Introducing Shamir's Secret Sharing

longcpp

longcpp9@gmail.com

March 18, 2020

Shamir' s Secret Sharing (SSS) 是 Adi Shamir 提出的算法, 用来将秘密信息进行分割成多个秘密份额, 给每个参与者一份独特的秘密份额 (Share). 需要重构秘密信息的时候, 需要一部分参与 (有最少个数要求, 也即阈值) 提供各自的秘密份额. 如果密钥份额的个数小于阈值, 则无法获得关于原始秘密的任何信息. 为方便讨论, 用 t-of-n SSS 表示将秘密信息分割成 n 份, 阈值为 t 的 SSS 方案, 也即最少需要 t 份秘密份额才能重构出原始的秘密信息.

通过将秘密信息进行分割, 可以避免单独的参与方能够完全控制秘密信息, 实现分权管理. 另外也实现了对秘密信息的冗余备份, 这是因为通常选择的阈值 t 小于 n, 即使有的参与方不小心丢失了自己的秘密份额, 只要还有 t 个参与方能够提供其的秘密份额就仍可以恢复出原始秘密信息, 避免了单点失败的风险. 基于同样的思路, 只要系统中被共谋作恶的参与方小于阈值, 即可保证恶人无法获取关于秘密信息的任何知识.

SSS 方案的核心思想是两点定线, 三点定圆, 或者更通用的说法: t 个点可以唯一确定次数为 $t-1$ 的多项式. 假设秘密信息是有限域 $\mathbb{F}$ 中的元素 s, 为了构建 t-of-n SSS 方案, 其中 $0 < t \leq n$, 可以随机选择 $a_1, ..., a_{t-1} \in_R \mathbb{F}$, 并且设置 $a_0 = s$, 则可以构建 $\mathbb{F}$ 上的多项式

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{t-1} x^{t-1}.$$

为了生成 n 份秘密份额, 设置 $i = 1, \ldots, n$, 并计算 $f(i)$, 则每个参与者获得的秘密份额为 $(i, f(i))$. 注意到 $f(x)$ 的常数项即为秘密信息 s, 所以为了重构出秘密信息, 只需要重构出多项式 $f(x)$ 即可. 由于 k 个点可以唯一确定阶为 $t-1$ 的多项式, 所以只需要 t 个秘密份额 $(i, f(i))$. 利用插值 (Interpolation) 方法即可重构多项式 $f(x)$, 从而重构秘密信息 s. 给定点的集合 $(x_0, y_0), \ldots, (x_j, y_j), \ldots, (x_{t-1}, y_{t-1})$, 则拉格朗日形式的插值多项式为

$$f(x) = \sum_{i=0}^{t-1} y_i \ell_i(x), \ \ell_i(x) = \prod_{0 \leq j \leq t-1, i \neq j} \frac{x - x_j}{x_i - x_j}$$

另外考虑到最终只需要 $f(0)$ 的值, 可以采用下面算式直接计算 $f(0)$ 的值:

$$f(0) = \sum_{i=0}^{t-1} y_i \prod_{j=0, j\neq i}^{t-1} \frac{x_j}{x_j - x_i}$$

在具体实现时候则需要考虑如何选取有限域 $\mathbb{F}$. 提供 128 比特安全性的分组密码的密钥长度通常为 128 比特, 所以理想情况下有限域 $\mathbb{F}$ 应该能够处理所有可能的秘密信息. 自然的想法是选取一个 128 比特的尽可能大的素数. 128 比特所能表示的最大的素数是 $2^{128} - 159$. 有限域 $\mathbb{F}_{2^{128}-159}$ 可以在绝大多数情况下满足我们想要处理 128 比特秘密信息的需求, 但是如果秘密信息位于 $2^{128} - 158 \sim 2^{128} - 1$ 这个区间中, 基于有限域 $\mathbb{F}_{2^{128}-159}$ 的 SSS 方案无法处理, 这种情况发生的概率是 $158/2^{128} \approx 2^{-120}$, 概率可忽略. Sage 实现 PoC 代码如下.

Listing 1: 基于二进制扩域 $\mathbb{F}_{2^m}$ 的 SSS 方案

```
gfp128 = GF(2^128-159)
t, n = 4, 6
secret = gfp128.random_element()
print "secret", hex(int(secret))

poly = [gfp128.random_element() for i in range(1, t)]
poly[0] = secret


print "poly"
for i in range(0, len(poly)):
    print i, ",", hex(int(poly[i]))

def eval_poly(x, poly):
    res = gfp128(0)
    for coeff in reversed(poly):
        res *= x
        res += coeff
    return res

shares = [(gfp128(i), eval_poly(gfp128(i), poly)) for i in range(1, n+1)]

print "shares"
for i in range(0, len(shares)):
    print "(", shares[i][0], ",", hex(int(shares[i][1])), ")"

def recover_secret(t, shares):
```

```
28      if len(shares) > t:
29          shares = shares[:t]
30
31      k = len(shares)
32
33      res = gfp128(0)
34      for i in range(k):
35          xi, yi = shares[i][0], shares[i][1]
36          print hex(int(xi)), hex(int(yi))
37
38          accum = gfp128(1)
39          for j in range(k):
40              xj = shares[j][0]
41              if i != j:
42                  accum *= xj / (xj-xi)
43
44          res += yi * accum
45      return res
46
47  print "recover with shares"
48  print hex(int(recover_secret(t, shares[0:4])))
49
50  print "recover with shares"
51  print hex(int(recover_secret(t, shares[1:5])))
52
53  print "recover with shares"
54  print hex(int(recover_secret(t, shares[2:6])))
```

2^{-120} 的概率的事件通常不会发生, 如果 128 比特的密钥是随机生成的, 则可以预期概率上不会出现基于有限域 $F_{2^{128}-159}$ 的 SSS 方案无法处理的情况. 但是有 158 个秘密值无法处理显得该 SSS 方案不够完美. 是否构造出能够处理所有的 128 比特秘密值的 SSS 的方案? 答案是肯定的, 这就要求重新选择有限域. 基于大素数的有限域显然不满足条件, 因为需要域中有 2^{128} 个元素, 而 2^{128} 显然不是素数. 此时需要的是二进制域的扩域. 因为对于正整数 m 的任意取值, $\mathbb{F}_{2^m}$ 确实是存在的并且在同构的意义下是唯一的. $\mathbb{F}_{2^m}$ 的构造方法: 选择 $\mathbb{F}_2$ 上的次数为 m 的不可约多项式 $p(x)$.

将 $\mathbb{F}_{2^m}$ 定义为 $\mathbb{F}_2$ 上的次数至多为 $m-1$ 的所有多项式构成的集合, 则该集合中总共有 2^m 个元素. $\mathbb{F}_{2^m}$ 中的加法定义为多项式的常规加法, 而 $\mathbb{F}_{2^m}$ 中的乘法定义为多项式的模乘, 模不可约多项式 $p(x)$. 现在我们需要的是 $\mathbb{F}_2$ 上的次数为 128 的不可约多项式 $p(x)$, 此处借用 GCM 工作模式中的定义 $p(x) = x^{128} + x^7 + x^2 + x + 1$. 选用这种有限域, 我们

可以构造能够处理任意 128 比特的秘密值的 SSS 方案. PoC 验证代码如下.

Listing 2: 基于二进制扩域 $\mathbb{F}_{2^m}$ 的 SSS 方案

```
gfp128 = GF(2^128, 'x', modulus = x^128 + x^7 + x^2 + x + 1)
t, n = 4, 6
secret = gfp128.random_element().integer_representation()
print "secret", hex(int(secret))

poly = [gfp128.random_element().integer_representation() for i in range(1, t)]
poly[0] = secret


print "poly"
for i in range(0, len(poly)):
    print i, ",", hex(int(poly[i]))

def eval_poly(x, poly):
    res = gfp128.fetch_int(0)
    for coeff in reversed(poly):
        res *= gfp128.fetch_int(x)
        res += gfp128.fetch_int(coeff)
    return res.integer_representation()

shares = [(i, eval_poly(i, poly)) for i in range(1, n+1)]

print "shares"
for i in range(0, len(shares)):
    print "(", shares[i][0], ",", hex(int(shares[i][1])), ")"


def recover_secret(t, shares):
    if len(shares) > t:
        shares = shares[:t]

    k = len(shares)

    res = gfp128.fetch_int(0)
    for i in range(k):
        xi, yi = shares[i][0], shares[i][1]
        print hex(int(xi)), hex(int(yi))

        accum = gfp128.fetch_int(1)
```

```
40          for j in range(k):
41              xj = shares[j][0]
42              if i != j:
43                  accum *= gfp128.fetch_int(xj) / (gfp128.fetch_int(xj)-gfp128.
                        fetch_int(xi))
44
45          res += gfp128.fetch_int(yi) * accum
46      return res.integer_representation()
47
48 print "recover with shares"
49 print hex(int(recover_secret(t, shares[0:4])))
50
51 print "recover with shares"
52 print hex(int(recover_secret(t, shares[1:5])))
53
54 print "recover with shares"
55 print hex(int(recover_secret(t, shares[2:6])))
```

值得注意的是, 虽然 t-of-n SSS 方案可以将秘密信息分割成多份, 可以在安全和防丢失方面提供一定程度的冗余. SSS 的问题在于, 要使用原始秘密信息时, 需要先重构出原始秘密信息, 原始秘密信息重构又重新引入了单点失败的分享, 这是在使用 SSS 方案时需要考虑的问题. 密码机等 HSM 会使用 SSS 来备份根密钥, 而分割和恢复的过程限定在 HSM 内部, 从而能够避免前述提及的单点失败的问题, 因为可认为 HSM 是安全的. 也因此 t-of-n SSS 方案是常见的用来备份根密钥的惯用手法. 一个很自然的问题是, 如何能够消除对可信方的依赖? 一个思路是利用安全多方计算的形式, 由多方共同完成依赖某项计算. 另一个思路是让每个参与方都选择自己的秘密信息并分割使得每个参与方都有所有参与方秘密的某个分片, 而真正用于计算的秘密信息是通过每个参与方的秘密信息计算而来, 这种思路的构造常用于构造多方阈值数字签名协议, 而其中为了保证协议安全需要引入同态加密, 零知识证明等密码学原语保证协议的安全性.