# 信安数基研究报告

2212794 曹瑜 密码科学与技术

## Topic 1 [离散对数问题]：（小组选题）

### 1、数学问题：

离散对数：如果对于一个整数 b，和质数 p 的一个原根 a，可以找到一个唯一的 z 指数 i，使得：$b = a^i (\mathrm{mod} p)$，其中 $0 \leq i \leq p-1$ 成立，那么指数 i 称为 b 的以 a 为基数的模 p 的离散对数；

离散对数难题：当已知一个质数$p$和它的一个原根$a$，如果给定一个$b$，要计算$i$的值是比较困难；给定群 G，设 g,h 属于 G，且有等式$h = g^x$ ，那么求解 x，就是离散对数问题（Discrete Logarithm Problem）。x 称为 h 以 g 为底的离散对数，用符号表示为$\log_g h$ ，或者 $DL_g(h)$。

### 2、算法介绍&特点：

Babystep-Giantstep 方法：

**算法原理**：Shanks 的 Babystep-Giantstep 方法是解决离散对数问题的一种有效算法；离散对数问题是指在给定元素 g 和 h 的情况下，在某个群 G 中找到一个整数，使得 $g^x=h$。这个问题在密码学中非常重要，尤其是在公钥加密和数字签名算法中；其核心思想是将可能的指数值分割成较小的区块，从而降低必须直接计算的指数的数量。算法主要分为两个阶段：Babysteps 和 Giantsteps。

**具体步骤**：

（1）选择整数 m：通常取 $m=\lceil \sqrt{n} \rceil$，其中 n 是群 G 的阶；

（2）Babysteps: 对于 j=0 到 m-1，计算 $g^j$，并将结果与 j 存储在一个表中；

（3）计算 $g^{(-m)}$：这是巨步的基础

（4）Giantsteps: 对于 i=0 到 m-1，计算 $h \cdot g^{-im}$ 并检查这个结果是否出现在 Babysteps 的表中。如果找到匹配，假设对应的 Babystep 是 $g^j$，则 $h=g^{im+j}$ 从而解为 x=im +j；

**特点**：

相比简单枚举，大大减少了搜索时间，算法直观易理解，易于实现，在非常大的群中，即使是 O(√n)的复杂度也可能导致不实际的计算时间和资源需求，但需要大量内存存储 Babysteps 的结果

Pollard ρ算法：

**算法原理**：Pollard ρ算法利用了随机漫步和伪随机数生成的思想来寻找循环，从而实现对数值问题的求解。其核心概念是通过构造一个伪随机序列来寻找重复值，进而确定循环长度，并最终找到目标值。

**具体步骤**：

（1）选择初始值 $x_0$ 和伪随机函数 f(x)；
（2）生成序列 $x^n+1=f(x^n)$并计算序列的 gcd 值;
（3）使用 Floyd 判环算法（乌龟和兔子算法）检测序列中的循环；
（4）一旦检测到循环，计算两个数值的最大公约数，即可得到因子。

**特点**：

内存占用低，在许多实际应用中展现出高效性，但本质上是概率算法，不能保证在所有情况下都能迅速找到解，算法的性能在一定程度上依赖于伪随机函数的选择

Pohlig-Hellman 算法：

**算法原理**：Pohlig-Hellman 算法利用中国剩余定理和模数的素因子分解，将离散对数问题 $g^x \equiv (\text{mod } p)$ 转化为多个小规模的离散对数问题。假设 p-1 的素因子分解为 $p-1=q_1^{e1}q_2^{e2}\cdots\cdots q_k^{ek}$，则原问题可以分解为模 $q_i^{ei}$ 的子问题。

**具体步骤**:

（1）将模数字 p-1 分解为素因子乘积 $p-1=q_1^{e1}q_2^{e2}\cdots\cdots q_k^{ek;}$
（2）对于每个素因子 $q^i$ 和对应的指数 $e^i$,求解模 $q_k^{ek}$ 的离散对数问题;
（3）利用中国剩余定理，将各个子问题的解合并，得到原问题的解；

**特点**:

适用于模数具有较小素因子的情况，能够显著降低计算复杂度;可扩展性：算法可以扩展到处理更大规模的离散对数问题;但算法性能依赖于模数 p-1 的素因子分解，如果 p-1 含有大素因子，算法效率会下降；但适用范围有限，主要适用于模数 p 为大质数的情况。

指数计算法：

**算法原理**：指数计算法(Index Calculus Method)是一种解决离散对数问题的算法，特别适用于大素数模数情况下的离散对数计算。该方法利用数论中的一些技巧，将离散对数问题转化为求解线性方程组，从而大大加快计算速度。

**具体步骤**:

（1）选择 $B=\{p^1,p^2,\ldots,p^k\}$，这些素数通常选取较小的素数；
（2）随机选择整数 a，计算 $g^a(\text{mod } p)$,然后检查 $g^a(\text{mod } p)$ 是否能分解为因子基

中的素数的乘积。如果可以，记录这个关系;

（3）将所有能分解的关系写成线性方程，形成矩阵 A 和向量 b，即 Ax=b.利用线性代数的方法解出 x，即 $\log_g p_j$.

（4）对于给定的 h，找到 a 使得 $g^a * h-1$ 能分解为因子基中的素数的乘积，通过前面的对数值计算出 x。

**具体步骤:**

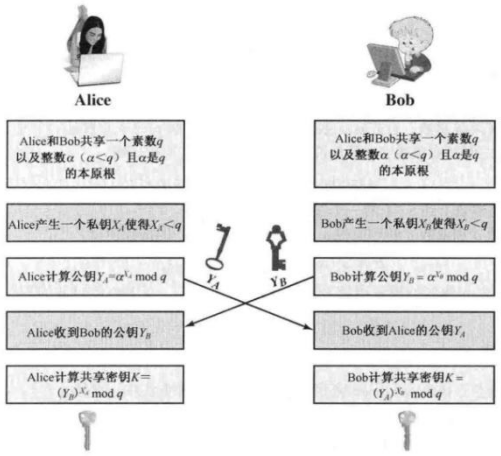
内存占用低，在许多实际应用中展现出高效性，但本质上是概率算法，不能保证在所有情况下都能迅速找到解，算法的性能在一定程度上依赖于伪随机函数的选择；

## 3、在密码学中的应用：

Diffie-Hellman 密钥交换协议：

Diffie-Hellman 算法由 Whitfield Diffie 和 Martin Hellman 提出，该算法的安全性也是基于一般有限域上的离散对数问题的难解性。该算法的目的是使两个用户能安全地交换密钥，以便在后续的通信中用该密钥对消息加密。该算法本身只限于进行密钥交换。

**通俗解释：**




**算法描述：**

Diffie-Hellman 算法主要用于两个用户密钥 K 的安全交换的场景下，利用一般有限域上的离散对数难解问题，使得敌手难以通过假定的已知信息推算得出密钥 K。

Diffie-Hellman 算法描述如下：

首先选取素数 q 和其本原根 α 为两个公开的整数，用户 Alice、Bob 再分别选取一个随机整数$(X < q)$并各自计算 $Y = \alpha^X (\mathrm{mod}\ q)$。在此过程中，用户只需保证各自所选取的 X 为私有的，计算所得出的 Y 则是可以公开访问的。在计算完毕各自的 Y 后，两用户通过信道交换得到对方的$Y'$，再根据各自私有的随机整数 X 计算 K：

$$K = (Y')^X (mod\ q)$$

两用户计算所得出的密钥 K 将是相等的:

$$
\begin{aligned}
K &= (Y_B)^{X_A} \ mod\ q \\
&= (\alpha^{X_B} mod q)^{X_A} \ mod\ q \\
&= (\alpha^{X_B})^{X_A} \ mod\ q \\
&= \alpha^{X_B X_A} \ mod\ q \\
&= (\alpha^{X_A})^{X_B} \ mod\ q \\
&= (\alpha^{X_A} mod q)^{X_B} \ mod\ q \\
&= (Y_A)^{X_B} \ mod\ q
\end{aligned}
$$

**应用场景：**

Diffie-Hellman 密钥交换协议广泛应用于网络通信和数据加密等领域。例如，HTTPS 协议中就使用了这种算法来建立安全连接。通过使用 Diffie-Hellman 密钥交换协议，两个通信方可以在不直接交换密钥的情况下，建立起一个共同的密钥，从而确保通信内容的安全性。Diffie-Hellman 算法也被广泛应用于 VPN、安全通信等场景。

## Topic 2 [RSA 问题]：

### 1、数学问题：

大整数因子分解问题：给定一个大合数 n，找到它的两个质因子 p 和 q 是一个极其困难的数学问题；在 RSA 算法中，n 是通过选择两个大质数 p 和 q 并计算它们的乘积得到的，即 n = p * q。这个 n 值将作为公钥和私钥的共同部分。由于大整数的因子分解困难，即使知道了 n 的值，也无法轻易得到 p 和 q 的值，从而保证了私钥的安全性；

求模的离散对数问题：
已知一个数 a、模数 n 和余数 b，求解指数 x，使得 $a^x \bmod n = b$。在大整数的情况下，这个问题是非常困难的，因为没有有效的算法可以在多项式时间内解决。在 RSA 算法中，这个问题与私钥的生成密切相关，具体来说，需要找到一个整数 e（公钥的加密指数），使得 $1 < e < \varphi(n)$且 e 与φ(n)互质（其中φ(n)是欧拉函数，表示小于 n 且与 n 互质的正整数的个数）。然后，需要计算 e 对于φ(n)的乘法逆元 d（私钥的解密指数），满足 $ed \equiv 1 \pmod{\varphi(n)}$。这个 d 的求解过程就涉及到了求模的离散对数问题

### 2、算法介绍&特点：

### 算法介绍：

RSA 算法由罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Leonard Adleman）在 1977 年提出。该算法使用一对密钥，即公钥和

私钥，来进行加密和解密操作。公钥用于加密数据，而私钥用于解密数据。RSA 算法的安全性基于以下两个事实：

大数分解的困难性：给定一个大合数 n，找到它的两个质因子 p 和 q 是一个极其困难的数学问题。RSA 算法中，n 是通过选择两个大质数 p 和 q 并计算它们的乘积得到的。

模幂运算和模逆元的存在性：RSA 算法利用模幂运算和模逆元的概念来进行加密和解密操作。

## 算法步骤：

参数表：

| 参数 | 解释 | 公式 | 描述 |
| --- | --- | --- | --- |
| P、Q | 质数 | P*Q=N | 分解模数N后得到的值 |
| N | 公共模数 | N=P*Q | 在RAS中进行模运算 |
| E | 公钥指数 | gcd(φ,e)=1 | 1<e<φ |
| D | 私钥指数 | gcd(φ,d)=1 | 1<d<φ |
| φ | 欧拉公式 | φ=(P-1)*(Q-1) | / |
| c | 密文 | $c=m^e \bmod N$ | / |
| d | 明文 | $m=c^d \bmod N$ | / |

加密算法：

$c \equiv m^e \bmod N$

解密算法：

$M \equiv c^d \bmod N$

密钥生成：

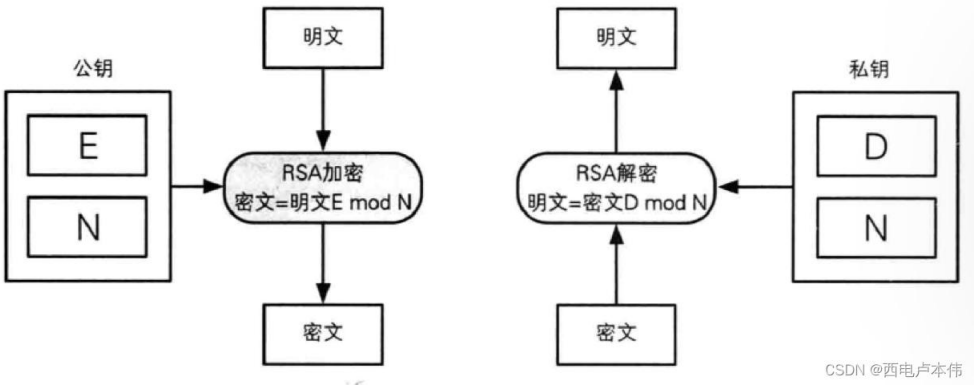
（1）选择两个大质数 p 和 q，计算 n = p * q。

（2）计算欧拉函数φ(n) = (p - 1) * (q - 1)。

（3）选择一个整数 e（公钥的加密指数），满足 1 < e < φ(n)且 e 与φ(n)互质。

（4）计算 d（私钥的解密指数），满足 ed ≡ 1 (mod φ(n))。

（5）公钥为(e, n)，私钥为(d, n)。



## 算法特点：

RSA 是一种非对称加密，使用一对密钥，且基于大数分解问题的困难性，不仅可以用于加密和解密数据，还可以用于数字签名，允许使用不同长度的密钥，与其他公钥加密算法相比，RSA 算法的实现相对容易理解，但在处理大数据时效率较低

### 3、在密码学中的应用：

（1）数据加密：
发送方可以使用接收方的公钥对数据进行加密，确保数据在传输过程中的机密性。只有拥有相应私钥的接收方才能解密数据，从而确保数据的安全性。这种加密方式在网络通信、云计算、移动设备等场景中得到了广泛应用，例如 HTTPS 协议就使用了 RSA 算法进行密钥交换和数据加密。
（2）数字签名：
RSA 算法也可以用于生成数字签名，用于验证数据的完整性和认证发送方的身份。发送方使用自己的私钥对数据进行签名，接收方使用发送方的公钥验证签名的有效性。
（3）身份认证：
RSA 算法还可以用于身份认证。例如，在网银等场景中，用户可以使用 RSA 算法生成一对公私钥，将公钥发送给银行。银行使用公钥对数据进行加密，只有用户拥有私钥才能解密，从而实现身份认证。
（4）密钥交换：
在网络通信中，RSA 算法还可以用于密钥交换。通信双方可以首先使用 RSA 算法交换对称加密算法的密钥，然后使用对称加密算法进行后续的数据传输。这种方式结合了 RSA 算法的安全性和对称加密算法的高效性，使得网络通信更加安全可靠。
（5）数字证书：
数字证书是一种由权威机构颁发的电子文件，用于证明某个公钥与某个实体（如个人、组织或设备等）之间的绑定关系。数字证书中通常包含公钥、实体信息、证书颁发机构信息等内容，并使用 RSA 算法进行签名以保证其真实性和完整性。通过验证数字证书的有效性，可以确定通信对方的身份和公钥的可靠性。

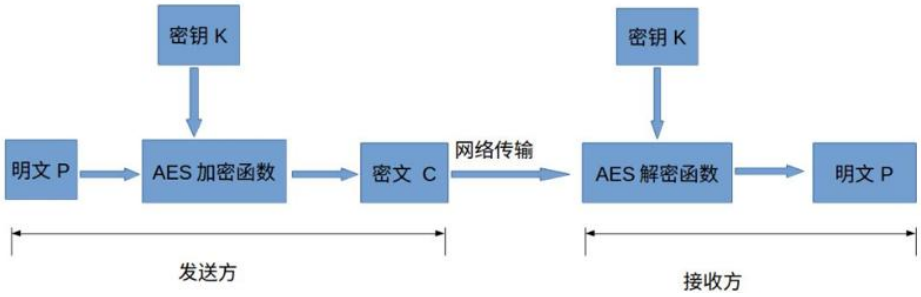
## Topic 3 [AES 加密]：

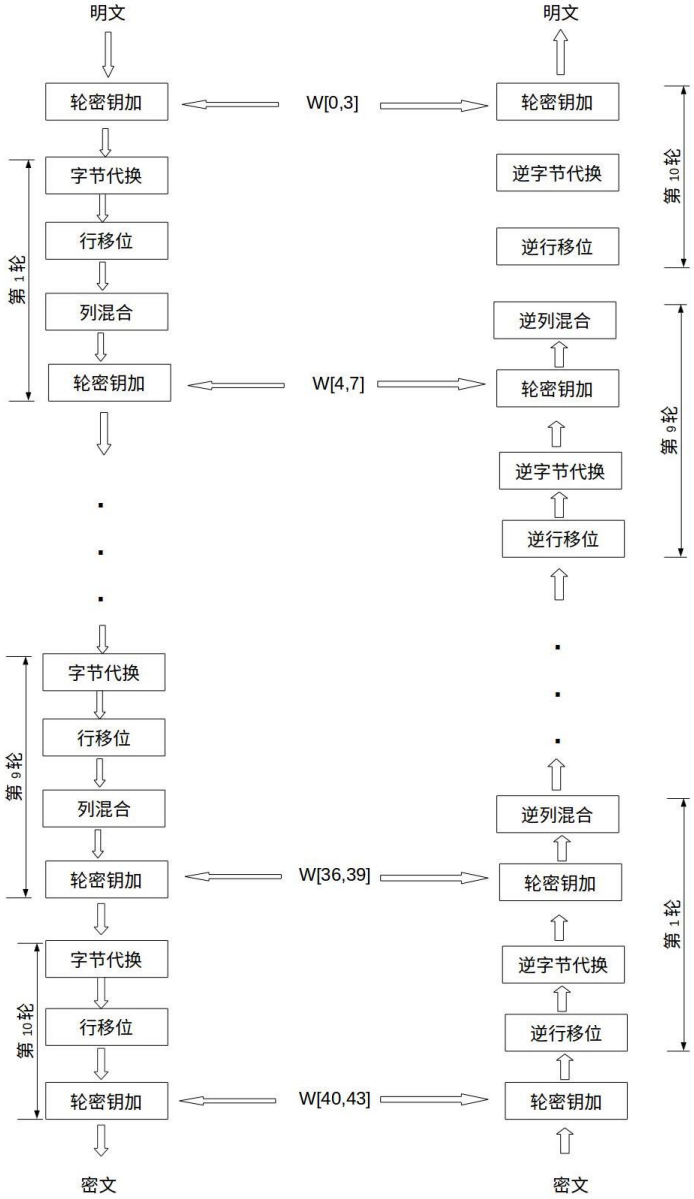
### 1、密码原理：

AES 加密算法是一种分组密码体制，其将明文按照固定大小进行分组，然后对每一分组进行加密。在加密过程中采用了多轮加密的方式，每一轮加密都包含了四种操作：SubBytes、ShiftRows、MixColumns 和 AddRoundKey。
AES 加密算法也是一种对称加密算法，加密和解密操作是相反的过程，因此需要使用相同的密钥作为加密和解密的关键参数。AES 算法支持三种密钥长度：

128 比特、192 比特和 256 比特，对于不同长度的密钥，AES 算法也会采用不同的轮数进行加密。

以 128 比特为例，一共进行 10 轮轮变换，前九轮依次进行 SubBytes、ShiftRows、MixColumns 和 AddRoundKey；最后一轮则只进行 SubBytes、ShiftRows 和 AddRoundKey；



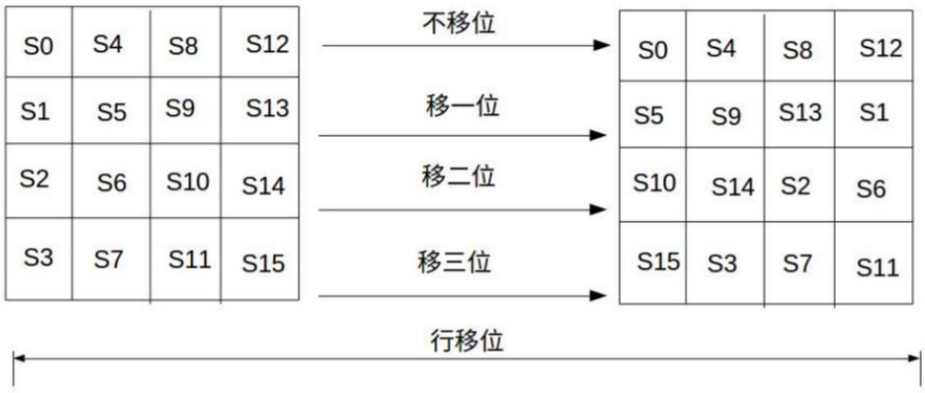
SubBytes：字节代换
根据字节内容进行查表代换
S 盒：

| 行/列 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0x63 | 0x7c | 0x77 | 0x7b | 0xf2 | 0x6b | 0x6f | 0xc5 | 0x30 | 0x01 | 0x67 | 0x2b | 0xfe | 0xd7 | 0xab | 0x76 |
| 1 | 0xca | 0x82 | 0xc9 | 0x7d | 0xfa | 0x59 | 0x47 | 0xf0 | 0xad | 0xd4 | 0xa2 | 0xaf | 0x9c | 0xa4 | 0x72 | 0xc0 |
| 2 | 0xb7 | 0xfd | 0x93 | 0x26 | 0x36 | 0x3f | 0xf7 | 0xcc | 0x34 | 0xa5 | 0xe5 | 0xf1 | 0x71 | 0xd8 | 0x31 | 0x15 |
| 3 | 0x04 | 0xc7 | 0x23 | 0xc3 | 0x18 | 0x96 | 0x05 | 0x9a | 0x07 | 0x12 | 0x80 | 0xe2 | 0xeb | 0x27 | 0xb2 | 0x75 |
| 4 | 0x09 | 0x83 | 0x2c | 0x1a | 0x1b | 0x6e | 0x5a | 0xa0 | 0x52 | 0x3b | 0xd6 | 0xb3 | 0x29 | 0xe3 | 0x2f | 0x84 |
| 5 | 0x53 | 0xd1 | 0x00 | 0xed | 0x20 | 0xfc | 0xb1 | 0x5b | 0x6a | 0xcb | 0xbe | 0x39 | 0x4a | 0x4c | 0x58 | 0xcf |
| 6 | 0xd0 | 0xef | 0xaa | 0xfb | 0x43 | 0x4d | 0x33 | 0x85 | 0x45 | 0xf9 | 0x02 | 0x7f | 0x50 | 0x3c | 0x9f | 0xa8 |
| 7 | 0x51 | 0xa3 | 0x40 | 0x8f | 0x92 | 0x9d | 0x38 | 0xf5 | 0xbc | 0xb6 | 0xda | 0x21 | 0x10 | 0xff | 0xf3 | 0xd2 |
| 8 | 0xcd | 0x0c | 0x13 | 0xec | 0x5f | 0x97 | 0x44 | 0x17 | 0xc4 | 0xa7 | 0x7e | 0x3d | 0x64 | 0x5d | 0x19 | 0x73 |
| 9 | 0x60 | 0x81 | 0x4f | 0xdc | 0x22 | 0x2a | 0x90 | 0x88 | 0x46 | 0xee | 0xb8 | 0x14 | 0xde | 0x5e | 0x0b | 0xdb |
| A | 0xe0 | 0x32 | 0x3a | 0x0a | 0x49 | 0x06 | 0x24 | 0x5c | 0xc2 | 0xd3 | 0xac | 0x62 | 0x91 | 0x95 | 0xe4 | 0x79 |
| B | 0xe7 | 0xc8 | 0x37 | 0x6d | 0x8d | 0xd5 | 0x4e | 0xa9 | 0x6c | 0x56 | 0xf4 | 0xea | 0x65 | 0x7a | 0xae | 0x08 |
| C | 0xba | 0x78 | 0x25 | 0x2e | 0x1c | 0xa6 | 0xb4 | 0xc6 | 0xe8 | 0xdd | 0x74 | 0x1f | 0x4b | 0xbd | 0x8b | 0x8a |
| D | 0x70 | 0x3e | 0xb5 | 0x66 | 0x48 | 0x03 | 0xf6 | 0x0e | 0x61 | 0x35 | 0x57 | 0xb9 | 0x86 | 0xc1 | 0x1d | 0x9e |
| E | 0xe1 | 0xf8 | 0x98 | 0x11 | 0x69 | 0xd9 | 0x8e | 0x94 | 0x9b | 0x1e | 0x87 | 0xe9 | 0xce | 0x55 | 0x28 | 0xdf |

逆 S 盒：

| 行/列 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0x52 | 0x09 | 0x6a | 0xd5 | 0x30 | 0x36 | 0xa5 | 0x38 | 0xbf | 0x40 | 0xa3 | 0x9e | 0x81 | 0xf3 | 0xd7 | 0xfb |
| 1 | 0x7c | 0xe3 | 0x39 | 0x82 | 0x9b | 0x2f | 0xff | 0x87 | 0x34 | 0x8e | 0x43 | 0x44 | 0xc4 | 0xde | 0xe9 | 0xcb |
| 2 | 0x54 | 0x7b | 0x94 | 0x32 | 0xa6 | 0xc2 | 0x23 | 0x3d | 0xee | 0x4c | 0x95 | 0x0b | 0x42 | 0xfa | 0xc3 | 0x4e |
| 3 | 0x08 | 0x2e | 0xa1 | 0x66 | 0x28 | 0xd9 | 0x24 | 0xb2 | 0x76 | 0x5b | 0xa2 | 0x49 | 0x6d | 0x8b | 0xd1 | 0x25 |
| 4 | 0x72 | 0xf8 | 0xf6 | 0x64 | 0x86 | 0x68 | 0x98 | 0x16 | 0xd4 | 0xa4 | 0x5c | 0xcc | 0x5d | 0x65 | 0xb6 | 0x92 |
| 5 | 0x6c | 0x70 | 0x48 | 0x50 | 0xfd | 0xed | 0xb9 | 0xda | 0x5e | 0x15 | 0x46 | 0x57 | 0xa7 | 0x8d | 0x9d | 0x84 |
| 6 | 0x90 | 0xd8 | 0xab | 0x00 | 0x8c | 0xbc | 0xd3 | 0x0a | 0xf7 | 0xe4 | 0x58 | 0x05 | 0xb8 | 0xb3 | 0x45 | 0x06 |
| 7 | 0xd0 | 0x2c | 0x1e | 0x8f | 0xca | 0x3f | 0x0f | 0x02 | 0xc1 | 0xaf | 0xbd | 0x03 | 0x01 | 0x13 | 0x8a | 0x6b |
| 8 | 0x3a | 0x91 | 0x11 | 0x41 | 0x4f | 0x67 | 0xdc | 0xea | 0x97 | 0xf2 | 0xcf | 0xce | 0xf0 | 0xb4 | 0xe6 | 0x73 |
| 9 | 0x96 | 0xac | 0x74 | 0x22 | 0xe7 | 0xad | 0x35 | 0x85 | 0xe2 | 0xf9 | 0x37 | 0xe8 | 0x1c | 0x75 | 0xdf | 0x6e |
| A | 0x47 | 0xf1 | 0x1a | 0x71 | 0x1d | 0x29 | 0xc5 | 0x89 | 0x6f | 0xb7 | 0x62 | 0x0e | 0xaa | 0x18 | 0xbe | 0x1b |
| B | 0xfc | 0x56 | 0x3e | 0x4b | 0xc6 | 0xd2 | 0x79 | 0x20 | 0x9a | 0xdb | 0xc0 | 0xfe | 0x78 | 0xcd | 0x5a | 0xf4 |
| C | 0x1f | 0xdd | 0xa8 | 0x33 | 0x88 | 0x07 | 0xc7 | 0x31 | 0xb1 | 0x12 | 0x10 | 0x59 | 0x27 | 0x80 | 0xec | 0x5f |
| D | 0x60 | 0x51 | 0x7f | 0xa9 | 0x19 | 0xb5 | 0x4a | 0x0d | 0x2d | 0xe5 | 0x7a | 0x9f | 0x93 | 0xc9 | 0x9c | 0xef |
| E | 0xa0 | 0xe0 | 0x3b | 0x4d | 0xae | 0x2a | 0xf5 | 0xb0 | 0xc8 | 0xeb | 0xbb | 0x3c | 0x83 | 0x53 | 0x99 | 0x61 |
| F | 0x17 | 0x2b | 0x04 | 0x7e | 0xba | 0x77 | 0xd6 | 0x26 | 0xe1 | 0x69 | 0x14 | 0x63 | 0x55 | 0x21 | 0x0c | 0x7d |

ShiftRows：行移位
将明文分组（一个 4x4 的字节矩阵）的每一行进行循环移位；第一行保持不变，第二行向左循环移位 1 个字节，第三行向左循环移位 2 个字节，第四行向左循环移位 3 个字节；




MixColumns：列混淆

列混淆操作使用了一个固定的矩阵（在 AES 标准中定义）与每一列进行乘法运算。这里的乘法不是常规的十进制乘法，而是基于伽罗瓦域（Galois Field）GF(2^8)上的运算

列混合变换是通过矩阵相乘来实现的，经行移位后的状态矩阵与固定的矩阵相乘，得到混淆后的状态矩阵，如下图的公式所示：

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

状态矩阵中的第j列(0 ≤j≤3)的列混合可以表示为下图所示：

$$s'_{0,j} = (2 * s_{0,j}) \oplus (3 * s_{1,j}) \oplus s_{2,j} \oplus s_{3,j}$$
$$s'_{1,j} = s_{0,j} \oplus (2 * s_{1,j}) \oplus (3 * s_{2,j}) \oplus s_{3,j}$$
$$s'_{2,j} = s_{0,j} \oplus s_{1,j} \oplus (2 * s_{2,j}) \oplus (3 * s_{3,j})$$
$$s'_{3,j} = (3 * s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 * s_{3,j})$$

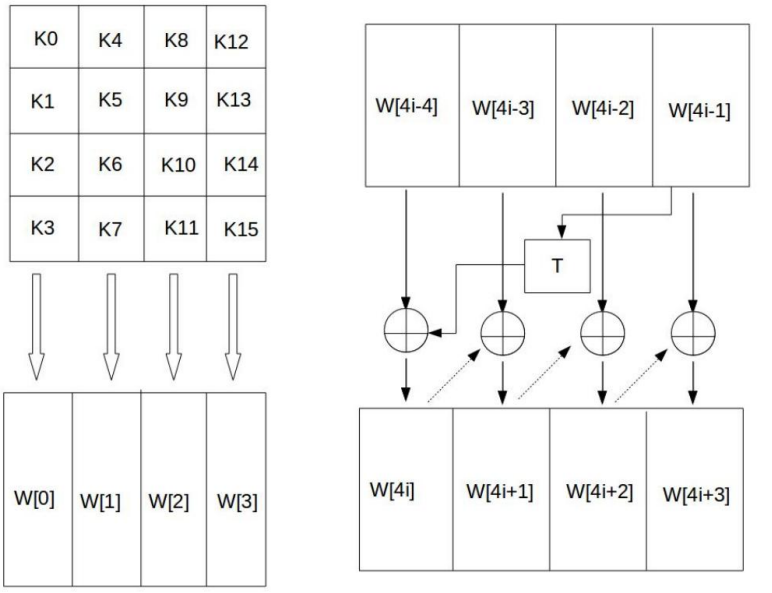
代码实现：

```
// MixColumns变换
unsigned char s0 = gfMul2(column[0]) ^ gfMul3(column[1]) ^ column[2] ^ column[3];
unsigned char s1 = column[0] ^ gfMul2(column[1]) ^ gfMul3(column[2]) ^ column[3];
unsigned char s2 = column[0] ^ column[1] ^ gfMul2(column[2]) ^ gfMul3(column[3]);
unsigned char s3 = gfMul3(column[0]) ^ column[1] ^ column[2] ^ gfMul2(column[3]);
```

AddRoundKey：轮密钥加

当前分组和对应的该轮扩展密钥进行按位异或，将输入或中间态 S 的每一列与一个密钥字 ki 进行按位异或，即将 128 位轮密钥 Ki 同状态矩阵中的数据进行逐位异或操作。

KeyExtend：轮密钥扩展




（1）扩展密钥数组

扩展过程：为了进行多轮加密，需要对 W 数组进行扩展，增加 40 个新的列，形成总共 44 列的扩展密钥数组。

新列的生成：

如果 i（新列的索引）不是 4 的倍数，则第 i 列由前一个 4 的倍数列（W[i-4]）和前一个列（W[i-1]）进行异或（⊕）运算得到：W[i] = W[i-4] ⊕ W[i-1]；

如果 i 是 4 的倍数，则第 i 列由前一个 4 的倍数列（W[i-4]）和经过函数 T 处理的前一个列（W[i-1]）进行异或运算得到：W[i] = W[i-4] ⊕ T(W[i-1])；

（2）函数 T

函数 T 用于生成 4 的倍数列时的特殊处理，它由以下三部分组成：

字循环（Word Shift）：将输入的 32 位字中的 4 个字节循环左移 1 个字节。即，如果输入字是[b0, b1, b2, b3]，则输出将是[b1, b2, b3, b0]。

字节代换（Byte Substitution）：对字循环的结果使用 AES 的 S 盒（Substitution-box）进行字节级的代换。S 盒是一个预定义的替换表，用于将每个输入字节映射到一个输出字节。

轮常量异或（Rcon XOR）：将前两步的结果（即字循环和字节代换后的字）与一个轮常量 Rcon[j]进行异或运算。这里的 j 表示当前的轮数（round number）。

轮常量 Rcon[j]：

```
string RconBox[4][10] = {
    {"01","02","04","08","10","20","40","80","1B","36"},
    {"00","00","00","00","00","00","00","00","00","00"},
    {"00","00","00","00","00","00","00","00","00","00"},
    {"00","00","00","00","00","00","00","00","00","00"},
};
```

## 2、应用场景：

数据保护：AES 算法被广泛用于保护敏感数据的机密性。这包括个人身份信息、商业机密、政府数据等。通过使用 AES 加密，可以确保数据在存储、传输或处理过程中不被未经授权的第三方访问或泄露。
网络通信安全：在网络通信中，AES 用于确保数据的完整性和保密性。无论是电子邮件、即时消息、文件传输还是在线交易，AES 都可以提供强大的加密保护，防止数据在传输过程中被截获或篡改。
软件安全：AES 也常用于软件安全领域，如加密软件、安全通信协议等。通过集成 AES 加密算法，软件可以提供更高级别的数据保护和身份验证功能。
数字内容保护：对于数字内容（如电影、音乐、电子书等），AES 加密算法可以用于防止未经授权的复制和分发。通过加密内容，可以确保只有拥有正确密钥的用户才能访问和使用这些内容。
硬件安全：在硬件领域，AES 加密算法也扮演着重要角色。例如，智能卡、安全芯片和加密设备等常常使用 AES 来保护存储在其中的敏感数据。

## 3、包含的数学问题：

### 有限域（GF(2^8)）

在 AES 中，使用的是 GF(2^8)，即包含 256 个元素的有限域。这些元素对应于一个字节（8 位）的所有可能值（从 00 到 FF 的十六进制数）。
加法与减法：在 GF(2^8)中，加法和减法实际上是相同的操作，因为它们是在二进制（模 2）下进行的。加法通过异或（XOR）操作实现，即如果两个对应的二进制位不同，则结果为 1，否则为 0。
乘法：乘法在 GF(2^8)中稍微复杂一些，因为它涉及到多项式乘法和模多项式除法。AES 使用一个固定的模多项式（称为不可约多项式）来定义乘法操作。
逆元：在 GF(2^8)中，每个非零元素都有一个乘法逆元，即存在一个元素使得与该元素相乘的结果为 1（模多项式的余数为 1）。逆元在 AES 的列混合步骤中用于实现矩阵的逆运算。
有限域的使用确保了 AES 算法中的运算都是可逆的，并且运算结果始终在有限的范围内。这为算法的加密和解密过程提供了数学基础。

### 线性代数（Linear Algebra）

线性代数在 AES 算法中主要体现在列混合步骤中。列混合是一个线性变换过程，它通过一个固定的 4x4 矩阵与状态数组（一个 4x1 的列向量）相乘来实现。在列混合步骤中，状态数组的每一列都被视为一个 4x1 的列向量，并与一个固定的 4x4 矩阵相乘。这个矩阵乘法操作是在 GF(2^8)上进行的，因此涉及到有限域中的乘法运算；线性代数的应用使得 AES 算法在加密过程中具有更高的复杂性和安全性。通过矩阵乘法和扩散性的实现，AES 能够抵御各种密码分析攻击，并保护数据的机密性和完整性。