零知识证明算法 zk-SNARK 调研 (上)

Shuang Wu, Jiaxuan Li

HPB 隐私计算小组

2020 年 2 月 9 日

摘要

本篇文章主要介绍了 zk-SNARK 算法的数学原理。第一部分为算法简介，第二部分和第三部分讲解了 zk-SNARK 用到的数学工具，从简单到复杂，循序渐进的引出 zk-SNARK 算法，并给出构造实例，使读者能理解 zk-SNARK 的构造思路，理解这部分不需要读者具备高深的密码学原理，但相关的基本概念，比如多项式，离散对数问题，还是要具备。第四部分是算法直接的数学阐释。第五部分是总结和后续工作。本文档只涉及算法，不涉及代码讲解。

目录

1 零知识算法简介 3

1.1 什么是零知识证明? 3
1.2 零知识证明应用场景 3
1.3 相关零知识证明算法 5
1.4 零知识证明 zk-SNARK 6

2 zk-SNARK 算法原理 1（多项式的零知识证明） 7

2.1 zk-SNARK 的证明媒介——多项式 7
2.2 多项式零知识证明 8
2.2.1 证明一个多项式 8
2.2.2 多项式的因式分解 9
2.2.3 模糊计算 10
2.2.4 约束多项式 13
2.2.5 零知识 15
2.3 非交互多项式零知识证明 16
2.3.1 加密值的相乘 17
2.3.2 可信任参与方的 setup 18
2.3.3 信任任意一个参与者 19
2.4 多项式的 SNARK 21
2.5 结论 22

3 zk-SNARK 算法原理 2（从程序到多项式的构造） 23

3.0.1 电路转化成多项式 23
3.0.2 程序转化成电路 26

4 算法原理（直接的数学表达） 27

4.1 算数电路和二次算数电路张成方案 27
4.2 算数电路 27
4.3 二次算数电路张成方案 27
4.3.1 构造 QAP 28
4.4 zk-SNARK 29

5 结论 32

1 零知识算法简介

1.1 什么是零知识证明?

zk-SNARK 是 Zero-knowledge succinct non-interactive arguments of knowledge 的缩写，他的意思是：

zero knowledge：零知识，即在证明的过程中不透露任何隐私数据：

succinct：简洁的，主要是指验证过程不涉及大量数据传输以及验证算法简单；

non-interactive：无交互。证明者与验证者之间不需要交互即可实现证明，交互的零知识证明要求每个验证者都要向证明者发送数据来完成证明，而无交互的零知识证明，证明者只需要计算一次产生一个 proof，所有的验证者都可以验证这个 proof。

zk-SNARK 是证明某个声明是真却不泄露关于该声明的隐私信息的一个很有创新性的算法，他可以证明某人知道某个秘密却不会泄露关于这个秘密的任何信息。这个算法可以解决什么问题呢?

1.2 零知识证明应用场景

- **验证隐私数据**
  - A 在银行里的存款余额大于 X
  - 去年，一家银行未与实体 Y 进行交易
  - 在不暴露全部 DNA 数据的前提下匹配 DNA
  - 一个人的信用评分高于 Z
- **匿名认证**
  - 在不揭露身份的情况下（比如登录密码），证明请求者 R 有权访问网站的受限区域
  - 证明一个人来自一组被允许的国家/地区列表中的某个国家/地区，但不暴露具体是哪个
  - 证明一个人持有地铁月票，而不透露卡号
- **外包计算**

  - 付款完全脱离任何一种身份
  - 纳税而不透露收入

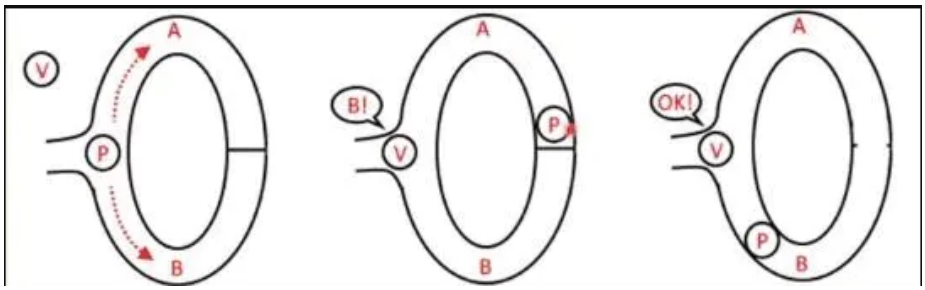
- **匿名支付**
  - 将昂贵的计算外包，并在不重新执行的情况下验证结果是否正确；它打开了一种零信任计算的类别
  - 改进区块链模型，从所有节点做同样的计算，到只需一方计算然后其它节点进行验证

在任何一个零知识证明系统中，都有一个证明者 (prover) 去向一个验证者（verifier) 证明他所发出的某一个声明是正确的，但又不会让验证者披露任何隐私信息。比如说，验证者知道证明着的银行账户存款余额大于 X，但无法知道银行账户余额的具体数值以及其他信息。一个零知识证明算法需要满足三个性质 [6]：

- 完整性——只要陈述是正确的，证明者就可以让验证者确信
- 可靠性——如果陈述是错误的，那么作弊的证明者就没有办法让验证者相信
- 零知识——协议的交互仅仅揭露陈述是否正确而不泄漏任何其它的信息

一个经常用来科普零知识证明的例子 [8]。下图中所示是一个山洞，入口处有两条路 A 和 B，而这两条路在山洞深处被一道门给隔开了，只有说出开门魔咒门才能打开。这里涉及两个角色 P（Proofer）和 V（Verifier），P 试图向 V 证明，他知道开门魔咒；如果属实，V 就饶 ta 不死。P 自然不能直接将魔咒告诉 V，因为万一 ta 知道后把自己干掉怎么办；V 则一定不会轻易相信 P。他们可以这么做：

1. P 从 A、B 两条路中随机选择一条走进去；这时，V 在洞外等着，对 P 选择了哪条路一无所知；
2. 等待足够长时间后，V 进入山洞，然后也从 A、B 中随机选择一个并且大声喊出来，譬如，“B！”；



3. P 听到 V 的声音后便从对应的那条路走出来。如果 P 确实知道开门魔咒，那么无论自己和 V 分别选择的是 A 还是 B，P 都能正确地从 V 报出的路走出来。相反，如果 P 不知道魔咒，那么 ta 只有 1/2 的概率会做到。而从 V 的角度来说，如果 ta 看到 P 从正确的路出来了，ta 便有 50% 的把握肯定 P 确实知道魔咒；

将第 1-3 步重复 N 次，如果 P 每次都能做对，那么 V 便有 $1-(0.5)^N$ 的把握相信 P。例如，N=5，可靠性就是 96.9%，已经足够好了。更重要的是，V 对于魔咒仍然一无所知。这便是零知识证明。

1.3 相关零知识证明算法

zk-SNARKs[4],zk-STARKs [1] 和 BulletProofs[2] 是应用在区块链上主要的零知识证明算法。以下从不同参考资料中得到这三个零知识证明算法的性能对比：

	SNARKs	STARKs	Bulletproofs
Algoritmic complexity: prover	O(N * log(N))	O(N * poly-log(N))	O(N * log(N))
Algoritmic complexity: verifier	~O(1)	O(poly-log(N))	O(N)
Communication complexity (proof size)	~O(1)	O(poly-log(N))	O(log(N))
- size estimate for 1 TX	Tx: 200 bytes, Key: 50 MB	45 kB	1.5 kb
- size estimate for 10.000 TX	Tx: 200 bytes, Key: 500 GB	135 kb	2.5 kb
Ethereum/EVM verification gas cost	~600k (Groth16)	~2.5M (estimate, no impl.)	N/A
Trusted setup required?	YES 😒	NO 😄	NO 😄
Post-quantum secure	NO 😒	YES 😄	NO 😒
Crypto assumptions	Strong 😒	Collision resistant hashes 😄	Discrete log 😒

图 1: 零知识证明算法对比 from awesome ZKP repo

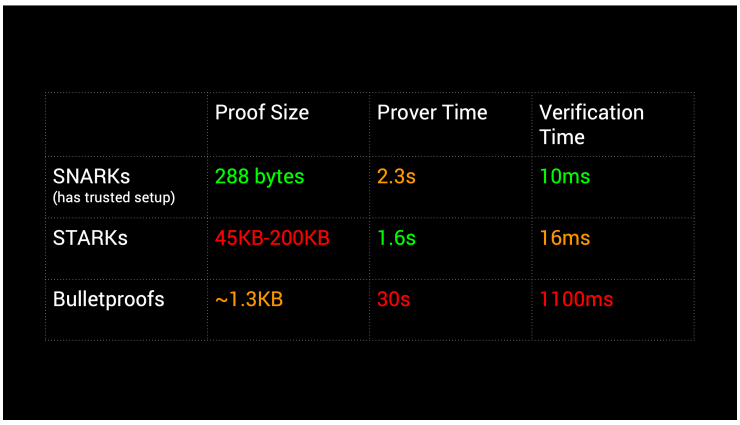


图 2: 零知识证明算法对比 from Elena Nadilinski's slides

由表格中分析可得，在 proof size 方面，SNARK 最具优势，并且产生 proof 所用的时间最短，但它需要 trusted setup。Bulletproof 和 STARK 不需要 trusted setup，Bulletproof 的计算和验证时间都比 STARK 的长，但它的 proof size 比 STARK 小的多。

1.4 零知识证明 zk-SNARK

如果证明者想要使用 zk-SNARK 证明某个声明，他首先要将这个声明转化成正确的"形式"，他要先将验证"证明者知道某一个秘密"的问题转化成"证明者知道某一个程序的正确的输入输出"问题。比如证明者要向验证者证明他知道某个密码，我们可以写一个简单的程序，将程序的输入做哈希然后和密码的正确的哈希值作比较，相同则输出 1，不同的输出 0. 这样，我们就把验证"证明者知道某一个密码"的问题，转化成了"证明者知道一个合法的输入，使这个程序的结果为 1"的问题。第二步是将程序转化成一个电路，再将电路转化为多项式 (QAP[3])，最终将问题转化为"证明者知道某一个多项式"的问题。

也就是说，我们索要经过的步骤是：证明某一个秘密 → 程序计算 → 算数电路 → 多项式 → 证明多项式。

这是 zk-SNARK 算法的概览，很抽象，接下来的部分我们将一步一步解释，为什么要做这些转换和如何做这样的转换。

2 zk-SNARK 算法原理 1（多项式的零知识证明）

2.1 zk-SNARK 的证明媒介——多项式

我们先考虑一个最简单的证明系统，先忽略它是否是零知识证明，也不用考虑它是不是非交互和可用性。

考虑一个含有 10 个比特的数组，我们希望向验证着证明这 10 个比特的值都为 1. 所以我们需要证明的声明是：

我知道一个所有元素都为 1 的长为 10 比特的数组。

验证者每次只能验证一个比特，为了验证声明的正确性，验证者随机挑选一个比特查看其值，如果所查看的比特的确是 1，那么声明正确的概率至少是 1/10. 如果所查看的比特值是 0，那么声明是不正确的。验证者可以多次随机抽取一个比特查看其值，直到他觉得足够相信声明是正确的为止。比如说，验证者要声明正确的可能性至少 50 才能接受这个声明是正确的的，那验证者可以进行以上实验 5 次。如果验证者要声明正确的可能性至少 95 才能接受这个声明是正确的的，那验证者需要验证所有的比特。这种验证方案的缺点是验证者需要验证的比特数与数组的长度成正比，当数组的长度很长的时候，方案将不再实际可行。

那让我们来考虑多项式和多项式的空间曲线

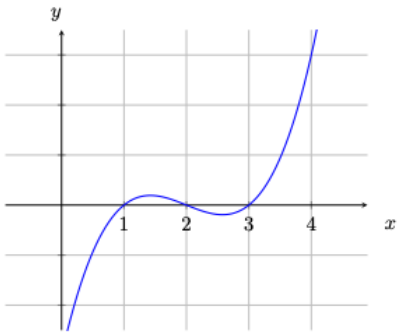


图 3: $f(x) = x^3 - 6x^2 + 11x - 6$

以上曲线由多项式 $f(x) = x^3 - 6x^2 + 11x - 6$ 得到。曲线的阶数由多项式中自变量的最高阶决定。这个例子中为 3。

多项式有一个好处就是，如果我们有两个最高阶为 d 的多项式，它们的曲线最多有 d 个交点，也就是穿过 d 个不同的点。根据这个性质，我们

可以设计一个证明算法来验证声明一个证明者知道某一个验证者也知道的多项式（'知道一个多项式'是说，知道这个多项式的所有系数），注意这个多项式的阶数可以无限大：

多项式零知识证明 version 1

- 验证者随机选取 x，计算多项式的值但将计算结果保密。
- 验证者将 x 发送给证明者，要求证明者也计算多项式在 x 的取值。
- 证明者计算多项式在 x 的取值，将结果发送给验证者。
- 验证者验证证明者提供的结果是否与自己计算的相同，如果相同，则有很大几率证明者的确知道某一多项式。

如果说自变量取值为 1 到 10^{77} 之间的所有整数，验证者一共验证了 d 个点，假设证明者所知道的多项式和验证者验证的多项式不一样，因为这两个不一样的多项式最多有 d 个交点，那么验证者验证的这个多项式的取值点恰好是这 d 个交点之一的概率只有 $d/10^{77}$。

这个证明方案中，验证者只需要进行一轮就能得到足够大的确认声明是否正确的概率，这就是为什么验证多项式就是 zx-SNARK 的中心思想。

2.2 多项式零知识证明

2.2.1 证明一个多项式

我们从证明"证明者 (prover) 知道某一个多项式"开始，再拓展到一般方法。在这个过程中我们会发现多项式的更多性质。

在之前的讨论中，我们所关注的零知识证明方案漏洞很多，证明者和验证者需要互相相信对方会遵守规则，因为在方案中并没有强制任何一方遵守规则的方法。比如说，一个证明者并不需要知道某一个多项式，它只要利用任何他能用到的方法去猜多项式在某一点值。如果多项式的值域不够大，比如说，只有 10 个值，那证明者就有不可忽略的概率猜到正确的值。这些问题我们需要一一解决。但首先我们需要知道，声明"prover 知道某一个多项式"是什么意思呢？我们知道一个多项式可以表达为（n 为多项式的阶数）：

$c_n x^n + ... c_1 x^1 + c_0 x^0$

如果一个证明者知道一个一阶多项式，也就是 $c_1 x^1 + c_0 x^0$, 他实际上知道的是这个一阶多项式的系数 c_1, c_0。多项式的系数可以为任何值，包括 0。

举个例子，假设一个证明者知道一个 3 阶多项式，使得多项式有两个零根 $x = 1$ 和 $x = 2$，那么这个多项式可能是 $x^3 - 3x^2 + 2x$

接下来让我们来剖析一下多项式的零根。

2.2.2 多项式的因式分解

代数学基本理论中，任何有零根的多项式都可以分解成一个或多个一阶多项式相乘，也就是说，我们可以将任何有零根的多项式用下面的因式相乘表示：

$(x - a_0)(x - a_1)...(x - a_n) = 0$

其中任何一个因式为零，则整个多项式为零，所以式中所有的 a_i 都为多项式的零根。

以上例子中的多项式可以被分解为 $x^3 - 3x^2 + 2x = x - 0\ (x - 2)(x - 1)$

一个证明者声明“prover 知道一个三阶多项式，并且这个三阶多项式有 1 和 2 两个零根”，这就意味着，证明者所知道的多项式有以下的结构：

$(x - 1)(x - 2)...$

换句话说，$(x - 1), (x - 2)$ 能够整除 prover 所知道的多项式。我们称这些声明中已知因式因子的乘积 $t(x) = (x - 1)(x - 2)$ 为**目标多项式**。未在声明中的因式因子我们记为 $h(x)$.

如此，假设 prover 所声明知道的多项式为 $p(x)$, 则

$p(x) = t(x) \cdot h(x)$

一个简单直接的计算 $h(x)$ 的方法是 $h(x) = \dfrac{p(x)}{t(x)}$, 如果 prover 不能找到这个 $h(x)$，说明他并不知道 $p(x)$ 这个多项式，他的声明就是错误的。基于以上分析，我们把之前的零知识正经方案改进：

多项式零知识证明 version 2

- 验证者随机选取 r，计算 $t = t(r)$ 的值, 将 r 发送给 prover。
- 证明者先计算得到多项式 $h(x) = \dfrac{p(x)}{t(x)}$，再计算 $h = h(r)$ 和 $p = p(r)$, 将 h, p 发送给验证者。
- 验证者验证 $p = h \cdot t$, 如果等式成立，则接受，否则拒绝。

如果验证者和证明者都能遵守规则，则这个方案是可以实现“零知识”这个属性的，因为验证者只能知道证明者有一个阶数为 3 的多项式，并且多项式有 0 和 1 两个零根。但验证者并不能推算出验证者所知道的多项式。

但这个方案也有很多漏洞：

1. 验证者可能不需要知道任何三阶多项式，它只要计算出 $t = t(x)$, 随便选两个值 p, h 使得 $p = h \cdot t$, 然后将 p, h 发送给验证者，就能使验证者接受他的声明。（从这一点来讲，这个方案很愚蠢）

2. 假设我们找到了一种方法使得证明者不能任意选取 p, h，那他可以自己构造任意的多项式 $h(x)$, 然后计算 $h = h(r)$ 再将 $p = h \cdot t, h$ 发送给验证者就好。

3. 假设我们又找到了一种方法，使得证明者必须要知道一个多项式 $p(x)$ 来计算 $h(x)$, 我们还是不能保证 $p(x)$ 的阶数就是 3. 因为更高阶数的多项式，只要有零根 1 和 0 就能通过验证。

所以，以上零知识证明方案，除了可以实现零知识，其它的安全需求都满足不了。但接下来我们将介绍一些数学工具，运用这些数学工具，我们能强迫验证者和证明者遵守规则，从而解决以上的漏洞。

2.2.3 模糊计算

以上方案的前两个问题都是因为证明者知道 r 和 $t(r)$ 的值，我们希望将方案改进，使证明者不知道验证者选择了那个随机值 r，但证明者仍然能进行相关计算。相当于证明者只收到一个黑箱，可进行操作但不会知道黑箱中的隐私数据。

同态加密 这正是同态加密可以实现的功能: 在加密数据上做计算。一种简单的方法是将加密数据作为指数，然后对幂进行操作：

比如，我们想将 3 加密，那么我们选取 5 作为底，3 作为指数：

$5^3 = 125$

如果我们想做计算 3×2, 那我们可以之间求幂的平方：

$(5^3)^2 = 15625 = 5^6$

这样我们相当于实现了加密数据的乘法操作。我们也可以通过幂与幂的乘法实现加密数据的加法操作, 比如我们计算 3+2：

$5^3 \times 5^2 = 5^5$

减法操作则通过相除就可以实现：

$\frac{5^5}{5^3} = 5^2$

但是呢，因为我们需要将底数 5 公开，这样直接将加密数值作为指数并不安全，因为攻击者只要将结果不断地除 5，就能得到指数。因此我们需要进行模运算。

模运算 我们将使用有限的数值 $0, 1, ..., n-1, n$ 进行以上算法，如果在计算过程中有数值超过了 n，我们就对这个值进行模 n 运算，得到的结果又将重新回到 $[0, n]$ 的范围。不使用模运算时，由加密结果可以计算出加密数值，比如以上例子，由 $5^2 = 25$ 我们可以推测出加密数值为 2，但如果我们使用了模运算，假设做模 25 运算，则加密的数值可能是 $2, 4, 8...$，由加密结果推测加密数值就更困难。如果模够大，实际上将不能找到这个加密数值。这也是秘密学经典的离散对数问题。

Remark: 通过这种加密方法，我们计算加密数值乘以一个已知数值的加密结果（$(5^2)^3$），我们也可以计算加密数值之间的加减法。但是我们不能计算两个加密数值之间的乘法。比如我们已知 5^a,5^b 我们不能计算 5^{ab}. 我们最终会发现，这些限制将成为 zk-SNARK 算法的基石。

加密的多项式 通过以上的工具，我们现在就可以介绍，我们如何能隐藏验证者选择的随机数 r 以及 $t(r)$ 的值。然后修改我们的零知识证明算法 version 2。

假如我们有一个多项式 $p(x) = x^3 - 3x^2 + 2x$, 我们想要证明者计算这个多项式在 $x = r$ 时的结果，又不能让证明者知道这个 r 的值和 $p(r)$, 我们就需要将 r 加密，让证明着直接在密文上运算来得到加密后的 $p(r)$, 也就是我们之前介绍的同态加密。我们采用的方法是将 r 加密得到密文 $E(r) = g^r$ (g 是循环群的生成元，也就是之前介绍的幂的底) 发送给证明者，则证明者这时可以计算 $2r$ 的加密结果，即为 $[E(r)]^2 = g^{2r}$。但证明者无法由 g^r 直接计算出 g^{r^2} （之前介绍的，这种同态加密的缺点是不能计算两个加密数值的乘法）。因此我们又需要将 r^2 和 r^3 加密后得到 $E(r^2) = g^{r^2}, E(r^3) = g^{r^3}$ 发送给证明者，此时证明者就可以计算：

$E(r^3) - [E(r^2)]^{-3} + [E(r)]^2 = g^{r^3 - 3r^2 + 2r}$

可见证明者只需知道 $E(r), E(r^2), E(r^3)$, 就能计算出多项式的加密后的结果。证明者不知道 r 的值，也不知道 $p(r)$ 的值。

现在我们可以改进之前的零知识证明 version2, 来验证证明者知道一个阶数为 d 的多项式。

多项式零知识证明 version 3

- 验证者:
  - 随机选取 r,
  - 计算 r 的幂的加密密文：$E(r^i) = g^{r^i}, i = 1, 2, ..., d$。
  - 计算目标多项式在 r 的取值：$t(r)$
  - 将密文 $E(r^i) = g^{r^i}, i = 1, 2, ..., d$ 发给证明者
- 证明者:
  - 计算得到多项式 $h(x) = \dfrac{p(x)}{t(x)}$.
  - 利用验证者发送过来的 $E(r^i)$, 和证明者知道的多项式的系数 $c_0, c_1, ..., c_n$, 计算 $E(p(r)) = g^{p(r)} = (g^{r^d})^{c_d}(g^{r^{d-1}})^{c_{d-1}}...(g^{r^0})^{c_0}$, 同样的，证明者还需利用多项式 $h(x)$ 的系数计算 $E(h(r)) = g^{h(r)}$。
  - 将计算结果 $E(p(r))$ 和 $E(h(r))$ 发送给验证者。
- 验证者:
  - 验证 $E(p(r)) = g^{p(r)}$ 是否与 $[E(h(r))]^{t(r)}$ 相等, 如果等式成立, 则接受，否则拒绝。

由于证明者现在不知道 r 的值，于是之前提到的前两个漏洞都可以被解决，但是第三个漏洞，也就是“不能保证证明者所知道的多项式确实为 d 阶”，这个问题不能解决。

比如说，证明者可以直接将 $h(x)$ 设为一个常数 h, 然后计算 $E(h) = g^h$ 则 $p(x) = h \times t(x)$, 证明者可以利用目标多项式 $t(x)$ 的系数计算 $E(t(r))$, 从而计算出 $E(p(r)) = E(t(r))^h$。将 $E(h)$ 和 $E(p(r))$ 发送给证明者，依然能通过验证。

所以，我们可以将第三个漏洞归结为，我们不能保证证明者将会使用 $E(r^i)$ 来计算相关的 $E(h)$ 和 $E(p(r))$，这个问题可以通过以下方法解决。

2.2.4 约束多项式

为了更清楚的阐释以上问题，我们来举一个简单的例子: 一个 1 阶多项式，只有一个变量和一个系数：$p(x) = c \cdot x$, 相对应的对变量的幂的加密则为 $E(r) = g^r$, 我们需要确定的是，证明者在计算 $p(r)$ 的加密密文的时候，证明者确实使用的是 g^r 来进行的同态加密乘法运算，也就是 $[g^r]^c$，而不是别的随机值 $[g^a]^c$。

为了解决这个问题，我们可以对原来的加密数值进行“偏移”，然后再将偏移后的数值加密，将原数值的密文和原数值偏移后加密的密文都发送给证明者，证明者需要在这两个密文上进行同样的操作，验证者在受到结果后可以检查两个结果之间是否还是存在原先的偏移量。这个偏移量，就像是“checksum”的作用。这种方法称为“Knowledge-of-Exponent Assumption”（KEA）更具体的来说:

- Alice 有一个值 a, 她希望 Bob 做 a 的任意次方的计算，唯一的要求是 Bob 只能以 a 为底，（因为离散对数问题，如果 Bob 随机选择 r 并计算 a^r,Alice 不能计算出 r 的数值, 也不能判断 Bob 给的数值是不是以 a 为底计算得来。）为了确保 Bob, 遵守要求，Alice 进行如下步骤:
  - 选择随机数 α
  - 计算 $a' = a^\alpha \ (mod\ n)$
  - 将元组 (a, a') 发送给 Bob,Bob 将对元组的元素做同样的指数运算，Bob 将运算后的结果 (b, b') 返回给 Alice, 此时元组 b, b' 应该仍然满足偏移量 $b^\alpha = b'$
- 由于离散对数问题，Bob 除了使用暴力破解外无法从元组 b, b' 中提取出偏移量 α, 所以只有使用 Alice 给出的元组，并对元组进行同样的操作，Bob 才能提供有效的返回值。Bob 需要进行的操作是:
  - 随机选择幂指数 c
  - 计算 $b = a^c \ (mod\ n)$ 和 $b' = a'^c \ (mod\ n)$
  - 返回 (b, b')
- 收到返回值后，Alice 检查有效性：$b^a \stackrel{?}{=} b'$
- 结论:

- Bob 必须对元组中的两个数据做同样的幂指数操作
- Bob 必须使用 Alice 提供的数据。
- Bob 知道幂指数 c。
- Alice 无法知道 c, 除非她进行暴力破解。

这个方案向 Alice 证明 Bob 的确对 Alice 发送给他的数据做了指数操作，Bob 不能对元组进行乘法或者加法操作，因为这样将不满足偏移量检查。

我们将这个方法应用到同态加密中，对于一个只有一个系数的一阶多项式：$p(x) = c \cdot x$

- 验证者选择一个随机数 r 和偏移量 α, 发送给证明者 $E(r) = g^r$ 和 $[E(r)]^\alpha = g^{r \cdot \alpha}$ $(g^r, g^{r\alpha})$
- 证明者以他所知道的多项式 $p(x) = c \cdot x$ 的系数 c 对元组进行幂指数运算，得到 $((g^r)^c, (g^{r\alpha})^c) = (g^{cr}, g^{cr\alpha})$
- 验证者检查 $(g^{cr})^\alpha \stackrel{?}{=} g^{cr\alpha}$

这个结构约束证明者只能用验证者提供的加密的 r 进行计算，因而 prover 只能将系数 c 赋给 verifier 提供的单项式。现在我们可以扩展这种单项式上的方法到多项式上，因为计算是先将每项的分配分开计算然后再“同态地”相加在一起的。所以如果给定 prover 一个指数为 r 的幂以及它们的偏移的加密值，他就可以计算原始的和偏移后的加密的多项式，这里也必须要满足同样的校验。对于阶数为 d 的多项式：

- 验证者提供加密值 $g^{s^0}, g^{s^1}, ..., g^{s^d}$ 和它们的偏移：$g^{\alpha s^0}, g^{\alpha s^1}, ..., g^{\alpha s^d}$。
- 证明者
  - 由证明者所知道的多项式的系数，证明者计算多项式的加密结果：$g^{p(r)} = \left(g^{r^0}\right)^{c_0} \cdot \left(g^{r^1}\right)^{c_1} ... \left(g^{r^d}\right)^{c_d} = g^{c_0 r^0 + c_1 r^1 + ... + c_d r^d}$
  - 计算多项式的偏移加密结果：$g^{\alpha p(r)} = \left(g^{\alpha r^0}\right)^{c_0} \cdot \left(g^{\alpha r^1}\right)^{c_1} ... \left(g^{\alpha r^d}\right)^{c_d} = g^{c_0 \alpha r^0 + c_1 \alpha r^1 + ... + c_d \alpha r^d}$
  - 将 $g^p = g^{p(r)}$ 和 $g^{p'} = g^{\alpha p(r)}$ 发送给验证者。
- 证明者校验 $(g^p)^\alpha = g^{p'}$

现在我们就可以确保证明者是用了验证者提供的多项式而不是其它值做计算的了，因为那样做不能够保持 α 偏移量的检验。当然如果 verifier 想要确保在 prover 的多项式中排除了 s 的某些次幂，如 j，他就不提供对应的密文及其变换：

与前面的协议相比，我们现在已经有了一个比较健壮的协议。但是尽管已经做了加密，在零知识性质上也还依然存在一个很明显的缺陷：即理论上多项式参数 c 是一个很广的取值范围内的值，实际上这个范围可能很有限（比如前面例子中的 6），这就意味着 verifier 可以在有限范围的系数组合中进行暴力破解，最终计算出一个与 prover 的答案相等的结果。比如我们将每个系数的取值范围定为 100，多项式阶数为 2，那么大概会有 100 万种不同的组合，这里可以认为暴力破解只需要少于 100 万次的迭代。更重要的是，即使在只有一个系数，值为 1 的例子中，安全协议也应该能够保证其安全。

Remark: 验证者能从证明者发送过来的加密数据暴力破解出证明者所知道的多项式，而证明者却不能暴力破解出验证者发过来的加密的 r 的幂指数，这似乎有些矛盾。原因是：证明者所知道的多项式的系数理论上虽然有无限取值，但当我们计算的时候，可观察到它的取值范围被限制在模运算的模的大小上，比如 (费马小定理)：

$g^{(p-1)x} \ mod \ p = g^x \ mod \ p$

验证者可以根据从证明者那里得到的数据推算出一个多项式，虽然这个多项式的系数可能会与证明者所知道的多项式的系数不同，但本质上，验证者还是通过证明者的返回值，提取出了信息。这就不满足零知识证明的性质。

2.2.5 零知识

因为验证者能够从证明者发送的数据中提取未知多项式 $p(x)$ 的知识，那么我们就来看一下这些证明者提供的数据（证明）：

$g^{p(r)}$, $g^{p'(r)}$, $g^{h(r)}$

它们参与到了下面的验证：

$g^{p(r)} = \left(g^{h(r)}\right)^{t(r)}$ (多项式 $p(x)$ 有根 $t(x)$)

$\left(g^{p(r)}\right)^{\alpha} = g^{p'(r)}$ (用了正确形式的多项式)

问题是我们如何选择证明使得这个校验依然有效，同时又保证没有知识能被提取?

前面章节给了我们一个答案：我们可以使用随机值 δ 来做一下偏移，如 $(g^p)^\delta$ 。之前为了保证证明者使用了正确的多项式的形式，验证者做了一个偏移，现在为了防止验证者提取知识，我们仍然做一个偏移，因为做了偏移后，等式的校验依然能保持不变，但却能隐藏证明者提供的信息。验证者想要提取信息，就必须要知道一个不可知的值 δ。并且，这种随机化在统计学上与随机值没有什么区别。

我们观察到验证者雁阵的等式两边每一边都有一个证明者提供的值。所以如果我们对每一边都偏移 δ, 这个 δ 有证明者随机选取，验证者无法提取这个 δ 的值，但等式依然保持相等。

具体来讲，就是证明者选择一个随机值 δ，并用它对证明中的值进行求幂:$\left(g^{p(r)}\right)^\delta$, $\left(g^{p'(r)}\right)^\delta$, $\left(g^{h(r)}\right)^\delta$, 将这些值提供给验证者做验证：

$g^{\delta p(r)} = \left(g^{\delta h(r)}\right)^{t(r)}$ (多项式 $p(x)$ 有根 $t(x)$)

$\left(g^{\delta p(r)}\right)^\alpha = g^{\delta p'(r)}$ (用了正确形式的多项式)

校验依然成功。注意零知识是如何轻而易举地融入到这个结构中去的，这通常也被称为" 无成本的" 零知识。借助这个"无成本的"技巧，就可以轻松实现零知了。但是这里实现零知识的方法和实际中的 Pinocchio 协议，还有 Groth16 方案略有不同。实际方案中是用乘法乘以 g^p

2.3 非交互多项式零知识证明

到现在为止，我们已经讲完了一个交互式的零知识方案。但为什么我们还需要有非交互式呢? 因为交互式证明只对原始的验证者有效，其他任何人（其他的验证者）都不能够信任这个证明，因为：

- 验证者可以和证明者串通，告诉他密文参数 r, α，有了这些参数, 证明者就可以伪造证明，就像前面零知识证明 version 2 提到的那些漏洞一样。
- 验证者也可以使用同样的方法自己伪造证明。
- 验证者必须保存 α and $t(r)$ 直到所有相关证明被验证完毕，这就带来了一个可能造成秘密参数泄漏的额外攻击面。

因而证明者就需要分别和每个验证者做交互来证明一个陈述（就是例子中指的多项式的知识）。

尽管交互式证明也有它的用处，例如一个证明者只想让一个特定的验证者（称为目标验证者）确信，就不能再重复利用同一个证明去向别人证明这个声明了，但是当一个证明者想让众多的参与者同时或者永久地确信的话，这种方法就很低效了。证明者需要保持一直在线并且对每一个验证者执行相同的计算。

因而，我们就需要一个可以被重复使用，公开，可信，又不会被滥用的秘密参数。

我们先来思考一下如何在 $(t(r), \alpha)$ 构造之后保证它的安全性。我们使用公共的秘密参数对其进行加密，方式与验证者在发送加密值给证明者之前对 r 的幂使用的加密方式一致。但是上中提到，我们使用的同态加密并不支持两个秘密值相乘，这一点对 $t(s)$ 和 h 的加密值相乘以及 p 和 α 的加密值相乘的验证都很重要。这个问题适合用 Pairing 配对操作来解决。

Remark: 这里非交互的证明协议将对参数加密，但引入了两个问题：

1. 同态加密无法对两个加密值做乘法，那如何验证加密后的参数呢？
2. 加密值一旦泄露，协议的信任关系将无法保证，如何确保参数的安全性？

2.3.1 加密值的相乘

配对操作（双线性映射）是一个数学结构，表示为函数 $e(g, g)$，它给定一个数据集中的两个加密的输入（即 g^a, g^b ），可以将加密值 a, b 的乘积确定性地映射到另一组不同的输出数据集上，即 $e(g^a, g^b) = e(g, g)^{ab}$ ：

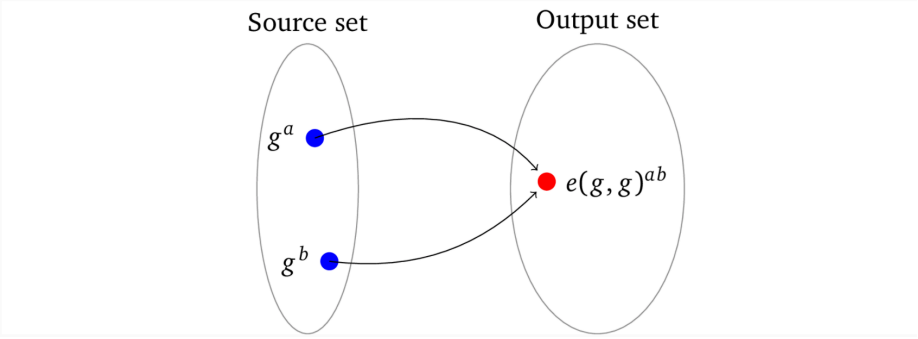


图 4: 双线性映射

因为源数据集和输出数据集（通常被称为一个 group）是不同的，所以一个配对的结果不能用做其他配对计算的输入。也就是说, $e(g,g)^a \cdot g^b$ 是不合法的，因为两个乘数不在一个数据集内。我们可以将输出集（也称为“目标集”）视为“不同的宇宙”。因而我们不能用另一个加密值乘以结果，而且配对这个名称本身也表明了，我们一次只能将两个加密值相乘。换句话说，配对只支持 $x \cdot y$ 这种两个值的乘法，但不支持三个或以上的值相乘，比如不支持 $x \cdot y \cdot z$。

在某种意义上，这个类似于一个哈希函数，他将所有可能的输入值映射到可能的输出值的集合中的一个元素上，通常情况下这个过程是不可逆的。

注意：乍一眼看过去，这个限制可能会阻碍相关功能的实现，但在 zk-SNARK 中这反而是保证安全模式的最重要性质。

配对的核心性质可以表示成下面的等式：

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

严格来讲一个配对的结果是在目标集的一个不同生成元 g 下对原始值乘积的加密，即 $e(g^a, g^b) = g^ab$。因而它具备同态加密的性质，也就是说我们可以把乘法配对的加密乘积放到一起：

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

注意：配对操作是通过改变椭圆曲线来实现这些性质的，现在我们用的符号 g^n 就代表曲线上一个由生成元自相加了 n 次的点，而不是我们前面用到的乘法群生成元。

2.3.2 可信任参与方的 setup

有了配对，我们现在就准备去设置安全公开且可复用的参数了。假定一下我们让一个诚实的参与方来生成秘密值 r 和 α。只要 α 和所有必要的 r 的幂及其对应的 α 偏移生成并被加密了，那么原始数据就必须要被删除 $(g^\alpha, g^{r^i}, g^{\alpha r^i}$ r 为 0 1, ..., d)。

这些参数通常被称为 common reference string 或者 CRS。CRS 生成后，任何的证明者和任何的验证者都可以使用它来构造非交互式的零知识证明协议。CRS 的优化版本将包含目标多项式的加密值 $g^t(r)$ ，尽管这个值并不重要。

把 CRS 分成两组（i 为 0 1, ..., d ）：

- 证明者需要使用的 proving key（也被称为 evaluation key: $(g^{r^i}, g^{\alpha r^i})$

- 验证者需要使用的 verification key:$(g^t(r), g^\alpha)$

只要能够加密值能够做乘法（之前我们在 remark 提到，在不使用双线性映射的情况下，只有两个未知数的加密值，我们无法计算这两个数的积的加密值），验证者就可以在协议的最后一步验证多项式了：

有了 verification key，验证者就可以处理从证明者那里得到的加密多项式的值 $g^{p(r)}, g^{h(r)}, g^{(p'(r))}$

- 在加密空间中校验 $\mathrm{p} = \mathrm{t} \cdot \mathrm{h}$：

  $e(g^{p(r)}, g^1) = e(g^{t(r)}, g^{h(r)})$

- 校验多项式的限制：

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

2.3.3 信任任意一个参与者

尽管受信任设置很有效率，但众多 CRS 用户也必须要相信生成者确实删除了 α 和 r ，这一点没有办法证明（proof of ignorance 是一个正在积极研究的领域，所以这种方法依然是无效的。因而很有必要去最小化或者消除这种信任。否则一个不诚实的参与方就可以构造假证明而不被发现。

一种解决办法就是由多个参与方使用前面小节中介绍的数学工具来生成一个组合式 CRS（多方计算），这样这些参与方就都不知道“秘密”了。下面是一个实现方案，我们假设有三个参与者 Alice，Bob 和 Carol ，对应为 A，B 和 C，其中 i 为 $1, 2, ..., d$：

- Alice 选择随机数 r_A 和 α_A，然后公开她的 CRS：

  $(g^{r_A^i}, g^{\alpha_A}, g^{\alpha_A r_A^i})$

- Bob 选择他的随机数 r_B 和 α_B, 然后通过同态乘法结合 Alice 的 CRS：

  $((g^{r_A^i})^{s_B^i}, (g^{\alpha_A})^{\alpha_B}, (g^{\alpha r^i})^{\alpha_B}) = (g^{(r_A r_B)^i}, g^{\alpha_A \alpha_B}, g^{\alpha_A \alpha_B (r_A r_B)^i})$

  然后公开两方 Alice-Bob 的 CRS 结果：

  $(g^{r_{AB}^i}, g^{\alpha_{AB}}, g^{\alpha_{AB} r_{AB}^i})$

- Carol 用她的随机数 r_C 和 α_C 做同样的事：

  $((g^{r_{AB}^i})^{r_C^i}, (g^{\alpha_{AB}})^{\alpha_C}, (g^{\alpha_{AB} r_{AB}^i})^{\alpha_C r_C^i}) = (g^{(r_A r_B r_C)^i}, g^{\alpha_A \alpha_B \alpha_C}, g^{\alpha_A \alpha_B \alpha_C (r_A r_B r_C)^i}$

然后公开 Alice-Bob-Carol 的 CRS:

$(g^{r^i_{ABC}}, g^{\alpha_{ABC}}, g^{\alpha_{ABC} r^i_{ABC}})$

- 这个协议最后我们就获得了一个混合的 r^i 和 α:

  $r^i = r^i_A r^i_B r^i_C, \alpha = \alpha_A \alpha_B \alpha_C$

除非他们串谋，否则参与者们互相之间并不知道其他人的秘密参数。实际上，一个参与者必须要和其它所有的参与者串谋才能得到 r 和 α，这样在所有的参与者中只要有一个是诚实的，就没有办法伪造证明。这个过程可以被尽可能多的参与者重复完成。

有一个问题是如何验证参与者在生成 CRS 时用的随机数值是一致的，因为攻击者可以生成多个不同的随机数 $r_1, r_2, \ldots$ 和 $\alpha_1, \alpha_2, \ldots$ 然后代入这些不同的随机数去执行 r 的不同次幂计算（或提供随机数作为一个 CRS 的扩充），从而使 CRS 无效或者不可用。

庆幸的是，因为我们可以使用配对来乘以加密值，所以我们就可以从第一个参数开始逐一执行一致性校验，并且确保了每个参数都源于前一个。

- 我们用 r 的 1 次幂作为标准来校验每一个其它次幂的值与之是否保持一致

  $e(g^{r^i}, g) = e(g^{r^1}, g^{r^{i-1}})|_{i \in 2,\ldots,d}$

- 在再验证一下前面步骤中 α 偏移后的值是否正确:

  $e(g^{r^i}, g^{\alpha}) = e(g^{\alpha r^i}, g)|_{i \in [d]}$

这里 $i \in 2, \ldots\ldots, d$ 是“i 值分别为 2 3 … d”的缩写，$[d]$ 是 $1, 2, \ldots, d$ 这个范围的缩写形式，在后面的章节这种表示方式更为常见。

当我们在验证每一个参与者秘密参数的一致性时，要注意参与者生成 CRS 的过程并没有强制后一个参与者（就是我们例子中的 Bob 和 Carol）都要使用前面已经公开的 CRS。因而如果一个攻击者是链上的最后一个参与者，他可以像链上的第一个参与者一样, 自己随便构造一个有效的 CRS，而忽略前面的 CRS。这样他就变成了唯一一个知道秘密 r 和 α 的人。

为了解决这个问题，我们可以额外再要求除了第一个以外的每一个参与者去加密然后公开他的参数。例如，Bob 同样公开了:

$(g^{r^i_B}, g^{\alpha_B}, g^{\alpha_B r^i_B})|_{i \in [d]}$

这就可以去验证 Bob 的 CRS 是乘以了 Alice 的参数后正常获得的，这里 i 为 $1, 2, \ldots, d$。

- $e(g^{r^i_{AB}}, g) = e(g^{r^i_A}, g^{r^i_B})$
- $e(g^{\alpha_{AB}}, g) = e(g^{\alpha_A}, g^{\alpha_B})$
- $e(g^{\alpha_{AB} r^i_{AB}}, g) = e(g^{\alpha_A r^i_A}, g^{\alpha_B r^i_B})$

同样的，Carol 也必须证明她的 CRS 是乘以了 Alice-Bob 的 CRS 后正常获得的。

这是一个健壮的 CRS 设置模式，它并不完全依赖于单个参与者。事实上，即使其它所有的参与者都串谋了，只要有一个参与者是诚实的，他能够删除并且永远不共享它的秘密参数，这个 CRS 就是有效的。所以在设置 CRS 有越多不相关的参与者参与，伪造证明的可能性就越低。当有相互竞争的参与方参与的时候，就几乎不可能伪造证明了。

现在有一些 zkSNARK 方案支持可升级的 CRS，任何怀疑 CRS 的参与方都可以对 CRS 进行更新。此外还有一些 zkSNARK 方案支持 Universal CRS，用不着对每一个电路进行受信任设置，而是只需要全局完成一次即可。除此之外，大量无需 Trusted Setup 的方案正在被充分研究。

2.4 多项式的 SNARK

我们现在准备来合并这个逐步演化出来的 zk-SNARKOP(Succinct Non-Interactive Argument of Knowledge of Polynomial) 协议。为简洁起见，我们将使用大括号来表示由旁边的下标填充的一组元素，例如：$\{r^i\}|_{i\in[d]}$ 表示一组数 $r^1, r^2, ..., s^d$.。

当证明着和验证者达成一致选取了目标多项式 $t(x)$ 和证明者的多项式阶数 d:

- Setup (多方计算运行)
  - 挑选随机值 r, α
  - 计算加密值 g^α 和 $\{g^{r^i}\}|_{i\in[d]}, \{g^{\alpha r^i}\}|_{i\in[d]}$
  - 生成 proving key：$(\{g^{r^i}\}|_{i\in[d]}, \{g^{\alpha r^i}\}|_{i\in[d]})$
  - 生成 verification key: $(g^\alpha, g^{t(r)})$
- Proving（证明者运行）

- 分配系数 $\{c_i\}|_{i\in\{0,1,...,d\}}$(即知识) 得到 $p(x) = c_d x^d + c_{d-1}x^{d-1} + ... + c_1 x^1 + c_0 x^0$.
- 求多项式 $h(x) = p(x)/t(x)$
- 由 proving key 中的 $\{g^{r^i}\}|_{i\in[d]}$ 和证明者所知道的 $p(x), h(x)$ 的系数，计算多项式加密值 $g^{p(r)}, g^{h(r)}$
- 由 proving key 中的 $\{g^{\alpha r^i}\}|_{i\in[d]}$ 和 $\{c_i\}|_{i\in\{0,1,...,d\}}$ 计算 $g^{\alpha p(r)}$ 的值。
- 选择随机数 δ
- 构造随机化的证明 $\pi = (g^{\delta p(r)}, g^{\delta h(r)}, g^{\delta \alpha p(r)})$

- Verification (验证者做验证)
  - 映射证明 π 为 $g^p, g^h, g^{p'}$,verification key:$(g^\alpha, g^{t(r)})$
  - 验证多项式约束 $e(g^{p'}, g) = e(g^\alpha, g^p)$
  - 验证多项式系数 $e(g^p, g) = e(g^{t(r)}, g^h)$

2.5 结论

我们用 zk-SNARK 协议来解决多项式问题的知识，不过这是一个有局限的例子。因为大家可以说 prover 只要用另外一个有界的多项式去乘以 $t(x)$ 就可以很容易得构造出一个能够通过测试的多项式 $p(x)$ ，并且这种结构也是有效的。

验证者知道证明者有一个有效的多项式，但是并不知道是哪一个。我们可以利用增加额外的证明来证明证明者知道的多项式有某些特定的性质，如额外证明: 被多个多项式整除，是某个多项式的平方。做好有一个通用的方式能来支持无数的应用。

总结一下这篇文章中一步一步解决了下面的几个问题 [7]:

保证 prover 的证明是按照规则正确构造的——>α 偏移（KEA）

保证知识的零知性——> “无成本的”δ 变换

可复用证明——> 非交互式

非交互中如何设置安全公开且可复用的参数——> 参数加密，verifier 借助密码配对进行验证

保证参数的生成者不泄密——> 多方的 Setup

至此，一个用来证明多项式知识的完整的 zk-SNARK 协议就构造出来了，不过现在的协议在通用性上依然还有很多限制，后面的文章将继续介绍如何构造通用的 zk-SNARK。

3 zk-SNARK 算法原理 2（从程序到多项式的构造）

以上章节我们介绍了多项式的非交互式零知识证明，但我们在应用的时候并不会去证明“某人是否知道一个多项式”，而是去证明“某人是否知道一个秘密的值，这个值满足一定的条件（比如哈希值为 x）”。那如何把证明“某人是否知道一个秘密的值，这个值满足一定的条件”的问题转化成“某人是否知道一个多项式”的问题呢？

我们可以写一个程序来检验这个值是不是满足条件，如果满足，则输出 1，不满足则输出 0. 而通过一些人们已经研究出的方法，程序可以转换成电路，电路的输入输出就是程序的输入输出。而电路又能转换成有特定零根的多项式（多项式的系数和电路的每个输入输出都有关）。验证者可以知道程序的代码，也可以知道转化而来的电路和由该电路转化的多项式应有的零根。这种情况下，验证证明者是否知道某一个秘密值，就成了验证证明者是否知道某一个有特定根的多项式。

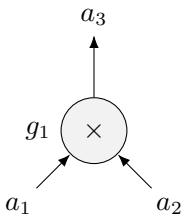
Remark: 这里你可能会有疑问，如果验证者只知道多项式的零根，那么证明者很容易就能构造出一个满足条件的多项式。但其实不然，验证者还知道电路的构造，这个电路的构造以及 trust set up 选定多项式必须经过某一点，这两个条件限制了证明者只有用满足电路输入输出的值才能构造出有效的多项式。所以这一章节我们就来介绍:

(1) 如何将一个程序转化成电路。

(2) 如何将一个电路转化成多项式。

3.0.1 电路转化成多项式

我们由实例电路来讲解如何将一个电路转化为多项式。先从最简单的，只有一个乘法门的电路开始。



为了用多项式表达电路，我们先将一个电路的输入输出都表示成多项式的形式，并且将一个电路的输入分为左输入和右输入，定义电路 C 的左输入为 $A(x)$, 右输入为 $B(x)$，输出为 $C(x)$。而我们需要寻找的多项式的形式为 $P(x) = A(x) \cdot B(x) - C(x)$,(我们先不在意这个输入变量 x，也不在意为什么要写成这种形式，接下来的讲解会有答案)，我们将来讲解如何将一个电路映射成为这样一个多项式。

将电路实际的输入输出和电路中的“导线”标记为 $a_1, a_2, ..., a_n$。为了将电路映射为多项式，也就是把电路映射为多项式的系数，我们使用 $a_1, a_2, ..., a_n$ 来表达多项式，也就是：

$A(x) = A_0(x) + a_1A_1(x) + ... + a_nA_n(x)$

$B(x) = B_0(x) + a_1B_1(x) + ... + a_nB_n(x)$

$C(x) = C_0(x) + a_1C_1(x) + ... + a_nC_n(x)$

(注意每个导线的“系数”都是一个关于 x 的函数 $A_0(x), B_0(x), ..., C_0(x)...$)

我们将电路的导线都映射到了多项式的系数中，但是只有这些“导线”显然不够表示这个电路。我们还需要表达出电路中的各个门对“导线”的约束：比如图中一个乘法门的约束为 $a_1 \cdot a_2 = a_3$, 我们需要让我们构造的多项式能够通过某种方式，表达这个约束条件。

我们观察到图中只有一个乘法门的电路可以直观的构造：

$A(x) = a_1x$

$B(x) = a_2x$

$C(x) = a_3x^2$

多项式为 $P(x) = a_1x \cdot a_2x - a_3x^2$

当多项式 $P(x)$ 取零根 $x = 1$ 的时候，多项式取值为

$P(1) = A(1) \cdot B(1) - C(1) = a_1a_2 - a_3 = 0$

这恰恰对应了电路中的乘法门 g_1 对输入输出的约束条件 $a_1 \cdot a_2 = a_3$。也就是我们构造的思路是：**希望多项式的每一个零根都能转化为电路中某一个门对应的约束条件**。那么这个多项式中的 $a_1, a_2, ..., a_n$ 值就能满足电路所有的约束条件，这个多项式就能表达这个电路的一个合法 $a_1, a_2, ..., a_n$ 的

取值实例。这个零根可以我们自己设定，上图的例子，我们也可以设多项式的零根为 2, 那么电路的输入输出多项式就表达为:

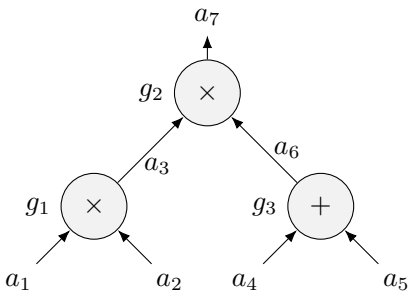
$A(x) = \frac{a_1}{2}x$

$B(x) = \frac{a_2}{2}x$

$C(x) = \frac{a_3}{4}x^2$

本质上，不论我们选取哪个店作为零根，证明者要一定要知道一组 a_1, a_2, a_3 使得 $a_1 \cdot a_2 = a_3$ 才能构造出合法的多项式。

理解构造的思想之后，我们再来看另一个复杂的电路的例子:



我们依然将电路实际的输入输出和电路中的“导线”标记为 $a_1, a_2, ..., a_n$ 来表达多项式:

$A(x) = A_0(x) + a_1A_1(x) + ... + a_7A_n(x)$

$B(x) = B_0(x) + a_1B_1(x) + ... + a_7B_n(x)$

$C(x) = C_0(x) + a_1C_1(x) + ... + a_7C_n(x)$

电路中有三个门，也就是要有三个零根，我们一个一个来看。

第一个门 g_1, 我们设对应这个门的零根为 1。约束条件为 $a_1 \cdot a_2 = a_3$ 我们得到:

$A_1(1) = a_1$,$B_2(1) = a_2$,$C_3(1) = a_3$ 其他值为 0。

第二个门 g_2, 我们设对应零根为 2。约束条件为 $a_3 \cdot a_6 = a_7$, 我们得到:

$A_3(2) = a_3$,$B_6(2) = a_6$,$C_7(2) = a_7$ 其他值为 0。

第三个门，对应零根为 3，这是个加法门，对应约束为 $a_4 + a_5 = a_6$ 所以我们把两个输入 a_4, a_5 都放在一边:

$A_4(3) = a_4, A_5(3) = a_5, B_0(3) = 1, C_6(3) = a_6$ 其他值为 0。

综上:

函数 $A_0(x)$ 要经过的点为 $(1, 0), (2, 0), (3, 0)$

函数 $A_1(x)$ 要经过的点为 $(1, a_1), (2, 0), (3, 0)$

函数 $A_2(x)$ 要经过的点为 $(1, 0), (2, 0), (3, 0)$

函数 $A_3(x)$ 要经过的点为 $(1,0),(2,a_3),(3,0)$

函数 $A_4(x)$ 要经过的点为 $(1,0),(2,0),(3,a_4)$

函数 $A_5(x)$ 要经过的点为 $(1,0),(2,0),(3,a_5)$

函数 $A_6(x)$ 要经过的点为 $(1,0),(2,0),(3,0)$

函数 $A_7(x)$ 要经过的点为 $(1,0),(2,0),(3,0)$

由拉格朗日插值法，我们可以得到满足以上条件的多项式:

$A_0(x) = 0$

$A_1(x) = \dfrac{a_1(x-2)(x-3)}{2}$

$A_2(x) = 0$

$A_3(x) = -a_3(x-1)(x-3)$

$A_4(x) = \dfrac{a_4(x-1)(x-2)}{2}$

$A_5(x) = \dfrac{a_5(x-1)(x-2)}{2}$

$A_6(x) = 0$

$A_7(x) = 0$

于是得到:

$$A(x) = \frac{a_1(x-2)(x-3)}{2} - a_3(x-1)(x-3) + \frac{a_4(x-1)(x-2)}{2} + \frac{a_5(x-1)(x-2)}{2}$$

同理可得:

$$B(x) = \frac{a_2(x-2)(x-3)}{2} - a_6(x-1)(x-3) + (x-1)(x-2)$$

$$C(x) = \frac{a_3(x-2)(x-3)}{2} - a_7(x-1)(x-3) + \frac{a_6(x-1)(x-2)}{2}$$

得到的多项式：$P(x) = A(x)B(x) - C(x)$ 有零根 $1,2,3$，即:

$P(1) = a_1 \cdot a_2 - a_3 = 0$

$P(2) = a_3 \cdot a_6 - a_7 = 0$

$P(3) = a_4 + a_5 - a_6 = 0$

证明者只要知道一组合法的导线值，就能构造出 $P(x)$ 有零根 $1,2,3$。

也就是，我们将" 证明某人是否知道一个电路的有效输入输出" 转化为了"某人是否知道有某些零根的特定结构的多项式"

3.0.2 程序转化成电路

理论上，任何涉及计算的程序都能够转化为电路，zk-snark 的代码库 libsnark 抽象出了 protoboard 和 gadget 方便开发者搭建电路。描述电路的语言为 $R1CS$，如何将程序转化成电路，还要对 libsnark 做具体的实例讲解，这篇文章只涉及 zk-SNARK 的理论讲解，所以这部分先省略过不谈。

4 算法原理（直接的数学表达）

在理解了 zk-SNARK 的构造思路后，这部分用数学语言 [5] 对算法进行了综合描述，并给出了一个具体的例子。使读者能有一个更全面的理解。

4.1 算数电路和二次算数电路张成方案

算数电路类似于布尔电路，我们可以用门还有'导线'组成的电路图来表示。把每个门的输入和输出都赋予一个变量。zk-SNARK 的思想就是利用二次算数电路张成方案 (quadratic airthmetic programs) 来将电路转化成一个多项式

4.2 算数电路

设一个映射 C 为 $C : \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$。这是一个将 $n+h$ 个来自有限域 $\mathbb{F}$ 的输入映射为同一值域 $\mathbb{F}$ 的 l 个输出。如果映射的输出由输入和对这些输入进行的计算操作 $+$ 或者 $\times$ 决定（允许有常量门），则映射 C 是一个算数电路。

一个对算数电路 C 有效的赋值是一个元组 $(a_1, ..., a_N) \in \mathbb{F}^N$, $N = (n+h)+l$ 是电路所有的输入输出的总和，满足 $C = (a_1, ..., a_{n+h}) = (a_{n+h+1}, ..., a_N)$

4.3 二次算数电路张成方案

二次算数方案张成在 zk-SNARK 中的作用是一个构造证明 π 的工具，这个证明 π 声明了某人知道一个算数电路 C 的有效输入元组 $(a_1, ..., a_N) \in \mathbb{F}^N$. 实际上，一个 QAP $Q(C) := (\vec{A}, \vec{B}, \vec{C}, Z)$ 向一个算数电路 C 提供了三个多项式集合：

$\vec{A} = (A_i(z))_{i=0}^m$, $\vec{B} = (B_i(z))_{i=0}^m$, $\vec{C} = (C_i(z))_{i=0}^m$ $(m \geq N)$

和算数电路的输入输出 $(a_1, ..., a_N) \in \mathbb{F}^N$ 以及一个目标多项式 $Z(z) \in F[z]$ 结合，多项式 $Z(z)$ 能整除多项式：

$$P(z) := \underbrace{(A_0(z) + \sum_{i=1}^{m} a_i A_i(z))}_{:=A(z)} \underbrace{(B_0(z) + \sum_{i=1}^{m} a_i B_i(z))}_{:=B(z)} - \underbrace{(C_0(z) + \sum_{i=1}^{m} a_i C_i(z))}_{:=C(z)}$$

当且仅当 $(a_1, ..., a_N) \in \mathbb{F}^N$ 是算数电路 C 有效输入输出的时候，这个多项式才能被证明者构造出来，验证者也能“轻松地”验证多项式 $P(z)$ 是否能被 $Z(z)$ 整除。

4.3.1 构造 QAP

这部分介绍如何构造一个算数电路的 QAP

我们先给出一个算数电路的例子:

$C(x_1, x_2, x_3, x_4) = ((x_1 + 7x_2)(x_2 - x_3), (x_2 - x_3)(x_4 + 1))$

这个电路的一个有效输入是 $(0, 1, 1, 1, 0, 0)$

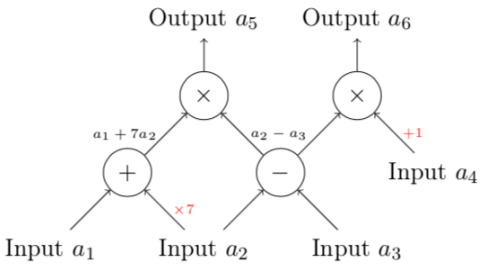


图 5: HTLC

1. 准备阶段：以一个输出门都为乘法门的电路为例，设 M 为包含该电路的所有乘法门的集合，集合中的乘法门以其输出变量作为索引，图示电路中两个乘法门分别为 g_5,g_6，则 $M = \{g_5, g_6\}$. 设 W 为包含电路的输入变量和电路乘法门的输出变量的集合 (the set of special wires)，$I_{g,L}$ 表示从左边输入的门 g 的输入边的集合,$I_{g,L}$ 表示从右边输入的门 g 的输入边的集合.

   则图示电路的集合:

   $W = \{a_1, a_2, a_3, a_4, a_5, a_6\}$,

   $I_{g_5,L} = \{a_1, a_2\}$ $\quad$ $I_{g_6,L} = \{a_2\ a_3\}$

   $I_{g_5,R} = \{a_2, a_3\}$ $\quad$ $I_{g_6,R} = \{a_4\}$

2. 目标多项式：设目标多项式为 $Z(z) = \Pi_{g \in M}(z - r_g)$，其零根为 r_i，$r_i \in \mathbb{F}$ 并且互不相等。

3. 左输入多项式和右输入多项式：电路的左输入多项式集合 $\vec{A}$ 和右输入多项式 $\vec{B}$ 由多项式在零根处的取值决定。

   乘法门左输入多项式在各个零根的取值：

   $A_i(r_g) = c_{g,L,i}$ if $i \in I_{g,L}$ otherwise $A_i(r_g) = 0$

   $B_i(r_g) = c_{g,R,i}$ if $i \in I_{g,R}$ otherwise $B_i(r_g) = 0$

   $C_{g,L,i}$ 表示输入门 g 的第 i 导线的系数。输入 g 的标量由 $A_0(r_g)$ 和 $B_0(r_g)$ 表示。

4. 输出多项式：构造输出多项式集合 $\vec{C}$ 使得：

   $C_i(r_g) = 1$ if $i = g$ otherwise $C_i(r_g) = 0$

   我们构造的多项式集合 $\vec{A}$,$\vec{B}$,$\vec{C}$ 就能表示电路中所有的门 $g \in M$

$$A(r_g) = A_0(r_g) + \sum_{i=1}^{m} a_i A_i(r_g) = A_0(r_g) + \sum_{i \in I_{g,L}} a_i c_{g,L,i}$$

$$B(r_g) = B_0(r_g) + \sum_{i=1}^{m} a_i B_i(r_g) = B_0(r_g) + \sum_{i \in I_{g,L}} a_i c_{g,R,i}$$

$$C(r_g) = C_0(r_g) + \sum_{i=1}^{m} a_i C_i(r_g) = a_g$$

   也就是说：

   $P(r_g) = A(r_g) \cdot B(r_g) - C(r_g) = 0$ for all gate $g \in M$

4.4 zk-SNARK

秘钥生成算法 key generation

input: Arithmetic circuit $C : \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$

1. Construct the QAP $Q(C) = (\vec{A}, \vec{B}, \vec{C}, Z)$ of C
2. Randomly sample $\tau, \rho_A\ \rho_B \in \mathbb{F}$, Set $\rho_C = \rho_A \cdot \rho_B$
3. Determine the coefficients $(h_i)_i^d = 0$ of $H(z) = \dfrac{A(z) \cdot B(z) - C(z)}{Z(z)}$

4. Generate the proving key $pk := (pk_A, pk_B, pk_C, pk_H)$ where

   $pk_A := (\underbrace{A_i(\tau)\rho_A P_1}_{pk_{A,i}})_{i=0}^{m}$ $pk_B := (\underbrace{B_i(\tau)\rho_B P_2}_{pk_{B,i}})_{i=0}^{m}$ $pk_C := (\underbrace{C_i(\tau)\rho_C P_1}_{pk_{C,i}})_{i=0}^{m}$

   $pk_H := (\underbrace{\tau^i P_1}_{pk_{H,i}})_{i=0}^{d}$

   Randomly sample $\alpha_A, \alpha_B\ \alpha_C \in \mathbb{F}$

   $pk'_A := (\underbrace{\alpha_A A_i(\tau)\rho_A P_1}_{pk_{A,i}})_{i=0}^{m}$ $pk'_B := (\underbrace{\alpha_B B_i(\tau)\rho_B P_2}_{pk_{B,i}})_{i=0}^{m}$

   $pk'_C := (\underbrace{\alpha_C C_i(\tau)\rho_C P_1}_{pk_{C,i}})_{i=0}^{m}$

5. Randomly sample $\beta, \gamma \in \mathbb{F}$

   Proving key:

   $pk_K = \underbrace{(\beta(\rho_A A_i(\tau) + \rho_B B_i(\tau) + \rho_C C_i(\tau))P_1)_{i=0}^{m}}_{pk_{K,i}}$

6. Generate the verification key :

   $vk_{IC} := (\underbrace{A_i(\tau)\rho_A P_1}_{vk_{IC,i}})_{i=0}^{n}$ $vk_Z = Z(\tau)\rho_C P_2$

   $vk_\gamma := \gamma P_2, vk^1_{\gamma\beta} := \gamma\beta P_1, vk^2_{\gamma\beta} := \gamma\beta P_2$

7. output:

   $pk_A, pk'_A, pk_B, pk'_B, pk_C, pk'_C, pk_H, pk_K$

   $vk_{IC}, vk_Z, vk_\gamma, vk^1_{\gamma\beta}, vk^2_{\gamma\beta}$

证明算法 Prover input: $pk, \vec{x} \in \mathbb{F}^n$, and $\vec{w} \in \mathbb{F}^h$

1. Randomly sample $\delta_1, \delta_2, \delta_3 \in \mathbb{F}$ set

$$A(z) = A_0(z) + \sum_{i=1}^{m} a_i A_i(z) + \delta_1 Z(z)$$

$$B(z) = B_0(z) + \sum_{i=1}^{m} a_i B_i(z) + \delta_2 Z(z)$$

$$C(z) = C_0(z) + \sum_{i=1}^{m} a_i C_i(z) + \delta_3 Z(z)$$

2. Construct the QAP $Q(C) = (\vec{A}, \vec{B}, \vec{C}, Z)$ of C
3. Compute a valid distribution $(a_1, ..., a_m) = QAPwit(C, \vec{x}, \vec{w})$
4. Determine the coefficients $(h_i)_i^d = 0$ of $H(z) = \dfrac{A(z) \cdot B(z) - C(z)}{Z(z)}$
5. Construct the proof $\pi := (\pi_A, \pi_B, \pi_C, \pi_H)$ where

$$\pi_A := \sum_{i=n+1}^{m} a_i pk_{A,i}$$

,

$$\pi_B := pk_{B,0} + \sum_{i=1}^{m} a_i pk_{B,i}$$

$$\pi_C := pk_{C,0} + \sum_{i=1}^{m} a_i pk_{C,i}$$

$$\pi_H := pk_{H,0} + \sum_{i=1}^{d} h_i pk_{H,i}$$

$$\pi'_A := \sum_{i=n+1}^{m} a_i pk'_{A,i}$$

,

$$\pi'_B := pk'_{B,0} + \sum_{i=1}^{m} a_i pk'_{B,i}$$

$$\pi'_C := pk'_{C,0} + \sum_{i=1}^{m} a_i pk'_{C,i}$$

$$\pi_K := pk_{K,0} + \sum_{i=1}^{m} a_i pk_{K,i}$$

6. Output: proof π of the statement "$\vec{x} \in L_C$"

验证算法 Verifier

1. 计算

$$vk_{\vec{x}} = vk_{IC,0} + \sum_{i=1}^{n} x_i vk_{IC,i}$$

2. 验证等式:

$$e(vk_{\vec{x}} + \pi_A, \pi_B) = e(\pi_H, vk_Z) \cdot e(\pi_C, P_2)$$

$$e(\pi_A, vk_A) = e(\pi'_A, P_2)$$

$$e(\pi_B, vk_B) = e(\pi'_B, P_2)$$

$$e(\pi_C, vk_C) = e(\pi'_C, P_2)$$

$$e(\pi_K, vk_\gamma) = e(vk_{\vec{x}} + \pi_A + \pi_C, vk^2_{\gamma\rho}) \cdot e(vk^1_{\gamma\rho}, \pi_B)$$

5 结论

zk-SNARK 十分复杂，但是这个算法功能强大，在隐私区块链上有很多应用。后续将对如何使用算法的库 libsnark 做调研。

参考文献

[1] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. `https://eprint.iacr.org/2018/046`.

[2] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Report 2017/1066, 2017. `https://eprint.iacr.org/2017/1066`.

[3] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct nizks without pcps. Cryptology ePrint Archive, Report 2012/215, 2012. `https://eprint.iacr.org/2012/215`.

[4] J. Groth. On the size of pairing-based non-interactive arguments. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49896-5.

[5] H. Mayer. zk-snark explained: Basic principles. URL `https://blog.coinfabrik.com/wp-content/uploads/2017/03/zkSNARK-explained_basic_principles.pdf`.

[6] M. Petkus. Why and how zk-snark works. *CoRR*, abs/1906.07221, 2019. URL `http://arxiv.org/abs/1906.07221`.

[7] 安比实验室. 从零开始学习 zk-snark. URL `https://learnblockchain.cn/article/287`.

[8] 逐舞传歌. URL `https://www.jianshu.com/p/7b772e5cdaef`.