/Bulletproof

	Proof size	Proving time	Verification time
zk-SNARK	•	●	•
zk-STARK	●	•	●
Bulletproof	●	●	●

3. dizk

4. 零知识证明加速 - GPU(cuda)

https://mp.weixin.qq.com/s/w4sLlvbyADuqW4k-FE_KiA

<https://mp.weixin.qq.com/s/0ubC3mTXycsCeWVrx4FJCA>

5.

Reference:

1. <https://www.law.upenn.edu/cf/faculty/jvagle/workingpapers/A%20Gentle%20Introduction%20to%20Elliptic%20Curve%20Cryptography.pdf> Jeffrey L. Vagle 2000.11.21 (比较偏科普的语气)
2. <https://wenku.baidu.com/view/b20f16be856a561252d36f85.html>
3. [https://zh.wikipedia.org/zh-hans/域_\(數學\)](https://zh.wikipedia.org/zh-hans/域_(數學))
4. <https://www.cnblogs.com/Colin-Cai/p/9438343.html>
5. <http://218.4.189.15:8090/download/9cf9ab3d-2f3a-44f2-9d84-13b94e80d1f2.pdf>
6. <https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/>
7. <https://andrea.corbellini.name/2015/05/23/elliptic-curve-cryptography-finite-fields-and-discrete-logarithms/>
8. <http://petkus.info/papers/WhyAndHowZkSnarkWorks.pdf>
9. <https://medium.com/@imolfar/why-and-how-zk-snark-works-1-introduction-the-medium-of-a-proof-d946e931160>
10. <https://medium.com/@imolfar/why-and-how-zk-snark-works-2-proving-knowledge-of-a-polynomial-f817760e2805>
11. <https://eprint.iacr.org/2016/260.pdf>
- 12.