密码学第十次实验报告

ElGamal 数字签名方案

原理

Elgamal 数字签名算法是采用 F_p 上离散对数困难性进行数字签名的. 实际上, 它就是 Elgamal 公钥加密算法的一个变种.

实际上, Elgamal 数字签名算法也用到了哈希函数, 通过把哈希函数值嵌入签名值中, 并且采用难以得到哈希值的嵌入方法, 使得伪造签名变得困难.

可以证明, 该签名算法是正确的, 而且要求哈希函数 $H(m)$ 具有抗碰撞性质.

事实上, 原来的 ElGamal 数字签名方案并没有用到哈希函数, 而是直接用消息 m 代替了 $H(m)$. 这样会使方案容易受到基于数论的攻击, 而哈希函数的安全性要求可以规避这些攻击.

伪代码

密钥生成算法

```
def gen_key() -> Tuple[int, [int, int, int]]:
    x = (1, p − 1) 之间的随机数
    y = g^x  mod p
    return x, (y, p, g)
```

签名算法

```
def sign(m: bytes, sk: int) -> Tuple[int, int]:
    m = H(m)
    x = sk
    k = 0
    while k > 1 和 gcd(k, p − 1) = 1 不能同时满足:
        k = (1, p − 1) 之间的随机数
    s_1 = g^k  mod p
    s_2 = k^{−1}(H(m) − xr)  mod p − e
    return s_1, s_2
```

验证算法

```
def verify(m: bytes, pk: Tuple[int, int, int],
           sig: Tuple[int, int]) -> bool:
    y, p, g = pk
    s_1, s_2 = sig
    v_1 = g^H(m)  mod p
    v_2 = y^s1 s_1^s2  mod p
    return v_1 == v_2
```

分析

设消息长度为 l, 哈希函数的时间复杂度和空间复杂度分别为 $\mathrm{O}_T(f(l))$ 和 $\mathrm{O}_S(f(l))$.

密钥生成算法

密钥生成算法实际上步骤确定, 由于 p 通常作为公开参数, 所以数据范围也确定. 所以算法的时空复杂度都是 $\mathrm{O}(1)$.

签名算法

和密钥生成算法类似, 除了哈希函数, 时空复杂度都是 $\mathrm{O}(1)$. 但是要考虑哈希函数对时空复杂度的影响, 所以算法的总时空复杂度分别是 $\mathrm{O}_T(f(l))$ 和 $\mathrm{O}_S(f(l))$.

验证算法

和签名算法类似, 数据范围也确定. 所以算法的时空复杂度分别是 $\mathrm{O}_t(f(l))$ 和 $\mathrm{O}_S(f(l))$.

测试

采用自动化测试脚本进行测试, 通过测试对正确的消息签名并验证, 和用错误消息和原来的正确签名得到签名错误的结果, 来证明算法的正确性. 脚本文件名为 `elgamal_dss_test.py`.

优化

对哈希函数进行优化

首先可以想到的是对哈希函数进行优化. 哈希函数在这里只是需要返回一个整数值, 所以更好优化. 但是这个整数值通常小于 p, 所以哪怕 p 足够大, 碰撞也会增加. 所以要考虑哈希函数内部状态泄露的问题. 而且由于模运算开销较大, 而且哈希函数一般不具备线性性, 所以这种优化方法其实不一定现实.

提前算出要验证的消息的哈希值

这种方法可以简化计算, 但是只适用于要验证的消息可能被复用的情况, 而且要注意保护算出来的哈希值, 避免敌手拿到, 从而简化攻击.

Schnorr 数字签名方案

原理

Schnorr 数字签名方案是相当有名的, 它的一个特点是生成的签名非常短, 而且也巧妙地应用了哈希函数. 其实, 它就是利用哈希函数缩短签名长度的.

系统中有的参数为: 素数 p, q, q 是 $p-1$ 的因子; g 满足 g^q 和 1 同余.

实际上, 素数 p 应该取得相当大, 大概与哈希函数的输出空间大小相当. 我在实验代码里取的参数并没有那么大, 因为这样方便演示.

Schnorr 也是基于有限域上离散对数困难问题的, 但是使用哈希函数的方式和基于的恒等式和 Elgamal 数字签名方案都不一样.

伪代码

结果为整数的哈希函数

```
def dss_hash_modular(m: bytes) -> int:
    return m 的 SHA-256 结果按大端序转换成整数后再模 p 的结果
```

密钥生成算法

```
def gen_key() -> Tuple[int, int]:
    s = (0, q) 中的随机数
```

```
    v = g^{-1s} mod p
    return s, v
```

签名算法

```
def sign(m: bytes, sk: int) -> Tuple[int, int]:
    s = sk
    r = (0, q) 中的随机数
    x = g^r mod p
    e = H(x ‖ m)
    y = (r + se) mod q
    return e, y
```

验证算法

```
def verify(m: bytes, pk: int, sig: Tuple[int, int]) -> bool:
    e, y = sig
    v = pk
    x = g^y v^e mod p
    return H(x ‖ m) == e
```

分析

设消息长度为 l.

结果为整数的哈希函数

由于该函数用到了 SHA-256 哈希算法, 而且剩余部分时空复杂度易知为常数, 所以总的时间复杂度和空间复杂度分别为 O(n) 和 O(1).

密钥生成算法

可以看出, 数据的界是给定的, 所以时空复杂度易知为 O(1).

签名算法

易知步骤也固定, 而且除了哈希函数的操作, 其它操作需要的时空复杂度都是 O(1). 所以总的时空复杂度分别是 O(n) 和 O(1).

验证算法

和签名算法类似, 除了哈希函数的操作, 其它操作需要的时空复杂度都是 $O(1)$. 所以总的时空复杂度也分别是 $O(n)$ 和 $O(1)$.

优化

结果为整数的哈希函数

其实也可以采取像 ElGamal 数字签名算法的方法, 进行模的优化, 但是用处也不大.

签名算法

我想不出很好的优化方案, 因为数据依赖程度比较高, 而且过程性比较强.

验证算法

实际上, 验证算法的哈希值和需要比较的结果都是可以并行计算的, 实际上这也有点意义, 因为哈希计算如果优化得好, 速度和快速模幂算法都差不多.

测试

采用自动化测试脚本进行测试, 脚本文件名为 `schnorr_dss_test.py`. 其中测试了对消息的数字签名、对正确签名应返回正确、对错误签名应返回错误的功能. 可以看出, 算法是正确的.

总结

ElGamal 数字签名方案

其实这是一种相对实用的数字签名方案. 这次的 ElGamal 数字签名方案实际上是在 F_p 上做的, 而不是椭圆曲线上. 这样其实是为了方便表示, 因为我试图写在椭圆曲线上的算法, 因为表示问题没写出来. 但是, 这个算法的精髓和 ElGamal 系列的其它算法是相同的, 而且都是基于离散对数困难问题.

Schnorr 数字签名方案

Schnorr 数字签名方案非常有名, 签名的结果也相对较短, 因为它用到了哈希函数. 但是, 它的安全性和正确性也能得到保证, 只要哈希函数能够满足安全性要求.

算法分析与优化

这两种数字签名方案基于的是数学, 在真正的算法上, 感觉更多是下层算法的优化, 比如对哈希函数和模幂运算的优化. 但是, 实际上这还是有意义的.

系统设计与维护

这次依赖了上次的哈希函数, 所以哈希函数要写好, 这也是系统模块化的体现. 其实, 由于这两种数字签名方案中实际上用到的是把结果转换成整数的哈希函数, 所以其实可以再在哈希函数上封装一层, 这也体现出模块化的设计思想.

对课程的建议

感觉这次的路子非常正, 能够考察我们对基本概念的理解, 也对学到的算法进行了深化. 下次应该是大作业了, 希望能够更加考验对程序的模块化和系统的鲁棒性, 和对各种原语的综合应用. 再就是希望能够用 Python, 因为很多算法我都是在 Python 下实现的.

总结

这次实验我巩固了数字签名相关算法, 深化了对它的理解. 我会努力面对以后的挑战.