第3章 数据加密

信息加密技术是保障信息安全的核心技术，已经渗透到大部分安全产品之中，并正向芯片化方向发展。通过数据加密技术，可以在一定程度上提高数据传输的安全性，保证传输数据的完整性。一个数据加密系统包括加密算法、明文、密文以及密钥，密钥控制加密和解密过程，一个加密系统的全部安全性是基于密钥的，而不是基于算法，所以加密系统的密钥管理是一个非常重要的环节。

3.1 数据加密技术

数据加密技术主要分为数据传输加密和数据存储加密。数据传输加密技术主要是对传输中的数据流进行加密，常用的有链路加密、节点加密和端到端加密3种方式。

(1) 链路加密是传输数据仅在物理层上的数据链路层进行加密，不考虑信源和信宿，它用于保护通信节点间的数据。接收方是传送路径上的各台节点机，数据在每台节点机内都要被解密和再加密，依次进行，直至到达目的地。

(2) 与链路加密类似的节点加密方法在节点处采用一个与节点机相连的密码装置，密文在该装置中被解密并被重新加密，明文不通过节点机，避免了链路加密节点处易受攻击的缺点。

(3) 端到端加密是为数据从一端到另一端提供的加密方式。数据在发送端被加密，在接收端解密，中间节点处不以明文的形式出现。端到端加密是在应用层完成的。在端到端加密中，数据传输单位中除报头外的报文均以密文的形式贯穿于全部传输过程，只是在发送端和接收端才有加、解密设备，而在中间任何节点报文均不解密。因此，不需要有密码设备，同链路加密相比，可减少密码设备的数量。另一方面，数据传输单位由报头和报文组成，报文为要传送的数据集合，报头为路由选择信息等(因为端到端传输中要涉及路由选择)。在链路加密时，报文和报头两者均须加密。而在端到端加密时，由于通路上的每一个中间节点虽不对报文解密，但为将报文传送到目的地，所以必须检查路由选择信息，因此只能加密报文，而不能对报头加密。这样就容易被某些通信分析发觉，而从中获取某些敏感信息。

链路加密对用户来说比较容易，使用的密钥较少，而端到端加密比较灵活，对用户可见。在对链路加密中各节点安全状况不放心的情况下也可使用端到端加密方式。

3.2 数据加密技术的发展

1. 密码专用芯片集成

密码技术也正向芯片化方向发展，在芯片设计制造方面，目前微电子工艺已经发展到很高水平，芯片设计的水平也很高。我国在密码专用芯片领域的研究起步落后于国外，近年来我国集成电路产业技术的创新和自我开发能力得到了加强，微电子工业得到了发展，从而推动了密码专用芯片的发展。加快密码专用芯片的研制将会推动我国信息安全系统的完善。

2. 量子加密技术的研究

量子技术在密码学上的应用分为两类，一类是利用量子计算机对传统密码体制进行分析；另一类是利用单光子的测不准原理在光纤一级实现密钥管理和信息加密，即量子密码学。量子计算机相当于一种传统意义上的超大规模并行计算系统，利用量子计算机，可以在几秒钟内分解 RSA 129 的公钥。根据互联网的发展，全光纤网络将是今后网络连接的发展方向，利用量子技术可以实现传统的密码体制，在光纤一级完成密钥交换和信息加密，其安全性是建立在 Heisenberg 的测不准原理之上的，如果攻击者企图接收并检测信息发送方的信息（偏振），则将造成量子状态的改变，这种改变对攻击者而言是不可恢复的，而对收发方则可很容易地检测出信息是否受到攻击。目前量子加密技术仍然处于研究阶段。

3.3 数据加密算法

数据加密算法有很多种，密码算法标准化是信息化社会发展的必然趋势，是世界各国保密通信领域的一个重要课题。按照发展进程来分，数据加密算法经历了古典密码、对称密钥密码和公开密钥密码阶段。古典密码算法有替代加密、置换加密；对称加密算法包括 DES 和 AES；非对称加密算法包括 RSA、背包密码、McEliece 密码、Rabin 及椭圆曲线等。目前在数据通信中使用最普遍的算法有 DES 算法、RSA 算法和 PGP 算法等。

3.3.1 古典密码算法

1. 凯撒密码

凯撒密码（Caesar Shifts，Simple Shift）也称凯撒移位，是最简单的加密方法之一，相传是古罗马凯撒大帝用来保护重要军情的加密系统，它是一种替代密码。

- 加密公式：密文＝（明文＋位移数）Mod 26。

• 解密公式：明文＝(密文－位移数) Mod 26。

以《数字城堡》中的一组密码“HL FKZC VD LDS”为例，只需把每个字母都按字母表中的顺序依次后移一个字母即可，即 A 变成 B，B 就成了 C，依此类推。因此明文为“IM GLAD WE MET”。

英文字母的移位以移 25 位为一个循环，移 26 位等于没有移位。所以可以用穷举法列出所有可能的组合。例如“phhw ph diwhu wkh wrjd sduwb”可以方便地列出所有组合，然后从中选出有意义的话，可知明文为“meet me after the toga party”。

2. 频率分析法

频率分析法可以有效破解单字母替换密码。

关于词频问题的密码，英文字母的出现频率如表 3-1 所示。

表 3-1 英文字母的出现频率

字母	频率/%	字母	频率/%	字母	频率/%	字母	频率/%
a	8.2	b	1.5	c	2.8	d	4.3
e	12.7	f	2.2	g	2.0	h	6.1
i	7.0	j	0.2	k	0.8	l	4.0
m	2.4	n	6.7	o	7.5	p	1.9
q	0.1	r	6.0	s	6.3	t	9.1
u	2.8	v	1.0	w	2.4	x	0.2
y	2.0	z	0.1				

词频法其实就是计算各个字母在文章中的出现频率，然后大概猜测出明码表，最后验证自己的推算是否正确。这种方法由于要统计字母的出现频率，需要花费时间较长。

3. 维吉尼亚密码

由于频率分析法可以有效破解单表替换密码，法国密码学家维吉尼亚于 1586 年提出一种多表替换密码，即维吉尼亚密码，也称维热纳尔密码。维吉尼亚密码引入了“密钥”的概念，即根据密钥来决定用哪一行的密表来进行替换，以此来对抗字频统计。

- 加密算法：例如密钥的字母为[d]，明文对应字母[b]。根据字母表的顺序[d]＝4，[b]＝2，那么密文就是[d]＋[b]-1＝4＋2-1＝5＝[e]，因此加密的结果为[e]。解密即做逆运算。
- 加密公式：密文 ＝(明文＋密钥) Mod 26－1。
- 解密公式：明文＝[26＋(密文－密钥)] Mod 26＋1。

假如对明文“to be or not to be that is the question”加密，当选定“have”作为密钥时，加密过程是：密钥第一个字母为[h]，明文第一个为[t]，因此可以找到在 h 行 t 列中的字母[a]，依此类推，得出对应关系如下。

密钥：ha ve ha veh av eh aveh av eha vehaveha

明文：to be or not to be that is the question

密文：ao wi vr isa tj fl tcea in xoe lylsomvn

4. 随机乱序字母

随机乱序字母即单字母替换密码。重排密码表 26 个字母的顺序，密码表会增加到四千亿亿亿多种，能有效防止用筛选的方法检验所有密码表。这种密码持续使用了几个世纪，直到阿拉伯人发明了频率分析法。

- 明码表：A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- 密码表：Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

例如明文为 forest，则密文为 gbmrst。

3.3.2 现代密码体制

古典密码学中提出的加密方案是一种算法保护的方案，在保密方案安全的情况下，可能收到一定的安全效果，但是，随着保密方案的泄露，被加密信息的安全就没有了安全保证。

除了算法的复杂度，现代密码学与古典密码学比较显著的不同之处在于，相对于古典密码学加解密流程（见图 3-1），现代密码学加解密流程中在加密端和解密端分别多了加密密钥和解密密钥，如图 3-2 所示。这样，对于加密的明文信息的保护转变为对密钥信息的保护，从而提高了加密的安全性和加密算法的生命力。

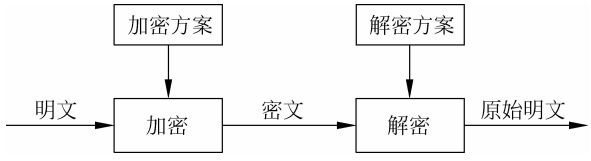

图 3-1 古典密码学加解密流程

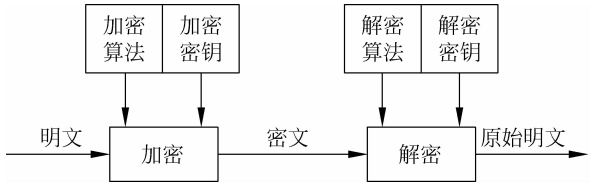

图 3-2 现代密码学加解密流程

现代密码学思想的核心内容是密码体系的强度不能依赖于对算法内部机制的保密。因为当密码内部机制被有意或无意泄露以后，一个强度高的密码体制还能依靠它的密钥来维持它的安全性，而一个强度依赖算法内部机理的体制，其安全问题难以保证。这是荷兰密码学家 A. Kerckhoffs（1835—1903）最早阐述的原则，即密码的安全必须完全寓于密钥之中。

根据密钥类型不同，现代密码体制可分为两类，一类是对称密码体制；另一类是非对称

密码体制。对称密码体制是加密和解密均采用同一把密钥，算法实现速度极快，因此有着广泛的应用，最著名的是美国数据加密标准 DES、AES（高级加密标准）和欧洲数据加密标准 IDEA。

非对称密码体制采用的加密钥匙（公钥）和解密钥匙（私钥）是不同的。该体制的安全性都基于复杂的数学难题。RSA 系统是最典型的方法，原理简单且易于使用。DSA（Data Signature Algorition）是基于离散对数问题的数字签名标准，它仅提供数字签名，不提供数据加密功能。另外还有安全性高、算法实现性能好的椭圆曲线加密算法 ECC（Elliptic Curve Cryptography）等。

在实际应用中，非对称密码体制并没有完全取代对称密码体制，这是因为非对称密码体制基于尖端的数学难题，计算非常复杂，安全性高，但实现速度却远远赶不上对称密码体制。因而非对称密码体制通常用来加密关键性的、核心的机密数据，而对称密码体制通常用来加密大量的数据。对于具有大存储量的图像来说，结合对称密码体制的加密方案将是科学的、实用的。

3.3.3 DES 算法

DES 即数据加密标准，最初是 IBM 的 W. Tuchman 和 C. Meyers 等人提出的一个数据加密算法 Lucifef，它于 1976 年被美国国家标准局正式用于商业和政府非要害信息的加密。DES 是典型的加解密钥相同的对称密码体制，其优势在于加解密速度快，算法易实现，安全性好。目前在我国国内，随着三金工程（尤其是金卡工程）的启动，DES 算法在 PQS、ATM、磁卡及智能卡（IC 卡）、加油站、高速公路收费站等领域被广泛应用，以此来实现关键数据的保密，如信用卡持卡人的 PIN 加密传输、IC 卡与 PQS 间的双向认证、金融交易数据包的 MAC 校验等，均用到 DES 算法。

DES 是分组加密算法，它以 64 位（二进制）为一组对数据加密，64 位明文输入，64 位密文输出。密钥长度为 56 位，但密钥通常表示为 64 位，并分为 8 组每组第 8 位作为奇偶校验位，以确保密钥的正确性。

1. 基于 DES 算法的数字图像加密

将 DES 算法用于数字图像加密，可以考虑将图像色彩的二维数据转化为一维数据，对一维数据按 64 位为一组进行分组加密。

2. DES 算法概要

（1）对输入的明文从右向左按顺序每 64 位分为一组（不足 64 位时在高位补 0），并按组进行加密或解密。

（2）进行初始置换。

（3）将置换后的明文分成左右两组，每组 32 位。

（4）进行 16 轮相同的变换，包括密钥变换，每轮变换如图 3-3 所示。

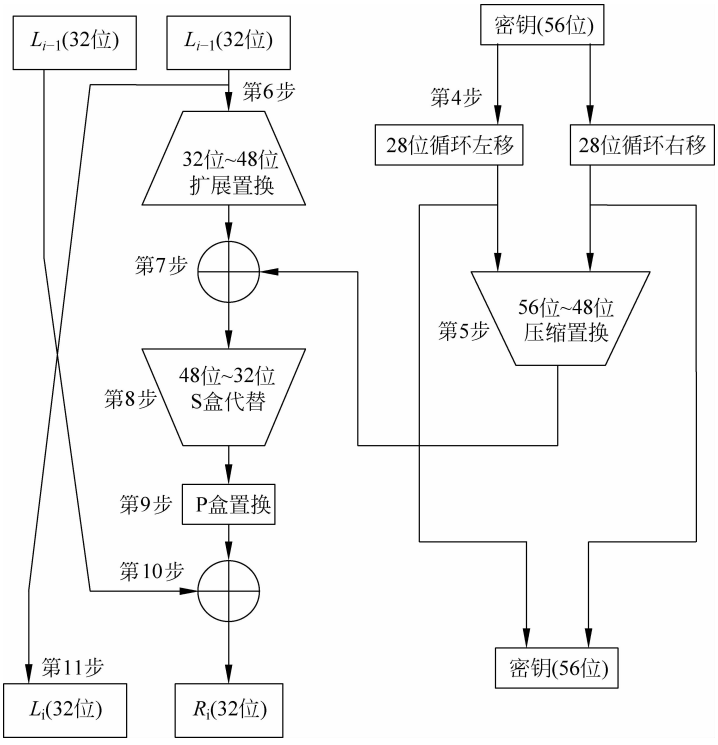

图 3-3 一轮 DES 变换

3. DES 算法加密过程

1）初始置换

初始置换就是按照矩阵 3-1 的规则对输入的 64 位二进制明文 P＝P1P2…P64 改变顺序，矩阵中的数字代表明文在 64 位二进制序列中的位置。

矩阵 3-1 初始置换

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

初始置换矩阵有 8 行 8 列，共 64 个元素，其元素的排列是有规律的，可以把上面 4 行和下面 4 行分成 2 组，取名为 L 和 R。

2）明文分组

将置换后的明文，即新的 64 位二进制序列，按顺序分为左、右两组，每组都是 32 位。

3）密钥置换

密钥置换就是按矩阵 3-2 的规则改变密钥的顺序。

矩阵 3-2 密钥置换

$$\begin{bmatrix} 57 & 49 & 41 & 33 & 25 & 17 & 9 \\ 1 & 58 & 50 & 42 & 34 & 26 & 18 \\ 10 & 2 & 59 & 51 & 43 & 35 & 27 \\ 19 & 11 & 3 & 60 & 52 & 44 & 36 \\ 63 & 55 & 47 & 39 & 31 & 23 & 15 \\ 7 & 62 & 54 & 46 & 38 & 20 & 22 \\ 14 & 6 & 54 & 46 & 45 & 37 & 29 \\ 21 & 13 & 5 & 28 & 20 & 12 & 4 \end{bmatrix}$$

密钥置换矩阵有 8 行 7 列，共 56 个元素。其元素的排列是有规律的，可以把上面 4 行和下面 4 行分为 2 组，取名为 KL、KR，由于取消了原 64 位密钥中的奇偶校验位，所以密钥置换矩阵 3-2 中不会出现 8、16、24、32、40、48、56、64 这些数值。

4）密钥分组、移位、合并

将置换后的 56 位密钥按顺序分成左右两个部分 KL、KR，每部分 27 位，根据 DES 算法轮数（迭代次数），分别将两个部分 KL、KR 循环左移 1 位或 2 位，每轮循环左移位数按照密钥移位个数而定，如表 3-2 所示。

表 3-2 每轮密钥循环左移位数

迭代次数	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
右移位数	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

例如，KL0＝c1c2…c28，KR0＝d1d2…d28，由于是第 1 次迭代，循环左移位数是 1，所以，KL0＝c2c3…c28c1，KR0＝d2d3…d28d1；KLKR 两组密钥循环左移后，再合并成 56 位密钥，例如 K1＝c2c3…c28c1d2d3…d28d1，合并后 56 位密钥一方面用于产生子密钥，另一方面为下次迭代运算做准备。

5）压缩置换

按照密钥压缩置换矩阵 3-3，从 56 位密钥中产生 48 位子密钥。密钥压缩置换矩阵中共有 48 位元素，其中看不到 9、18、22、25、35、38、43、54 这 8 个元素，因为这些元素已被压缩了。

矩阵 3-3 密钥压缩置换

$$\begin{bmatrix} 14 & 17 & 11 & 24 & 1 & 5 \\ 3 & 28 & 15 & 6 & 21 & 10 \\ 23 & 19 & 12 & 4 & 26 & 8 \\ 16 & 7 & 27 & 20 & 13 & 2 \\ 41 & 52 & 31 & 37 & 47 & 55 \\ 30 & 40 & 51 & 45 & 33 & 48 \\ 44 & 49 & 39 & 56 & 34 & 53 \\ 46 & 42 & 50 & 36 & 29 & 32 \end{bmatrix}$$

6）扩展置换

将原明文数据的右半部分 R 从 32 位扩展成 48 位，扩展置换按照扩展置换矩阵 3-4 规则进行。

矩阵 3-4 扩展置换

$$\begin{bmatrix} 32 & 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 6 & 7 & 8 & 9 \\ 8 & 9 & 10 & 11 & 12 & 13 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 16 & 17 & 18 & 19 & 20 & 21 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 24 & 25 & 26 & 27 & 28 & 29 \\ 28 & 29 & 30 & 31 & 32 & 1 \end{bmatrix}$$

7）子密钥和扩展置换后的数据异或运算

将子密钥和扩展置换后的数据按位进行异或运算，然后，将得到的 48 位结果送到 S 盒替换。

8）S 盒替换

将 48 位数据按顺序每 6 位分为一组，共分成 8 组，分别输入 S1、S2…S8 盒中，每个 S 盒的输出为 4 位，再将每个 S 盒的输出拼接成 32 位，S 盒，结构如图 3-4 所示。

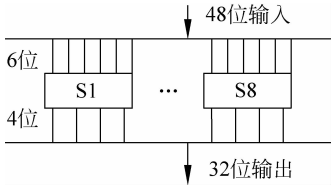

图 3-4 S 盒

DES 的 S 盒的使用方法是，设 S 盒的输入为 6 位二进制数 b1b2b3b4b5b6，把 b1、b6 两位二进制数转换成十进制，并作为 S 盒的行号 i，把 b2、b3、b4、b5 这 4 位二进制数转换成十进制数，并作为 S 盒的列号 j，则对应 S 盒的 (i,j) 元素就为 S 盒的十进制输出，再将该十进制数转换为二进制数，就得到 S 盒的 4 位二进制输出。

S 盒替换是 DES 的核心部分，整个变换过程是非线性的（而 DES 算法的其他变换都是线性的），提供了很好的混乱数据效果，比 DES 算法其他步骤提供的安全性更好。

9）P 盒置换

将 S 盒输出的 32 位二进制数据按 P 盒置换矩阵 3-5 进行置换。

矩阵 3-5 P 盒置换矩阵

$$\begin{bmatrix} 16 & 7 & 20 & 21 & 29 & 12 & 28 & 17 \\ 1 & 15 & 23 & 26 & 5 & 28 & 31 & 10 \\ 2 & 8 & 24 & 14 & 32 & 27 & 3 & 9 \\ 19 & 13 & 30 & 6 & 22 & 11 & 4 & 25 \end{bmatrix}$$

例如将 S 盒输出的第 16 位变换成第 1 位，S 盒输出的第 1 位变换成第 9 位。

10）P 盒输出与原 64 位数据进行异或运算

将 P 盒输出的 32 位二进制与原 64 位数据分组的左半部分 Li 进行异或运算，得到分组的右半部分 Ri。

11）Ri-1-Li

将原分组的右半部分 R$i-1$ 作为分组的左半部分 Li。

12) 循环

重复 4)～11)步,循环操作 16 轮。

13) 逆初始置换

经过 16 轮的 DES 运算后,将输出的 L16、R16 合并起来,形成 64 位的二进制数,最后按照逆初始置换矩阵 3-6 进行逆初始置换,就可以得到密文。

矩阵 3-6　逆初始置换

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

4. DES 算法解密过程

DES 算法加密和解密过程使用相同的算法,并使用相同的加密密钥和解密密钥,两者的区别如下。

(1) DES 加密时是从 L0、R0 到 L15、R15 进行变换,而解密时是从 L15、R15 到 L0、R0 进行变换的。

(2) 加密时各轮的加密密钥为 K0、K1、…、K15,而解密时各轮的解密密钥为 K15、K14、…、K0。

(3) 加密时密钥循环左移,而解密时循环右移。

5. 三重 DES 算法

三重 DES 加密的基本方法是用两个密钥对一个分组进行三次加密,即加密时,先用第一个密钥加密,然后用第二个密钥解密,最后再用第三个密钥加密。解密时,先用第一个密钥解密,然后用第二个密钥加密,最后再用第三个密钥解密,过程示意如图 3-5 所示。

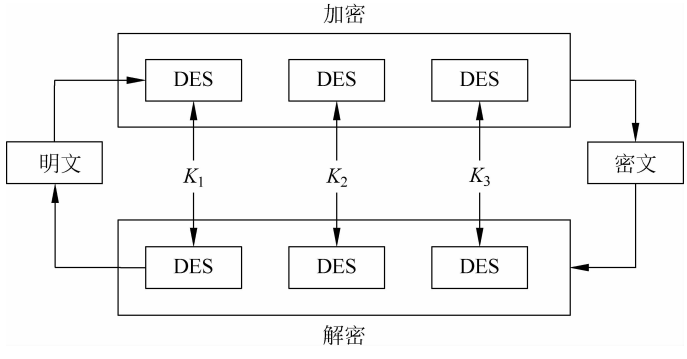

图 3-5　三重 DES 算法

3.3.4 RSA 公开密钥密码体制

RSA 是第一个比较完善的公开密钥算法，它既能用于加密，也能用于数字签名。RSA 以它的 3 个发明者 Ron Rivest、Adi Shamir、Leonard Adleman 的名字首字母命名，

根据数论寻求两个大素数比较简单，而把两个大素数的乘积分解则极其困难。在这一体制中，每个用户有两个密钥，即加密密钥 pk=$\{e,n\}$和解密密钥 sk=$\{d,n\}$。用户把加密密钥公开，使得任何其他用户都可以使用；而对解密密钥中的 d 则保密。

这里，n 为两个大素数 p 和 q 的乘积(素数 p 和 q 一般为 100 位以上的十进制数)。e 和 d 满足一定的关系，当敌手已知 e 和 n 时并不能求出 d。

1. 加密算法

若用整数 X 表示明文，用整数 Y 表示密文(X 和 Y 均小于 n)，则加密和解密的公式如表 3-3 所示。

表 3-3 加密和解密公式

公钥 KU	n：两素数 p 和 q 的乘积(p 和 q 必须保密)； e：与$(p-1)(q-1)$互质
私钥 KR	d：$e^{-1}(\mathrm{mod}\ (p-1)(q-1))$
加密	$C\equiv m^e \bmod n$
解密	$m\equiv c^d \bmod n$

2. 密钥的产生

(1) 选择一对不同的、足够大的素数 p 和 q，计算 n。

用户秘密地选择两个大素数 p 和 q，计算出 $n=pq$，n 称为 RSA 算法的模数，明文必须用小于 n 的数来表示，实际上 n 是几百比特长的数。加密消息 m 时，首先将它分成比 n 小的数据分组(采用十六进制数，选取小于 n 的 16 的最大次幂)，也就是说，如果 p 和 q 为 100 位的素数，那么 n 将有 200 位，每个消息分组 m_i 应小于 200 位长(如果需要加密固定的消息分组，那么可以在它的左边填充一些 0 并确保该数比 n 小)。加密后的密文 c，将由相同长度的分组 c_i 组成。加密公式简化如下：

$$c_i = m_i^e(\mathrm{mod}\ n) \tag{3-1}$$

解密时，取每一个加密后的分组 c_i 并计算 m_i，公式如下：

$$m_i = c_i^d(\mathrm{mod}\ n) \tag{3-2}$$

(2) 计算$\not\subset(n)$，即再计算出 n 的欧拉函数，如公式(3-3)：

$$\not\subset(n) = (p-1)*(q-1).\ \not\subset(n) \tag{3-3}$$

(3) 选择 e。从[0，$\not\subset(n)-1$]中选择一个与$\not\subset(n)$互素的数 e 作为公开的加密指数。

(4) 计算 d，计算出满足公式(3-4)的 d：$ed=1 \bmod \not\subset(n)$ 作为解密指数。

$$d = e^{-1} \bmod ((p-1)(q-1)) \tag{3-4}$$

(5) 得出所需要的公开密钥和私有密钥,如下。

- 公开密钥 (即加密密钥): $pk=\{e,d\}$。
- 私有密钥 (即解密密钥): $sk=\{d,n\}$。

【实例 3-1】 假设用户 A 需要将明文 key 通过 RSA 加密后传递给用户 B,过程如下。

(1) 设计公私密钥(e,n)和(d,n)。

令 $p=3,q=11$,得出 $n=p\times q=3\times 11=33$; $f(n)=(p-1)(q-1)=2\times 10=20$; 取 $e=3$,(3 与 20 互质)则 $e\times d\equiv 1 \bmod f(n)$,即 $3\times d\equiv 1 \bmod 20$。$d$ 取值可以用试算的办法来寻找,试算结果如表 3-4 所示。

表 3-4 计算结果

d	$e\times d=3\times d$	$(e\times d) \bmod (p-1)(q-1)=(3\times d) \bmod 20$
1	3	3
2	6	6
3	9	9
4	12	12
5	15	15
6	18	18
7	21	1
8	24	3
9	27	6

通过试算找到,当 $d=7$ 时,$e\times d\equiv 1 \bmod f(n)$同余等式成立。因此,可令 $d=7$。从而可以设计出一对公私密钥,加密密钥(公钥)为 KU $=(e,n)=(3,33)$,解密密钥(私钥)为 KR $=(d,n)=(7,33)$。

(2) 英文数字化。

将明文信息数字化,并将每块两个数字分组。假定明文英文字母编码表为按字母顺序排列数值,如表 3-5 所示,则得到分组后的 key 的明文信息为 11、05、25。

表 3-5 英文字母编码表

字母	a	b	c	d	e	f	g	h	i	j	k	l	m
码值	01	02	03	04	05	06	07	08	09	10	11	12	13
字母	n	o	p	q	r	s	t	u	v	w	x	y	z
码值	14	15	16	17	18	19	20	21	22	23	24	25	26

(3) 明文加密。

用户利用加密密钥(3,33)将数字化明文分组信息加密成密文。由 $c\equiv m^e \pmod n$得 m_1、m_2 和 m_3 如下。

$$m_1\equiv(c_1)^d \pmod n=11^7 \pmod{33}=11$$

$$m_2\equiv(c_2)^d \pmod n=31^7 \pmod{33}=05$$

$$m_3\equiv(c_3)^d \pmod n=16^7 \pmod{33}=25$$

因此,可得到相应的密文信息为 11、05、25。

(4) 密文解密。

用户 B 收到密文,若将其解密,只需要计算 $m \equiv c^d \pmod{n}$,即有如下结果。

$$m_1 \equiv (c_1)^d \pmod{n} = 11^7 \pmod{33} = 11$$

$$m_2 \equiv (c_2)^d \pmod{n} = 31^7 \pmod{33} = 05$$

$$m_3 \equiv (c_3)^d \pmod{n} = 16^7 \pmod{33} = 25$$

可知用户 B 得到明文信息为 11,05,25。根据上面的编码表将其转换为英文,即可得到恢复后的原文“key”。

由于 RSA 算法的公钥私钥的长度(模长度)要到 1024 位甚至 2048 位才能保证安全,因此,p、q、e 的选取及公钥私钥的生成、加密解密模指数运算都有一定的计算程序,需要仰仗计算机高速完成。

3. RSA 的安全性

在 RSA 密码应用中,公钥 KU 是被公开的,即 e 和 n 的数值可以被第三方窃听者得到。破解 RSA 密码的问题就是从已知的 e 和 n 的数值(n 等于 pq)想法求出 d 的数值,这样就可以得到私钥来破解密文。从公式:$d \equiv e^{-1} \pmod{((p-1)(q-1))}$或 $de \equiv 1 \pmod{((p-1)(q-1))}$可以看出,密码破解的实质问题是从 pq 的数值求出$(p-1)$和$(q-1)$。换句话说,只要求出 p 和 q 的值,就能求出 d 的值,进而得到私钥。

当 p 和 q 是一个大素数的时候,从它们的积 pq 去分解因子 p 和 q,这是一个公认的数学难题。比如当 pq 大到 1024 位时,迄今为止还没有人能够利用任何计算工具完成分解因子的任务。因此,RSA 从提出到现在已近 20 年,经历了各种攻击的考验,逐渐为人们接受,普遍认为是目前最优秀的公钥方案之一。

然而,虽然 RSA 的安全性依赖于大数的因子分解,但并没有从理论上证明破译 RSA 的难度与大数分解难度等价。即 RSA 的重大缺陷是无法从理论上把握它的保密性能如何。

此外,RSA 的缺点还有两个方面,一是产生密钥很麻烦,受到素数产生技术的限制,因而难以做到一次一密。二是分组长度太大,为保证安全性,n 至少也要 600 位以上,使得运算代价很高,尤其是速度较慢,较对称密码算法慢几个数量级;且随着大数分解技术的发展,这个长度还在增加,不利于数据格式的标准化。因此,使用 RSA 只能加密少量数据,大量的数据加密还要靠对称密码算法。

3.3.5 AES 简介

NIST(National Institute of Standards and Technology)于 1999 年发布了一个新版本的 DES 标准,该标准指出 DES 仅能用于遗留的系统,同时,3DES 将取代 DES 成为新的标准。然而,3DES 的根本缺点在于用软件实现该算法的速度比较慢。3DES 中轮的数量三倍于 DES 中轮的数量,故其速度慢得多。另外,DES 和 3DES 的分组长度均为 64 位,就效率和安全性而言,分组长度应该更长。

2000 年 10 月,美国国家标准和技术协会宣布从 15 种候选算法中选取出 Rijndael 算法作为新的对称加密算法标准,称为 AES。AES 算法的加密、解密流程图如图 3-6 所示。

可见,在解密时,只需将所有操作的逆变换逆序进行,并逆序使用密钥编排方案即可。

而 AES 算法有其特殊性，即解密本质上和加密有相同的结构，因而存在“等价逆密码”。这个“等价逆密码”能通过原变换的一系列逆变换来实现 AES 算法的解答，这些逆变换按与 AES 算法加密相同的顺序进行。只是密钥扩展有所不同，即先应用原密钥扩展，再将 InvMixColumns 应用到除第 1 轮和最后一轮外的所有轮密钥上。此解密算法称为直接解密算法。在这个算法中，不仅步骤本身与加密不同，而且步骤出现的顺序也不相同。为了便于实现，通常将唯一的非线性步骤（SubBytes）放在轮变换的第 1 步。Rijndael 的结构使得有可能定义一个等价的解密算法，其中所使用的步骤次序与加密相同，只是将每一步改成它的逆，并改变密钥编排方案。

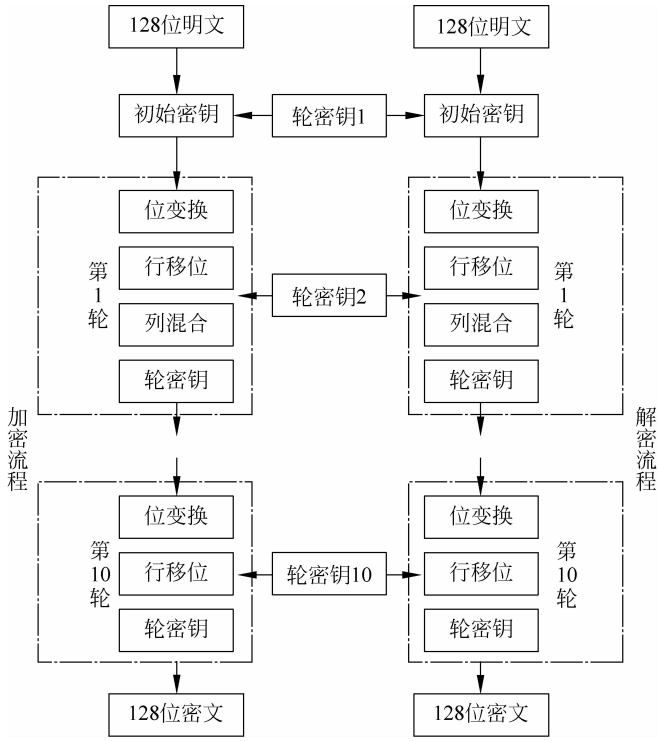

图 3-6　AES 算法加密解密流程图

3.3.6　MD5

MD5（Message-Digest Algorithm 5，信息-摘要算法）是由 MD2、MD3、MD4 发展而来的一种单向函数算法（也就是 Hash 算法），它是国际著名的公钥加密算法标准 RSA 的第一设计者 R. Rivest 于 20 世纪 90 年代初开发出来的。MD5 的最大作用在于将不同格式的大容量文件信息在用数字签名软件来签署私人密钥前“压缩”成一种保密的格式，关键之处在于这种“压缩”是不可逆的。

MD5 可以为任何文件（不管其大小、格式、数量）产生一个独一无二的“数字指纹”，如果任何人对文件做了任何改动，其 MD5 值也就是对应的“数字指纹”都会发生变化。

MD5 的典型应用是对一段 Message（字节串）产生 fingerprint（指纹），以防止被“篡改”。

比如对某个文件产生一个 MD5 的值并记录在案，然后传播这个文件，如果有人修改了文件中的任何内容，对这个文件重新计算 MD5 时就会发现两个 MD5 值不相同。如果再有一个第三方的认证机构，用 MD5 还可以防止文件作者的“抵赖”，这就是所谓的数字签名应用。

MD5 还广泛用于加密和解密技术，在很多操作系统中，用户的密码是以 MD5 值（或类似的其他算法）的方式保存的，用户登录的时候，系统是把用户输入的密码计算成 MD5 值，然后再去和系统中保存的 MD5 值进行比较，而系统并不“知道”用户的密码是什么。

根据密码学的定义，如果内容不同的明文通过散列算法得出的结果（密码学称为信息摘要）相同，就称为发生了“碰撞”。因为 MD5 值可以由任意长度的字符计算出来，所以可以把一篇文章或者一个软件的所有字节进行 MD5 运算得出一个数值，如果这篇文章或软件的数据改动了，那么再计算出的 MD5 值也会产生变化，这种方法常常用作数字签名校验。因为明文的长度可以大于 MD5 值的长度，所以可能会有多个明文具有相同的 MD5 值，如果找到了两个相同 MD5 值的明文，那么就是找到了 MD5 的“碰撞”。

散列算法的碰撞分为两种，即强无碰撞和弱无碰撞。已知某 MD5 值，假如能够找出某个单词，它的 MD5 值和已知的某 MD5 值一致，那么就找到了 MD5 的“弱无碰撞”，其实这就意味着已经破解了 MD5。如果不给指定的 MD5 值，随便去找任意两个相同 MD5 值的明文，即找强无碰撞，显然要相对容易些了，但对于好的散列算法来说，做到这一点也很不容易了。现在的电脑大约一两个小时就可以找到一对碰撞。找到强无碰撞在实际破解中没有什么真正的用途，所以现在 MD5 仍然是很安全的。

实现 MD5 算法主要经过以下 5 个步骤。

1. 补位

补位的目标是使输入的消息长度，从任意值变成一个新的长度 n，使得 $n=448(\mathrm{mod}\ 512)$，即通过补位使消息长度差 64 位成为 512 的整数倍，即使原消息的长度正好满足要求，也需要进行补位。补位的补丁包括一个 1，剩下的全是 0，在原消息之后。特别地，如果原消息的长度正好满足要求，则补位包括一个 1 和 512 个 0。

2. 追加长度

在追加长度前，通过补位，消息长度已经变成模 512 余 448，接下来的追加长度将在消息后继续补充 64 位的信息，新消息将是 512 的整数倍。追加长度的信息由 64 位表示，被追加到已补的信息后，如果原消息长度超过 64 位，只使用低 64 位。追加的长度是原消息的长度，而不是补位后的信息长度。

3. 缓冲区初始化

为了计算 Hash 函数的结果，需首先设置 128 位的缓冲区。缓冲区除接受 Hash 函数最终结果外，还记录中间结果。

缓冲区分成 4 等份，即 4 个 32 位寄存器（A，B，C，D），每个 32 位寄存器称为字，如图 3-7、图 3-8 所示。

赋初值 A：0x01234567、B：0x89abcdef、C：0xfedcba98、D：0x76543210，ABCD 即构成 Buffer 0。

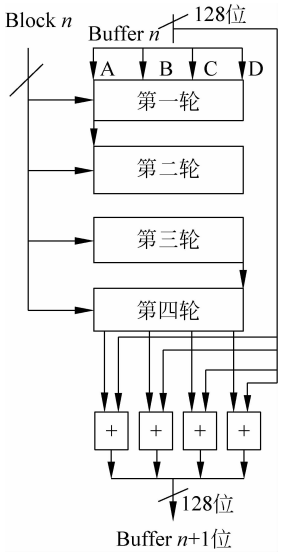

图 3-7 缓冲区 $n \rightarrow n+1$ 示意图

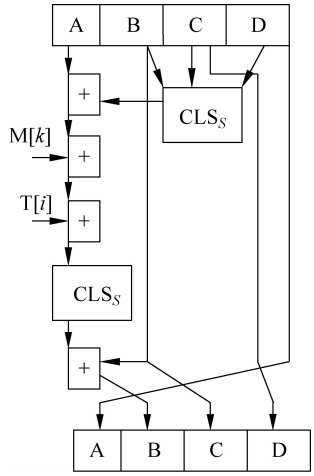

图 3-8 四轮算法

4. 消息迭代

从 Buffer 0 开始进行算法的主循环，循环的次数是消息中 512 位消息分组的数目，将上面 4 个变量复制到另外的变量中：A 到 a，B 到 b，C 到 c，D 到 d。主循环有 4 轮，每轮很相似，每一轮进行 16 次操作，每次操作对 a、b、c 和 d 中的 3 个作一次线性函数运算，然后将所得的结果加上第 4 个变量，文本的一个子分组和一个常数，再将所得的结果向右环移一个不定的数，最后用该结果取代 a、b、c 或 d 中之一。主循环的运算过程如图 3-9 所示。

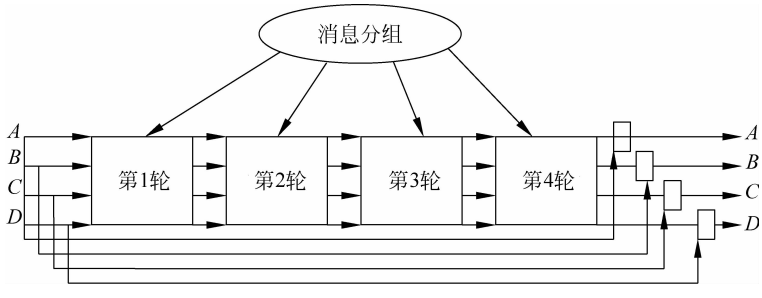

图 3-9 MD5 主循环

4 轮运算中有 4 种函数，分别为 $F(X,Y,Z)$、$G(X,Y,Z)$、$H(X,Y,Z)$ 和 $I(X,Y,Z)$。

```
F(X,Y,Z) = (X and Y) or (not (X) and Z)
G(X,Y,Z) = (X and Z) or (Y and not (Z))
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X or not(Z))
```

这些函数是这样设计的：如果 X、Y 和 Z 的对应位是独立和均匀的，那么结果的每一位也应是独立和均匀的。函数 F 是按逐位方式操作，如果 X，那么 Y，否则 Z。函数 H 是逐

位奇偶操作符。

设 Mj 表示消息的第 j 个子分组(从 0 到 15),<<<s 表示循环左移 s 位,则操作表达式如下所述。

```
FF(a,b,c,d,Mj,s,ti)表示 a = b + ((a + (F(b,c,d) + Mj + ti)<<< s)
GG(a,b,c,d,Mj,s,ti)表示 a = b + ((a + (G(b,c,d) + Mj + ti)<<< s)
HH(a,b,c,d,Mj,s,ti)表示 a = b + ((a + (H(b,c,d) + Mj + ti)<<< s)
II(a,b,c,d,Mj,s,ti)表示 a = b + ((a + (I(b,c,d) + Mj + ti)<<< s)
```

这四轮(64 步)迭代过程分别如下所述。

第 1 轮:

```
FF (a, b, c, d, M[ 0], 11, 0xd76aa478);
FF (d, a, b, c, M[ 1], 12, 0xe8c7b756);
FF (c, d, a, b, M[ 2], 13, 0x242070db);
FF (b, c, d, a, M[ 3], 14, 0xc1bdceee);
FF (a, b, c, d, M[ 4], 11, 0xf57c0faf);
FF (d, a, b, c, M[ 5], 12, 0x4787c62a);
FF (c, d, a, b, M[ 6], 13, 0xa8304613);
FF (b, c, d, a, M[ 7], 14, 0xfd469501);
FF (a, b, c, d, M[ 8], 11, 0x698098d8);
FF (d, a, b, c, M[ 9], 12, 0x8b44f7af);
FF (c, d, a, b, M[10], 13, 0xffff5bb1);
FF (b, c, d, a, M[11], 14, 0x895cd7be);
FF (a, b, c, d, M[12], 11, 0x6b901122);
FF (d, a, b, c, M[13], 12, 0xfd987193);
FF (c, d, a, b, M[14], 13, 0xa679438e);
FF (b, c, d, a, M[15], 14, 0x49b40821);
```

第 2 轮:

```
GG (a, b, c, d, M[ 1], 21, 0xf61e2562);
GG (d, a, b, c, M[ 6], 22, 0xc040b340);
GG (c, d, a, b, M[11], 23, 0x265e5a51);
GG (b, c, d, a, M[ 0], 24, 0xe9b6c7aa);
GG (a, b, c, d, M[ 5], 21, 0xd62f105d);
GG (d, a, b, c, M[10], 22,  0x2441453);
GG (c, d, a, b, M[15], 23, 0xd8a1e681);
GG (b, c, d, a, M[ 4], 24, 0xe7d3fbc8);
GG (a, b, c, d, M[ 9], 21, 0x21e1cde6);
GG (d, a, b, c, M[14], 22, 0xc33707d6);
GG (c, d, a, b, M[ 3], 23, 0xf4d50d87);
```

```
GG (b, c, d, a, M[ 8], 24, 0x455a14ed);
GG (a, b, c, d, M[13], 21, 0xa9e3e905);
GG (d, a, b, c, M[ 2], 22, 0xfcefa3f8);
GG (c, d, a, b, M[ 7], 23, 0x676f02d9);
GG (b, c, d, a, M[12], 24, 0x8d2a4c8a);
```

第 3 轮：

```
HH (a, b, c, d, M[ 5], 31, 0xfffa3942);
HH (d, a, b, c, M[ 8], 32, 0x8771f681);
HH (c, d, a, b, M[11], 33, 0x6d9d6122);
HH (b, c, d, a, M[14], 34, 0xfde5380c);
HH (a, b, c, d, M[ 1], 31, 0xa4beea44);
HH (d, a, b, c, M[ 4], 32, 0x4bdecfa9);
HH (c, d, a, b, M[ 7], 33, 0xf6bb4b60);
HH (b, c, d, a, M[10], 34, 0xbebfbc70);
HH (a, b, c, d, M[13], 31, 0x289b7ec6);
HH (d, a, b, c, M[ 0], 32, 0xeaa127fa);
HH (c, d, a, b, M[ 3], 33, 0xd4ef3085);
HH (b, c, d, a, M[ 6], 34, 0xe4881d05);
HH (a, b, c, d, M[ 9], 31, 0xd9d4d039);
HH (d, a, b, c, M[12], 32, 0xe6db99e5);
HH (c, d, a, b, M[15], 33, 0x1fa27cf8);
HH (b, c, d, a, M[ 2], 34, 0xc4ac5665);
```

第 4 轮：

```
II (a, b, c, d, M[ 0], 41, 0xf4292244);
II (d, a, b, c, M[ 7], 42, 0x432aff97);
II (c, d, a, b, M[14], 43, 0xab9423a7);
II (b, c, d, a, M[ 5], 44, 0xfc93a039);
II (a, b, c, d, M[12], 41, 0x655b59c3);
II (d, a, b, c, M[ 3], 42, 0x8f0ccc92);
II (c, d, a, b, M[10], 43, 0xffeff47d);
II (b, c, d, a, M[ 1], 44, 0x85845dd1);
II (a, b, c, d, M[ 8], 41, 0x6fa87e4f);
II (d, a, b, c, M[15], 42, 0xfe2ce6e0);
II (c, d, a, b, M[ 6], 43, 0xa3014314);
II (b, c, d, a, M[13], 44, 0x4e0811a1);
```

```
II (a, b, c, d, M[ 4], 41, 0xf7537e82);
II (d, a, b, c, M[11], 42, 0xbd3af235);
II (c, d, a, b, M[ 2], 43, 0x2ad7d2bb);
II (b, c, d, a, M[ 9], 44, 0xeb86d391);
```

5. 输出结果

最后将 a、b、c 和 d 还原为 A、B、C、D，再将 ABCD 组合起来，就构成原消息的摘要。

用 C 语言实现 MD5 加密算法，md5.h 代码如下。

```
/*                  md5.h            */
#ifndef _MD5_H_
#define _MD5_H_
#define R_memset(x, y, z) memset(x, y, z)
#define R_memcpy(x, y, z) memcpy(x, y, z)
#define R_memcmp(x, y, z) memcmp(x, y, z)
typedef unsigned long UINT4;
typedef unsigned char * POINTER;
/* MD5 context. */
typedef struct {
  /* state (ABCD) */
  /* 四个 32bits 数,用于存放最终计算得到的消息摘要.当消息长度> 512bits 时,也用于存放每个
512bits 的中间结果 */
  UINT4 state[4];
  /* number of bits, modulo 2 ^64 (lsb first) */
  /* 存储原始信息的 bits 数长度,不包括填充的 bits,最长为 2 ^64 bits,因为 2 ^64 是一个 64 位
数的最大值 */
  UINT4 count[2];
  /* input buffer */
  /* 存放输入的信息的缓冲区,512bits */
  unsigned char buffer[64];
} MD5_CTX;
void MD5Init(MD5_CTX *);
void MD5Update(MD5_CTX *, unsigned char *, unsigned int);
void MD5Final(unsigned char [16], MD5_CTX *);
#endif /* _MD5_H_ */
```

md5.cpp 代码及说明如下。

```
/*     md5.cpp    */
#include "stdafx.h"
/* Constants for MD5Transform routine. */
/* MD5 转换用到的常量,算法本身规定的 */
#define S11 7
#define S12 12
#define S13 17
#define S14 22
#define S21 5
#define S22 9
```

```
#define S23 14
#define S24 20
#define S31 4
#define S32 11
#define S33 16
#define S34 23
#define S41 6
#define S42 10
#define S43 15
#define S44 21
static void MD5Transform(UINT4 [4], unsigned char [64]);
static void Encode(unsigned char *, UINT4 *, unsigned int);
static void Decode(UINT4 *, unsigned char *, unsigned int);
/*用于 bits 填充的缓冲区,为什么要 64 个字节呢?因为当欲加密的信息的 bits 数被 512 除余数为
448 时,需要填充的 bits 数的最大值为 512 = 64 * 8 */
static unsigned char PADDING[64] = {
  0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
/*接下来的这几个宏定义是 MD5 算法规定的,是对信息进行 MD5 加密都要做的运算。*/
/* F, G, H and I are basic MD5 functions. */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^(y) ^(z))
#define I(x, y, z) ((y) ^((x) | (~z)))
/* ROTATE_LEFT rotates x left n bits. */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32 - (n))))
/* FF, GG, HH, and II transformations for rounds 1, 2, 3, and4.
  Rotation is separate from addition to prevent recomputation. */
#define FF(a, b, c, d, x, s, ac) { \
  (a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
}
#define GG(a, b, c, d, x, s, ac) { \
  (a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
}
#define HH(a, b, c, d, x, s, ac) { \
  (a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
}
#define II(a, b, c, d, x, s, ac) { \
  (a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
}
```

```
/* MD5 initialization. Begins an MD5 operation, writing a new context. */
/*初始化 MD5 的结构*/
void MD5Init (MD5_CTX *context)
{
  /*将当前的有效信息的长度设成 0*/
  context->count[0] = context->count[1] = 0;
  /* Load magic initialization constants.*/
  /*初始化链接变量*/
  context->state[0] = 0x67452301;
  context->state[1] = 0xefcdab89;
  context->state[2] = 0x98badcfe;
  context->state[3] = 0x10325476;
}
/* MD5 block update operation. Continues an MD5 message-digest operation, processing another
message block, and updating the context. */
/*将与加密的信息传递给 MD5 结构,可以多次调用。
context: 初始化过了的 MD5 结构。
input: 欲加密的信息,可以任意长。
inputLen: 指定 input 的长度*/
void MD5Update(MD5_CTX *context,unsigned char * input,unsigned int  inputLen)
{
unsigned int i, index, partLen;
/* Compute number of bytes mod 64 */
/*计算已有信息的 bits 长度的字节数的模 64, 64bytes = 512bits。用于判断已有信息加上当前传
过来的信息的总长度能不能达到 512bits,如果能够达到则对凑够的 512bits 进行一次处理*/
index = (unsigned int)((context->count[0] >> 3) & 0x3F);
/* Update number of bits *//*更新已有信息的 bits 长度*/
if((context->count[0] += ((UINT4)inputLen << 3)) < ((UINT4)inputLen << 3))
  context->count[1]++;
context->count[1] += ((UINT4)inputLen >> 29);
/*计算已有的字节数长度还差多少字节可以凑成 64 的整倍数*/
partLen = 64 - index;
/* Transform as many times as possible. */
/*如果当前输入的字节数大于已有字节数长度补足 64 字节整倍数所差的字节数*/
if(inputLen >= partLen)
    {
  /*用当前输入的内容把 context->buffer 的内容补足 512bits*/
  R_memcpy((POINTER)&context->buffer[index], (POINTER)input, partLen);
  /*用基本函数对填充满的 512bits(已经保存到 context->buffer 中)做一次转换,转换结果保
存到 context->state 中*/
  MD5Transform(context->state, context->buffer);
/* 对当前输入的剩余字节做转换(如果剩余的字节在输入的 input 缓冲区中大于 512bits),转换结
果保存到 context->state 中 */
  for(i = partLen; i + 63 < inputLen; i += 64)/*把 i+63<inputLen 改为 i+64<=inputLen
更容易理解*/
   MD5Transform(context->state, &input[i]);
        index = 0;
    }
    else
  i = 0;
/* Buffer remaining input */
```

```
/* 将输入缓冲区中的不足填充满 512bits 的剩余内容填充到 context->buffer 中,留待以后再作
处理 */
R_memcpy((POINTER)&context->buffer[index], (POINTER)&input[i], inputLen-i);
}
/* MD5 finalization. Ends an MD5 message-digest operation, writing the
  the message digest and zeroizing the context. */
/* 获取加密的最终结果
digest: 保存最终的加密串
context: 前面初始化并填入了信息的 MD5 结构 */
void MD5Final (unsigned char digest[16],MD5_CTX *context)
{
unsigned char bits[8];
unsigned int index, padLen;
/* Save number of bits */
/* 将要被转换的信息(所有的)的 bits 长度复制到 bits 中 */
Encode(bits, context->count, 8);
/* Pad out to 56 mod 64. */
/* 计算所有 bits 长度的字节数的模 64, 64bytes = 512bits */
index = (unsigned int)((context->count[0] >> 3) & 0x3f);
/* 计算需要填充的字节数,padLen 的取值范围在 1～64 之间 */
padLen = (index < 56) ? (56 - index) : (120 - index);
/* 这一次函数调用绝对不会再导致 MD5Transform 的被调用,因为这一次不会填满 512bits */
MD5Update(context, PADDING, padLen);
/* Append length (before padding) */
/* 补上原始信息的 bits 长度(bits 长度固定的用 64bits 表示) */
MD5Update(context, bits, 8);
/* Store state in digest */
/* 将最终的结果保存到 digest 中。 */
Encode(digest, context->state, 16);
/* Zeroize sensitive information. */
R_memset((POINTER)context, 0, sizeof(*context));
}
/* MD5 basic transformation. Transforms state based on block. */
/*
对 512bits 信息(即 block 缓冲区)进行一次处理,每次处理包括四轮。
state[4]: MD5 结构中的 state[4]用于保存对 512bits 信息加密的中间结果或者最终结果;
block[64]: 欲加密的 512bits 信息
*/
static void MD5Transform (UINT4 state[4], unsigned char block[64])
{
UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];
Decode(x, block, 64);
/* Round 1 */
FF(a, b, c, d, x[ 0], S11, 0xd76aa478); /* 1 */
FF(d, a, b, c, x[ 1], S12, 0xe8c7b756); /* 2 */
FF(c, d, a, b, x[ 2], S13, 0x242070db); /* 3 */
FF(b, c, d, a, x[ 3], S14, 0xc1bdceee); /* 4 */
FF(a, b, c, d, x[ 4], S11, 0xf57c0faf); /* 5 */
FF(d, a, b, c, x[ 5], S12, 0x4787c62a); /* 6 */
FF(c, d, a, b, x[ 6], S13, 0xa8304613); /* 7 */
FF(b, c, d, a, x[ 7], S14, 0xfd469501); /* 8 */
```

```
FF(a, b, c, d, x[ 8], S11, 0x698098d8); /* 9 */
FF(d, a, b, c, x[ 9], S12, 0x8b44f7af); /* 10 */
FF(c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
FF(b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
FF(a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
FF(d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
FF(c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
FF(b, c, d, a, x[15], S14, 0x49b40821); /* 16 */
/* Round 2 */
GG(a, b, c, d, x[ 1], S21, 0xf61e2562); /* 17 */
GG(d, a, b, c, x[ 6], S22, 0xc040b340); /* 18 */
GG(c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
GG(b, c, d, a, x[ 0], S24, 0xe9b6c7aa); /* 20 */
GG(a, b, c, d, x[ 5], S21, 0xd62f105d); /* 21 */
GG(d, a, b, c, x[10], S22,  0x2441453); /* 22 */
GG(c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
GG(b, c, d, a, x[ 4], S24, 0xe7d3fbc8); /* 24 */
GG(a, b, c, d, x[ 9], S21, 0x21e1cde6); /* 25 */
GG(d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
GG(c, d, a, b, x[ 3], S23, 0xf4d50d87); /* 27 */
GG(b, c, d, a, x[ 8], S24, 0x455a14ed); /* 28 */
GG(a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
GG(d, a, b, c, x[ 2], S22, 0xfcefa3f8); /* 30 */
GG(c, d, a, b, x[ 7], S23, 0x676f02d9); /* 31 */
GG(b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

/* Round 3 */
HH(a, b, c, d, x[ 5], S31, 0xfffa3942); /* 33 */
HH(d, a, b, c, x[ 8], S32, 0x8771f681); /* 34 */
HH(c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
HH(b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
HH(a, b, c, d, x[ 1], S31, 0xa4beea44); /* 37 */
HH(d, a, b, c, x[ 4], S32, 0x4bdecfa9); /* 38 */
HH(c, d, a, b, x[ 7], S33, 0xf6bb4b60); /* 39 */
HH(b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
HH(a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
HH(d, a, b, c, x[ 0], S32, 0xeaa127fa); /* 42 */
HH(c, d, a, b, x[ 3], S33, 0xd4ef3085); /* 43 */
HH(b, c, d, a, x[ 6], S34,  0x4881d05); /* 44 */
HH(a, b, c, d, x[ 9], S31, 0xd9d4d039); /* 45 */
HH(d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
HH(c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
HH(b, c, d, a, x[ 2], S34, 0xc4ac5665); /* 48 */
/* Round 4 */
II(a, b, c, d, x[ 0], S41, 0xf4292244); /* 49 */
II(d, a, b, c, x[ 7], S42, 0x432aff97); /* 50 */
II(c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
II(b, c, d, a, x[ 5], S44, 0xfc93a039); /* 52 */
II(a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
II(d, a, b, c, x[ 3], S42, 0x8f0ccc92); /* 54 */
II(c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
```

```
II(b, c, d, a, x[ 1], S44, 0x85845dd1); /* 56 */
II(a, b, c, d, x[ 8], S41, 0x6fa87e4f); /* 57 */
II(d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
II(c, d, a, b, x[ 6], S43, 0xa3014314); /* 59 */
II(b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
II(a, b, c, d, x[ 4], S41, 0xf7537e82); /* 61 */
II(d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
II(c, d, a, b, x[ 2], S43, 0x2ad7d2bb); /* 63 */
II(b, c, d, a, x[ 9], S44, 0xeb86d391); /* 64 */
state[0] += a;
state[1] += b;
state[2] += c;
state[3] += d;
/* Zeroize sensitive information. */
R_memset((POINTER)x, 0, sizeof(x));
}
/* Encodes input (UINT4) into output (unsigned char). Assumes len is
  a multiple of 4. */
/*将 4 字节的整数复制到字符形式的缓冲区中。
output: 用于输出的字符缓冲区。
input: 欲转换的四字节的整数形式的数组。
len: output 缓冲区的长度,要求是 4 的整数倍
*/
static void Encode(unsigned char *output, UINT4 *input,unsigned int  len)
{
unsigned int i, j;
for(i = 0, j = 0; j < len; i++, j += 4) {
  output[j] = (unsigned char)(input[i] & 0xff);
  output[j + 1] = (unsigned char)((input[i] >> 8) & 0xff);
  output[j + 2] = (unsigned char)((input[i] >> 16) & 0xff);
  output[j + 3] = (unsigned char)((input[i] >> 24) & 0xff);
}
}
/* Decodes input (unsigned char) into output (UINT4). Assumes len is
  a multiple of 4. */
/*与上面的函数正好相反,这一个把字符形式的缓冲区中的数据复制到 4 字节的整数中(即以整数
形式保存)。
output: 保存转换出的整数。
input: 欲转换的字符缓冲区。
len: 输入的字符缓冲区的长度,要求是 4 的整数倍
*/
static void Decode(UINT4 *output, unsigned char *input,unsigned int  len)
{
unsigned int i, j;
for(i = 0, j = 0; j < len; i++, j += 4)
  output[i] = ((UINT4)input[j]) | (((UINT4)input[j + 1]) << 8) |
   (((UINT4)input[j + 2]) << 16) | (((UINT4)input[j + 3]) << 24);
}
```

md5test.cpp 代码及说明如下。

```
//md5test.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include "string.h"
int main(int argc, char * argv[])
{
MD5_CTX md5;
MD5Init(&md5);                                          //初始化用于 MD5 加密的结构
unsigned char encrypt[200];                             //存放欲加密的信息
unsigned char decrypt[17];                              //存放加密后的结果
scanf("%s",encrypt);                                    //输入加密的字符
MD5Update(&md5,encrypt,strlen((char *)encrypt));        //对欲加密的字符进行加密
MD5Final(decrypt,&md5);                                 //获得最终结果
printf("加密前:%s\n 加密后:",encrypt);
for(int i=0;i<16;i++)
  printf("%2x ",decrypt[i]);
  printf("\n\n\n 加密结束!\n");
  return 0;
}
```

3.3.7 PGP 技术

1. 概念

PGP(Pretty Good Privacy)加密技术是一个基于 RSA 公钥加密体系的邮件加密软件，提出了公共钥匙或不对称文件的加密技术。PGP 加密技术的创始人是美国的 Phil Zimmermann。他的创造性把 RSA 公钥体系和传统加密体系结合起来，并且在数字签名和密钥认证管理机制上有巧妙的设计，因此 PGP 成为目前几乎最流行的公钥加密软件包。

由于 RSA 算法计算量极大，在速度上不适合加密大量数据，所以 PGP 实际上用来加密的不是 RSA 本身，而是采用传统加密算法 IDEA，IDEA 加解密的速度比 RSA 快得多。PGP 随机生成一个密钥，用 IDEA 算法对明文加密，然后用 RSA 算法对密钥加密；收件人同样是用 RSA 解出随机密钥，再用 IEDA 解出原文。这样的链式加密既有 RSA 算法的保密性(Privacy)和认证性(Authentication)，又保持了 IDEA 算法速度快的优势。

2. PGP 加密软件

使用 PGP8.0.2i 可以简洁而高效地实现邮件或者文件的加密、数字签名。

【实验 3-1】 网络软件下载安全性检验

【实验目的】

(1) 理解 MD5 的含义。

(2) 掌握 MD5 的一般应用，完成网络下载安全性检查。

【实验步骤】

(1) 以下载国泰君安软件为例,打开网站“http://www.gtja.com/jccy/softdownload.html”,下载官方提供的 MD5 码计算工具和国泰君安大智慧软件。

(2) 运行 MD5 码计算工具 GtjaMD5.exe。

(3) 浏览所下载的国泰君安大智慧软件,选择计算等待生成 MD5 码,如图 3-10 所示。

图 3-10 生成 MD5 码

(4) 检查生成的 MD5 码是否与网站提供下载的 MD5 码相同。

【思考与练习】

网络上可以很容易找到 MD5 的破解网站,那么它们是有效的吗? 如果有效,方法是什么? 是否说明 MD5 有问题呢?

【实验 3-2】 PGP 加密应用实验

(1) 安装 PGP 加密软件。

PGP 的安装向导界面如图 3-11 所示。

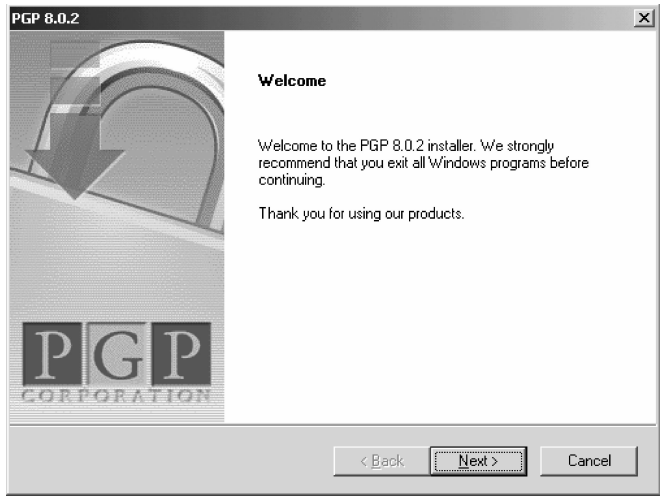

图 3-11 安装向导

下面的几步全采用默认的安装设置，因为是第一次安装，所以在用户类型设置界面中选择“No, I'm a New User”单选按钮，如图 3-12 所示。

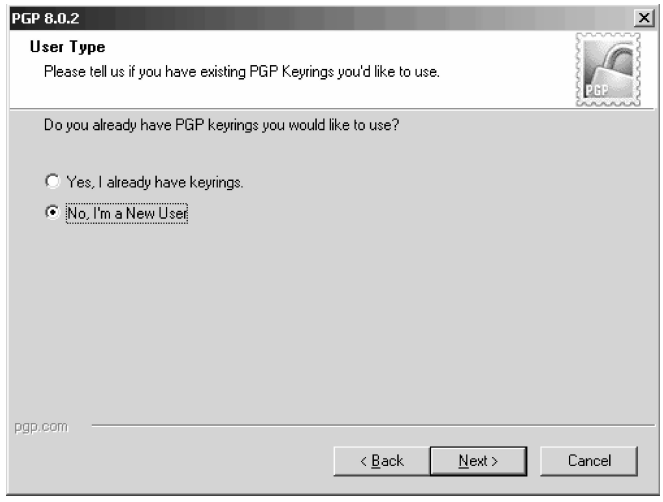

图 3-12　用户类型选择

根据需要选择安装的组件一般根据默认选即设置项可，PGPdisk Volume Security 的功能是提供磁盘文件系统的安全性；PGPmail for Microsoft Outlook/Outlook Express 提供邮件的加密功能，如图 3-13 所示。

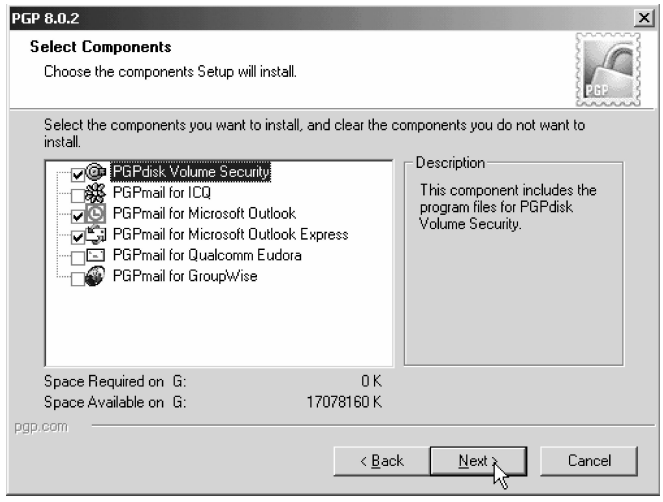

图 3-13　选择功能

(2) 使用 PGP 产生密钥。

用户类型选择了新用户，在计算机启动以后，会自动提示建立 PGP 密钥，如图 3-14 所示。

单击“下一步”按钮，进入用户信息设置界面，输入相应的姓名和电子邮件地址，然后单击“下一步”按钮，如图 3-15 所示。

在 PGP 密码输入框中输入 8 位以上的密码并确认，如图 3-16 所示。

图 3-14 提示建立 PGP 密钥

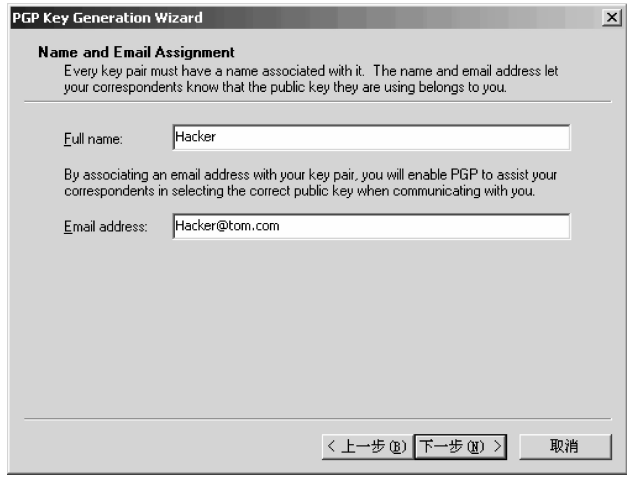

图 3-15 设置用户信息

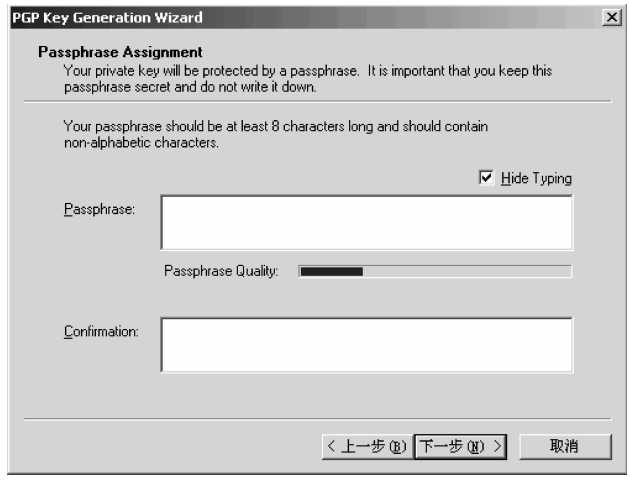

图 3-16 设置密码信息

然后 PGP 会自动产生 PGP 密钥，生成的密钥如图 3-17 所示。

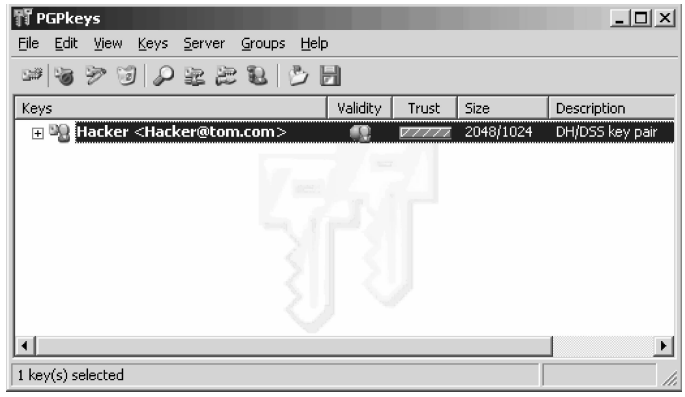

图 3-17　生成密钥

(3) 使用 PGP 加密文件。

使用 PGP 可以加密本地文件，右击要加密的文件，在弹出的快捷菜单中选择 PGP→Encrypt 命令，如图 3-18 所示。

图 3-18　选择命令

系统自动出现对话框，让用户选择要使用的加密密钥，选中一个密钥后单击 OK 按钮，如图 3-19 所示。

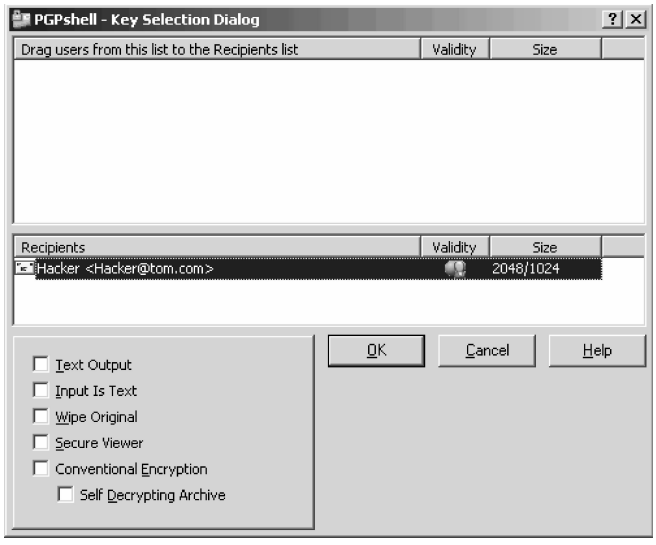

图 3-19　选择密钥

图　3-20

此时目标文件被加密了，当前目录下自动产生一个新的文件，如图 3-20 所示。

打开加密后的文件时，程序会要求输入密码，并要求输入建立该密钥时设置的密码，如图 3-21 所示。

图 3-21 设置生效

(4) 使用 PGP 加密邮件。

PGP 的主要功能是加密邮件,安装完毕后,PGP 会自动和 Outlook 或者 Outlook Express 关联。和 Outlook Express 关联的界面显示如图 3-22 所示。

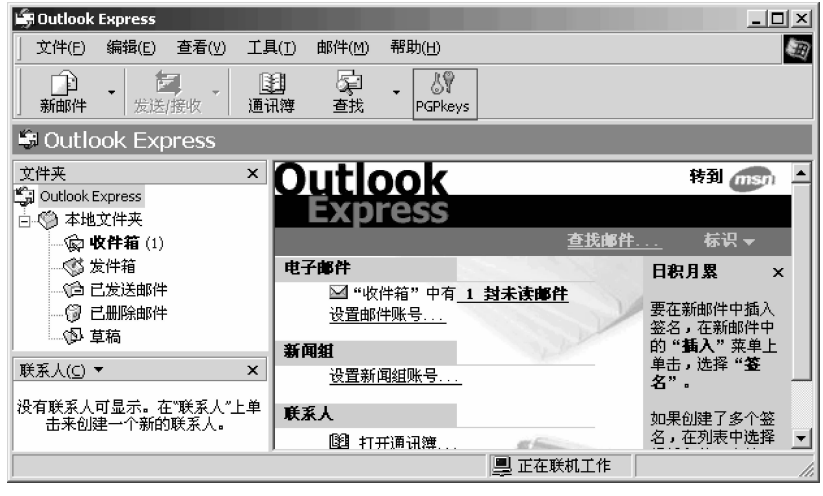

图 3-22 关联界面

利用 Outlook 建立邮件后,可以选择利用 PGP 进行加密和签名,如图 3-23 所示。

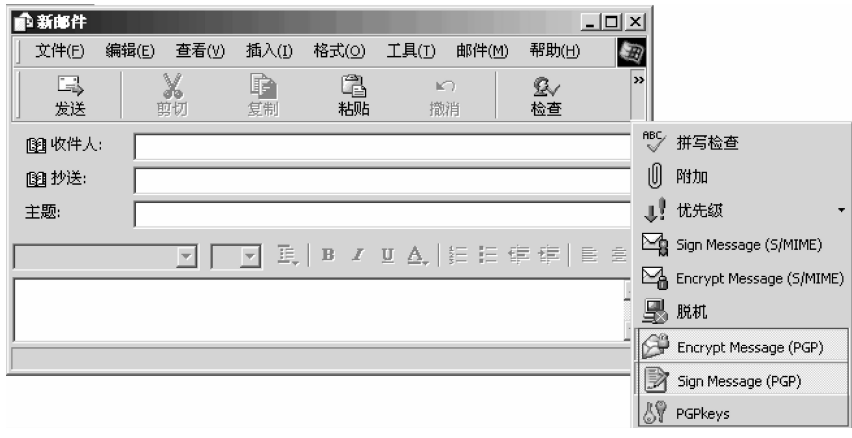

图 3-23 加密邮件

【思考与练习】

(1) 找到一个你认为好用的 DES 加密软件,对自己指定的文件进行加密解密操作,提交报告(带截图及说明)。并请设计一个你认为对自己来说最有可能应用 DES 的情境。

(2) 请解释什么是 APT。并找到一款 APT 工具,试用它验证有效性。

(3) 请解释什么是 CC 攻击。尝试找到一款 CC 攻击工具,试用并验证它的有效性。

以上操作要确保不在真实网络中使用以免带来危害,最好在虚拟机环境下测试完成。