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

算法 1: ElGamal 数字签名方案密钥生成算法

输入: 无

输出: 私钥 x，公钥 (y, p, g)，其中 g 是素数 p 的一个原根

数据: p, g，其中 p 是素数，g 是 p 的一个原根

$x \leftarrow (1, p-1)$ 中的一个随机数

$y \leftarrow g^x \mod p$

return $x, (y, p, g)$

算法 2: ElGamal 数字签名方案签名算法

H

输入: 消息 m，私钥 x

输出: 签名 (s_1, s_2)

数据: 公钥 (y, p, g)

$k \leftarrow 0$

while *not (*$k > 1$ *and* $\gcd(k, p-1) = 1$*)* **do**

  $k \leftarrow (1, p-1)$ 之间的随机数

end

$s_1 \leftarrow g^k \mod p$

$s_2 \leftarrow k^{-1}(H(m) - xr) \mod p - e$

return s_1, s_2

算法 3: ElGamal 数字签名方案验证算法

输入: 消息 m，公钥 (y, p, g)

输出: 签名是否有效

$v_1 \leftarrow g^{H(m)} \mod p$

$v_2 \leftarrow y^{s_1} {s_1}^{s_2} \mod p$

if $v_1 = v_2$ **then**

  return *true*

else

  return *false*

end

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.

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

算法 4: Schnorr 数字签名方案密钥生成算法

输入: 无

输出: 私钥 s，公钥 v

数据: 素数 p 及其一个原根 g

$s \leftarrow (0, q)$ 中的随机数

$v \leftarrow (g^{-1})^s \mod p$

return s, v

算法 5: Schnorr 数字签名方案签名算法

输入: 消息 m，私钥 s

输出: 签名 c, y

数据: 素数 p 及其一个原根 g

$r \leftarrow (0, q)$ 中的随机数

$x \leftarrow g^r \mod p$

$e \leftarrow H(x \parallel m)$

$y \leftarrow (r + se) \mod q$

算法 6: Schnorr 数字签名算法验证算法

输入: 消息 m，私钥 s，签名 c, y

输出: 签名是否有效

数据: 素数 p 及其一个原根 g

$x = g^y v^e \mod p$

if $H(x \parallel m) = e$ **then**
  return *true*

else
  return *false*

end

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 $\mathrm{O}(n)$ 和 $\mathrm{O}(1)$.

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 $\mathrm{O}(1)$.

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 $\mathrm{O}(1)$. 所以总的时空复杂度分别是 $\mathrm{O}(n)$ 和 $\mathrm{O}(1)$.

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 $\mathrm{O}(1)$. 所以总的时空复杂度也分别是 $\mathrm{O}(n)$ 和 $\mathrm{O}(1)$.

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.

SM2 数字签名方案

原理

SM2 数字签名方案实际上也是基于椭圆曲线上离散对数的困难问题形成的。同时，它也利用了哈希函数的安全性质。但是，它也具有一定的认证性，因为系统参数中加入了用户的杂凑值 Z。而且，它的鲁棒性比一般的基于椭圆曲线的数字签名方案要好，因为它在生成用户的可辨别标识的时候，不仅把椭圆曲线的各个参数考虑进去，而且把用户的公钥也考虑进去了。这样，可以加大敌手伪造用户身份的成本。同时，生成的签名也可以验证用户身份。

伪代码

算法 7: SM2 数字签名方案密钥生成算法

输入: 无

输出: 有限域上的点（私钥）d，椭圆曲线上的点（公钥）P

数据: 椭圆曲线的参数 a, b，它的生成元 G

$d \leftarrow [1, n-2]$ 中的随机数

$P \leftarrow dG$

return d, P

算法 8: SM2 数字签名方案用户杂凑值生成算法

输入: 用户的可辨别标识 ID，用户的公钥 P

输出: 用户的杂凑值 Z

数据: 椭圆曲线的参数 a, b，它的生成元 G

$entlen \leftarrow ID$ 的比特长度

if $entlen \geq 2^{16}$ **then**

  抛出错误

end

把 $entlen$ 转换成 2 个字节长的字节串

$x_G, y_G \leftarrow G$ 的横坐标和纵坐标

$x_A, y_A \leftarrow P$ 的横坐标和纵坐标

把 a, b, x_G, y_G, x_A, y_A 转换成字节串

$Z \leftarrow H(entlen \parallel ID \parallel a \parallel b \parallel x_G \parallel y_G \parallel x_A \parallel y_A)$

return Z

算法 9: SM2 数字签名方案签名算法

输入: 消息 m，私钥 d，用户的杂凑值 Z

输出: 签名 (r, s)

数据: 椭圆曲线的生成元 G，它上面的点构成的交换群的阶 n

$\bar{M} \leftarrow Z \parallel M$

$e \leftarrow H(\bar{M})$

把 e 转换为整数

while *true* **do**

  $k \leftarrow [1, n-1]$ 中的整数

  $x_1 \leftarrow kG$ 的横坐标

  把 x_1 转换为整数

  $r \leftarrow (e + x_1) \mod n$

  if $r = 0$ *or* $r + k = n$ **then**

    continue

  end

  $s \leftarrow ((1 + d)^{-1}(k - rd)) \mod n$

  if $s \neq 0$ **then**

    break

  end

end

把 r, s 转换成字节串

return r, s

算法 10: SM2 数字签名方案验证算法

输入: 消息 m，公钥 P，签名 (r, s)，签名用户的杂凑值 Z

输出: 签名是否有效

数据:

把 (r, s) 从字节串形式转换成整数

if $r \notin [1, n-1]$ *or* $s \notin [1, n-1]$ **then**
  return *false*
end

$\bar{M} \leftarrow Z \parallel M$

$e \leftarrow H(\bar{M})$

把 e 转换成整数

$t \leftarrow (r + s) \mod n$

if $t = 0$ **then**
  return *false*
end

$x_1 \leftarrow (sG + tP)$ 的横坐标

$R \leftarrow (e + x_1) \mod n$

if $R = r$ **then**
  return *true*
else
  return *false*
end

分析

设消息长度为 l，签名用户的可辨别标识 ID 的长度为 l_{ID}。

密钥生成算法

该算法与其它数字签名方案的密钥生成算法类似，尤其是基于有限域上离散对数困难问题的那些算法。由于算法步骤固定，易知时空复杂度都是 $\mathrm{O}(1)$。

用户杂凑值生成算法

易知对 a, b, x_G, y_G, x_A, y_A 的获取与转换，时空复杂度都是 $\mathrm{O}(1)$。而求哈希值时，消息的长度由 l_{ID} 决定。根据哈希函数的时空复杂度，可以得出总时空复杂度分别为 $\mathrm{O}(l_{ID})$ 和 $\mathrm{O}(1)$。

签名算法

签名算法要调用用户杂凑值生成算法，而这个算法的时空复杂度分别为 $\mathrm{O}(l_{ID})$ 和 $\mathrm{O}(1)$。之后，计算 $\bar{M}$ 的杂凑值，这一步的时空复杂度根据哈希函数的时空复杂度可以得出。它的时间复杂度为 $\mathrm{O}(l)$，空间复杂度为 $\mathrm{O}(1)$。之后生成随机数、计算椭圆曲线上的点这两步，运算上界和空间上界可以确定，因此时空复杂度都是 $\mathrm{O}(1)$。之后两步判断可以当成结果固定成有利于算法往下执行的结果，因为考虑它们的话，整个算法就会变成随机算法，难以推断出它的时空复杂度了。而且，它们发生的概率很小。最后一步字节串转换的运算上界和空间上界都可以同样被确定，因此算法总时间复杂度为 $\mathrm{O}(l_{ID}) + \mathrm{O}(l) = \mathrm{O}(l + l_{ID})$，空间复杂度为 $\mathrm{O}(1)$。

验证算法

对它的分析和对签名算法的分析类似。首先前两个检验时空复杂度显然都是 $\mathrm{O}(1)$。之后也有求用户杂凑值的步骤，时空复杂度与签名算法中的该步骤相同。在之后求 $H(\bar{M})$ 的时候时空复杂度也与签名算法中同样的步骤相同。之后的加法和计算椭圆曲线的点，以及几次数据转换，也易知运算上界和空间上界固定，因此时空复杂度也是 $\mathrm{O}(1)$。因此，算法的总时间复杂度为 $\mathrm{O}(l + l_{ID})$，空间复杂度为 $\mathrm{O}(1)$。

优化

密钥生成算法

感觉这个算法比较简单，我想不出很好的优化方案。但是，可以通过对子算法的优化，来优化该算法。

用户杂凑值生成算法

实际上，各个用户的杂凑值可以提前算出。虽然这样系统就不用再关心每个用户的椭圆曲线参数具体是什么了，而且运算量减少，间接地提高了安全性，但是也有缺点。一是用户的密钥会更新，同步是个问题。二是提前算出可能会被敌手修改，会引起新的安全性问题。不过，用户的可辨别标识和椭圆曲线参数是公开的，这样提前算出问题能够减小一些。因此，提前算出杂凑值的方式应用起来要谨慎。

另外一种优化方案是提前算好哈希函数的初始向量，就像对 HMAC 优化那样。但是，这里哈希函数的参数一开头是用户的 ID，没法预测，因此这样也不行。其实，这也可能是防止调整哈希函数初始向量的一种安全手段。

还有一种优化方法，是缓存常见的椭圆曲线参数，避免转换时的开销。但是，一旦用上了缓存，碰到没有缓存好的参数，时间长度就变了。这样不但效率会降低，也可能受到侧信道攻击。一种避免的方式是缓存和提前算出的杂凑值这两种优化方案混合使用，并在时间上加入干扰（jitter）或者故意延时，这样攻击者获取的时间不但样本少，方差也大，攻击在实际中难以奏效。

签名算法和验证算法

这两种算法结构相似，因此可以一块考虑。首先，在签名算法中，kG 的横坐标 x_1 可以提前生成，而且 kG 可以提前丢弃。这样能提高效率，而且能提高算法的局部性，有利于现代处理器架构执行该算法。但缺点是 k 需要保留供以后使用，需要做好保密，不能让攻击者知道 k。其次，$(k-rd)=(k-ed-x_1d)$，因此在签名算法中，对应的 x_1d 也可以提前算出来。然后，实际上在两个算法中，可以利用类似于计算 HMAC 时的算法，来优化 $H(\bar{M})=H(Z\,\bar{\|}\,M)$ 的计算。最后，由于 $sG+tP=s(G+P)+rP$，所以可以提前把 $G+P$ 算出来，这样能够减少计算步骤。不过，还是要计算 t，因为这是必要的验证。

测试

采用自动化脚本进行测试。测试脚本文件名为 `sm2_dss_test.py`。脚本通过 SM2 标准文档中的样例测试签名算法是否正确，不过如果想得到和样例中完全一样的签名，需要更改签名算法中关于生成 k 的部分，因此测试脚本中比对签名的部分被注释掉了。脚本也通过自己生成一对公私钥对来测试密钥生成算法是否正确。

同时，该脚本也测试了各种异常情况，比如 m 和 Z 被篡改。

经过测试，算法能够生成与样例相同的结果，而且签名算法能够有效地验证签名。所以，实现一般是正确的。

DSA 数字签名方案

原理

DSA 数字签名方案也是基于有限域上的离散对数的困难问题，不过是基于 F_q 上的。它通过哈希函数的抗碰撞性以及其它安全性质，和数论的一些性质，实现了短签名的特点，但又与 Schnorr 数字签名方案有所区别。该算法需要用到几个系统参数，为了避免攻击，系统参数有严格的规定。

DSA 数字签名方案依赖的其实是子群上的离散对数的困难性，这种困难性并没有经过严格证明。因此，这可能是 DSA 数字签名方案安全性中的一个薄弱点。

伪代码

算法 11: DSA 数字签名方案密钥生成算法

输入: 无

输出: 私钥 x，公钥 y

数据: 子群的原根 g

$x \leftarrow [1, q-1]$ 中的随机数

$y \leftarrow g^x \mod p$

return x, y

算法 12: DSA 数字签名方案签名算法

输入: 消息 m，私钥 x

while *True* **do**

  $k \leftarrow [1, q-1]$ 中的随机数

  $r \leftarrow (g^k \mod p) \mod q$

  $s \leftarrow (k^{-1}(H(m) + xr)) \mod q$

  if $r \neq 0$ *and* $s \neq 0$ **then**

    return r, s

  end

end

算法 13: DSA 数字签名算法验证算法

输入: 消息 m，公钥 y，签名 (r, s)

输出: 签名是否有效

数据: 较小的素数 q, 子群的原根 g

if *not* $(0 < r < q$ *and* $0 < s < q$ **then**

  return *false*

end

$w \leftarrow s^{-1} \mod q$

$u_1 \leftarrow H(m)w \mod q$

$u_2 \leftarrow rw \mod q$

$v \leftarrow (g^{u_1} y^{u_2} \mod p) \mod q$

if $v = r$ **then**

  return *true*

end

return *false*

分析

设消息 m 的长度为 l。

密钥生成算法

比较容易看出，该算法的空间上界和计算上界都是常数，只要系统参数给定。因此，该算法的时空复杂度都是 $\mathrm{O}(1)$。

签名算法和验证算法

这两个算法都用到了哈希函数 $H(m)$。但是，算法的其它部分也类似可知时空复杂度都是常数。因此，该算法的时间复杂度为 $\mathrm{O}(l) + \mathrm{O}(1) = \mathrm{O}(l)$，空间复杂度为 $\mathrm{O}(1)$。

优化

密钥生成算法

这个算法可以把 x 和对应的 y 提前算出来。但是，不能重复使用 x，因为一旦重复使用，就有已经被证实的可以恢复出私钥的攻击方法。

签名算法和验证算法

可以通过费马小定理来优化求模逆的过程。不过，实际上两种算法的时间复杂度差不多，都是 $\mathrm{O}(\log q)$，因此没有太大必要。

测试

采用自动化脚本进行测试。测试脚本文件名为 `dss_test.py`。其中测试了密钥生成，以及对消息的签名和对签名的验证。对签名的验证中，故意篡改消息，得到的结果应该是签名无效。

经过测试，对有效的签名，能够得出有效的结果，反过来也是。因此，实现一般是正确的。

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.

SM2 数字签名方案

SM2 数字签名方案是椭圆曲线上的数字签名方案, 但是有比较先进的感觉. 它不但能够起到一定的验证用户身份的作用, 而且能够做到数字签名本身的基础功能。这提高了它的实用性.

DSA 数字签名方案

DSA 数字签名方案也是基于离散对数问题困难性的数字签名方案, 不过它借鉴了 Schnorr 数字签名方案, 使用了短签名的思想. 这种签名方案结果应该也是比较短的, 因此比较实用.

算法分析与优化

这四种数字签名方案基于的是数学, 在真正的算法上, 前两种数字签名方案和 DSA 数字签名方案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. 最后的 SM2 数字签名方案, 其实在数学上和算法上都是有一定优化空间的. 因此, 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, 由于前两种数字签名方案和 DSA 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. 在 SM2 数字签名方案中, 用到了椭圆曲线的相关算法, 体现的模块化的思想也是类似的.

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.