目录

第一章　绪论

1.1　什么是密码学

如果我们要求一个从未接触过密码学的人处理一段文字，把这段文字尽可能地加密，让别人无法破解。那么，大多数人在深思熟虑之后，总能提出一些加密算法。这时候，一般人的思路可能会有几个方向。

有的人的方向是把这段文字中的字母之间通过各种复杂的运算进行组合。比如说，把要加密的文字中每两个字母在字母表中的位置进行相加，形成密码。比如说，“cryptography” 中，“cr” 变成了 $3+18=21$, “yp” 变成了 $25+16=41$, “cryptography” 对应的密码就是 “214135252133”。但这样的密码，显然忽略了一点：密码的可解密性。我们之所以要进行加密，是为了安全地传递信息。但接收信息者必须要有解密的方法。但这种加密方式则没有对应的解密方法。所以说，这不是一个合格的加密方式。

这时，足够聪明的人，则考虑到了解密的方法。他们想出了一些类似于古代使用的加密方法。比如说，著名的凯撒密码：把一段由英文字母组成的语句，每个字母都在字母表中往后移 3 个位置。“veni vidi vici” 也就变成了 “yhql ylgl ylfl”. 这种密码的解密方法，也就是每个字母在字母表中向前移动 3 个位置。当问起这种密码体制里最重要的是什么，大部分人都会说是加密算法。如果把加密算法告诉了别人，那这种密码就相当于被破解了。

如果进而问起一个密码体制的组成，大部分人的回答，就是加密算法。也许有些自诩比普通人聪明一点的人，还会加上一个解密算法，也就是，将处理过后的密码变为正常文字的算法。如果用 m 代表要被加密的文字，c 代表被加密出来的密码，E() 代表加密算法，D() 代表解密算法，那么大多数人理解的密码学，本质就是以下这两个式子:

$$c = \mathrm{E}(m)$$
$$m = \mathrm{D}(c)$$

1.1.1　密钥

但是，我们考虑一下实际的情况。根据我们之前的常识，会发现，如果要用密码来传递信息，首先通信的双方必须要在一个安全信道中传递一些额外的信息。比如说，告诉接收者加密的方式，或者更直接地，告诉接收者解密算法。不存在一种加密的方式，能不事先通过安全信道传递信息，而使得只有接收者能够解密。那么，为了使通信更加安全，通信双方对安全信道的使用应该尽可能少。如果 Alice 和 Bob 使用凯撒密码进行通信，而 Alice 事先在安全信道中要对 Bob 说 “我的这种加密方式是把一段由英文字母组成的语句，每个字母都在字母表中往后移 3 个位置。” 这段话是如此之长，而如果考虑是在战场上，由通信员用摩尔斯电码发送，那么花费的这么长的时间显然是不合理的。再者，现代的通信方式一般都是把信息编码成二进制进行传递，那么，相比把一段话编码成二进制，不如找一些数字来代表这个密码。在我们

的这段思想博弈中，一个重要的概念呼之欲出。

拿凯撒密码为例，在其加密算法中，还有一个关键的量：3. 试想，如果通信双方 Alice 和 Bob, Alice 在场的所有人她用的是凯撒密码的加密方式加密她的话，又在安全信道中告诉 Bob“4”, 意味着她使用的凯撒密码里，是将每个字母在字母表的位置上向后移动 4 个位置。那么，即使在场有人能偷听到 Alice 给 Bob 的密码，也无法破解 Alice 想说的是什么，只有掌握了规则“向后移动 4 个位置”的 Bob, 才能正确地破解密码。

如果还是按照之前的说法，一个密码体制包括加密算法和解密算法，那么向后移动 3 个位置的凯撒密码和向后移动 4 个位置的凯撒密码，就成了两个密码体制。但是，这两种加密方式是极其类似的，差别也就只在于一个向后移动 3 个位置，一个向后移动 4 个位置。我们对于这些密码的研究，也许也就十分类似。因此，把这两种加密方式归类为同一种密码体制，似乎是更好的选择。因此，**密钥**就应运而生了。所谓密钥，我们可以粗略地理解成加密算法、解密算法中的参数，也就是我们之前说的 3、4. 通过一个密码体制通信的双方，需要首先在保密信道中确定密钥。而一个密码体制，也就可以由加密算法、解密算法和密钥构成。因此，如果以 k 代表密钥，那么之前的式子就变成了

$$c = \mathrm{E}_k(m)$$
$$m = \mathrm{D}_k(c)$$

这也就是现在通用的密码学。

那么我们思考几个问题：在一次加密通信过程中，加密算法和解密算法中使用的密钥是否必须要相同？密钥是否只能是一个数？可不可以没有密钥？

针对第一个问题，确实存在某些高端的技巧，使得加密算法和解密算法使用的密钥是不同的。事实上，在密码学领域中，根据加密算法和解密算法使用的密钥是否相同，人们将加密方式分为对称加密与非对称加密，分别对应使用同一密钥和使用不同密钥。非对称加密方式也许难以理解，我们也会在充分学习对称加密后再介绍非对称加密。所以，我们接下来讨论的对称加密过程中，请大家记住，加密算法和解密算法使用的是相同的密钥。

此外，对于对称加密算法，密钥也并不一定是一个数。比如说，加密算法

$$\mathrm{E}_{a,b}(m) = am + b$$

其密钥为 (a, b), 但我们仍认为其使用单一密钥，也就是说，把 (a, b) 看作一个密钥。

此外，可不可能不存在密钥呢？确实有这样的密码体制，但这样却也十分不安全。比如说，加密算法

$$\mathrm{E}(m) = m$$

就是一个无密钥的加密算法。

1.2 密码体制的基本要素

根据之前的讨论，我们就可以得出一个密码体制的基本要素：

- 明文空间 $\mathcal{M}$

  所有可以被加密算法加密的元素组成的集合，加密算法的定义域。明文空间的元素叫做明文 $m \in \mathcal{M}$.

  例如，在凯撒密码中，明文空间就为所有由英文字母组成的字符串。

- 密文空间 $\mathcal{C}$

  所有可以由加密算法输出的元素组成的集合，加密算法的值域。密文空间的元素叫做密文 $c \in \mathcal{C}$.

  我们需要注意到的是，这里的值域，可以理解成二元函数 $\mathrm{E}(k, m)$ 的值域。比如说，对于加密算法

  $$\mathrm{E}_k(m) = m^2 + k^2$$

  其密文空间为 $[0, +\infty)$ 而非 $[k^2, +\infty)$.

- 密钥空间 $\mathcal{K}$

  所有密钥组成的集合。密钥空间的元素叫做密钥 $k \in \mathcal{K}$.

  在非对称加密中，密钥空间分为加密密钥空间和解密密钥空间。

- 加密算法 $\mathrm{E}_k(m)$

  根据密钥生成的特定算法，将明文转化为密文。

- 解密算法 $\mathrm{D}_k(c)$

  根据密钥生成的特定算法，将密文转化为明文。

  对于对称加密算法，也就是只使用一个密钥的加密体制，解密算法与加密算法满足

  $$\mathrm{D}_k(\mathrm{E}_k(m)) = m$$

在我们讨论密码体制的一些性质时，密钥生成算法有时也是必要的。什么是密钥生成算法呢？回忆之前 Alice 和 Bob 的例子，凭什么 Alice 选择的密钥是 4 而不是 25 呢？这就涉及到了密钥生成算法。在这个例子中，密钥生成算法就是 Alice 自己想到哪个密钥就输出哪个密钥。但是，从严格意义上来讲，密钥生成算法是一种概率算法。所谓概率算法，就是在算法的步骤中涉及到了某些概率。比如说，在某个密钥生成算法中，在 $(0, 1)$ 中等概率随机生成一个数 t, 而生成的密钥 k 满足

$$k = \begin{cases} 1 & t \in (0.5, 1) \\ 0 & t = 0.5 \\ -1 & t \in (0, 0.5) \end{cases}$$

这就是一个典型的概率算法。其特点就是在两次运行中输出的结果不一定相同。

因此，我们称一个加密方案包含三个要素：加密算法 E, 解密算法 D, 密钥生成算法 G. 根据定义，我们可以说，一个密码体制由一个加密方案 $(\mathrm{E}, \mathrm{D}, \mathrm{G})$ 及一个明文空间 $\mathcal{M}$ 完全定义。

因此，Alice 和 Bob 的一次加密通信的过程包括：

1. Alice 根据密钥生成算法 G 生成密钥 $k \in \mathcal{K}$.
2. Alice 通过安全信道将 k 告诉 Bob.
3. Alice 将想要传达的明文 $m \in \mathcal{M}$ 根据加密算法加密成密文 $c = \mathrm{E}_k(m)$ 告诉 Bob.
4. Bob 根据之前 Alice 告诉自己的 k、密文 c 及解密算法得出明文 $m = \mathrm{D}_k(c)$

那么，在一个密码体制中，哪个最重要呢？是不是之前我们说的加密算法呢？这里，就不得不提 Kerckhoffs 原则。用现代的语言来说，Kerckhoffs 原则阐述的是：

提倡安全性不能建立在对算法的保密上。

也就是说，我们如果要证明一个加密体制的安全性，不能指望算法的保密性。我们应默认加解密算法可以被所有人知道（事实上也确实如此）。也就是说，真正值得保密的，是密钥。如果潜在的敌手获得密钥，那么根据公开的解密算法，那么他就可以从窃得的密文中获得明文。

1.2.1 弱密钥与半弱密钥

在我们构造密码体系的时候，有的人会想，利用已有的密码体系加密两次怎么样？即：

$$\mathrm{E}_k\left(\mathrm{E}_k\left(m\right)\right)$$

或者

$$\mathrm{E}_{k_1}\left(\mathrm{E}_{k_2}\left(m\right)\right)$$

的安全性如何？

这里提出了弱密钥与半弱密钥的概念：

定义 1.1. *对于加密方式 $\mathrm{E}_k\left(m\right)$, 若密钥 k 使得对于任意 $m \in \mathcal{M}$, 有*

$$\mathrm{E}_k\left(\mathrm{E}_k\left(m\right)\right) = m$$

则称 k 为弱密钥。

若密钥 k_1, k_2 若密钥 k 使得对于任意 $m \in \mathcal{M}$, 有

$$\mathrm{E}_{k_1}\left(\mathrm{E}_{k_2}\left(m\right)\right) = m$$

则称 k_1, k_2 为一对半弱密钥。

由上述定义可知，如果我们想要利用已有的密码体系加密两次，那么一定要避开的就是弱密钥与半弱密钥。

1.3 经典密码

1.3.1 基础概念

我们讨论了密码体制的基本要素之后，就可以介绍一些经典的密码，让大家更好地理解这些术语了。

值得指出的是，这些密码都是古代欧洲人的研究成果，当时并没有如今“数字化”的概念。因此，这些密码，都是针对拉丁字母进行的加密。因此，我们首先要引入一些概念：

函数 $C\left(m\right)$ 将拉丁字母 m 映射到它在字母表中的位置上，比如 $C(a) = 1, C(z) = 26.$ 函数 $I\left(n\right)$ 将位于 1 和 26 之间的数字映射到字母表中相应位置的拉丁字母上，比如 $I(1) = a, I(26) = z.$

在讨论经典密码时，一些极其基础的数论记号及知识可以让我们更加方便、更加简洁地叙述、理解这些经典密码。

我们用 $\gcd\left(a, b\right)$ 表示 a 与 b 的最大公因数。

用 $a \bmod b$ 表示整数 a 除以 b 后的余数（取值范围为 0 到 $b-1$）, 比如说 $15 \bmod 6 = 3, 12 \bmod 6 = 0, (-4) \bmod 6 = 2.$

若 $a \bmod b = 0$, 即 a 能整除 b, 我们则记为 $a \mid b$. 如 $2 \mid 4, 3 \nmid 4$.

若 $(a-b) \mid c$, 我们则称 a 与 b 模 c 同余，记作 $a \equiv b \pmod{c}$. 如 $16 \equiv 23 \pmod{7}$.

对于整数 a, b, 若存在整数 c 使得 $ac \equiv 1 \pmod{b}$, 则称 c 为 a 在模 b 时的逆，记作 a^{-1}. 并非所有的整数都有逆，如在模 4 的情况下，整数 2 就没有逆。对于整数 a, b, 在模 b 时 a 存在逆的充分必要条件为 $\gcd(a, b) = 1$.

1.3.2 单表代换密码

单表代换密码的典型，就是凯撒密码。如果用我们上述的记号来表示凯撒密码的过程，那么如果设明文为字符串 "$m_1 m_2 \cdots m_n$", 密文为字符串 "$c_1 c_2 \cdots c_n$", m_i, c_i 均代表一个拉丁字母。凯撒密码的加密算法

$$c_i = \mathrm{E}(m_i) = C((I(m_i) + 3) \bmod 26) \tag{1.1}$$

解密算法

$$m_i = \mathrm{D}(c_i) = C((I(c_i) - 3) \bmod 26) \tag{1.2}$$

它通过对字母表中每个字母进行固定的代换，得到密码。单表替换密码则是凯撒密码的推广，引入了密钥。从数学意义上，可以作如下定义：

设明文为字符串 "$m_1 m_2 \cdots m_n$", 密文为字符串 "$c_1 c_2 \cdots c_n$", m_i, c_i 均代表一个拉丁字母。如果把整数对 (a, b) 作为密钥，其中 $a \neq 0$, 那么其加密算法

$$c_i = \mathrm{E}_{a,b}(m_i) = C((aI(m_i) + b) \bmod 26)$$

解密算法

$$m_i = \mathrm{D}_{a,b}(c_i) = C\left(\left(a^{-1}(I(c_i) - b)\right) \bmod 26\right)$$

其中 a^{-1} 为 a 模 26 的逆。

事实上，如果我们令 $n_i = I(m_i), q_i = I(\mathrm{E}_{a,b}(m_i)), e_i = I(\mathrm{E}_{a,b}(m_i)), d_i = I(\mathrm{D}_{a,b}(\mathrm{E}_{a,b}(m_i)))$, 那么

$$q_i \equiv a n_i + b \pmod{26}$$

故

$$\begin{aligned} d_i &\equiv a^{-1}(q_i - b) \\ &\equiv a^{-1}(a n_i + b - b) \\ &\equiv n_i \pmod{26} \end{aligned}$$

也就是说，

$$I(\mathrm{D}_{a,b}(\mathrm{E}_{a,b}(m_i))) \equiv m_i \pmod{26}$$

故

$$\mathrm{D}_{a,b}(\mathrm{E}_{a,b}(m_i)) = m_i$$

这也就证明了这个加密算法是正确的算法。

让我们不要再纠结于繁琐的数学符号，我们来从直观上看一看这个加密算法。任意取一个密钥，比如说，$a = 3, b = 7$, 就会对应的生成一张加密表和解密表：

表 1.1: $a = 3, b = 7$ 时的加密表

明文	a	b	c	d	e	f	g	h	i	j	k	l	m
密文	h	k	n	q	t	w	z	c	f	i	l	o	r
明文	n	o	p	q	r	s	t	u	v	w	x	y	z
密文	u	x	a	d	g	j	m	p	s	v	y	b	e

表 1.2: $a = 3, b = 7$ 时的解密表

密文	a	b	c	d	e	f	g	h	i	j	k	l	m
明文	p	y	h	q	z	i	r	a	j	s	b	k	t
密文	n	o	p	q	r	s	t	u	v	w	x	y	z
明文	c	l	u	d	m	v	e	n	w	f	o	x	g

那么我们根据这张表，就可以很快地进行加密和解密的工作了。

我们再回到之前所说的加密体制的基本要素：其明文空间 $\mathcal{M}$ 为由拉丁字母组成的任意长度的字符串组成的集合，密文空间 $\mathcal{C} = \mathcal{M}$. 密钥空间 $\mathcal{K} = \{(a, b) \mid a, b \in \mathbb{Z}, \gcd(a, 26) = 1\}$(这里 $\gcd(a, 26) = 1$ 的条件是因为解密算法中要求 a 模 26 的逆存在。).

1.3.3 单表代换密码的破解

上述的单表代换密码看似十分安全，但是如果用来加密由拉丁字母组成的用语言逻辑形成的一句话时，却有一个致命的弱点。虽然这些密码不会直接暴露明文，但却会暴露明文中各个字母出现的频率。我们知道，在任何一门由字母组成的语言文字中，每个字母出现的频率在语句十分长时是趋向于一个定值的。比如说，在英文中，有一个著名的短语：“ETAOIN SHRDLU”. 这个短语是英文中出现频率最高的 12 个字母，从高到低排列。根据这些频率，就可以找到破解这种密码的方法。

为什么暴露明文出现的频率就会使加密系统不安全呢？我们可以用一个例子来说明：我们利用单表代换密码的原理，加密东南大学的校徽：

图 1.1: 单表代换加密前

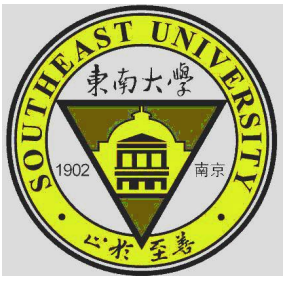


图 1.2: 单表代换加密后

显而易见，加密效果近似于无。

利用这个原理，有两种最常用的方法：频率分析法与巧合指数法。

频率分析法

假设利用单表代换密码加密的语言为英文，那么根据统计学家的知识，英文字母出现的频率从高到低依次是 “ETAOIN SHRDLU”. 如果我们可以统计出一个相当长的密文中各个字母

出现的频率，那么极有可能出现频率最高的字母对应的就是明文中的 “E”. 这就是频率分析法的基本思想。

巧合指数法

巧合指数法的想法非常直接：对于一段文字，任意取两个字母，这两个字母相同的概率称为巧合指数 IC。可以证明，对于一段长度为 N 的文字，共有 c 个字母，每个字母出现的次数为 $n_i, i = 1, 2, \ldots, c$. 那么巧合指数的值为

$$\mathrm{IC} = \frac{\sum_{i=1}^{c} n_i \left(n_i - 1\right)}{N \left(N - 1\right)} \tag{1.3}$$

如果这段文字充分长，那么我们有

$$\frac{\sum_{i=1}^{c} n_i \left(n_i - 1\right)}{N \left(N - 1\right)} \approx \frac{\sum_{i=1}^{c} n_i^2}{N^2} \tag{1.4}$$

如果记 $p_i = \dfrac{n_i}{N}$ 表示第 i 个字母在这段文字中出现的频率，那么

$$\mathrm{IC} \approx \sum_{i=1}^{c} p_i^2 \tag{1.5}$$

而我们先前提到，统计学家已经统计出在英文文本中各个字母出现的频率，因此，带入上述式子，可以得出英文文本的巧合指数 $\mathrm{IC} \approx 0.0686$.

接着我们统计加密过后的密文的巧合指数，假设密钥为 k, 我们对 $k = 0, 1, \ldots, 25$ 依次去试，如果出现了解密后的文本的巧合指数接近于 0.0686, 就说明这个 k 有很大可能是密钥。

1.3.4 多表代换密码

为了进一步提高安全性，古代的人们想到也许一张表并不足够安全，不妨使用多张表。因此，多表代换密码应运而生。

假设一共有 t 张加密表，人们是怎么做的呢？从明文的第一个字符开始，第一个字符使用第一张加密表进行加密，第二个字符使用第二张加密表进行加密，以此类推，第 t 个字符使用第 t 张加密表进行加密。到了第 $t+1$ 个字符，则又回到第一张加密表进行加密。用数学的语言怎么叙述这件事呢？

对于明文字符串 $M = m_1 m_2 \cdots m_n$, 我们要求其长度满足 $n = tp$. 我们将明文字符串等分成 p 个列向量 $M_1, M_2, \ldots, M_p$, 其中 $M_i = \left(m_{p(i-1)+1}, m_{p(i-1)+2}, \ldots, m_{p(i-1)+t}\right)^{\mathrm{T}}$. 对密文字符串 C 也作同样的划分 $C_1, C_2, \ldots, C_p$. 取密钥为 (A, B), 其中矩阵 A 为 $t \times t$ 的可逆矩阵，且满足 $\gcd\left(|A|, 26\right) = 1$, B 为 t 维列向量。

那么多表代换密码的加密算法为

$$C_i = \mathrm{E}_{A,B}\left(M_i\right) = C\left(\left(AI\left(M_i\right) + B\right) \bmod 26\right) \tag{1.6}$$

解密算法为

$$M_i = \mathrm{D}_{A,B}\left(C_i\right) = C\left(\left(A^{-1} I\left(C_i - B\right)\right) \bmod 26\right) \tag{1.7}$$

其中 A^{-1} 满足

$$A^{-1}A \bmod 26 = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

我们可以用一个具体的例子来理解这些抽象的公式：

假设我们一共有三张表，分别使用密钥为 $(1,1),(3,3),(5,7)$ 的单表替换密码生成。明文字符串 $M = m_1 m_2 \cdots m_n$ 的长度 $n = 3p$. 我们将其等分为 p 个列向量 $M_1, M_2, \ldots, M_p$, 其中 $M_i = \left(m_{p(i-1)+1}, m_{p(i-1)+2}, m_{p(i-1)+3}\right)^{\mathrm{T}}$. 对于密文字符串 C 也作同样的划分 $C_1, C_2, \ldots, C_p$. 取

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{pmatrix}, B = \begin{pmatrix} 1 \\ 3 \\ 7 \end{pmatrix}$$

为密钥。

那么其加密算法为

$$\begin{aligned} \begin{pmatrix} c_{p(i-1)+1} \\ c_{p(i-1)+2} \\ c_{p(i-1)+3} \end{pmatrix} = C_i = \mathrm{E}_{A,B}(M_i) &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{pmatrix} M_i + \begin{pmatrix} 1 \\ 3 \\ 7 \end{pmatrix} \bmod 26 \\ &= \begin{pmatrix} m_{p(i-1)+1} + 1 \\ 3m_{p(i-1)+2} + 3 \\ 5c_{p(i-1)+3} + 7 \end{pmatrix} \bmod 26 \end{aligned}$$

其解密算法为

$$M_i = \mathrm{D}_{A,B}(C_i) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 21 \end{pmatrix} \left(C_i - \begin{pmatrix} 1 \\ 3 \\ 7 \end{pmatrix} \right) \bmod 26$$

那么我们可以看到，从明文的第一个字符开始，每隔三个字符的第一个字符使用的加密方法为 $c_{p(i-1)+1} = \left(m_{p(i-1)+1} + 1\right) \bmod 26$, 第二个字符使用的加密方法为 $c_{p(i-1)+2} = \left(3m_{p(i-1)+2} + 3\right) \bmod 26$, 第三个字符使用的加密方法为 $c_{p(i-1)+3} = \left(5m_{p(i-1)+3} + 7\right) \bmod 26$. 这也就是我们设计多表代换密码的原意。

我们发现，在上述例子中，密钥里的 A 似乎有许多 0 的位置。但是，我们之前在数学上严格定义多表代换密码的时候，并没有要求这些位置一定要是 0. 事实上，这些位置如果不是 0，就意味着密文中特定位置的字符并不是由明文中对应位置的字符确定，而是明文中对应位置与前后位置的字符一起确定。这也是可行的。

多表代换密码也是不安全的。我们利用多表代换密码同样加密上面提到的东南大学校徽，得到的结果也很不理想：

图 1.3: 多表代换加密前

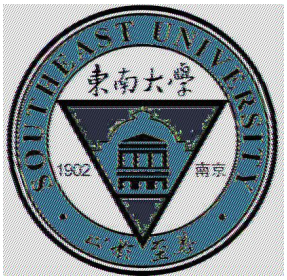


图 1.4: 多表代换加密后

1.3.5 一次一密

之前我们说道，单表代换密码可以根据每个字母出现的频率来破解，其实多表代换密码的破解也很类似。那么，有没有什么方法能使密文不显示明文中每个字母出现的频率呢？一次一密的方法就是答案。

为了加密某个长度为 n 的字符串，我们取另一个长度为 n 的字符串作为密钥。所得的密文就是明文每个字符在字母表中的位置与密钥每个字符在字母表中的位置相加。由密文得到明文也就是密文中每个字符在字母表中的位置与密钥每个字符在字母表中的位置相减。

这种方式之所以称为一次一密，是因为同一串密钥只能使用一次。试想如果有人窃得了用同一串密钥加密的两个密文 C_1, C_2, 将这两个字符串中的每个字符按其在字母表中的位置相减，那么如果出现 0, 那么对应位置就就有可能是出现频率比较高的几个字母。当然，更严谨的论证可以用之后提到的概率论的方式证明。

最后，值得一提的是，现代使用的量子加密中，最常用的加密机制就是一次一密。

1.4 加密系统的安全性

为了从数学上定义加密系统的安全性，我们必须引入一些和概率有关的定义。这时，请回忆一下之前定义的加密方案 $(\mathrm{E}, \mathrm{D}, \mathrm{G})$. 其中，E 代表加密算法，D 代表解密算法，G 代表密钥生成算法。此外，还有明文空间 $\mathcal{M}$, 密文空间 $\mathcal{C}$, 密钥空间 $\mathcal{K}$.

对于 $m \in \mathcal{M}$, M 为表示明文的随机变量，用 $\Pr[M = m]$ 表示明文为 m 的概率。这一定义看似难以理解，为什么明文会出现概率呢？从某种意义上，可以理解成假想的敌手在没有任何信息的情况下猜测明文的概率。比如说，在没有任何信息的情况下，一个敌手可能会假定明文为 “attack tomorrow” 或 “don’t attack”. 且敌手认为明文为 “attack tomorrow” 的概率为 0.6, 明文为 “don’t attack” 的概率为 0.4.

对于 $k \in \mathcal{K}$, K 为表示密钥值的随机变量，用 $\Pr[K = k]$ 表示随机算法 G 输出 k 的概率。(由常识可知，K 与 M 是独立的)

对于 $c \in \mathcal{C}$, C 为表示密文的随机变量，用 $\Pr[C = c]$ 表示密文为 c 的概率。由于密文是完全由明文及密钥确定的，所以我们可以知道：

$$\Pr[C = c] = \sum_{c = \mathrm{E}_k(m)} \Pr[M = m] \Pr[K = k] \tag{1.8}$$

我们定义完善保密 (perfect secrecy):

定义 1.2. *对于明文空间为 $\mathcal{M}$ 的加密方案 $(\mathrm{E}, \mathrm{D}, \mathrm{G})$，若对 $\mathcal{M}$ 上的任何概率分布，任何明*

文 $m \in \mathcal{M}$、任何密文 $c \in \mathcal{C}$ 且 $\Pr[C = c]$ 有

$$\Pr[M = m \mid C = c] = \Pr[M = m] \tag{1.9}$$

则称加密方案 $(\mathrm{E}, \mathrm{D}, \mathrm{G})$ 是完善保密。

用一句通俗的话来讲，就是敌手在窃取密文之后并不会对明文有任何知识。而用概率论的说法，则是随机变量 M 与 C 是独立的。

我们可以通过一些概率论上的技巧，证明“一次一密”的加密方式是完善保密。

但是，完善保密也有其胶柱鼓瑟之处：

定理 1.1. *若明文空间为 $\mathcal{M}$ 的加密方案 $(\mathrm{E}, \mathrm{D}, \mathrm{G})$ 是完善保密，则*

$$|\mathcal{K}| \geq |\mathcal{M}|, |\mathcal{C}| \geq |\mathcal{M}| \tag{1.10}$$

如果用通俗的语言解释上述定理，则是说：要想实现完善保密，则密钥至少要和明文一样长，而密文则也至少要和明文一样长。

我们之前说过，要想实现加密通信，通信双方一定要事先在安全信道中沟通密钥。如果密钥至少和明文一样长，那与其沟通密钥，不如直接把明文告诉对方了。此外，生成的密文也至少和明文一样长，这也是十分浪费通信资源的手段。因此我们可以看到，完善保密确实是一个难以实现的目标。

对于感兴趣的同学，我们可以介绍一个判断一个加密方案是否是完善保密的简单方法，即香农定理：

定理 1.2. *对于明文空间为 $\mathcal{M}$ 的加密方案 $(\mathrm{E}, \mathrm{D}, \mathrm{G})$, 若 $|\mathcal{K}| = |\mathcal{M}| = |\mathcal{C}|$, 则当且仅当下列条件成立时，此方案是完善保密加密：*

1. *由 G 产生的任意密钥 $k \in \mathcal{K}$ 的概率都是 $\dfrac{1}{|\mathcal{K}|}$*
2. *对任意明文 $m \in \mathcal{M}$ 和任意密文 $c \in \mathcal{C}$, 只存在唯一的密钥 $k \in \mathcal{K}$ 使得 $c = \mathrm{E}_k(m)$.*

关于完善保密，我们的讨论就告一段落。最后，介绍一下对加密系统安全性的分类。

对加密系统安全性的分类，现在主流学界习惯上以敌手的算力及时间进行划分：

- 无条件安全

  如果假设攻击者在无限资源的前提下，也无法破译加密算法，就认为相应的密码体制是无条件安全的。

  这里的无限资源，可以包括无限算力和无限时间。

  可以把无条件安全的加密方式理解成完美加密。

- 计算安全

  使用目前最好的方法攻破它所需要的计算远远超出攻击者的计算资源水平，则可以定义这个密码体制是计算安全的。

  比如说，如果要破解某个加密算法需要用当今最好的计算机连续工作一万年，那么我们就可以认为这个密码体制是计算安全的。

- 可证明安全

  如果破译某加密算法的困难性与破解某些困难数学命题的困难性相同（如大整数的因数分解），则可以定义这个密码体制是可证明安全的。

值得指出的是，关于加密系统的安全性，尽管我们使用日常的语言叙述的这些概念，但这些概念实际上都是有严格的数学定义的。鉴于我们目前的数学水平有限，在这里引入这些数学概念是不适合的。因此，我们仅从感性上理解这些概念即可。

第二章　流密码

2.1　数学上的基本概念

从这章开始，我们就不再拘泥于古代的加密拉丁字母的加密方式，而开始转向数字化。我们知道，在现代科技中，密码学最常应用的领域就是计算机领域，而计算机领域则是由二进制的 0, 1 串构成。因此，在接下来的讨论中，我们都将明文空间及密文空间看作由 0 和 1 构成的二进制串组成的集合。

因此，为了接下来能更顺畅地进行关于流密码的讨论，这里先介绍一些数学上关于这方面的基础知识。

2.1.1　GF (2) 上的加法与乘法

由于我们讨论的仅是 0 和 1 及它们的运算, 因此，我们定义一个有限域 $\mathrm{GF}(2) = \{0, 1\}$.

有限域 GF (2) 上的加法被定义成逻辑上的异或，也可以理解成模 2 加法。以下为 GF (2) 上的加法表：

表 2.1: GF (2) 上的加法表

$+$	0	1
0	0	1
1	1	0

如果我们定义一个集合中元素 a 的加法逆元 $-a$ 满足

$$a + (-a) = (-a) + a = 0$$

则 GF (2) 上的加法逆元表为：

表 2.2: GF (2) 上的加法逆元表

a	0	1
$-a$	0	1

而 GF (2) 上的减法则可以定义成与加法逆元的加法，即

$$a - b = a + (-b) \tag{2.1}$$

GF (2) 上的乘法则被定义成了逻辑上的与。以下为 GF (2) 上的乘法表：

表 2.3: GF (2) 上的乘法表

·	0	1
0	0	0
1	0	1

根据上述的定义，我们可以得出以下 GF (2) 上常用的运算规则：

- $$\forall x \in \mathrm{GF}(2), x + x = 0 \tag{2.2}$$

- $$\forall x, y \in \mathrm{GF}(2), x - y = y - x = x + y \tag{2.3}$$

- $$\forall x \in \mathrm{GF}(2), x \cdot x = x \tag{2.4}$$

2.1.2 GF (2) 上的多项式

此外，还有一个我们未曾接触过的知识：GF (2) 上的多项式。为此，不妨先介绍一下多项式理论。

对于表达式

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = \sum_{i=0}^{n} a_i x_i \tag{2.5}$$

我们称其系数为 $a_0, a_1, \ldots, a_n$. 其系数集 S 为 $a_0, a_1, \ldots, a_n$ 的取值范围。当 $a_n \neq 0$ 时，称该多项式为 S 上的 n 次多项式。比如说，如果其系数的取值仅限于 0 和 1, 则称这个多项式为 GF (2) 上的多项式。

值得指出的是，我们研究多项式理论时，多项式在我们眼中仅仅是一个表达式，我们并不需要去对每一个 x 的取值进行多项式的求值。它就相当于一个集合的元素，一个多项式就是一个最小的单位。其加、减、乘、除就应该像我们定义复数集那样重新地进行定义。也就是说，我们通过定义来确定多项式 $f + g = h$, 而不是通过 $\forall x, h(x) = f(x) + g(x)$ 来定义多项式之和。尽管结果确实如此，但这应该理解为自洽的定义，而非推导。

多项式的加法

两个多项式之和的多项式的系数，等于其对应系数之和。即：

若 $m \geq n$, 则

$$\sum_{i=0}^{m} a_i x^i + \sum_{j=0}^{n} b_j x^j = \sum_{k=0}^{n} (a_k + b_k) x^k + \sum_{i=n+1}^{m} a_i x^i \tag{2.6}$$

其中 $a_k + b_k$ 的加法应理解成系数集 S 上的加法。

用更形象的方法来说，我们不妨考虑实数集上的多项式 $f = x^2 + 1$ 与 $g = x^3 + x^2$, 那么其和我们可以类似于小学时的竖式来计算：

$$\begin{array}{rcccccc} f & = & & & x^2 & + & 1 \\ g & = & x^3 & + & x^2 & + & 1 \\ \hline f+g & = & x^3 & + & 2x^2 & + & 2 \end{array}$$

但是，我们这里也需要注意到刚刚说的，$a_k + b_k$ 的加法应理解成系数集 S 上的加法。比如说还是刚刚的两个多项式，但其在 GF (2) 上的加法为：

$$\begin{array}{rcccccc} f & = & & & x^2 & + & 1 \\ g & = & x^3 & + & x^2 & + & 1 \\ \hline f+g & = & x^3 & & & & \end{array}$$

这里 $2x^2$ 与 2 之所以不见了，是因为在 GF (2) 上，$1 + 1 = 0$.

多项式的减法

类似于多项式的加法的定义，两个多项式之差的系数，等于其对应系数之差。在 GF (2) 上的多项式之差的例子：

$$\begin{array}{rcccccc} f & = & & & x^2 & + & 1 \\ g & = & x^3 & + & x^2 & + & 1 \\ \hline f-g & = & x^3 & & & & \end{array}$$

这里是由于 $(0-1)x^3 = (0+1)x^3 = x^3, (1-1)x^2 = 0, 1-1 = 0$.

多项式的乘法

我们可以形式上地用乘法分配律进行计算，并且约定 $x^i x^j = x^{i+j}$. 下面演示一下 GF (2) 上多项式的乘法运算：

$$\begin{aligned} &(x+1)(x+1) \\ =&xx + (1+1)x + 1 \\ =&x^2 + 1 \end{aligned}$$

注意到在 GF (2) 上，$(1+1)x = 0$.

同时，我们记 $f^n = f \cdot f^{n-1}$.

多项式的整除

对于多项式 f 和 g, 如果存在多项式 h, 使得 $f = gh$, 则称 g 整除 f, 记作 $g \mid f$, 同时 $\frac{f}{g} = h$.

注意这仍是在系数集 S 上的。比如说，在 GF (2) 中，我们有

$$\left(x^2+1\right) \mid (x+1)^2$$

此外，若一个多项式 f 是不可约的，则说明不存在两个多项式 g, h, 使得 $gh = f$ 且 g, h 的次数均小于 f 的次数。

GF (2) 上的常用等式

加法交换律

$$f + g = g + f \tag{2.7}$$

加法结合律

$$(f+g)+h=f+(g+h) \tag{2.8}$$

乘法交换律

$$fg=gf \tag{2.9}$$

乘法结合律

$$(fg)\,h=f\,(gh) \tag{2.10}$$

乘法对加法的分配律

$$f\,(g+h)=fg+fh \tag{2.11}$$

等比数列求和公式：对于多项式 f 和 $g\neq 1$

$$f+gf+fg^2+\cdots+fg^{n-1}=\sum_{i=0}^{n-1}fg^i=\frac{f-fg^n}{1-g} \tag{2.12}$$

总结

在 GF (2) 这个有限域上的运算和我们在实数集上的运算很不一样，所以在后文中我们应该着重注意运算是定义在 GF (2) 上的还是定义在实数集上的。同时，我们也该清楚定理叙述的是多项式之间的关系还是值之间的关系。

2.2 流密码的基本概念

之前我们讲到“一次一密”的加密方式是完善保密，同时，“一次一密”的缺点也十分显著：密钥过长。那么，有什么办法能规避这样的缺点呢？事实上，“一次一密”的加密方式之所以是完善保密的，最重要的一点是每次密钥的字符串是随机生成的。通过之前讲的香农定理我们可以知道，每个密钥字符串生成的概率均是相等的。如果我们可以降低一点这种随机性的要求，那也许就能实现短密钥 + 强保密的目标。

为此，我们引入伪随机数序列的概念。由于这个概念的严格定义需要高超的概率论及算法知识，我们只需要感性地理解伪随机数序列为一种，由确定的算法产生的（即相同的初始条件下的输出是相同的），与真随机数序列性质几乎一样的序列。

那么流密码的工作模式可以简单地看作：对于给定长度的一串明文 $M=m_1m_2\cdots m_n$, 我们输入密钥 k, 通过某种算法产生一个同样长度的伪随机序列 $Z=z_1z_2\cdots z_n$ 作为密钥流, 输出结果 $Y=y_1y_2\cdots y_n$ 为明文串与伪随机序列按位异或的结果 $y_i=m_i\oplus z_i$. 根据之前在 GF (2) 上的讨论，解密算法也是将密文串与密钥流进行按位异或，即 $m_i=y_i\oplus z_i$.

根据上述的定义，流密码与一次一密的区别就在于，在与明文串进行按位异或的过程中，一次一密使用的是真随机序列，流密码使用的是伪随机序列。如何能使伪随机序列的表现足够像真随机序列，则是流密码安全性的关键。因此，对流密码的研究，主要就在于产生伪随机序列的算法上。

设 $f(k,\sigma_i)$ 为一个能产生伪随机序列的算法，其中 k 为输入的密钥，σ_i 为当前时刻系统的状态。在每个时刻，$f(k,\sigma_i)$ 输出一个伪随机数，同时系统状态改变为 σ_{i+1}. 常把一个用于加密算法的伪随机序列称为密钥流，产生伪随机序列的算法称为密钥流产生器。

那么流密码的过程可以理解为：如果设明文为二进制串 $X=x_1x_2\cdots x_n$, 密钥为 k. 在初始状态下，输入 x_1 和 k, 密钥流生成器根据当前的状态输出一个伪随机数 z_1, 输出密文 $y_1=x_1\oplus z_1$. 接着输入 x_2, 密钥流生成器根据当前的状态输出一个伪随机数 z_2, 输出密文 $y_2=x_2\oplus z_2$. 以此类推。

2.3 密钥流产生器

之前在讲到密钥流生成器的时候，我们提到了系统当前的状态 σ_i. 这里的状态，是根据当前的输入和输出而变化的。每输入一个数，密钥流生成器每输出一个数，当前系统的状态就发生了变化。之后会介绍一些具体的例子让大家更加理解“系统的状态”的含义。在这里，我们将流密码分为同步流密码和自同步流密码。如果密钥流生成器中的状态与输入的明文有关，则称为自同步流密码，反之则称为同步流密码。我们接下来讨论的都是同步流密码。

为了生成近似真随机的伪随机序列，常见的密钥流产生器由线性部分和非线性部分组合。但是，并非所有的密钥流产生器都是这种工作原理。还有别的能产生伪随机序列密钥流的方法，如 RC4 算法等。

2.3.1 密钥流产生器的线性部分

LFSR

根据上述的讨论，密钥流产生器应有两个功能：输出密钥流与更新系统状态。密钥流产生器的线性部分也有两个功能：输出中间的过渡字符，并且更新系统状态。所谓中间的过渡字符，意思是该部分的输出并非作为最终输出的伪随机数，而是一个中间用于之后计算的数。

常见的处理方法为：系统的状态 $\sigma_i = a_{n,i}a_{n-1,i}\cdots a_{1,i}$ 由 n 个二进制数构成。在启动之前，具有初始状态 $a_{n,0}a_{n-1,0}\cdots a_{1,0}$. 之后，每次的输出 b_i 为

$$b_i = a_{1,i-1} \tag{2.13}$$

而状态更新的方法为

$$a_{j,i} = \begin{cases} a_{j+1,i} & 1 \le j \le n-1 \\ f\left(a_{1,i-1}, a_{2,i-1}, \ldots, a_{n,i-1}\right) & j = n \end{cases} \tag{2.14}$$

其中反馈函数 $f\left(a_{1,i-1}, a_{2,i-1}, \ldots, a_{n,i-1}\right)$ 是一个 GF (2) 上的线性函数。即：

$$\begin{aligned} a_{n,i} &= f\left(a_{1,i-1}, a_{2,i-1}, \ldots, a_{n,i-1}\right) \\ &= c_n a_{1,i-1} + c_{n-1}a_{2,i-1} + \cdots + c_1 a_{n,i-1} \qquad (2.15) \\ &= \sum_{k=1}^{n} c_{n+1-k}a_{k,i-1} \qquad (2.16) \end{aligned}$$

其中 $c_k \in \text{GF}(2)$, 即 c_k 的取值为 0 或 1. 这些数字都是固定的，由密钥流产生器本身决定。而这里的加法、乘法运算均为 GF (2) 上的运算。

我们通过一个例子来熟悉：

设初始状态为 110, 反馈函数 $f(a_{3,i}, a_{2,i}, a_{1,i}) = a_{3,i} + a_{1,i}$. 则我们可以通过下表来了解这个线性部分的输出：

i	0	1	2	3	4	5	6	7	$\cdots$
$f(a_3, a_2, a_1)$	1	0	1	0	0	1	1	1	$\cdots$
a_3	1	1	0	1	0	0	1	1	$\cdots$
a_2	1	1	1	0	1	0	0	1	$\cdots$
a_1	0	1	1	1	0	1	0	0	$\cdots$
b		0	1	1	1	0	1	0	$\cdots$

其输出序列就为 011101　011101$\cdots$.

我们由上面的表可以发现，从每一列来看，随着 i 的递增，上一行的数会传给下一行的数。这似乎是某种线性寄存器的工作形式。因此，我们称密钥流产生器的线性部分为一个线性反馈移位寄存器 (Linear Feedback Shift Register, LFSR). 若其状态具有 n 个二进制数字，且 $c_n \neq 0$, 则称其为 n 级 LFSR.

LFSR 的输出序列

如果设 $B = b_1 b_2 \cdots b_i \cdots$ 表示 LFSR 的输出序列，则由上述的讨论也可以知道

$$(b_{i+1} b_{i+2} \cdots b_{i+n}) = (a_{1,i-1} a_{2,i-1} \cdots a_{n,i-1}) \tag{2.17}$$

我们还可以认为，每一个 b_i 都可以代表 LFSR 的一个状态。那么，由于 LFSR 的一个状态中有 n 个二进制数，故一个 LFSR 至多有 2^n 种状态。如果称状态 $\sigma = 00\cdots0$ 为零状态，下面我们证明：零状态的下一个状态是零状态，任何非零状态的下一个状态均不是零状态。

前半句话显然，设 $\forall 1 \leq j \leq n, a_{j,i} = 0$. 对于 $j \leq n-1, a_{j,i+1} = a_{j+1,i} = 0$. 而

$$a_{n,i} = c_n a_{1,i-1} + c_{n-1} a_{2,i-1} + \cdots + c_1 a_{n,i-1} = 0$$

对于后半句话，若某个非零状态 $\sigma_i = a_{n,i} a_{n-1,i} \cdots a_{1,i}$ 的下一个状态为零状态，则由定义可知，$\forall 2 \leq j \leq n, a_{j,i} = a_{j-1,i+1} = 0, a_{n,i+1} \neq 0$. 且

$$0 = a_{n,i+1} = c_n a_{1,i} + c_{n-1} a_{2,i} + \cdots + c_1 a_{n,i} = c_n a_{1,i}$$

又由于 $c_n \neq 0$, 故 $a_{1,i} = 0$ 矛盾。

由上述讨论可知，一个 LFSR 中的状态，至多 $2^n - 1$ 个之后即达成循环。也就是说，一个 LFSR 产生的序列，周期至多为 $2^n - 1$.

定义 2.1. *我们称由 LFSR 产生的周期为 $2^n - 1$ 的序列为一个 m 序列。*

m 序列相关定理

上述的讨论中，我们提到，反馈函数

$$a_{n,i} = c_n a_{1,i-1} + c_{n-1} a_{2,i-1} + \cdots + c_1 a_{n,i-1}$$

中的 $c_1, c_2, \ldots, c_n$ 由 LFSR 本身决定。因此：

定义 2.2. *我们称 GF (2) 上的多项式*

$$p(x) = 1 + c_1 x + \cdots + c_{n-1} x^{n-1} + c_n x^n \tag{2.18}$$

为 LFSR 的特征多项式。

对于 LFSR 生成的一个序列 $a_1 a_2 \cdots a_n \cdots$, 称幂级数

$$A(x) = \sum_{i=1}^{\infty} a_i x^{i-1} \tag{2.19}$$

为该序列的生成函数。

对于使用给定的 LFSR, 由于初始状态不同而产生的所有 $2^n - 1$ 个非零序列构成的集合记作 $G(p(x))$.

下面叙述一个在证明中很有用的定理：

定理 2.1. *设 LFSR 的特征多项式 $p(x)=1+c_1x+\cdots+c_{n-1}x^{n-1}+c_nx^n$, $A(x)$ 为 $G(p(x))$ 中任意序列 $\{a_n\}$ 的生成函数，则对于 GF(2) 上的多项式 $p(x)$ 和 $A(x)$, 满足*

$$A(x)=\frac{\phi(x)}{p(x)} \tag{2.20}$$

其中

$$\phi(x)=\sum_{i=1}^{n}\left(c_{n-i}x^{n-i}\sum_{j=1}^{i}a_jx^{j-1}\right) \tag{2.21}$$

我们可以对 $\phi(x)$ 进行展开：

$$\begin{aligned}\phi(x)&=\sum_{i=1}^{n}\left(c_{n-i}x^{n-i}\sum_{j=1}^{i}a_jx^{j-1}\right)\\&=\sum_{i=1}^{n}\left(\sum_{j=1}^{i}c_{n-i}a_jx^{n-i+j-1}\right)\\&\xlongequal{k=n-i+j-1}\sum_{i=1}^{n}\sum_{k=n-i}^{n-1}c_{n-i}a_{k-n+i+1}x^k\end{aligned}$$

因此，我们可以发现，$\phi(x)$ 的次数不超过 $n-1$.

定义 2.3. *对于 GF(2) 上的多项式 $p(x)$, 若 $p(x)\mid x^p-1$, 则称最小的 p 为 $p(x)$ 的阶。*

一个与之相关的定理是：

定理 2.2. *设 $p(x)$ 是 n 次不可约多项式，其阶为 m. 则 $\forall\{a_n\}\in G(p(x))$, $\{a_n\}$ 的周期为 m.*

定义 2.4. *若 n 次不可约多项式 $p(x)$ 的阶为 2^n-1, 则称 $p(x)$ 是本原多项式。*

下面叙述的是最关键的一个定理：

定理 2.3. *设 $\{a_n\}\in G(p(x))$, 则 $\{a_n\}$ 为 m 序列的充要条件是 $p(x)$ 是本原多项式。*

LFSR 是伪随机数产生器

为了叙述本节的定理，我们引入两个概念：

定义 2.5. *对于序列 $\{a_n\}$, 长度最大为 n 的连续的 0 或者 1 称为一个长度为 n 的 0 游程或 1 游程。*

对于 GF(2) 上周期为 2 的序列 $\{a_n\}$, 称其异相关函数为

$$R(\tau)=\frac{1}{T}\sum_{k=1}^{T}(-1)^{a_k}(-1)^{a_{k+\tau}},0<\tau\le T-1 \tag{2.22}$$

之前我们提到的伪随机数序列，我们在这里给出一种定义方法：

定义 2.6. *对于周期序列 $\{a_n\}$, 其为伪随机序列的条件为*

1. *在序列的一个周期内，0 与 1 的个数至多相差 1.*
2. *在序列的一个周期内，长为 i 的游程占游程总数的 $\frac{1}{2^i}$, 且其中 0 游程与 1 游程个数相等。*
3. *该序列的异相关函数是个常数。*

那么我们可以证明，一个 n 长 m 序列是这种意义下的伪随机序列。

LFSR 密码的破译

在本节介绍的破译方式中，值得强调的是，敌手是知道 LFSR 为 n 级，且敌手获得了一串长度为 $2n$ 的明密文对。

假设敌手获得的明密文对为 $x_1x_2\cdots x_{2n}$ 和 $y_1y_2\cdots y_{2n}$, 其需要破译的是 LFSR 的特征多项式的系数 $c_1, c_2, \ldots, c_n$. 那么由于在 GF (2) 上

$$y_i = x_i + z_i$$

其中 z_i 为产生的密钥流的第 i 位。

故在 GF (2) 上

$$x_i + y_i = x_i + x_i + z_i = z_i$$

从而敌手就获得了一段长度为 $2n$ 的密钥流 $z_1z_2\cdots z_{2n}$.

如果记

$$S_i = (z_{i+1}, z_{i+2}, \ldots, z_{i+n})^{\mathrm{T}}, i = 0, 1, \ldots, n-1 \tag{2.23}$$

且

$$X = (S_0, S_1, \ldots, S_{n-1}) \tag{2.24}$$

故根据表达式

$$z_{n+i} = c_n z_i + c_{n-1} z_{i+1} + \cdots + c_1 z_{i+n-1} \tag{2.25}$$

可得：

$$(z_{n+1}, z_{n+2}, \ldots, z_{2n}) = (c_n, c_{n-1}, \cdots, c_1)\, X \tag{2.26}$$

而我们可以证明 X 是可逆的。故

$$(c_n, c_{n-1}, \cdots, c_1) = (z_{n+1}, z_{n+2}, \ldots, z_{2n})\, X^{-1} \tag{2.27}$$

2.3.2 密钥流产生器的非线性部分

由上述的讨论我们可以发现，线性的密钥流产生器产生的密钥流一定是周期的。而周期较短的密钥流则相当于一种多表代换密码，相对容易破解。因此，只有加上非线性的部分来处理，才能提高安全性。

密钥流产生器的非线性部分的主要工作是接受一个或多个 LFSR 的输入，以非线性的方式，输出密钥流。

我们综合判定一个输出密钥流的优劣，可以从两个方面：周期和线性复杂度。周期越长，线性复杂度越高的密钥流越安全。周期我们可以直接衡量，而线性复杂度我们该如何衡量呢? 我们称一个序列的线性复杂度为生成该序列的最短 LFSR 的级数。即若该序列周期 T 满足 $2^{n-1} - 1 < T \leq 2^n - 1$, 则该序列的线性复杂度为 n.

Geffe 序列生成器

Geffe 序列生成器接受三个 LFSR 的输入，其输入分别为序列 $\left\{a_n^{(1)}\right\}, \left\{a_n^{(2)}\right\}, \left\{a_n^{(3)}\right\}$.

其输出序列 $\{b_k\}$ 可以表示为

$a_k^{(2)}$	b_k
0	$a_k^{(3)}$
1	$a_k^{(1)}$

若 $\left\{a_n^{(1)}\right\}, \left\{a_n^{(2)}\right\}$ 和 $\left\{a_n^{(3)}\right\}$ 的周期分别为 $2^{n_1}-1, 2^{n_2}-1$ 和 $2^{n_3}-1$ 且 n_1, n_2, n_3 两两互素，则 $\{b_k\}$ 的周期为 $(2^{n_1}-1)(2^{n_2}-1)(2^{n_3}-1)$, 线性复杂度为 $(n_1+n_3)n_2+n_3$

JK 触发器

JK 触发器接受两个 LFSR 的输入，其输入分别为序列 $\left\{a_n^{(1)}\right\}, \left\{a_n^{(2)}\right\}$.

其输出序列 $\{b_k\}$ 可以表示为

$a_k^{(1)}$	$a_k^{(2)}$	b_k
0	0	b_{k-1}
0	1	0
1	0	0
1	1	$\overline{b_{k-1}}$

若 $\left\{a_n^{(1)}\right\}$ 和 $\left\{a_n^{(2)}\right\}$ 的周期分别为 2^n-1 和 2^m-1 且 m, n 互素，$a_0^{(1)}+a_0^{(2)}=1$, 则 $\{b_k\}$ 的周期为 $(2^n-1)(2^m-1)$

钟控序列生成器

钟控序列接受两个 LFSR 的输入，其输入分别为序列 $\left\{a_n^{(1)}\right\}, \left\{a_n^{(2)}\right\}$. 前一个序列控制后一个序列的时钟周期 n_k.

其输出序列 $\{b_k\}$ 可以表示为

$a_k^{(1)}$	n_k	b_k
0	n_{k-1}	$a_{n_k}^{(2)}$
1	$n_{k-1}+1$	$a_{n_k}^{(2)}$

若 $\left\{a_n^{(1)}\right\}$ 和 $\left\{a_n^{(2)}\right\}$ 的周期分别为 p_1 和 p_2, 且记 $w=\sum_{i=0}^{p_1-1} a_i^{(1)}$, 则 $\{b_k\}$ 的周期为 $\dfrac{p_1p_2}{\gcd(w,p_2)}$. 若 $p_1=2^m-1, p_2=2^n-1$, 则其线性复杂度为 $n(2^m-1)$.

第三章　分组密码

3.1　设计密码系统的方法

我们之前提到如凯撒密码、多表代换密码等经典密码的时候，讲到了破解这些密码的一种方法，就是利用英文中每个字母出现的频率不同。一次一密的方法可以抵御这种破解，原因是对于明文中的每个字符，其移位都是随机的，因此在密文中完全没有明文中字母出现的频率的信息。但是，一次一密的成本又太高，我们有没有什么办法，能尽可能地掩盖在密文中出现的字母频率的信息，来抵御这种频率攻击呢？香农给出了一种解决方法。

3.1.1　扩散与混淆

所谓扩散，指的是如果我们改变明文中的一个字符，加密得到的密文中的多个字符也会得到改变；如果我们改变密文中的一个字符，解密得到的明文中的多个字符也同样会得到改变。因此，明文中一个字符的信息被“扩散”到密文中的多个字符。因此，实现扩散的方法，就可以是用明文中的多个字符去生成密文中的一个字符。

扩散是将明文和密文之间的关系变得复杂使我们很难获得密钥，而混淆则是将密钥和密文之间的关系变得复杂。例如，密钥中的一个字符的改变会导致密文中多个字符的改变。因此，即使攻击者通过密文，知道了一些关于明文的统计信息，也很难获得密钥。

我们可以发现，凯撒密码的加密方式并没有实现扩散和混淆，因此，它可以被频率攻击轻易破解。

3.1.2　置换与代换

为了实现扩散，常用的方法是置换。也就是说，把输入的一部分与另一部分进行交换，然后输出。置换中最常用的结构为 P 盒。其接受 m 位二进制输入，m 位二进制输出。其输出是把输入的比特按一定规则打乱顺序后输出。

为了实现混淆，常用的方法是代换。而代换中最常用的结构为 S 盒。其接受 m 位二进制输入，n 位二进制输出。因此，其输入共有 2^m 中，输出共有 2^n 种。我们可以把 S 盒理解成一种查找表。对于 2^m 个输入中的每一种输入，我们可以在这个表中查找到一个 n 位的输出，而且 S 盒需要保证不同的输入对应的输出也不同。从编程上，我们也可以将此理解成一个长度为 2^m 的数组，它的每个元素为 n 比特的数字。因此，其所占的空间为 $n2^m$.

3.2　分组密码的定义

事实上，在非对称密码发展之前，大多数著名的密码体系，其核心都是扩散与混淆。但是，在我们上述谈到扩散与混淆的时候，有一个值得注意的地方：实现扩散与混淆的器件，即

P 盒与 S 盒，其接受的都是固定长度的输入。这与我们之前谈到的流密码不同，流密码的输入可以是任意长度的。因此，为了更好地实现扩散与混淆，我们引入了分组密码。

分组密码就是一个较好地实现扩散与混淆的密码系统。它的核心思想是分组。首先，将明文分成若干个等长的组，然后对每个组利用密钥依次进行加密，生成等长的密文组。

对于分组密码，我们要研究的有：每组内如何根据输入和密钥进行加密，以及各组之间的关系。最简单的方法是各组的输入是之前分好的明文的各个分组，密钥是相同的密钥。但是，也可以使用相对复杂的方法，使加密变得更加复杂（参见“分组密码的运行模式”一节）。

由于密文是明文按组生成的，因此，分组密码具有扩散性；而通过采用特别设定的 S 盒，也可以实现混淆性。

这里我们要注意的是，密码学中研究的分组密码，并不仅仅是将明文分组的加密方式。我们之前提到的同步密码，实际上也可以看作是将明文分组进行加密，每个组的长度为其伪随机密钥流的周期。但是，同步流密码依然是逐比特加密，因此，失去了扩散性。所以，只有这个组的每个比特都参与到了密文的生成中的加密方法，才是我们这一章研究的重点。

此外，在讨论为分组密码的运行模式，即各组之间的联系之后，我们接下来讨论的 DES, IDEA, AES 等都是每组内的加密方法。由于分好了组，所以这些密码体制有一个特点，即明文或者密文是固定长度的。

3.3 分组密码的设计方法

之前讲到，对于分组密码，我们要研究的有：每组内如何根据输入和密钥进行加密，以及各组的输入和密钥是什么。本节讨论的是每组内的加密方法。也就是说，在本节内，我们提到的“输入”、“明文”等，都是指被分好组以后的每个组的输入、明文。

常见的分组密码有两种核心设计方法：费斯妥密码与 SP 网络。它们都有一个相同的特点：需要将同一个步骤重复多轮。而在加密过程中，任何核心步骤都是同时需要明文和密钥的信息的。因此，为了保证可靠性，每一轮步骤的输入和密钥都是不一样的。步骤的输入可以根据上一轮的输出来改变，而密钥怎么办呢？这时，就需要**子密钥**。所谓子密钥，就是根据密钥，生成的不同序列。比如说，某种方法需要 16 轮步骤，那么，我们就应设计一种算法，使密钥能生成 16 个子密钥。

3.3.1 费斯妥密码

费斯妥密码每一轮的步骤，接受上一轮的输出 O_{i-1} 为本轮的输入 I_i, 同时接受子密钥 K_i, 输出本轮输出 O_i.

首先，将输入的二进制串 O_{i-1} 分成左右两个相等长度的子串 L_{i-1} 和 R_{i-1}, 然后计算

$$L_i = R_{i-1} \tag{3.1}$$

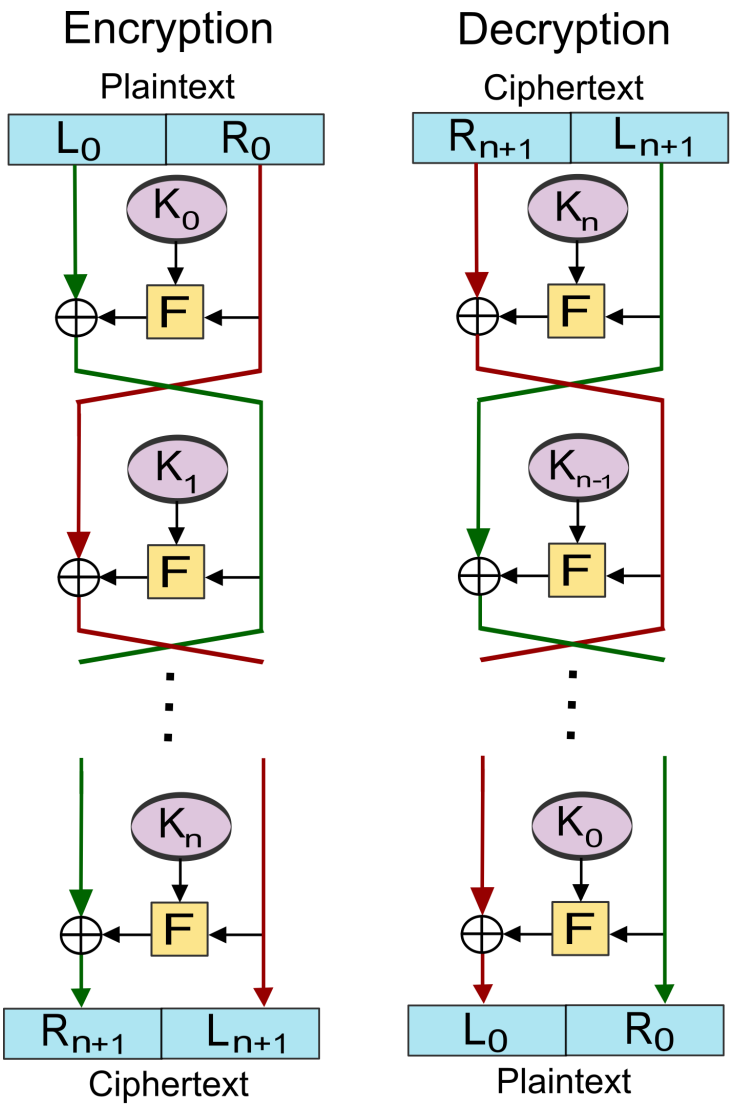
$$R_i = L_{i-1} \oplus F\left(R_{i-1}, K_i\right) \tag{3.2}$$

最后输出是把 L_i 和 R_i 拼起来成 O_i.

其中 $F\left(R_{i-1}, K_i\right)$ 称为轮函数，是不同具体的加密算法的核心。

而由于等式 $a \oplus b \oplus b = a$, 因此，其解密过程与加密过程完全相同，只不过需要把子密钥倒着顺序使用。

可以用下图形象地理解费斯妥密码（图源 wiki）：

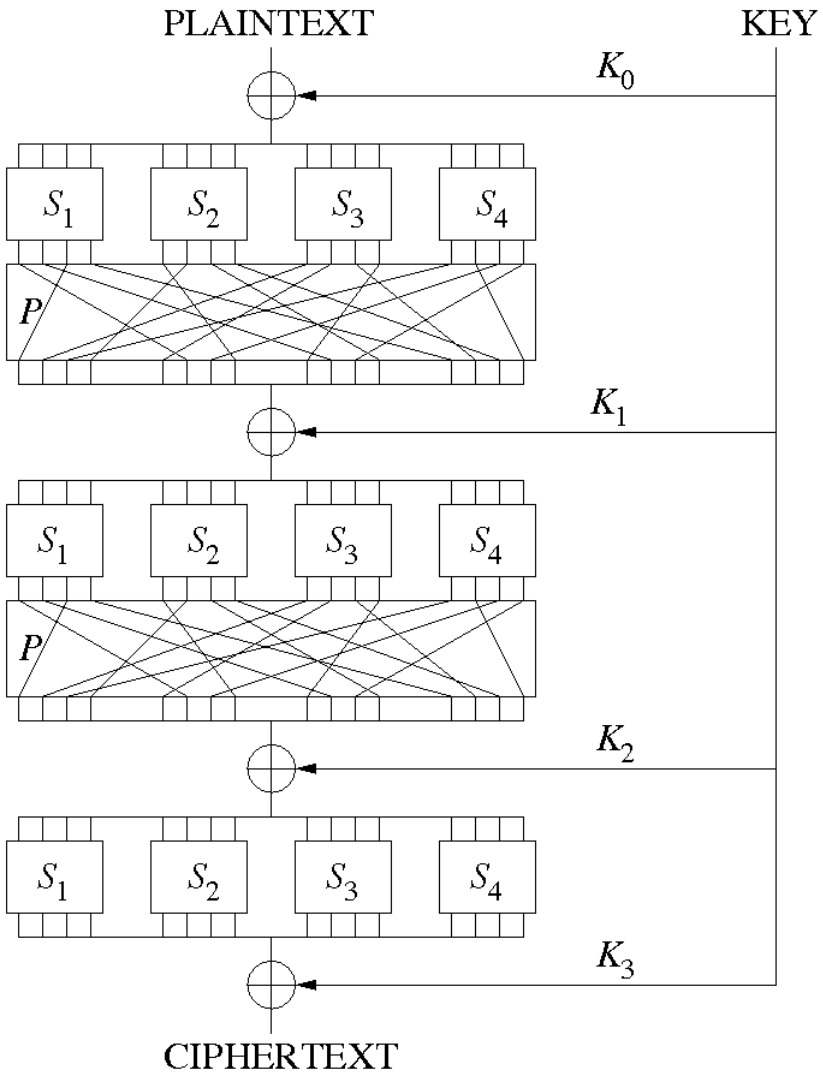


值得注意到的一点是，在最后一次输出的时候，不再进行交换，也就是说，输出的并不是 (L_{n+1}, R_{n+1}), 而是 (R_{n+1}, L_{n+1}).

3.3.2 SP 网络

SP 网络每轮接受到输入之后，首先，将输入与该轮的子密钥进行异或，然后是对输入再次分组（也就是对分过组的明文的每组内容再次进行分组），接着将每个组通过不同的 S 盒进行代换，代换后的结果再拼成一个新的串，经过一个 P 盒的置换进行输出。

可以通过下图形象地理解 SP 网络（图源 wiki）：



3.4 分组密码的运行模式

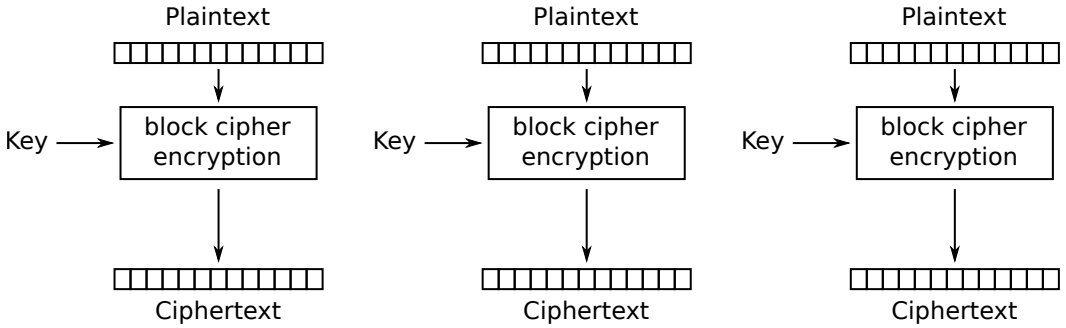
上一节讨论了在对明文分好组后，每组内常见的加密方式。本节讨论的，则是组与组之间输入和输出的关系。在这里，假设将明文 M 分成了 $m_1, m_2, \ldots, m_n$ 共 n 个等长的组，在每组内，加密算法为 $\mathrm{E}_k(m)$, 解密算法为 $\mathrm{D}_k(c)$.

3.4.1 电码本模式 (ECB)

对于每组，其加密的输入为明文分好的组 m_i, 密钥为同一个密钥串 k. 也就是说，

$$c_i = \mathrm{E}_k(m_i) \tag{3.3}$$

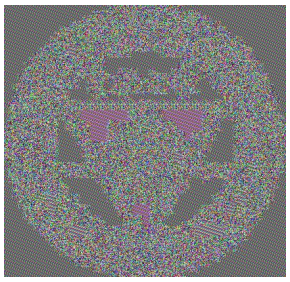
其加密过程如图所示（图源 wiki）:



Electronic Codebook (ECB) mode encryption

ECB 模式极不安全。比如说，我想用 ECB 模式的 AES 密码体系加密东南大学的校徽：

结果为



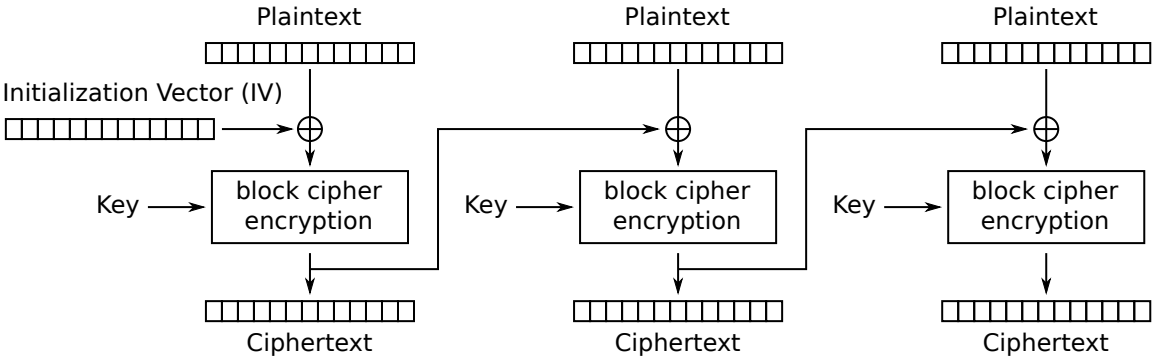
3.4.2 密码分组链接模式 (CBC)

对于每组，其加密的输入为当前明文组与前一密文组的异或。也就是说，

$$c_i = \mathrm{E}_k\left(m_i \oplus c_{i-1}\right) = \mathrm{E}_k\left(m_i \oplus \mathrm{E}_k\left(m_{i-1}\right)\right) \tag{3.4}$$

对于第一组明文组，由于没有 m_0, 因此，需要一个初始的二进制串，常记作 IV, 来与 m_1 异或。

其加密过程如图所示（图源 wiki）：



Cipher Block Chaining (CBC) mode encryption

3.4.3 密码反馈模式 (CFB)

CFB 模式需要一个移位寄存器。同时, CFB 模式也提供了一个可选的参数 j, 通常取 $j = 8$. 其过程如下：

对于每组，首先需要进一步分组，使每组的长度为 j 个比特。然后对于每个新分好的组，先将移位寄存器左移 j 个比特，然后将上一组输出的 j 个比特输入到移位寄存器的右边。然后将移位寄存器内存储的二进制串用密钥 k 进行加密，其输出取前 j 个比特与本组的 j 个比特的输入进行异或输出。

与 CBC 类似，处理第一组时移位寄存器内的值也需要一组初始的二进制串 IV.

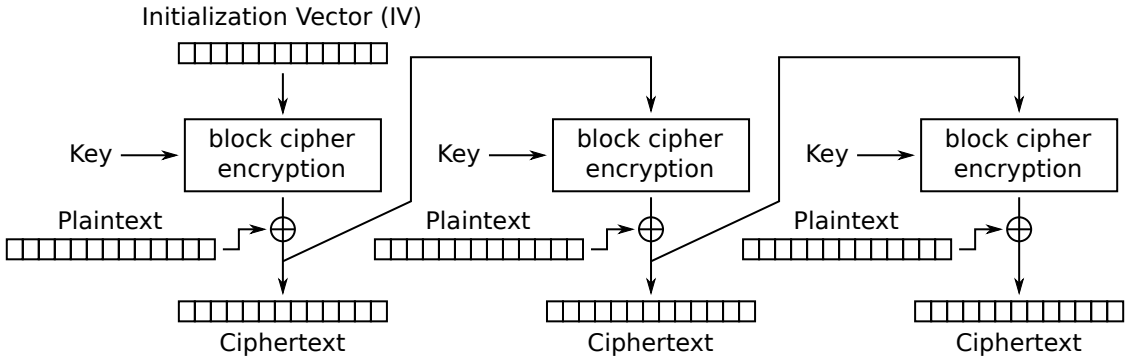
因此，如果记 $H_j(m)$ 代表取 m 的左边 j 位，P_i 表示将每组进一步分为的 j 比特的分组，则 CFB 模式的加密过程可用公式描述为：

$$c_i = H_j\left(\mathrm{E}_k\left(S_{i-1}\right)\right) \oplus P_i \tag{3.5}$$

其中 S_i 为移位寄存器的值。移位寄存器的工作方式为

$$S_i = \left(\left(S_{i-1} << x\right) + c_i\right) \bmod 64 \tag{3.6}$$

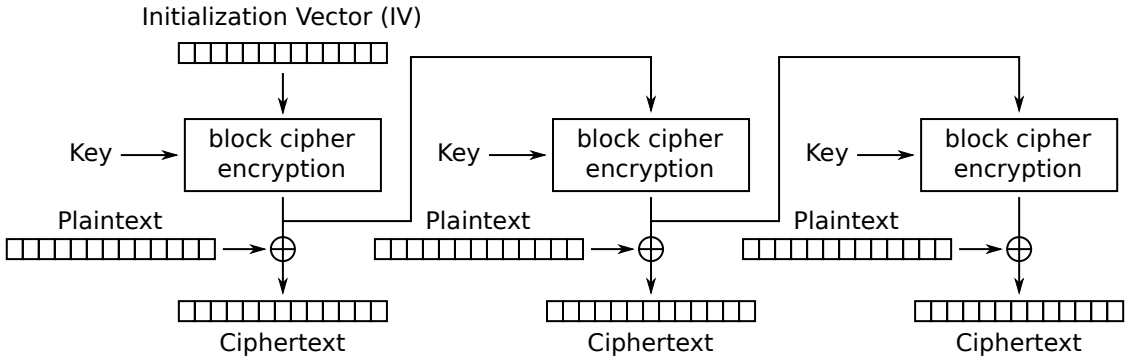
如果忽略移位寄存器，其大致的工作原理可由下图表示（图源 wiki）：



Cipher Feedback (CFB) mode encryption

3.4.4 输出反馈模式 (OFB)

OFB 模式与 CFB 模式极其类似，区别仅在于每组向移位寄存器内的输入为上一组内与明文异或之前的输出。如下图所示（图源 wiki）：



Output Feedback (OFB) mode encryption

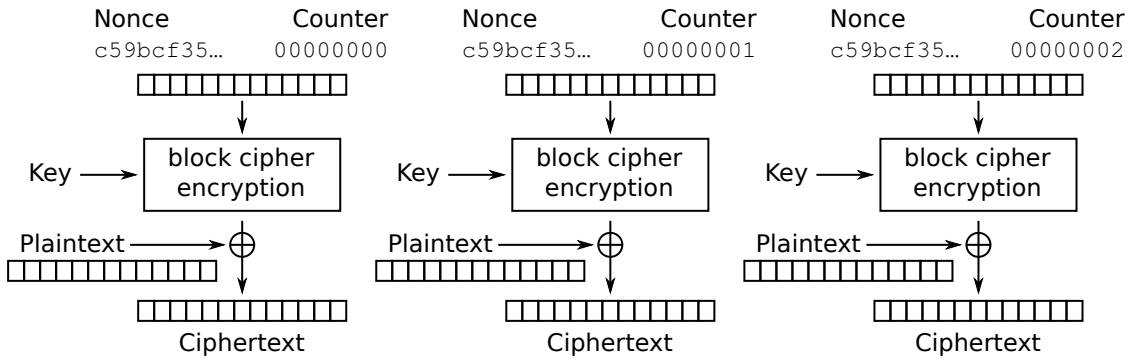
从图中可以看到，每组之间传递的数据与明文无关。因此，在 OFB 中，明文出错只会影响该组的密文，之后的密文都不会被影响。

我们如果再仔细看一下 OFB 的模式图，会发现，事实上，OFB 模式把分组密码变成了流密码。真正经过分组密码体系加密的是密钥，而明文则是通过与密钥加密出来的结果异或产生的密文。因此，如果要使 OFB 模式的安全性高，则要求由分组密码加密出来的密钥为伪随机序列。

3.4.5 计数器模式 (CTR)

在借鉴了 OFB 中本组明文不参与下一组加密的经验之后，引入了 CTR 模式。在 CTR 模式中，存在一个计数器函数 f. 其接受一个初始值，并在每组加密完成后，进行计数，累加到初始值之上。然后每组加密的时候，只需要将该函数的返回值输入分组加密算法中，输出值与当前明文组异或产生密文输出。

其过程如图所示（图源 wiki）：



Counter (CTR) mode encryption

根据上图，我们可以发现 CTR 的独一无二的好处：可以并行加密。其每组加密不需要上组的任何信息，只需要该组对应的计数器值即可。

3.5 DES

接下来，我们讨论的是，在每组内的分组加密算法。

DES 是最著名的分组密码之一。我们可以先大致地讨论其算法，然后再讨论其一些性质。

3.5.1 算法与代码

DES 采用了费斯妥密码的结构，并对它进行了一定的改进。之前我们提到的费斯妥密码，可以改进的地方有其最开始的输入、子密钥、轮函数以及最后的输出。因此，我们分别就上述几个方面来分块解释 DES 的算法。

主体：费斯妥密码

首先，我们介绍 DES 的主体——费斯妥密码。正如之前说的，费斯妥密码为一个步骤的多轮操作。其核心公式为

$$L_i = R_{i-1} \tag{3.7}$$

$$R_i = L_{i-1} \oplus F\left(R_{i-1}, K_i\right) \tag{3.8}$$

其中，L_{i-1}, R_{i-1} 为上一轮输出的左右两半，L_i, R_i 为本轮输出的左右两半，$F\left(R_{i-1}, K_i\right)$ 为轮函数，K_i 为本轮的子密钥。

此外，从数学上可以证明，费斯妥密码的解密过程的核心公式也是上述公式，只不过使用的密钥的顺序与加密正好相反。

DES 密码一共进行 16 轮这样的步骤，并且其输入为 64 位二进制串，子密钥为 48 位二进制串，输出为 64 位二进制串。

其 C 程序代码如下：

```
bitset<64> round(bitset<64> input, bitset<48> ki)
{
    bitset<64> output;
    bitset<32> previousLeftPart;
    bitset<32> leftPart;
    bitset<32> rightPart;

    for (int i = 0; i < 32; i++)
        previousLeftPart[31 - i] = input[63 - i];

    for (int i = 0; i < 32; i++)
        leftPart[31 - i] = input[31 - i];

    rightPart = previousLeftPart ^ F(leftPart, ki);

    for (int i = 0; i < 32; i++)
        output[63 - i] = leftPart[31 - i];

    for (int i = 32; i < 64; i++)
        output[63 - i] = rightPart[63 - i];
```

```
    return output;
}
```

费斯妥密码的轮函数

在费斯妥密码的轮函数这里，实际上是采用了 SP 网络的思想，也就是将轮函数的输入经过 S 盒的代换来混淆和 P 盒的置换来扩散。

轮函数接受 32 位的输入和 48 位的子密钥。首先，将 32 位的输入扩充成 48 位的二进制串（可以理解成通过了一个 32 位到 48 位的 S 盒，称为选择扩展运算 E），然后将其逐比特与 48 位的子密钥异或，输出的 48 位二进制串作为一个 48 位到 32 位的 S 盒的输入。接着，将 S 盒输出的 32 位二进制串经过一个 P 盒（称为置换运算 P）输出。

这里的 48 位到 32 位的 S 盒实际上是由 8 个 6 位到 4 位的 S 盒组成。其将输入的 48 位分组输入，然后再分组输出。

因此，轮函数所做的事情有：将 32 位的输入扩展成 48 位，进行异或，输入 S 盒，输入 P 盒。其 C 程序代码如下：

```
int E[] = {32,  1,  2,  3,  4,  5,
            4,  5,  6,  7,  8,  9,
            8,  9, 10, 11, 12, 13,
           12, 13, 14, 15, 16, 17,
           16, 17, 18, 19, 20, 21,
           20, 21, 22, 23, 24, 25,
           24, 25, 26, 27, 28, 29,
           28, 29, 30, 31, 32,  1};

int S_BOX[8][4][16] = {
    {
        {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7},
        {0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8},
        {4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0},
        {15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}
    },
    {
        {15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10},
        {3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5},
        {0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15},
        {13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9}
    },
    {
        {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8},
        {13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1},
        {13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7},
        {1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12}
```

```
    },
    {
        {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15},
        {13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9},
        {10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4},
        {3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14}
    },
    {
        {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9},
        {14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6},
        {4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14},
        {11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3}
    },
    {
        {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11},
        {10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8},
        {9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6},
        {4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13}
    },
    {
        {4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1},
        {13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6},
        {1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2},
        {6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12}
    },
    {
        {13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7},
        {1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2},
        {7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8},
        {2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11}
    }
};

int P[] = {16,  7, 20, 21,
           29, 12, 28, 17,
            1, 15, 23, 26,
            5, 18, 31, 10,
            2,  8, 24, 14,
           32, 27,  3,  9,
           19, 13, 30,  6,
           22, 11,  4, 25};
```

```
bitset<4> S_boxi(bitset<6> input, int i)
{
    int row = 2 * input[5] + input[0];
    int column = 8 * input[4] + 4 * input[3] + 2 * input[2]
                + input[1];
    int outputint = S_BOX[i][row][column];
    bitset<4> output(outputint);

    return output;
}

bitset<32> S_box(bitset<48> input)
{
    bitset<32> output;
    bitset<6> SiInput[8];

    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 6; j++)
            SiInput[i][5 - j] = input[47 - (j + i * 6)];

        bitset<4> SiOutput = S_boxi(SiInput[i], i);

        for (int j = 0; j < 4; j++)
            output[31 - (j + i * 4)] = SiOutput[3 - j];
    }

    return output;
}

bitset<32> F(bitset<32> rightPart, bitset<48> ki)
{
    bitset<32> output;
    bitset<48> expandedInput;

    for (int i = 0; i < 48; i++)
        expandedInput[47 - i] = rightPart[32 - E[i]];

    bitset<48> S_boxInput = expandedInput ^ ki;
    bitset<32> S_boxOutput = S_box(S_boxInput);

    for (int i = 0; i < 32; i++)
```

```
        output[31 - i] = S_boxOutput[32 - P[i]];

    return output;
}
```

费斯妥密码的最初输入

DES 加密算法接受 64 位明文输入，DES 解密算法接受 64 位密文输入。为了更好地实现扩散性，首先，需要将输入的 64 位二进制串经过一个 P 盒。在 DES 算法中，这个 P 盒被称作初始置换 IP. 随后，将经过置换后的 64 位二进制串作为费斯妥密码的输入。

其 C 程序代码如下：

```
int IP[] = {58, 50, 42, 34, 26, 18, 10, 2,
            60, 52, 44, 36, 28, 20, 12, 4,
            62, 54, 46, 38, 30, 22, 14, 6,
            64, 56, 48, 40, 32, 24, 16, 8,
            57, 49, 41, 33, 25, 17, 9,  1,
            59, 51, 43, 35, 27, 19, 11, 3,
            61, 53, 45, 37, 29, 21, 13, 5,
            63, 55, 47, 39, 31, 23, 15, 7};

bitset<64> getInitialPermutation(bitset<64> input)
{
    bitset<64> initialPermutation;

    for (int i = 0; i < 64; i++)
        initialPermutation[63 - i] = input[64 - IP[i]];

    return initialPermutation;
}
```

费斯妥密码的最终输出

之前我们再三强调，费斯妥密码的最后一轮输出后，还要将左右两边互换，也就是最终输出为 (R_{16}, L_{16}) 而非 (L_{16}, R_{16}). 此外，为了使 DES 的加密和解密算法能尽可能复用，我们将输出再经过一个 P 盒才形成最终的 DES 的输出。其中，输出时经过的 P 盒要是输入时 P 盒的逆，被称为逆初始置换 IP^{-1}. 这样的话，我们假设明文为 m, 密文为 c, 中间的费斯妥密码部分 (包括最后的左右交换)，加密为 $f(x)$, 解密为 $f^{-1}(x)$. 那么，DES 加密的过程为

$$c = \mathrm{IP}^{-1}\left(f\left(\mathrm{IP}\left(m\right)\right)\right)$$

而只需要把中间的 f 换成 f^{-1}:

$$\begin{aligned}
&\mathrm{IP}^{-1}\left(f^{-1}\left(\mathrm{IP}\left(c\right)\right)\right)\\
=&\mathrm{IP}^{-1}\left(f^{-1}\left(\mathrm{IP}\left(\mathrm{IP}^{-1}\left(f\left(\mathrm{IP}\left(m\right)\right)\right)\right)\right)\right)\\
=&\mathrm{IP}^{-1}\left(f^{-1}\left(f\left(\mathrm{IP}\left(m\right)\right)\right)\right)\\
=&\mathrm{IP}^{-1}\left(\mathrm{IP}\left(m\right)\right)\\
=&m
\end{aligned}$$

即可实现解密。

因此，DES 输出包括交换费斯妥密码输出的左右位置，以及通过逆初始置换 IP^{-1}. 其 C 程序代码如下：

```
int IP_1[] = {40, 8, 48, 16, 56, 24, 64, 32,
              39, 7, 47, 15, 55, 23, 63, 31,
              38, 6, 46, 14, 54, 22, 62, 30,
              37, 5, 45, 13, 53, 21, 61, 29,
              36, 4, 44, 12, 52, 20, 60, 28,
              35, 3, 43, 11, 51, 19, 59, 27,
              34, 2, 42, 10, 50, 18, 58, 26,
              33, 1, 41,  9, 49, 17, 57, 25};

bitset<64> exchangeLeftAndRight(bitset<64> input)
{
    bitset<64> output;

    for (int i = 0; i < 32; i++)
        output[63 - i] = input[31 - i];

    for (int i = 32; i < 64; i++)
        output[63 - i] = input[95 - i];

    return output;
}

bitset<64> getInversePermutation(bitset<64> input)
{
    bitset<64> output;

    for (int i = 0; i < 64; i++)
        output[63 - i] = input[64 - IP_1[i]];

    return output;
}
```

密钥的处理

对于输入的密钥，我们需要让其生成 16 个子密钥。类似于费斯妥密码，这里的 16 次生成也是同样的步骤循环 16 次。但首先，我们需要处理的是 DES 算法输入的 64 位密钥。

DES 算法输入的 64 位密钥中，通常包含 8 位奇偶校验位。首先，我们将奇偶校验位去除，得到 56 位的真正的密钥。然后，再将其通过一个 P 盒（称为置换选择 1:PC_1），作为接下来生成子密钥的算法的输入。

其 C 程序代码为：

```
int PC_1[] = {57, 49, 41, 33, 25, 17, 9,
               1, 58, 50, 42, 34, 26, 18,
              10,  2, 59, 51, 43, 35, 27,
              19, 11,  3, 60, 52, 44, 36,
              63, 55, 47, 39, 31, 23, 15,
               7, 62, 54, 46, 38, 30, 22,
              14,  6, 61, 53, 45, 37, 29,
              21, 13,  5, 28, 20, 12,  4};

bitset<56> getKeyPermutation(bitset<64> key)
{
    bitset<56> output;

    for (int i = 0; i < 56; i++)
        output[55 - i] = key[64 - PC_1[i]];

    return output;
}
```

子密钥的生成

由于 DES 算法中的费斯妥密码部分一共需要 16 轮循环，因此共需要 16 个子密钥。在 DES 算法中，采用了同一个步骤循环 16 次的方式生成子密钥。该步骤接受 56 位二进制串的输入，生成 48 位的子密钥和 56 位的输出。其包含两个操作：循环移位和置换选择 2。

之前我们提到，在 DES 密码算法中，加密和解密仅有的区别就是子密钥的使用顺序。因此，这种区别就体现在了子密钥生成的算法上。

在循环移位步骤中，其接受 56 位的输入，然后将这 56 位的二进制串分为左右两个 28 位的二进制串。并在每一轮中，将这两个二进制串分别循环移位。加密过程是左循环移位，解密过程是右循环移位，并且每一轮移动的位数不同。

在循环移位操作完成后，将左右两个二进制串重新拼成一个 56 位的二进制串作为输出和下一轮操作的输入，同时，再将 56 位的二进制串经过一个 56 位到 48 位的 S 盒（称为置换选择 2: PC_2）, 作为本轮的子密钥。

其 C 程序代码为：

```
enum ShiftStyle {
    leftShift,
    rightShift
};

int shiftBits[] = {1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2,
                   2, 2, 1};
int inverseShiftBits[] = {0, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2,
                          2, 2, 2, 2, 1};

int PC_2[] = {14, 17, 11, 24,  1,  5,
               3, 28, 15,  6, 21, 10,
              23, 19, 12,  4, 26,  8,
              16,  7, 27, 20, 13,  2,
              41, 52, 31, 37, 47, 55,
              30, 40, 51, 45, 33, 48,
              44, 49, 39, 56, 34, 53,
              46, 42, 50, 36, 29, 32};

bitset<56> shiftKey(bitset<56> key, int round,
                    ShiftStyle shiftStyle)
{
    bitset<56> output;

    bitset<28> previousLeftPart;
    for (int i = 0; i < 28; i++)
        previousLeftPart[27 - i] = key[55 - i];

    bitset<28> previousRightPart;
    for (int i = 0; i < 28; i++)
        previousRightPart[27 - i] = key[27 - i];

    int *shift;
    switch (shiftStyle)
    {
        case leftShift:
            shift = shiftBits;
            break;

        case rightShift:
            shift = inverseShiftBits;
            break;
```

```
        default:
            break;
    }

    bitset<28> leftPart;
    bitset<28> rightPart;
    int shiftBit = shift[round];

    switch (shiftStyle)
    {
        case leftShift:
            for (int i = 0; i < 28; i++)
            {
                leftPart[27 - i] = previousLeftPart[(27 - i
                                    - shiftBit + 28) % 28];
                rightPart[27 - i] = previousRightPart[(27 - i
                                    - shiftBit + 28) % 28];
            }
            break;

        case rightShift:
            for (int i = 0; i < 28; i++)
            {
                leftPart[27 - i] = previousLeftPart[(27 - i
                                    + shiftBit + 28) % 28];
                rightPart[27 - i] = previousRightPart[(27 - i
                                    + shiftBit + 28) % 28];
            }
            break;

        default:
            break;
    }

    for (int i = 0; i < 28; i++)
        output[55 - i] = leftPart[27 - i];

    for (int i = 28; i < 56; i++)
        output[55 - i] = rightPart[55 - i];

    return output;
```

```
}

bitset<48> getSubkey(bitset<56> key, int round,
                     ShiftStyle shiftStyle)
{
    bitset<48> output;

    for (int i = 0; i < 48; i++)
        output[47 - i] = key[56 - PC_2[i]];

    return output;
}
```

DES 加密和解密

以上就是 DES 密码的每个组成部分。我们可以把它们组合起来，实现 DES 的加密和解密。其 C 程序代码如下：

```
bitset<64> DES_ENC(bitset<64> plainText, bitset<64> key)
{
    bitset<64> cipher;
    bitset<64> roundInput = getInitialPermutation(plainText);
    bitset<64> roundOutput;
    bitset<56> permutatedKey = getKeyPermutation(key);
    bitset<56> previousShiftOutput = shiftKey(permutatedKey, 0,
                                              leftShift);
    for (int i = 0; i < 16; i++)
    {
        bitset<48> subkey = getSubkey(previousShiftOutput, i,
                                      leftShift);
        roundOutput = round(roundInput, subkey);
        roundInput = roundOutput;
        previousShiftOutput = shiftKey(previousShiftOutput,
                                      i + 1, leftShift);
    }
    cipher = getInversePermutation(
                exchangeLeftAndRight(roundOutput));
    return cipher;
}

bitset<64> DES_DEC(bitset<64> cipher, bitset<64> key)
{
    bitset<64> plainText;
```

```
    bitset<64> roundInput = getInitialPermutation(cipher);
    bitset<64> roundOutput;
    bitset<56> permutatedKey = getKeyPermutation(key);
    bitset<56> previousShiftOutput = shiftKey(permutatedKey, 0,
                                              rightShift);
    for (int i = 0; i < 16; i++)
    {
        bitset<48> subkey = getSubkey(previousShiftOutput, i,
                                      rightShift);
        roundOutput = round(roundInput, subkey);
        roundInput = roundOutput;
        previousShiftOutput = shiftKey(previousShiftOutput,
                                      i + 1, rightShift);
    }
    plainText = getInversePermutation(
                    exchangeLeftAndRight(roundOutput));
    return plainText;
}
```

3.5.2 多重 DES

我们可以看出，DES 密码使用了费斯妥密码，并且局部也使用了 SP 网络，这样使这种分组密码的安全性较高。在几乎 30 年的大量研究之后，已知对 DES 的最好的实用攻击仍然只是对密钥空间的穷举搜索。但是，DES 使用的密钥在去除奇偶校验位之后的实际长度只有 56 位，在如今的计算机水平下，变得十分容易破解。在 2017 年，通过计算机更是创下了在 25 秒内破解 DES 的记录。

鉴于此，人们选择了多重 DES 加密。如二重 DES 加密：使用一个 112 位的密钥 K, 将其分为 K_1 和 K_2 两个 56 位的密钥。如果记 $\mathrm{E}_k(m)$ 为 DES 的加密过程，$\mathrm{D}_k(m)$ 为 DES 的解密过程，那么，其加密过程为

$$\mathrm{E}_{K_1}(\mathrm{E}_{K_2}(m))$$

解密过程为

$$\mathrm{D}_{K_2}(\mathrm{D}_{K_1}(c))$$

而如今最常用的是二密钥的三重 DES，简称为 3DES 密码。其使用一个 112 位的密钥 K, 将其分为 K_1 和 K_2 两个 56 位的密钥，其加密过程为

$$\mathrm{E}_{K_1}(\mathrm{D}_{K_2}(\mathrm{E}_{K_1}(m)))$$

解密过程为

$$\mathrm{D}_{K_1}(\mathrm{E}_{K_2}(\mathrm{D}_{K_1}(c)))$$

3.5.3 结构特性

除了 DES 的密钥过短，从数学角度来看，DES 密码拥有一些结构特性，也降低了其破解的难度。

互补特性

对于二进制串 m, 如果我们记 $\overline{m}$ 为 m 按位取补，并用 $\mathrm{DES}_k(m)$ 表示通过密钥 k, 二进制串 m 的 DES 加密的密文，那么，我们可以证明：

$$\mathrm{DES}_{\overline{k}}(\overline{m}) = \overline{\mathrm{DES}_k(m)} \tag{3.9}$$

因此，在使用穷举搜索破解时，可以使工作量减少一半。假设有一个使用已知明文攻击的攻击者，他可以选择明密文对 (M, C) 和 $\left(\overline{M}, C^*\right)$. 那么，在所有的 2^{56} 个可能的密钥，也就是 2^{55} 对互补的密钥二进制串中，他只需要每对互补的二进制串中取一个，一共搜索 2^{55} 个密钥。对于每个尝试的密钥 l, 如果 $\mathrm{DES}_l(M) = C$ 或 $\overline{C^*}$, 就说明密钥是 l 或 $\bar{l}$.

弱密钥与半弱密钥

在多重 DES 加密的过程中，有一些密钥十分危险。比如说弱密钥：

定义 3.1. *在 DES 加密的过程中，如果存在一个密钥 w, 使得*

$$\mathrm{E}_w(\mathrm{E}_w(m)) = m \tag{3.10}$$

则称 w 为弱密钥。

从另一个角度解释公式3.10, 也就是说，

$$\mathrm{E}_w(m) = \mathrm{D}_w(m) \tag{3.11}$$

而我们之前提到，DES 密码的加密与解密过程唯一的区别就是子密钥的使用顺序。据此我们可以很容易地构造弱密钥 w, 也就是使其生成子密钥时加密与解密循环移位的结果相同即可。在 56 位的密钥中，共有 4 个弱密钥。

此外，还有半弱密钥对

定义 3.2. *在 DES 加密的过程中，如果存在一对密钥 w_1, w_2, 使得*

$$\mathrm{E}_{w_1}(\mathrm{E}_{w_2}(m)) = m \tag{3.12}$$

则称 w_1, w_2 为一对半弱密钥。

在 3DES 密码中，如果选取的两个密钥是一对半弱密钥，后果不堪设想。在 56 位的密钥中，共有 6 对半弱密钥。

3.6 IDEA

3.6.1 符号说明

在介绍 IDEA 之前，首先，先介绍一些 IDEA 加密过程中用到的数学符号。

- 逐比特异或 $\oplus$

  $m_1 \oplus m_2$ 即将两个 16 位的二进制串逐比特异或。

- 模 2^{16} 整数加法 $\boxplus$

  即对于 16 位二进制数 m_1, m_2, $m_1 \boxplus m_2 = (m_1 + m_2) \bmod 2^{16}$.

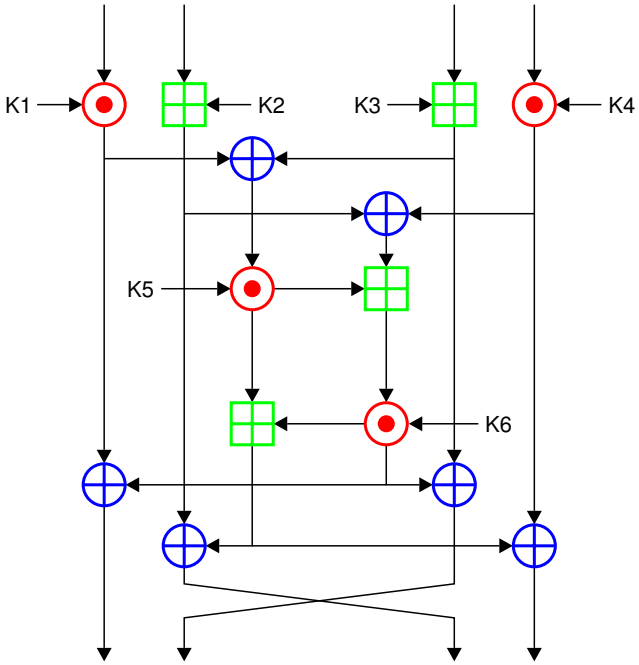
- 模 $2^{16}+1$ 整数乘法 $\odot$

  即对于 16 为二进制数 m_1, m_2, $m_1 \odot m_2 = (m_1 \cdot m_2) \bmod (2^{16}+1)$.

  这里特别指出，如果 $m_1 = 00\ldots0$, 应把 m_1 看作 2^{16}. 这是由于 $2^{16}+1 = 65537$ 为素数，故由数论知识我们可以知道，模 $2^{16}+1$ 的非零整数乘法构成一个群，即所有非零整数都有逆元。此外，这样也可以保证输出一定不会超过 16 位。由于 $2^{16}+1$ 为素数，所以如果 $(m_1 m_2) \bmod (2^{16}+1) = 0$, 则表明 m_1 或 m_2 必然是 $2^{16}+1$ 的倍数。而 m_1, m_2 均为 16 位字符串，所以这是不可能的。

3.6.2 轮结构

和其他分组密码类似，IDEA 也是采用了多次重复轮结构的步骤。其轮结构如图所示（图源 wiki）:

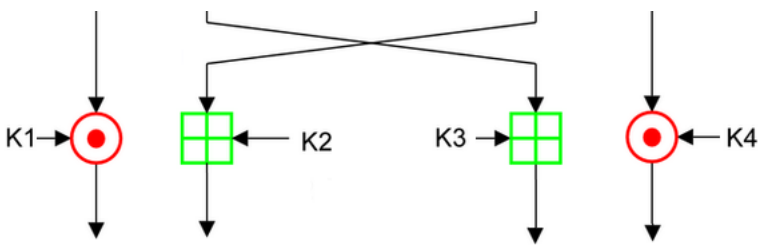


在每一轮中，一共需要输入 6 个 16 位的子密钥，并且输入为 4 组 16 位的二进制串，输出也为 4 组 16 位的二进制串。

3.6.3 加密过程

IDEA 加密算法接受 64 位明文和 128 位密钥。对于密钥，将其通过一个子密钥生成器生成 48 个 16 位的子密钥用于 8 轮轮结构，加上 6 个 16 位的子密钥用于输出变换。

首先，将 64 位明文分成 4 组等长的子串，然后输入轮结构中。在经过 8 轮轮结构后，将输出的结构通过如图所示的输出变换（图源 wiki），得到密文。



而子密钥的生成算法则相对比较直接：

前 8 个子密钥 $Z_1, Z_2, \ldots, Z_8$ 直接在加密密钥中依次选取。然后，将加密密钥循环左移 52 位，再依次取接下来的 8 个子密钥。以此类推，直到 52 个子密钥全部生成。

3.7 AES

3.7.1 输入与输出

AES 密码接受的明文与密钥的长度可独立选择 128 位、192 位或 256 位。这些选择之间的区别仅仅在于加密运算的轮数以及子密钥的调度不同。但是在 AES 密码标准中，明文长度固定为 128 位，密钥长度可以选择为 128, 192 或 256 位，分别叫做 AES-128, AES-192 和 AES-256.

在处理过程中，我们对明文和密钥进一步分组。AES 密码的每个步骤的处理单位是一个“字节”，即 8 个比特。因此，我们将明文和密钥分成每组长度为 8 比特的分组，每个 8 比特的明文分组称为一个“状态”。

对于由字节组成的明文，我们第二次进行分组，使其分成 4 个等长的分组，记明文的每个分组的长度为 N_b (在实际操作中明文总是 128 位，因此 $N_b = 4$). 此外，我们也记以字节为单位的密钥的长度除以 4 为 N_k (N_k 的可能取值为 $4, 6, 8$). 我们如果假定由字节组成的明文是一个长度为 $4N_b$ 的数列 $\{M_n\}$, 那么，我们可以按下表的顺序填充分组：

M_0	M_4	$\cdots$	M_{4N_b-3}
M_1	M_5	$\cdots$	M_{4N_b-2}
M_3	M_6	$\cdots$	M_{4N_b-1}
M_4	M_7	$\cdots$	M_{4N_b}

由明文的分组组成的阵列称为状态阵列。在 AES 的实际过程中，都是对状态阵列进行操作。最终的密文就是从状态序列中，按写入顺序拿出的序列。

因此，在加密的输入阶段，我们要做的事是将明文和密钥按字节分组，然后将明文再填入状态阵列中。在加密的输出阶段，就是将密文从状态阵列中读出，再变成以比特为单位的二进制串（但事实上，明文或密文的按字节分组可以直接在填入状态阵列的时候做。因此，我们只需要实现比特串向字节串的转化用于密钥，而不需要实现字节串向比特串的转化）。在解密阶段，我们要做的也是同样的步骤，只不过是明文和密文对调。

此环节的 C 程序代码如下：

```
void bitToByte(bitset<128> inputBits, bitset<8> outputBytes[64])
{
    for (int byte = 0; byte < 64; byte++)
        for (int bit = 0; bit < 8; bit++)
            outputBytes[byte][bit] = inputBits[8 * byte + bit];
}

void bitToState(bitset<128> input, bitset<8> state[4][4])
{
    for (int column = 0; column < 4; column++)
```

```
        for (int row = 0; row < 4; row++)
            for (int bit = 0; bit < 8; bit++)
                state[row][column][bit] = input[32 * column
                                                + 8 * row + bit];
}

void stateToBit(bitset<8> state[4][4], bitset<128> &output)
{
    for (int column = 0; column < 4; column++)
        for (int row = 0; row < 4; row++)
            for (int bit = 0; bit < 8; bit++)
                output[32 * column + 8 * row + bit]
                    = state[row][column][bit];
}
```

3.7.2 轮函数

AES 的轮函数由 4 个计算部件组成，分别为字节代换 (ByteSub), 行移位 (ShiftRow), 列混合 (MixColumn), 密钥加 (AddRoundKey).

字节代换

字节代换可以看作一个 128 位到 128 位的 S 盒。其接受输入状态阵列，然后对状态阵列实现 S 盒的操作。

在 C 程序实现里，加密过程中，S 盒为`bitset<8> S_box[16][16]`; 解密过程中，S 盒为`bitset<8> Inv_S_Box[16][16]`. 这两个 S 盒的值在 AES 中是固定的，其生成方式可以看后面的数学论证部分。

其 C 程序实现如下：

```
void SubByte(bitset<8> state[4][4], CryptoMode cryptoMode)
{
    bitset<8> (*crypto_S_box)[16];

    switch (cryptoMode)
    {
        case Enc:
            crypto_S_box = S_box;
            break;

        case Dec:
            crypto_S_box = Inv_S_Box;
            break;

        default:
```

```
            break;
    }

    for (int row = 0; row < 4; row++)
        for (int column = 0; column < 4; column++)
        {
            bitset<8> previousValue = *(*(state + row) + column);
            int S_box_row = previousValue[7] * 8
                                + previousValue[6] * 4
                                    + previousValue[5] * 2
                                        + previousValue[4];
            int S_box_column = previousValue[3] * 8
                                + previousValue[2] * 4
                                    + previousValue[1] * 2
                                        + previousValue[0];
            *(*(state + row) + column)
                = crypto_S_box[S_box_row][S_box_column];
        }
}
```

行移位

在之前处理数据的时候，我们提到把明文分为 4 组。每组可以看作一行，每行包括 N_b 个字节。

所谓行移位，就是将各行进行循环移位，不同的行的位移量不同。第一行不移位，第二行左移 C_1, 第三行左移 C_2, 第四行左移 C_3. C_1, C_2, C_3 都与 N_b 有关。

在解密时，只需要反向循环移位即可。

其 C 程序实现如下：

```
int shiftBytes[4] = {0, 1, 2, 3};

void ShiftRows(bitset<8> state[4][4], CryptoMode cryptoMode)
{
    for (int row = 0; row < 4; row++)
    {
        bitset<8> previousRow[4];
        for (int column = 0; column < 4; column++)
            previousRow[column] = state[row][column];
        switch (cryptoMode)
        {
            case Enc:
                for (int column = 0; column < 4; column++)
                    state[row][column]
```

```
                        = previousRow[(column
                                      - shiftBytes[row] + 4) % 4];
                break;

            case Dec:
                for (int column = 0; column < 4; column++)
                    state[row][column]
                        = previousRow[(column
                                      + shiftBytes[row]) % 4];
                break;

            default:
                break;
        }
    }
}
```

列混合

该步骤是对状态阵列的每一列进行变换。为了方便我们理解以及后面的数学论证，我们将这一步理解成矩阵乘法。由于每一列共有 4 个元素，因此，我们可以把它看作一个四维列向量 $(a_0, a_1, a_2, a_3)^{\mathrm{T}}$. 其输出也为 4 位列向量 $(b_0, b_1, b_2, b_3)^{\mathrm{T}}$.

在加密过程中，其计算方法为

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \tag{3.13}$$

在解密过程中，其计算方法为

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{3.14}$$

这里要特别指出的是，在矩阵相乘时，每一项之间的乘法并不是我们平时用到的十进制乘法。这种乘法遵循特定的乘法表。在 C 程序实现中，我们发现，只需要使用与 01, 02, 03, 09, 0b, 0d, 0e 的乘法。因此，对于这 7 个数，我们各有一张长度为 256 的表，用于与一个字节对应的比特（共 $2^8 = 256$ 种）相乘（a_0, a_1, a_2, a_3 都是一个字节）的结果。

其 C 程序实现如下：

```
bitset<8> *M[4][4] = {
    {Mul_02, Mul_03, Mul_01, Mul_01},
    {Mul_01, Mul_02, Mul_03, Mul_01},
    {Mul_01, Mul_01, Mul_02, Mul_03},
```

```
    {Mul_03, Mul_01, Mul_01, Mul_02}
};

bitset<8> *Inv_M[4][4] = {
    {Mul_0e, Mul_0b, Mul_0d, Mul_09},
    {Mul_09, Mul_0e, Mul_0b, Mul_0d},
    {Mul_0d, Mul_09, Mul_0e, Mul_0b},
    {Mul_0b, Mul_0d, Mul_09, Mul_0e}
};

void MixColumns(bitset<8> state[4][4], CryptoMode cryptoMode)
{
    bitset<8>* (*crypto_M)[4][4];

    switch (cryptoMode)
    {
        case Enc:
            crypto_M = &M;
            break;

        case Dec:
            crypto_M = &Inv_M;
            break;

        default:
            break;
    }
    for (int column = 0; column < 4; column++)
    {
        bitset<8> previousColumn[4];
        for (int row = 0; row < 4; row++)
            previousColumn[row] = state[row][column];
        for (int row = 0; row < 4; row++)
        {
            state[row][column] = 0;
            for (int i = 0; i < 4; i++)
                state[row][column]
                    ^= *(*(*(*crypto_M + row) + i)
                        + previousColumn[row].to_ulong());
        }
    }
}
```

密钥加

在我们输入密钥之后，首先会通过密钥编排算法，得到若干长度为 N_b 的轮密钥。密钥加即为将轮密钥与当前的状态阵列逐比特异或。解密时再次异或即可。

其 C 程序实现如下：

```
void AddRoundKey(bitset<8> state[4][4], bitset<32> ki[4])
{
    for (int row = 0; row < 4; row++)
        for (int column = 0; column < 4; column++)
            for (int bit = 0; bit < 8; bit++)
                state[row][column][bit]
                    = ki[row][column * 4 + bit]
                        ^ state[row][column][bit];
}
```

轮函数总的步骤

在每一轮中，对当前的状态阵列，轮函数依次进行字节代换、行移位、列混合和与当前轮密钥的密钥加，最后输出。

在最后一轮中，不进行列混合。

3.7.3 密钥编排算法

之前我们讲到，我们输入的密钥会通过密钥编排算法变成若干长度为 N_b 的轮密钥。密钥编排算法分为密钥扩展和轮密钥选取两个部分。

考虑到每个轮密钥长度都为 N_b, 假设我们需要进行 N_r 轮，则一共需要 $N_b\,(N_r+1)$ 长度的密钥。因此，我们首先需要将输入的密钥扩展成对应长度的扩展密钥。

然后，第一轮轮密钥取扩展密钥的第一个 N_b 长度个字，第二轮轮密钥取接下来的 N_b 长度个字，以此类推。

密钥扩展算法

在密钥扩展算法中，我们需要用到 N_r 个轮常数Rcon，以及一些辅助工作，如对密钥的循环移位RotWord和对密钥的替换（使用的 S 盒与之前的是同一个）SubWord. 此外，对于不同的 N_r, 密钥扩展算法也不同。

这里以 AES-128 为例，其 C 程序代码如下：

```
bitset<32> Rcon[10] = {0x01000000, 0x02000000, 0x04000000,
0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000,
0x1b000000, 0x36000000};

bitset<32> RotWord(bitset<32> input)
{
    bitset<32> output;
```

```
    for (int i = 0; i < 32; i++)
        output[i] = input[(i - 8 + 32) % 32];

    return output;
}

bitset<32> SubWord(bitset<32> input)
{
    bitset<32> output;

    for (int byte = 0; byte < 4; byte++)
    {
        int row = input[8 * byte + 7] * 8
                    + input[8 * byte + 6] * 4
                        + input[8 * byte + 5] * 2
                            + input[8 * byte + 4];
        int column = input[8 * byte + 3] * 8
                        + input[8 * byte + 2] * 4
                            + input[8 * byte + 1] * 2
                                + input[8 * byte];
        bitset<8> value = S_box[row][column];
        for (int bit = 0; bit < 8; bit++)
            output[8 * byte + bit] = value[bit];
    }

    return output;
}

void KeyExpansion(bitset<8> *key, bitset<32> *w)
{
    int Nk = 4;
    int Nr = 10;

    for (int word = 0; word < Nk; word++)
        for (int byte = 0; byte < 4; byte++)
            for (int bit = 0; bit < 4; bit++)
                w[word][4 * byte + bit] = key[byte][bit];

    for (int word = Nk; word < 4 * (Nr + 1); word++)
    {
        bitset<32> tmp = w[word - 1];
```

```
        if (!word % Nk)
            tmp = SubWord(RotWord(tmp)) ^ Rcon[word / Nk];
        w[word] = w[word - Nk] ^ tmp;
    }
}
```

3.7.4 总的加解密过程

加密过程

首先，将明文写入状态阵列，密钥变成字节串。然后，求出扩展密钥。接着，对前 4 个密钥使用密钥加算法。然后，在进行 10 轮的字节替换、行移位、列混合、密钥加，其中最后一轮不进行列混合。最后，将此时的状态阵列读出为密文。

其 C 程序代码如下：

```
bitset<128> AES_128_ENC(bitset<128> plainText, bitset<128> key)
{
    bitset<128> cipher;

    bitset<8> state[4][4];
    bitToState(plainText, state);

    bitset<8> keys[16];
    bitToByte(key, keys);

    bitset<32> expanedKeys[44];
    KeyExpansion(keys, expanedKeys);

    bitset<32> subkeys[4];
    for (int i = 0; i < 4; i++)
        subkeys[i] = expanedKeys[i];
    AddRoundKey(state, subkeys);

    for (int round = 0; round < 9; round++)
    {
        SubByte(state, Enc);
        ShiftRows(state, Enc);
        MixColumns(state, Enc);
        for (int i = 0; i < 4; i++)
            subkeys[i] = expanedKeys[i + 4 + round * 4];
        AddRoundKey(state, subkeys);
    }

    SubByte(state, Enc);
```

```
    ShiftRows(state, Enc);
    for (int i = 0; i < 4; i++)
        subkeys[i] = expanedKeys[i + 40];
    AddRoundKey(state, subkeys);

    stateToBit(state, cipher);

    return cipher;
}
```

解密算法

首先，将密文写入状态阵列，密钥变成字节串。然后，求出扩展密钥。接着，对最后 4 个密钥使用密钥加算法。然后，在进行 10 轮的字节替换、行移位、密钥加、列混合，其中最后一轮不进行列混合。最后，将此时的状态阵列读出为明文。

解密与加密的区别在于：轮密钥的使用顺序恰好相反（但每一轮内的顺序是相同的）。

其 C 程序代码如下：

```
bitset<128> AES_128_DEC(bitset<128> cipher, bitset<128> key)
{
    bitset<128> plainText;

    bitset<8> state[4][4];
    bitToState(cipher, state);

    bitset<8> keys[16];
    bitToByte(key, keys);

    bitset<32> expanedKeys[44];
    KeyExpansion(keys, expanedKeys);

    bitset<32> subkeys[4];
    for (int i = 0; i < 4; i++)
        subkeys[i] = expanedKeys[40 + i];
    AddRoundKey(state, subkeys);

    for (int round = 0; round < 9; round++)
    {
        SubByte(state, Dec);
        ShiftRows(state, Dec);
        MixColumns(state, Dec);
        for (int i = 0; i < 4; i++)
            subkeys[i] = expanedKeys[i + 36 - round * 4];
```

```
            AddRoundKey(state, subkeys);
        }

        SubByte(state, Dec);
        ShiftRows(state, Dec);
        for (int i = 0; i < 4; i++)
            subkeys[i] = expanedKeys[i];
        AddRoundKey(state, subkeys);

        stateToBit(state, plainText);

        return plainText;
    }
```

3.7.5 数学基础

有限域 GF (2^8)

我们之前在讲流密码的时候提到过有限域 GF (2). 在这里，我们就要正式地引入域的概念：

所谓域，就是可交换的除环。更确切地说：

定义 **3.3.** 对于集合 F 和它上面的两个运算 $+, \cdot$, 如果满足如下性质：

1. 加法和乘法的封闭性

   $\forall a, b \in F, a+b \in F, a \cdot b \in F$

2. 加法和乘法的结合律

   $\forall a, b, c \in F, (a+b)+c = a+(b+c), (a \cdot b) \cdot c = a \cdot (b \cdot c)$

3. 加法和乘法的交换律

   $\forall a, b \in F, a+b = b+a, a \cdot b = b \cdot a$

4. 乘法对加法的分配率

   $\forall a, b, c \in F, a \cdot (b+c) = a \cdot b + a \cdot c$

5. 加法单位元

   $\exists a \in F, \text{s.t.} \forall b \in F, a+b = b+a = b$

   常将加法单位元 a 记作 0

6. 乘法单位元

   $\exists a \in F, \text{s.t.} \forall b \in F \text{且} b \neq 0, a \cdot b = b \cdot a = b$

   常将乘法单位元 a 记作 1

7. 加法逆元

   $\forall a \in F, \exists -a \in F, \text{s.t.} a+(-a) = (-a)+a = 0$

8. 乘法逆元

$$\forall a \in F \text{且} a \neq 0, \exists a^{-1} \in F, \text{s.t.}\, a \cdot a^{-1} = a^{-1} \cdot a = 0$$

则称 F 和它上面的两个运算 $+, \cdot$ 构成一个域。

如果 F 内的元素个数有限，为 n 个，则称 F 为有限域，记作 GF (n). 在 AES 中，我们主要研究有限域 GF (2^8).

熟悉抽象代数的同学可能会知道，对于任意素数 p 和正整数 n, 所有有限域 GF (p^n) 都是同构的。因此，我们可以使用一个尽可能简单的表示方法来研究 GF (2^8).

我们考虑集合 $F = \left\{ \sum_{i=0}^{7} a_i x^i \mid a_i \in \text{GF}\,(2) \right\}$ 为一个多项式的集合。我们想通过定义其上的加法与乘法使其变成一个 GF (2^8). 这时，就需要我们之前提到的 GF (2) 上的多项式。为了方便叙述，我们记 G 为 GF (2) 上的多项式组成的集合，定义 $\oplus$ 和 $\odot$ 为 G 上的加法与乘法，$+$ 和 $\cdot$ 为 F 上的加法与乘法。

如果定义其元素加法为：$\forall p_1 = \sum_{i=0}^{7} a_i x^i \in F, p_2 = \sum_{i=0}^{7} b_i x^i \in F$:

$$p_1 + p_2 = \sum_{i=0}^{7} (a_i + b_i)\, x^i \tag{3.15}$$

其中 $a_i + b_i$ 是 GF (2) 上的加法，也就是模 2 加法，或者说是异或。因此，F 上的加法与 G 上的加法相同。

而为了定义其元素乘法，首先引入一个 8 次不可约多项式

$$m\,(x) = x^8 + x^4 + x^3 + x + 1 \tag{3.16}$$

同时，定义 G 上的取模运算为：对于 $p_1, p_2 \in G$, 如果存在多项式 $q \in G, r \in G$, 使得 $p_1 = q \odot p_2 \oplus r$, 且 r 的次数低于 p_2, 则记作 $p_1 \bmod p_2 = r$.

接着，我们就可以定义 F 上的乘法为：$\forall p_1 = \sum_{i=0}^{7} a_i x^i \in F, p_2 = \sum_{i=0}^{7} b_i x^i \in F$:

$$p_1 \cdot p_2 = (p_1 \odot p_2) \bmod m\,(x) \tag{3.17}$$

可以证明，F 对 $+, \cdot$ 构成一个有限域 GF (2^8).

此外，我们还需要在 F 上定义一个一元运算，称为 x 乘法：对于 $p \in F$, 记

$$\text{xtime}\,(p) = x \cdot p \tag{3.18}$$

这里需要强调的是，虽然我们用多项式来表示 GF (2^8), 但这并不意味着 GF (2^8) 的元素就是多项式。我们不在意它的元素是什么，我们关注的是它具有 2^8 个元素，并且它上面的乘法和加法表符合我们之前利用多项式定义的乘法和加法表。

GF (2^8) 上的多项式

类似于我们之前在流密码中讲到的 GF (2) 上的多项式，在我们定义了 GF (2^8) 的加法与乘法之后，就可以定义 GF (2^8) 上的多项式了。但是在 AES 中，与 GF (2) 上的多项式不同的是，GF (2^8) 上的多项式为系数属于 GF (2^8) 且次数小于 4 的多项式。即集合 $\left\{ \sum_{i=0}^{3} a_i x^i \mid a_i \in \text{GF}\left(2^8\right) \right\}$.

同时，我们定义 GF (2^8) 上的多项式的模 x^4+1 乘法 $\otimes$. 即：

$$p_1 \otimes p_2 = (p_1 \cdot p_2) \bmod \left(x^4+1\right) \tag{3.19}$$

其中 $\cdot$ 是 GF (2^8) 上的多项式的普通乘法。

此外，如果我们假设 $p_1 = \sum_{i=0}^{3} a_i x^i, p_2 = \sum_{i=0}^{3} b_i x^i$, 那么

$$\begin{aligned} p_1 \cdot p_2 &= \sum_{i=0}^{3}\sum_{j=0}^{3} a_i b_j x^{i+j} \\ &= a_0b_0 + (a_0b_1 + a_1b_0)\,x + (a_0b_2 + a_1b_1 + a_2b_0)\,x^2 + (a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)\,x^3 \\ &+ (a_1b_3 + a_2b_2 + a_3b_1)\,x^4 + (a_2b_3 + a_3b_2)\,x^5 + a_3b_3x^6 \end{aligned}$$

如果我们记 t_{ij} 为 $p_1 \cdot p_2$ 中 x^i 的系数中，b_j 前面的系数（如 $t_{00} = a_0, t_{10} = a_1$），那么

$$\begin{aligned} p_1 \otimes p_2 &= \left(\sum_{i=0}^{6}\left(\sum_{j=0}^{3} t_{ij}b_j\right)x^i\right) \bmod \left(x^4+1\right) \\ &= \sum_{i=0}^{3}\left(\sum_{j=0}^{3} t_{ij}b_j\right)x^i + \sum_{j=4}^{6}\left(\sum_{j=0}^{3} t_{ij}b_j\right)\left(x^j \bmod \left(x^4+1\right)\right) \end{aligned}$$

如果记 $x^i \bmod \left(x^4+1\right) = \sum_{j=0}^{3} \alpha_{ij}x^j$, 同时如果我们假设 $p_1 \otimes p_2 = \sum_{i=0}^{3} c_i x^i$, 那么

$$\begin{aligned} \sum_{i=0}^{3} c_i x^i &= \sum_{i=0}^{3}\left(\sum_{j=0}^{3} t_{ij}b_j\right)x^i + \sum_{k=4}^{6}\left(\sum_{m=0}^{3} t_{km}b_m\right)\left(\sum_{l=0}^{3}\alpha_{kl}x^l\right) \\ &= \sum_{i=0}^{3}\left(\sum_{j=0}^{3} t_{ij}b_j\right)x^i + \sum_{l=0}^{3}\left(\sum_{k=4}^{6}\left(\sum_{m=0}^{3} t_{km}b_m\right)\alpha_{kl}\right)x^l \\ &= \sum_{i=0}^{3}\left(\sum_{j=0}^{3}\left(t_{ij} + \sum_{k=4}^{6} t_{kj}\alpha_{ki}\right)b_j\right)x^i \end{aligned}$$

因此

$$c_i = \sum_{j=0}^{3}\left(t_{ij} + \sum_{k=4}^{6} t_{kj}\alpha_{ki}\right)b_j \tag{3.20}$$

注意到这个式子中，$t_{ij}, t_{kj}, \alpha_{ki}$ 都与 p_2 无关，因此可以提前算出来。从而，我们可以得到：

定理 3.1. *对于多项式 $p_1 = a_0 + a_1x + a_2x^2 + a_3x^3$ 和 $p_2 = b_0 + b_1x + b_2x^2 + b_3x^3$, 若 $p_1 \otimes p_2 = c_0 + c_1x + c_2x^2 + c_3x^3$, 则*

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_2 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{3.21}$$

这里矩阵里元素的乘法是 GF (2^8) 上的乘法。

同时还有一个推论：

定理 3.2. GF (2^8) 上的多项式 $a_3x^3+a_2x^2+a_1x+a_0$ 是模 x^4+1 可逆的，当且仅当矩阵

$$\begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_2 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix}$$

在 GF (2^8) 上是可逆的。

AES 加密过程

在 AES 加密的过程中，我们将长度为 128 个比特的明文分成了 4 行 $N_b=4$ 列的状态阵列。每个阵列元素为一个长度为 8 比特的字节。因此，一共有 $2^8=256$ 种字节。所以，我们可以将一个字节看作一个 GF (2^8) 中的元素。

加密过程中的字节代换分为两步：

1. 将字节看作 GF (2^8) 上的元素，将其映射到自己的乘法逆元。全 0 字节映射到自己。

2. 接着，对字节作以下 GF (2) 上的运算，输出 $(y_0,y_1,\ldots,y_7)$:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{3.22}$$

在解密过程中，将上述的矩阵换为其逆矩阵即可。

对于加密过程中的列混合：

由于明文被分为 4 组，每组看作一行，那么对于每列来说，都含有四个元素，且每个元素都属于 GF (2^8). 因此，可以将每列看作 GF (2^8) 上的多项式。接着，记

$$c(x)=(03)_{16}x^3+(01)_{16}x^2+(01)_{16}x+(02)_{16} \tag{3.23}$$

其中 $(03)_{16},(01)_{16},(02)_{16}$ 都是用两个 16 进制数表示的 8 比特二进制串。

然后，对于每列对应的 GF (2^8) 上的多项式，将其与 $c(x)$ 作 GF (2^8) 上多项式的模 x^4+1 乘法 $\otimes$ 来输出。利用之前的讨论，我们可以得到

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \tag{3.24}$$

其中的运算作 GF (2^8) 上的运算。

而通过数学上的计算可知，如果记

$$d(x)=(0B)_{16}x^3+(0D)_{16}x^2+(09)_{16}x+(0E)_{16} \tag{3.25}$$

则在解密运算中，将每列与 $d(x)$ 作 GF (2^8) 上多项式的模 x^4+1 乘法 $\otimes$ 来输出即可。类似地，也有一个对应的矩阵

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{3.26}$$

对于密钥编排算法：

密钥编排算法中用到了轮常数Rcon. 对于Rcon的第 i 个元素的形式为 $((rc_i)_{16}, 00, 00, 00)$, 其算法为

$$rc_i = 2 \cdot rc_{i-1} \tag{3.27}$$

其中 $\cdot$ 为 GF (2^8) 中的乘法，$rc_0 = (01)_{16}$.

第四章　公钥密码

4.1　简介

之前我们讲到，最安全的密码体系是一次一密。但是，由于其需要用安全信道传输的密钥长度至少于明文一样长，并且一个密钥只能用一次，因此，与其将密钥传输，不如将明文传输。所以，一次一密的缺陷也十分明显。为了解决这一问题，之前我们采取的方法是使用各种手法，包括流密码以及分组密码等，来缩短密钥的长度。而公钥密码则提供了另一种不同的思路：减弱对安全信道的需求。其基本思想为：

1. 由解密方生成一对密钥，称为公钥（记作 SK）和私钥（记作 PK）。
2. 解密方将公钥传送给加密方（不需要通过安全信道）。
3. 加密方利用公钥加密明文，传递给解密方。
4. 解密方利用私钥解密密文。

因此，和之前讲到的对称密码体系类似，公钥密码体系包含的三个关键算法是：公钥-私钥生成算法，加密算法，解密算法。

要在实践中实现这个思想，我们得满足：

- 生成公钥、密钥对的算法较容易
- 用公钥加密明文的算法较容易
- 用密钥解密密文的算法较容易
- 由公钥不能（或者很难）得到对应的密钥
- 由密文和公钥不能（或者很难）得到对应的明文
- 公钥、私钥可交换。即：

$$\mathrm{D}_{PK}\left(\mathrm{E}_{SK}\left(m\right)\right)=\mathrm{D}_{SK}\left(\mathrm{E}_{PK}\left(m\right)\right) \tag{4.1}$$

上述的六个要求，实质上就是需要我们找到如下的一种“单向陷门函数”$f(x)$:

对于从 X 到 Y 的函数 $f(x)$, 如果 $\forall x\in X$, $f(x)$ 的计算较容易，而对于几乎所有 Y 中的元素 y, 找出其对应的 x 都是计算不可行的。但是，如果掌握一个“陷门”z, 则求逆较容易。则称 $f(x)$ 为一个单向陷门函数。

比如说将一个怀表拆成许多零件是容易的，将零件重新装回一个怀表是几乎不可能的。但如果我们拥有怀表的构造说明书，那么用零件装回怀表又是较容易的。这就是现实中的一个单项陷门函数。

如果我们找到了这样一种单向陷门函数，那么我们可以用它来构造公钥。

但是，这个定义中，“较容易”、“几乎所有”、“计算不可行”都是一种直观上的感性词语。其真正的严格数学定义需要用到概率多项式时间等高深的方法，这里不再介绍。

此外，公钥密码体系还有一个重要的特点是，其安全性基于数论知识，而非类似于分组密码的代换与替换。并且，在之后的算法细节中我们会了解到，公钥密码体系的算法比对称密码体系的算法要慢许多。同时，由于并非一次一密，因此其安全性从本质上说并不比对称密码体系高。

公钥密码体系也称为非对称密码体系，因为用于加密的公钥不能解密其密文。

同时，公钥密码除了解决了密钥分配问题，即对安全信道的需求问题，其还解决了另外一个问题：数字签名问题。即，在以往的对称加密体系中，我们无法得知收到的密文是否来自于我们选择的加密方。

4.2 RSA 密码

在现在的互联网安全中，RSA 密码体系承担了大部分的工作。如 Outlook 等加密邮件都是使用 RSA 的密码体系。

4.2.1 基本框架

之前讲到，公钥密码体系包括公钥-私钥生成算法，加密算法和解密算法。下面介绍其基本框架。值得指出的是，这里只是写了其操作步骤，而没有说明其具体实现方法。具体的实现方法会在后面再介绍。

公钥-私钥生成算法

1. 取两个素数 p, q, 且 $p \neq q$
2. 计算 $n = pq$ 和 $\phi(n) = (p-1)(q-1)$
3. 取 e, 使得 $1 < e < \phi(n)$ 且 $(\phi(n), e) = 1$, 即 $\phi(n)$ 与 e 互素
4. 计算 d, 使得 d 满足
$$de \equiv 1 \pmod{\phi(n)} \tag{4.2}$$
5. 公钥为 (e, n), 私钥为 (d, n).

加密算法

对于公钥 (e, n), 加密者需要加密的二进制字符串对应的二进制数为 m. 要求 $m < n$. 加密操作为

$$c \equiv m^e \pmod{n} \tag{4.3}$$

再将每个 c 拼起来形成密文串。

解密算法

对于私钥 (d, n), 解密者将接收到的密文 c 作解密操作

$$m \equiv c^d \pmod{n} \tag{4.4}$$

4.2.2 算法细节

上一小节讨论了 RSA 的整体算法结构。接下来，分几个部分介绍一下具体的算法细节。

大整数

在 RSA 中，需要使用到许多大整数。这些大整数规模在 1024 比特左右。因此，我们在程序实现时，采用一个大整数类`BigInteger`, 其将大整数以二进制串的形式存储，成员变量为符号位`sign`(布尔值，非负为`true`)，二进制串长度`length`, 以及由低位向高位存储的二进制串`value`.

大整数相乘模运算

最常用的运算是求两个大整数的乘积对于另一个大整数的模。为了使中间结果的位数不太大，我们可以考虑如下结论：

若 $a = \sum_{i=0}^{n} a_i 2^i$, 那么

$$ab \bmod c = \sum_{i=0}^{n} \left(a_i b 2^i \bmod c\right) \tag{4.5}$$

比如说，如果 $a = (1001)_2 = 2^3 + 2^0$, 那么

$$ab \bmod c = \left(2^3 b \bmod c\right) + \left(2^0 b \bmod c\right)$$

利用这种想法，其 C 程序实现如下：

```
BigInteger mulmod(const BigInteger mul1,
                  const BigInteger mul2, const BigInteger mod)
{
    if (mul1 == 0 || mul2 == 0)
        return 0;

    BigInteger product = 0;
    BigInteger modMul1 = mul1;
    for (int digit = 0; digit < mul2.length; digit++)
    {
        modMul1 = (modMul1 << 1) % mod;
        if (mul2.value[digit])
            product = (product + modMul1) % mod;
    }

    return product;
}
```

大整数幂模运算

此外，还有一个常用的运算是对于大整数 a, b, c, 求 $a^b \bmod c$. 这里常用的方法为快速指数法。其基本思想为：

假设 b 的二进制表示为 $b_k b_{k-1} \cdots b_0$, 即

$$b = b_k 2^k + b_{k-1} 2^{k-1} + \cdots + 2b_1 + b_0 \tag{4.6}$$

那么

$$a^b = \left(\cdots \left(\left(a^{b_k} \right)^2 a^{b_{k-1}} \right)^2 \cdots a^{b_1} \right)^2 a^{b_0} \tag{4.7}$$

比如说，如果 $b = 9$, 那么我们可以

$$a^b \bmod c = a^{8+1} \bmod c = \left(\left(a^2 \right)^2 \right)^2 \cdot a \bmod c$$

将原本需要算 9 次的乘法改进成算 4 次。此外，我们还可以每一次乘法都求模，这样可以使中间结果更小，如：

$b = (1001)_2 = 2^3 + 2^0$, 那么

$$a^b \bmod c = \left(\left(\left(a^2 \bmod c \right)^2 \bmod c \right)^2 \bmod c \right) \cdot a$$

其 C 程序实现为：

```
BigInteger fastExp(const BigInteger base,
                   const BigInteger exponent,
                   const BigInteger mod)
{
    BigInteger power = 1;

    for (int digit = exponent.length - 1; digit >= 0; digit--)
    {
        power = mulmod(power, power, mod);
        if (exponent.value[digit])
            power = mulmod(power, base, mod);
    }

    return power;
}
```

互素判定

判断两个数是否互素，我们常用欧几里德辗转相除法求两个数的最大公因数。其基本原理为

定理 4.1. 对于整数 a, b

$$(a, b) = (b, a \bmod b) \tag{4.8}$$

其 C 程序实现如下：

```
BigInteger gcd(BigInteger a, BigInteger b)
{
    if (a % b == 0)
        return b;
```

```
    return gcd(b, a % b);
}
```

求乘法逆元

由裴蜀定理可知，对于整数 x, y, 存在整数 a, b, 使得

$$ax + by = (x, y) \tag{4.9}$$

那么，如果 $(x, y) = 1$, 那么存在整数 a, b, 使得

$$ax + by = 1 \tag{4.10}$$

因此

$$ax = -by + 1$$

故

$$ax \equiv 1 \pmod{y} \tag{4.11}$$

因此，我们如果可以由 x, y 找出对应的 a, b, 那么 a 就是 x 模 y 的乘法逆元。

因此，我们使用扩展欧几里德算法。其 C 程序实现如下：

```
BigInteger extendGcd(BigInteger x, BigInteger y,
                     BigInteger &a, BigInteger &b)
{
    if (y == 0)
    {
        a = 1;
        b = 0;
        return x;
    }

    BigInteger r = extendGcd(y, x % y, a, b);
    BigInteger tmp = a;
    a = b;
    b = tmp - x / y * b;
    return r;
}
```

从而，求乘法逆元的 C 程序实现如下：

```
BigInteger inverse(const BigInteger a, const BigInteger mod)
{
    BigInteger inverse = 0;
    BigInteger tmp = 0;
    extendGcd(a, mod, inverse, tmp);
```

```
    return inverse;
}
```

素数 p, q 的选取

在之后会介绍，RSA 的安全性取决于 p, q 要是大素数。那么，我们就需要判断素数。最初等的方法是从 3 到 $\sqrt{n}$ 挨个判断是否是 n 的因子。但是对于极大的数，这样判断方法是不现实的。因此，下面介绍一下常用的判断素数的方法：Miller-Rabin 素数测试。

Miller-Rabin 素数测试基于一个基本定理：

定理 4.2. *对于奇数 $n = 2^s d + 1$, 其中 d 为奇数。若存在 a 满足 $\forall 0 \leq r \leq s-1$, 有*

$$a^d \not\equiv 1 \pmod{n} \tag{4.12}$$

$$a^{2^r d} \not\equiv 1 \pmod{n} \tag{4.13}$$

则 p 不是素数。

由此定理，我们取充分多的 a, 对于每个 a 我们测试所有的 $0 \leq r \leq s-1$, 只要有一个不满足，那么 p 就不是素数。如果我们取的 a 充分多，并且都没有找到不满足定理的值，那么 p 就可以被看作一个素数。

那么，我们取多少个 a 比较合适呢？事实上，如果奇数 n 是 k 位二进制数，并对它进行 t 次 Miller-Rabin 测试均返回成功，那么其为合数的概率满足

$$P < \begin{cases} k^2 4^{2-\sqrt{k}} & k \geq 2 \\ k^{\frac{3}{2}} 2^t t^{-\frac{1}{2}} 4^{2-\sqrt{tk}} & y = 2, k \geq 88 \text{或} 3 \leq t \leq \dfrac{k}{9}, k \geq 21 \\ \dfrac{7}{20} k 2^{-5t} + \dfrac{1}{7} k^{-\frac{k}{2}-2t} + 12k 2^{-\frac{k}{4}-3t} & t \geq \dfrac{k}{9}, k \geq 21 \\ \dfrac{1}{7} k^{\frac{15}{4}} 2^{-\frac{k}{2}-2t} & t \geq \dfrac{k}{4}, k \geq 21 \end{cases} \tag{4.14}$$

对于 1024 比特的 n, 选取 40 个 a 以后 n 为合数的概率要小于 2^{-83}. 而事实上，我们也常对 n 使用 40 次 Miller-Rabin 测试。其 C 程序实现如下：

```
bool Miller_Rabin(BigInteger n, int round)
{
    BigInteger m = n - 1;
    int k = 0;
    while (!m.getValue()[0])
    {
        k++;
        m = m >> 1;
    }

    for (int i = 0; i < round; i++)
    {
        BigInteger a = BigInteger::getRand() % (n - 1) + 1;
        BigInteger b = BigInteger::fastExp(a, m, n);
        if (b == 1)
```

```
            return true;

        for (int j = 0; j < k; j++)
        {
            if (b == n - 1)
                return true;
            b = BigInteger::mulmod(b, b, n);
        }
    }
    return false;
}
```

上述讲的是如何测试 p, q 是否为素数。那么如何生成 p, q 呢？常用的方法是：

随机生成一个 1024 比特的奇数，然后对其进行 40 轮 Miller-Rabin 测试。如果不是素数，则将其自增 2.

看似这个方法很没有效率，但是，根据素数定理，在 0 到 N 之间，每两个相邻的素数之间的平均距离为 $\ln N$. 因此，对于 1024 比特的奇数 n, 如果其不为素数，那么其前后两个素数之间的距离约为 $\ln n$. 故其平均需要再往后测试

$$\frac{\ln n}{2} \approx 354$$

次即可。

综上，公钥-私钥对的产生的 C 程序算法如下：

```
void generateKeys(BigInteger &pub, BigInteger &pri)
{
    pub = getRandBit(1024);
    if (!pub.getValue()[0])
        pub += 1;

    while (!Miller_Rabin(pub, 40))
        pub += 2;


    pri = getRandBit(1024);
    if (!pri.getValue()[0])
        pri += 1;

    while (!Miller_Rabin(pri, 40))
        pri += 2;
}
```

e 的选取

由于 RSA 算法的安全性主要在于 p, q 的选取，因此，作为公钥的 e 的选取就没有必要是随机的。常用的 e 取自 $3, 5, 17, 257, 65537$. 判断 e 与 $\phi(n) = (p-1)(q-1)$ 是否互素可以用欧

几里德辗转相除法gcd()来求其最大公因数，判断其是否为 1.

其 C 程序实现如下：

```
BigInteger generateE(BigInteger phi)
{
    BigInteger list[5] = {65537, 257, 17, 5, 3};
    for (int i = 0; i < 5; i++)
        if (BigInteger::gcd(phi, list[i]) == 1)
            return list[i];

    return -1;
}
```

d 的求值

由定义，d 是 e 模 $\phi(n)$ 的乘法逆元。因此，我们采用扩展欧几里德算法求 d.

加密

RSA 的加密过程实际上就是求大整数的幂的模。因此，我们可以采用快速指数法。

解密

RSA 的解密过程是解密方进行的操作。而解密方拥有的数有 $p, q, n, \phi(n), e, d$ 以及密文 c.

解密方可计算

$$\begin{cases} d_p = d \bmod (p-1) \\ d_q = d \bmod q - 1 \end{cases}, \begin{cases} m_p = c^{d_p} \bmod p \\ m_q = c^{d_q} \bmod q \end{cases} \tag{4.15}$$

于是由费马小定理可化简得到

$$\begin{cases} m_p \equiv m \pmod p \\ m_q \equiv m \pmod q \end{cases}$$

运用中国剩余定理:

$$m \equiv qe_p m_p + pe_q m_q \pmod{pq} \tag{4.16}$$

其中 $qe_p \equiv 1 \pmod p, pe_q \equiv 1 \pmod q$.

如果我们采用快速指数法计算 m_p, m_q, 采用扩展欧几里德算法计算 e_p, e_q, 即可得到 m.

其 C 程序实现如下：

```
BigInteger RSA_DEC(BigInteger cipher, BigInteger d,
                   BigInteger p, BigInteger q)
{
    BigInteger dp = d % (p - 1);
    BigInteger dq = d % (q - 1);

    BigInteger mp = fastExp(cipher, dp, p);
    BigInteger mq = fastExp(cipher, dq, q);
```

```
    BigInteger ep = inverse(q, p);
    BigInteger eq = inverse(p, q);

    BigInteger n = p * q;

    BigInteger tmp1 = mulmod(q, ep, n);
    tmp1 = mulmod(tmp1, mp, n);

    BigInteger tmp2 = mulmod(p, eq, n);
    tmp2 = mulmod(tmp2, mq, n);

    return tmp1 + tmp2;
}
```