南開大學
Nankai University

《信息安全数学基础》探究报告

学 号： 2212046

姓 名：王昱

学 院：网络空间安全学院

专业：信息安全

2023 年 6 月 6 日

大整数分解问题

一、数学问题定义与背景

- 背景：数论中一个最基本、最古老而当前仍然受到人们重规的问题就是将大整数分解成素因子乘积。在历史上，这个问题曾经吸引了包括费马、欧拉、勒让德和高斯在内的大批数学家，他们花费了大量的时间和精力去研究这个问题。高斯在其著名的《算术探讨》中称道：“把素数同合数鉴别开来及将合数分解成素因子乘积被认作为算术中最重要最有用的问题之一.”我国的《易经》中也对这个问题作了研究。

  大整数分解问题在计算上是一个非常困难的问题，我们都知道，已知两个很大的素数去将其相乘，是一件十分简单的事；但是当事情反一下，已知一个大数是两个大素数的乘积，这样的话我们就不是很好求解这两个素数了。这种计算上的困难性是公钥密码学，特别是 RSA 公钥密码系统的基础。1977 年，基于大整数分解难题的密码体系 RSA 诞生，命名自 Ron Rivest、Adi Shamir 和 Leonard Adleman 三位提出者姓氏的首字母。RSA 算法在公钥密码学上作出重要贡献并对全世界产生重要影响，三位提出者于 2002 年获该年度图灵奖。整数分解问题是当今几乎所有公钥密码安全性领域中最重要的课题之一，如果攻击者能够分解这个整数，就可以破解 RSA 密码。因此，大整数分解问题在密码学和信息安全领域有着极其重要的地位，对于理解和应对现代的网络安全威胁有着重要作用。
- 定义：大整数分解问题是指将一个非常大的整数 N 分解为两个或多个质数的乘积，我们寻找素因子 p, q,使其满足 $N = p * q$，当 N 足够大的时候解决这个问题十分困难。

二、算法思想与特点

对于大整数分解，目前已经有了一些常见的算法来解决该问题：

2. 1 暴力穷举算法

算法介绍：

暴力穷举算法是一种最基本且直观的大整数分解方法。其基本思想是从最小的素数开始，逐个尝试将给定的大整数 N 进行整除。如果某个素数 p 能够整除 N，则记录 p 并将 N 除以 p 后继续对商进行相同的操作，直到 N 被完全分解。

算法步骤：

1.初始化一个空列表用于存储因子。

2.从最小的因子 2 开始到 N 一个一个进行尝试是否能够整除 N。

3.如果可以那么就把这个数 d 加入列表，并且把 N/d 作为新的 N 记录下来。

4.重复上述步骤直到 N=1。

算法伪代码：

如下为算法的 python 伪代码，我们输入一个需要分解的整数 N，随后输出一个列表，该列表记录了其所有质因子的结果。

```
def factorize(N):
    factors = []  # 初始化一个空列表用于存储因子
d = 2  # 从最小的因子 2 开始
    while N > 1:
     while N % d == 0:  # 检查是否能够整除 N
          factors.append(d)  # 如果可以，加入因子列表
          N //= d  # 更新 N 的值
       d += 1  # 尝试下一个因子
    return factors
```

算法特点：

该算法的优点是朴素直观，简单易实现，逻辑直接，适用于分解较小的整数。它逐一尝试每个可能的因子，直到找到所有素因子。但是，由于其时间复杂度高，最坏情况是 O（$\sqrt{N}$），处理大整数效率很低，因此实践中很少用到。

2. 2 费马分解法

算法思想：

对于一个 300 位的十进制数字进行质因数分解，普通电脑需要上百万年时间。但是如果这两个大质数比较接近，那使用费马质数分解就不需要几百万年了。费马分解法的核心思想是利用任何奇数 N 可以表示为两个平方数之差的性质，算法从最小的整数 a 开始，满足 $a > \sqrt{N}$ ，计算$b^2 = a^2 - N$,如果是完全平方数，则找到了 a 和 b，找到这两个数之后 a+b 和 a-b 就是 p 和 q 了。如果不是，则增加 a 的值，重复上述步骤直到找到合适的 b。这种方法对因子接近的整数效果较好，比试除法更高效一些。

算法流程：

1. 对给定的奇整数 N，将 N 进行开方后取整得到 x。

2.计算$x^2 - N$=y,如果 y 为一个完全平方数，那么就找到了对应的 x。

3.分别计算$(x + \sqrt{y})$和$(x - \sqrt{y})$,即为计算得到的因子对。

算法伪代码：

输入 N，输出其因子对:

```
def fermat_factorization(N):
    a = math.ceil(math.sqrt(N))
    b2 = a * a - N
    while not is_perfect_square(b2):
        a += 1
        b2 = a * a - N
    b = int(math.sqrt(b2))
    return (a - b, a + b)
def is_perfect_square(x):
    s = int(math.sqrt(x))
    return s * s == x
```

算法特点：

费马分解法基于将奇数表示为两个平方数之差的数学性质，在某些情况下可以很快找到分解结果。但是在实际应用中费马质数分解的效率其实不高，因为实际上两个质因数相近的情况是比较少见的。分解法在处理因子相近的数时效率较高，比试除法更有效，但对于差距较大的因子效率较低。

2. 3 Pollard's ρ

算法思想：

Pollard's ρ算法是 John Pollard 发明的一种能快速找到大整数的一个非 1、非自身的因子的算法。是一种用于整数分解的随机化算法，尤其适合分解大整数。其核心思想是利用伪随机数生成器在一个循环结构中产生数列，通过寻找数列中的周期性来找到非平凡因子。Pollard's ρ是一个非常玄学的方式，用于在 $O(n^{\frac{1}{4}})$的期望时间复杂度内计算合数 N 的某个非平凡因子。可以使用 Floyd 循环检测算法或 Brent 循环检测算法来检测循环的位置，事实上 Pollard's ρ 算法在实际环境中运行的相当不错。

算法步骤：

1.初始设置：选择函数 $f(x) = (x^2 + c) \bmod N$,选择随机初始值x_0和常数 c。

2.生成序列：初始化 x 和 y 为随机值，然后使用递归方法生成序列：$x_{i+1} = f(x_i), y_{i+1} = f(f(y_i))$。

3.寻找周期：使用 Floyd 判环法，通过计算 $\gcd(|x - y|, N)$检测周期。如果 gcd 非 1 且小于 N，找到因子。

4.如果检测到环或无法找到因子，重新选择随机值和常数。

5.累积若干次计算的结果，减少频繁求 gcd 的次数，提高效率。

伪代码：

该代码输入 N,输出其因子：

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
def f(x, c, n):
    return (x * x + c) % n
def pollard_rho(n):
```

```
    if n % 2 == 0:
        return 2
    x = random.randint(2, n - 1)
    y = x
    c = random.randint(1, n - 1)
    d = 1
    while d == 1:
        x = f(x, c, n)
        y = f(f(y, c, n), c, n)
        d = gcd(abs(x - y), n)
    return d if d != n else None
```

算法评价：

Pollard's ρ 算法是一种高效且实用的大整数分解方法，具有随机化和时间复杂度为$O(n^{\frac{1}{4}})$的优点。它通过生成伪随机数序列并使用 Floyd 判环法来检测周期性，从而找到非平凡因子。尽管存在随机性和不确定性问题，但实际应用证明了其有效性。该算法适用于因子差距较小的整数，在实际环境中运行表现出色，广泛应用于数论和密码学领域。

2.4 Lenstra椭圆曲线方法

算法思想：

Lenstra 椭圆曲线方法基于 Pollard's ρ 方法，但是在椭圆曲线上进行操作，而不是在模 N 的环上。Lenstra ECM 是一种基于椭圆曲线的高效分解大整数的方法，特别适用于寻找大整数的小因子。然而，由于其概率性，实际应用中可能需要多次运行才能成功找到因子。此外，选择适当的参数 B 对于算法的效率至关重要。

算法步骤:

ECM 算法的效率很大程度取决于群运算的快慢，最关键的是模 N 的求逆运算。

1.随机选取椭圆曲线 E 与一个点 P 在 E 上，并且选择一个合适的 k 值。

2.计算 kP:在计算过程中，如果遇到无法求逆的模 N 值，则该值为 N 的一个非平凡因子。

3.如果没有找到非平凡因子，选择另一条椭圆曲线和另一个点，重复上述步骤。

伪代码：

```
function lenstra_ecm(N, B):
    // 选择一个随机的椭圆曲线 E 和一个随机的点 P 在 E 上
    E, P = random_curve_and_point(N)
    // 对于每个在范围内的整数 k
    for k in range(2, B):
        try:
            // 计算点 kP
            Q = multiply_point(k, P, E, N)
        except NonInvertibleElementException as e:
            // 如果在计算过程中发现一个非平凡的因子，那么返回这个因子
            return gcd(e.element, N)
// 如果没有找到因子，那么尝试另一条曲线
return lenstra_ecm(N, B)
```

算法评价：

Lenstra 椭圆曲线方法通过在不同的椭圆曲线群上操作，提高了找到非平凡因子的概率，主要优点是运行时间与待分解的数 N 的最小素因子的大小有关，而不是 N 的大小。这使得它在寻找大整数的小因子时表现优秀。其效率依赖于群运算的快慢，尤其是模 N 的求逆运算,相比于 Pollard's ρ 算法，该在实际应用中通常更高效，适用于大整数分解。

2.5 二次筛选法

算法思想：

二次筛法(Quadratic Seive)是由 Pomerance 于 1981 年提出的，直到 1993 年是世界上渐进最快的通用大整数因子分解方法，第一的位置后来被数域筛所取代，不过对于 120 位以下的整数，二次筛还是要比数域筛快一些。二次筛选法是用于整数分解的有效算法，通过寻找平方同余来实现分解。它是一种通用的分解算法，特别适用于中等大小的数。QS 的核心思想是通过构建一个由小质数组成的光滑数因数基，筛选出满足特定条件的数值，并通过线性代数方法找到这些数值的平方同余。

算法步骤：

1.选择光滑数上界 B：确定一个适当的上界 B，定义一个包含所有小于 B 的素数的因数基底。

2.使用筛选方法，找到满足 $y(x) \equiv x^2 - N (\mathrm{mod}\ N)$的光滑数，使得这些数值只包含因数基底中的质因数。

3.对找到的光滑数进行质因数分解，生成每个数的指数向量。

4.将这些指数向量排列成一个矩阵，通过高斯消去法找到模 2 的线性相依向量组合。

5.从线性相依向量中找到平方同余$a^2 \equiv b^2 (\mathrm{mod}\ N)$从而求解出 N 的因子。

伪代码：

```
function quadratic_sieve(N, B):
    factor_base = generate_factor_base(B)
    smooth_x = []
    smooth_y = []
    // 对于每个整数 k
    for k in range(1, ∞):
        // 计算 x 和 y
        x = floor(sqrt(N)) + k
        y = x^2 - N
        // 检查 y 是否为 B-平滑数
        if is_B_smooth(y, factor_base):
            smooth_x.append(x)
            smooth_y.append(y)
            // 如果已找到足够多的平滑数
            if len(smooth_x) >= B:
                break
    // 构建矩阵并求解
    matrix = build_matrix(smooth_x, smooth_y, factor_base)
    solution = solve_matrix(matrix)
    // 从解中找到因子
```

```
        for s in solution:
            factor = gcd(N, s)
            if 1 < factor < N:
                return factor
        // 如果没有找到因子，返回失败
        return "Failure"
    // 生成因子基
    function generate_factor_base(B):
        factor_base = []
        for p in range(2, B):
            if is_prime(p):
                factor_base.append(p)
        return factor_base
    // 检查 y 是否为 B-平滑数
    function is_B_smooth(y, factor_base):
        for p in factor_base:
            while y % p == 0:
                y /= p
        return y == 1
    // 构建矩阵
    function build_matrix(smooth_x, smooth_y, factor_base):
        matrix = []
        for y in smooth_y:
            row = []
            for p in factor_base:
                count = 0
                while y % p == 0:
                    y /= p
                    count += 1
                row.append(count % 2)
            matrix.append(row)
        return matrix
    // 求解矩阵
    function solve_matrix(matrix):
        // 这里假设使用高斯消元法或其他线性代数方法求解
        // 返回一个解向量
        return gaussian_elimination(matrix)
```

算法评价：

二次筛选法在分解中等大小的整数方面表现优异，效率较高，并且具有良好的并行化特性。然而，由于其实现复杂性和对内存的需求较高，对于一些特定应用场景（如极大整数分解）可能不如某些其他算法（如数域筛选法）高效。对于需要分解中等大小整数的实际应用，二次筛选法是一个非常有效的选择。但在使用前，需要仔细考虑其实现复杂性和资源需求。

三、在密码学中的应用

1. RSA 算法

RSA 算法是现代密码学中最常用的公钥加密算法之一，其安全性依赖于大整数分解的困难性。RSA 算法通过生成一对密钥（公钥和私钥）实现加密和解密。密钥生成步骤包括：选择两个大素数 p 和 q,计算它们的乘积 $n = pq$，以及欧拉函数 $\emptyset(n) = (p-1)(q-1)$。选择一个与$\emptyset(n)$互素的整数 e 作为公钥指数，计算私钥指数 d，使得 $e * d \equiv 1(\mathrm{mod}\ \emptyset(n))$。公钥由$(e, n)$组成，私钥由$(d, n)$组成。加密时，将明文 m 变为密文 c 通过 $c \equiv m^e(\mathrm{mod}\ n)$计算，解密时通过 $m \equiv c^d(\mathrm{mod}\ n)$恢复明文。RSA 算法的广泛应用包括安全数据传输和数字签名，其核心安全性便是在于当前技术下因式分解大整数极为困难。

2. 数字签名

RSA 数字签名是一种用于验证消息完整性和发送者身份的加密技术。发送者对消息进行哈希运算生成消息摘要，然后使用其私钥对摘要进行加密生成签名，并将签名与消息一同发送。接收者使用发送者的公钥解密签名并比对解密后的摘要与消息的哈希值，如果一致，则证明消息未被篡改且确实由发送者签名。RSA 数字签名广泛应用于电子邮件、软件分发和金融交易中，确保数据的完整性和认证发送者身份。

3. 数字货币和区块链

一些数字货币和区块链技术也依赖于大整数分解问题。虽然比特币主要依赖于椭圆曲线加密，但其他一些加密货币系统可能使用基于大整数分解的加密算法来确保交易的安全性。

4. 密钥交换协议

RSA 密钥交换协议利用大整数分解的困难性来安全地交换密钥。尽管传统的 Diffie-Hellman 密钥交换依赖离散对数问题，RSA 版本的密钥交换协议依赖于大整数分解：

- 生成密钥对：每个参与方生成自己的 RSA 密钥对，并交换公钥。
- 交换密钥：通过 RSA 加密和解密步骤，双方能够安全地交换会话密钥

总的来说，大整数分解问题在密码学中的应用广泛，并且这个问题的难度直接关系到许多密码系统的安全性。

离散对数问题

一、数学问题定义与背景

- 背景：离散对数问题的研究起源于 1801 年高斯的《算术研究》，但在 1976 年 Diffie 和 Hellman 提出 Diffie-Hellman 密钥交换协议后，才在密码学中获得广泛关注。随后，ElGamal 在 1980 年代初提出了 ElGamal 加密系统，进一步推动了其应用。尽管现有的多种攻击算法和量子计算的发展对其构成威胁，离散对数问题依然是密码学中理解和设计安全系统的重要基础。
- 定义：离散对数问题在数学和密码学中具有重要意义。具体来说，离散对数问题可以定义如下：给定一个素数 p,一个生成元 g 和一个在模 p 环中的整数 h，求一个整数 x，使得$g^x \equiv h(\mathrm{mod}\ p)$。换句话说，就是找到 x 使得 g 的 x 次幂与 h 在模 p 下同余。这个问题在数学上可以类比为求解指数方程，但由于离散对数问题在有限域上进行，因而其难度和连续对数问题有本质的不同。离散对数问题的难度在于没有已知的高效算法可以在多项式时间内解决这一问题。正是由于这一特点，离散对数问题在密码学中广

泛应用。

二、算法思想与特点

目前离散对数问题仍不存在多项式时间经典算法（离散对数问题的输入规模是输入数据的位数）。在密码学中，基于这一点人们设计了许多非对称加密算法：

2.1Baby-step Giant-step 算法

算法思想：

该算法的核心思想是将离散对数问题分解为两个子问题，通过预计算和查找来有效地寻找离散对数。具体来说，离散对数问题是求解方程 $g^x \equiv h(\mathrm{mod}\ p)$，其中 g 是素数 p 的一个原根。该算法利用了时间和空间的平衡，预先计算并存储一部分幂值，然后在另一部分计算过程中进行匹配查找。通过构造两个列表，一个包含 g 的小次幂，另一个包含 h 乘以 g 的负大次幂，算法可以通过查找相同元素的方式快速找到 x 的值。该方法显著降低了求解离散对数问题的时间复杂度，较传统的蛮力搜索有明显的改进。

在算法竞赛中，BSGS 常用于求解离散对数问题。形式化地说，对 $a, b, m \in Z^+$，该算法可以在 $O(\sqrt{m})$的时间内求解：

$$a^x \equiv b(\mathrm{mod}\ m)$$

其中 $a \perp m$,方程的解满足 $0 \leq x \leq m$。

算法步骤：

1.计算 n 的值，$n = \lceil \sqrt{p} \rceil + 1$,这是因为将问题分解为两个大小为 n 的子问题，预计算这些子问题的结果并进行查找。

2.构造两个列表:

List1:计算并储存 g 的所有小次幂，即$[1, g^1, g^2, g^3, \ldots, g^n]$

List2:计算并储存 h 乘以 g 的负大次幂，即$[h, hg^1, hg^2, hg^3, \ldots, hg^n]$

3.在两个列表中找到相同的元素 $g^i = hg^{-jn}$,即 $\mathrm{List1}[j] = \mathrm{List2}[j]$,这表明 $g^{i+jn} \equiv h(\mathrm{mod}\ p)$。

4.通过找到的 i 和 j 计算出离散对数 x，即 $x = i + jm$。

伪代码：

```
function sDLP(g, h, p):
    // 初始化
    n = ⌈ √p⌉ + 1
    L1 = init_list(1, g, n, p)
    aa = (g^(-n)) mod p
    L2 = init_list(h, aa, n, p)

    // 查找相同的元素
    i = 0
    j = 0
    while True:
        if (i >= n or j >= n):
            break
        while (L1[i].vue < L2[j].vue and i < n):
            i += 1
        while (L1[i].vue > L2[j].vue and j < n):
```

```
                j += 1
            if (L1[i].vue == L2[j].vue):
                x = L1[i].num + L2[j].num * n
                return x
        return None

    function init_list(first, g, n, p):
        List = []
        temp = node(first, 0)
        List.append(temp)
        for i = 1 to n:
            temp = node()
            temp.num = i
            temp.vue = (List[i-1].vue * g) mod p
            List.append(temp)
        List.sort by vue
        return List

    class node:
        function _init_(self, vue=0, num=0):
            self.vue = vue
            self.num = num
```

算法特点：

该算法在求解离散对数问题时表现出色。它将时间复杂度从蛮力搜索的 $O(p)$降低到 $O(\sqrt{p})$，显著提高了计算效率，同时作为确定性算法，确保每次执行都能找到准确的解。然而，该算法的空间复杂度较高，需要存储两个大小为$\sqrt{p}$的列表，这对大规模问题可能会导致内存需求过高。总体而言，该算法适用于中等规模的离散对数问题，是密码学和数论领域中分析和破解加密系统的有效工具。

2.2 Pohlig − Hellman离散对数算法

算法思想：

Pohlig − Hellman 算法是一种求解离散对数问题的有效方法，尤其适用于模数 p 的阶 $p-1$ 有小素因子的情况。该算法通过将大规模问题分解为若干较小的子问题，并最终使用 CRT 合并解来得到最终结果。

算法步骤：

1.根据唯一分解定理：$N = q_1^{e_1} q_2^{e_2}, \ldots, q_t^{e_t}$对 N 进行分解，得到因数列表 List q^e,即$q_i^{e_i} = [q_1^{e_1}, q_2^{e_2}, \ldots, q_i^{e_i}]$。

2.用已知的每一项$q_i^{e_i}$， List $g^{(p-1)/q^e}$,List $h^{(p-1)/q^e}$

3.对于每一组$g^{(p-1)/q^e}$,$h^{(p-1)/q^e}$,计算出$(g^{(p-1)/q^e})^x = h^{(p-1)/q^e}$,中的 x,并且存入 List x。

4.最后一步利用 CRT 算法，对于 List 中对应的各 i 项，$x = \text{List } x[i] (\text{mod List } q^e[i])$构成同余方程组。

伪代码：

```
    function PohligHellman(g, h, p):
        // 1. 因数分解 p-1
```

```
    qe = factorize(p-1)

    // 2. 计算 List g 和 List h
    Lg = [g^((p-1)/q^e) mod p for each q^e in qe]
    Lh = [h^((p-1)/q^e) mod p for each q^e in qe]

    // 3. 求解每个小问题，得到 List x
    La = []
    for i in range(len(qe)):
        La.append(solve_dlp(Lg[i], Lh[i], q^e))
    // 4. 使用中国剩余定理合并结果
    X = CRT(qe, La)
    return X
```

算法特点：

Pohlig-Hellman 算法通过将离散对数问题分解为多个小问题并利用中国剩余定理（CRT）合并解，能够在模数 p 的阶 $p-1$ 有小素因子时高效地求解。其主要优点在于高效性和利用分解与合并策略处理大规模问题。在实际应用中，该算法通过对 $p-1$ 进行因数分解，将复杂的大问题转化为多个小问题，从而降低计算复杂度。然而，Pohlig-Hellman 算法的效率依赖于 $p-1$ 的因数结构，若 $p-1$ 具有大素因子，算法效率会显著降低。此外，实现该算法需要精确的因数分解和 CRT 合并，对于大规模问题实现较为复杂。总体而言,Pohlig − Hellman 算法在适用范围内表现优异，是密码学中解决离散对数问题的重要工具。

三、在密码学中的应用

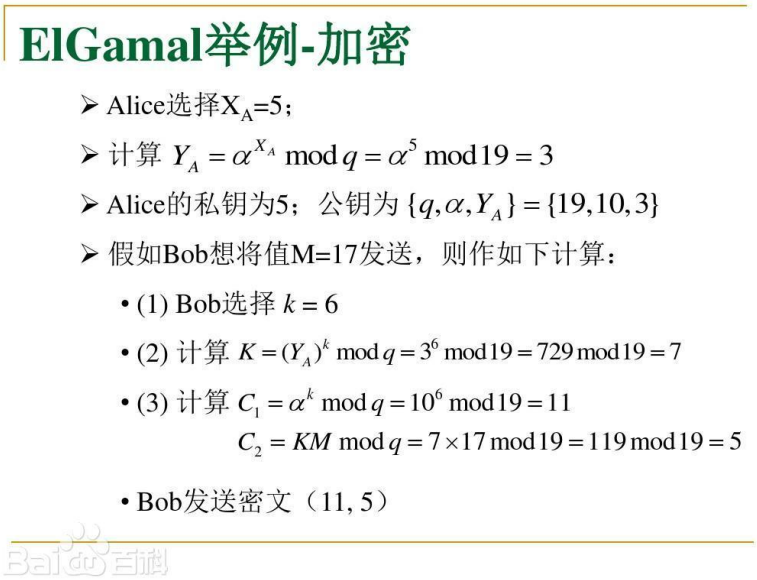
1. Diffie − Hellman密钥

Diffie − Hellman 密钥交换是一种用于在公共信道上安全生成共享密钥的协议，其安全性基于离散对数问题的困难性。该过程包括以下步骤：首先，通信双方选择一个公共素数 p 和一个生成元 g。每一方分别选择一个私有密钥 a 和 b，并计算各自的公钥 $A \equiv g^a \pmod p$和 $B \equiv g^b \pmod p$。双方交换公钥后，各自计算共享密钥 s，计算方式为 $sg^{ab} \equiv B^a \pmod p$和 $sg^{ab} \equiv A^b \pmod p$。由于$g^{ab} \equiv g^{ba} \pmod p$，双方得到相同的共享密钥 s。这一协议确保了即使攻击者截获了公钥，也无法计算出共享密钥，因而实现了安全的密钥交换。

2. ElGamal 加密系统

ElGamal 加密系统是一种基于离散对数问题的公钥加密方案，用于确保数据的保密性，其安全性直接依赖于离散对数问题的计算困难性。过程包括以下步骤：首先，接收方生成一个大素数 p 和一个生成元 g，选择私钥 x，并计算公钥 $h \equiv g^x \pmod p$。发送方在加密消息 m 时，选择一个随机数 y，计算$c_1 \equiv g^y \pmod p$ 和$c_2 \equiv m * h^y \pmod p$，加密后的消息为 (c_1, c_2) 。解密时，接收方使用私钥 x 计算 $m \equiv c_2 / c_1^x \pmod p$。ElGamal 加密的安全性依赖于离散对数问题的困难性，即计算$g^x \equiv h \pmod p$的 x 在大多数情况下是不可行的,从而保证了消息的保密性和传输的安全性。

以下是一个例子：



3. 数字签名算法

数字签名算法是一种基于离散对数问题的数字签名方案，用于验证消息的完整性和发送者的身份。其过程包括生成大素数 p 和 q 及生成元 g，选择私钥 x，并计算公钥 $y \equiv g^x (\mathrm{mod}\ p)$。对消息 m 生成签名时，选择随机数 k，计算签名对 (r, s)，其中 $r \equiv (g^k \bmod p) \bmod q, s \equiv (H(m) + xr)k^{-1} \bmod q$。验证时，接收方计算 $w \equiv s^{-1} \bmod q, u_1 \equiv H(m)w \bmod q, u_2 \equiv rw \bmod q$，并验证 $v \equiv (g^{u_1} y^{u_2} \bmod p) \bmod q$ 是否等于 r。DSA 的安全性依赖于离散对数问题的难解性。

4. 椭圆曲线密码学

离散对数问题在椭圆曲线密码学中被扩展到椭圆曲线离散对数问题（ECDLP），这是现代密码学中广泛应用的一种形式。

- 椭圆曲线 Diffie-Hellman（ECDH）：用于安全的密钥交换。
- 椭圆曲线数字签名算法（ECDSA）：用于数字签名。

AES 加密

一、介绍密码原语

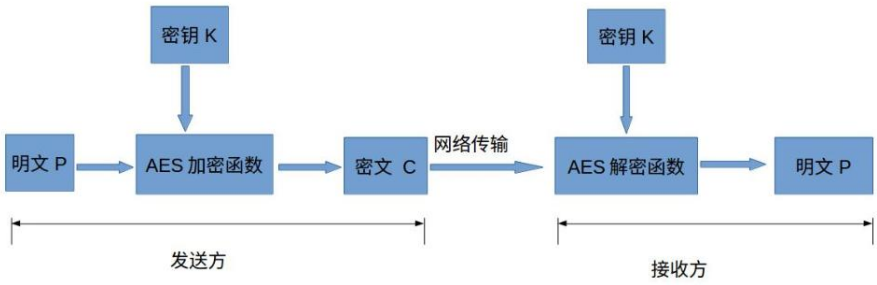
- 介绍：高级加密标准(AES,Advanced Encryption Standard)为最常见的对称加密算法(微信小程序加密传输就是用这个加密算法的)，是由美国国家标准与技术研究院于 2001 年公布的一种对称密钥加密标准，设计用于取代原先的数据加密标准。AES 是一种块密码，它将数据分为固定大小的块（在 AES 中，这个大小为 128 位）进行加密和解密。AES 的设计目标是提供高度的安全性，同时保持较高的计算效率和易用性。
- 密钥长度和加密轮数：AES 使用的密钥长度可以为 128、192 或 256 位。根据使用的密钥长度不同，加密过程的轮数也有所不同，分别为 10 轮、12 轮和 14 轮。每一轮的操作都包括四个主要步骤：字节代换、行移位、列混淆和轮密钥加。这些步骤通过组合，确保了数据的高度混淆和扩散，增加了加密的复杂性和安全性。

- 设计目标：AES 的设计目标是在提供高度安全性的同时保持较高的计算效率。其安全性通过复杂的 S 盒代换、行移位和列混淆操作来实现，使得加密后的密文具有高度的不可预测性和复杂性，从而防止攻击者通过分析密文破解密钥。AES 的结构设计确保即使一个位的改变也会导致输出的显著变化，增加了对抗各种攻击的能力。
- 特点：AES 的设计考虑到了易用性和通用性。作为一个标准，AES 需要能在各种硬件和软件环境中实现，包括嵌入式系统、移动设备和大型计算机。此外，AES 还需要能适应各种应用需求，包括数据保密、身份验证和数据完整性保护。同时，AES 在效率方面的设计使得其可以在各种硬件和软件环境中高效运行。AES 在硬件实现中尤为高效，使其适用于从低功耗嵌入式系统到高性能服务器的广泛设备。AES 的算法设计尽可能地减少所需的计算资源，如时间和内存，确保在各种环境中都能高效运行。

  AES 通过一系列复杂但高效的操作，确保了数据在传输和存储过程中的安全性和完整性。其设计兼顾了安全性、效率、易用性和通用性，使其成为当今最广泛使用的加密标准。随着技术的发展，AES 仍将是保护信息安全的重要基石。

二、AES 加密工作原理

具体的加密流程如下图：



核心操作：

1.字节代换：使用一个固定的 S 盒（Substitution box）对每个字节进行非线性代换，增强数据的混淆性。S 盒是一个预先计算好的查找表，用于将输入字节替换为输出字节，通过非线性代换增加加密的强度。

2.行移位：将数据块中的每一行循环左移不同的字节数，增加数据的扩散性。具体操作是：第一行不变，第二行左移一字节，第三行左移两字节，第四行左移三字节。这一操作通过重新排列数据块中的位来增加数据的复杂性。

3.列混淆：使用一个固定的多项式对每列进行混合运算，以进一步扩散数据。每列的四个字节被视为一个多项式，并与一个固定多项式相乘，这样可以使得每个字节影响多个字节，从而增加数据的混淆性和扩散性。

4.轮密钥加：将每一轮生成的子密钥与数据块进行按位异或（XOR）运算，增加数据的保密性。这个操作将当前状态和密钥结合，使得每一轮的加密都与密钥紧密相关，确保加密过程的安全性。

三、AES 加密应用场景

AES 加密算法被广泛应用在各种需要数据保密的场景中，以下是一些常见的 AES 加密应用场景：

1. 数据传输安全

AES 广泛用于保护数据在传输过程中的安全性，防止数据被窃取或篡改。

- HTTPS：在互联网通信中，HTTPS 协议使用 AES 加密数据，确保在客户端和服务器之间传输的信息保密和完整。
- 虚拟专用网络：VPN 利用 AES 加密用户的网络流量，确保用户数据在公共网络上的传输安全。

2. 存储数据加密

AES 用于对存储在硬盘、固态硬盘（SSD）和其他存储介质上的数据进行加密，防止未经授权的访问：

- 全磁盘加密：操作系统（如 Windows BitLocker 和 macOS FileVault）使用 AES 加密整个硬盘或分区的数据，确保即使设备丢失或被盗，数据仍然安全。
- 数据库加密：许多数据库系统（如 Microsoft SQL Server 和 Oracle）使用 AES 加密存储的敏感数据，保护数据隐私。

3. 无线通信安全

AES 在无线通信中用于加密数据，确保无线网络的安全性。如 Wi-Fi 安全：无线网络安全协议（如 WPA2 和 WPA3）使用 AES 加密 Wi-Fi 网络中的数据流，防止无线通信被窃听或劫持。

4. 移动设备安全

移动设备（如智能手机和平板电脑）使用 AES 加密存储的敏感数据，保护用户隐私和安全。例如应用数据加密，许多移动应用程序使用 AES 加密存储的用户数据，如密码、支付信息和个人信息，防止数据泄露；设备加密，移动操作系统（如 Android 和 iOS）提供设备加密功能，使用 AES 加密整个设备上的数据。

5. 文件加密

AES 用于加密单个文件或文件夹，保护文件内容的机密性。例如：

- 文件加密工具：许多文件加密工具（如 VeraCrypt、AxCrypt 和 7-Zip）使用 AES 加密文件，确保文件在存储和传输过程中的安全。
- 云存储安全：云存储服务（如 Google Drive、Dropbox 和 OneDrive）使用 AES 加密存储在云端的文件，保护用户数据免受未经授权的访问。

AES 作为一种高效、安全、通用的对称密钥加密算法，广泛应用于各种数据加密需求中。其应用场景涵盖了数据传输、安全存储、无线通信、移动设备安全、文件加密、电子邮件加密以及支付和金融安全等领域。AES 的广泛应用得益于其高效的加密性能、强大的安全性和易用性，确保了数据在各个环节中的安全和完整。

四、AES 加密包含的数学问题

AES 加密问题涉及到许多数学问题，主要如下所示：

1. 有限域（Galois Field）运算

AES 中的所有操作都在有限域 $GF(2^8)$上进行，这意味着每个字节被视为一个在不可约多项式$x^8 + x^4 + x^2 + x + 1$ 下的多项式。在此有限域内，所有运算（加法、乘法、求逆等）都可以通过多项式运算实现。字节代换使用 S 盒进行非线性代换，而 S 盒的生成基于有限域 $GF(2^8)$中每个字节的逆元，通过仿射变换进一步复杂化。这种有限域运算确保了数据在加密过程中被高度混淆，使得简单的线性关系无法被利用进行攻击，从而增加了加密的安全性。

2. 非线性和线性变换

AES 通过一系列非线性和线性变换确保加密的复杂性和安全性。非线性变换包括字节

代换和行移位，前者通过 S 盒替换每个字节，后者通过循环左移数据块中的行增加数据的扩散性。线性变换包括列混淆和轮密钥加，其中列混淆通过矩阵与向量的乘法运算，使得每个字节的变化影响整列数据，增加数据的扩散性。轮密钥加通过按位异或操作将状态矩阵与每轮的子密钥结合，确保密钥的影响贯穿整个加密过程。这些变换相结合，确保了 AES 加密的高度复杂性和不可预测性。

3. 密钥扩展和轮密钥加

AES 的密钥扩展过程将初始密钥生成多个独特的轮密钥，确保每一轮的加密过程都是独一无二的。密钥扩展算法涉及到移位、S 盒代换和异或操作，这些步骤生成了一组用于每轮加密的子密钥。轮密钥加在每一轮中将当前状态与相应的子密钥进行按位异或运算，确保密钥的影响在整个加密过程中持续存在。这一过程确保了即使初始密钥被部分泄露，也难以通过分析一个或少量轮次的密文破解整个加密过程，从而增加了加密系统的安全性和鲁棒性。