Data Encryption Standard Algorithm

By

Syed Siraj Ul Haque
Ammar Hassan

July 2000

Table of Contents

1.1 Introduction

The DES (Data Encryption Standard) algorithm is the most widely used encryption algorithm in the world. For many years, and among many people, "secret code making" and DES have been synonymous. And despite the recent coup by the Electronic Frontier Foundation in creating a $220,000 machine to crack DES-encrypted messages, DES will live on in government and banking for years to come through a life- extending version called "triple-DES."

How does DES work? This article explains the various steps involved in DES-encryption, illustrating each step by means of a simple example. Since the creation of DES, many other algorithms (recipes for changing data) have emerged which are based on design principles similar to DES. Once you understand the basic transformations that take place in DES, you will find it easy to follow the steps involved in these more recent algorithms.

But first a bit of history of how DES came about is appropriate, as well as a look toward the future. On May 15, 1973, during the reign of Richard Nixon, the National Bureau of Standards (NBS) published a notice in the Federal Register soliciting proposals for cryptographic algorithms to protect data during transmission and storage. The notice explained why encryption was an important issue.

Over the last decade, there has been an accelerating increase in the accumulations and communication of digital data by government, industry and by other organizations in the private sector. The contents of these communicated and stored data often have very significant value and/or sensitivity. It is now common to find data transmissions which constitute funds transfers of several million dollars, purchase or sale of securities, warrants for arrests or arrest and conviction records being communicated between law enforcement agencies, airline reservations and ticketing representing investment and value both to the airline and passengers, and health and patient care records transmitted among physicians and treatment centers.

The increasing volume, value and confidentiality of these records regularly transmitted and stored by commercial and government agencies has led to heightened recognition and concern over their exposures to unauthorized access and use. This misuse can be in the form of theft or defalcations of data records representing money, malicious modification of business inventories or the interception and misuse of confidential information about people. The need for protection is then apparent and urgent.

It is recognized that encryption (otherwise known as scrambling, enciphering or privacy transformation) represents the only means of protecting such data during transmission and a useful means of protecting the content of data stored on various media, providing encryption of adequate strength can be devised and validated and is inherently integrable into system architecture. The National Bureau of Standards solicits proposed techniques and algorithms for computer data encryption. The Bureau also solicits recommended techniques for implementing the cryptographic function: for generating, evaluating, and protecting cryptographic keys; for maintaining files encoded under expiring keys; for

making partial updates to encrypted files; and mixed clear and encrypted data to permit labelling, polling, routing, etc. The Bureau in its role for establishing standards and aiding government and industry in assessing technology, will arrange for the evaluation of protection methods in order to prepare guidelines.

NBS waited for the responses to come in. It received none until August 6, 1974, three days before Nixon's resignation, when IBM submitted a candidate that it had developed internally under the name LUCIFER. After evaluating the algorithm with the help of the National Security Agency (NSA), the NBS adopted a modification of the LUCIFER algorithm as the new Data Encryption Standard (DES) on July 15, 1977.

DES was quickly adopted for non-digital media, such as voice-grade public telephone lines. Within a couple of years, for example, International Flavors and Fragrances was using DES to protect its valuable formulas transmitted over the phone ("With Data Encryption, Scents Are Safe at IFF," *Computerworld* 14, No. 21, 95 (1980).)

Meanwhile, the banking industry, which is the largest user of encryption outside government, adopted DES as a wholesale banking standard. Standards for the wholesale banking industry are set by the American National Standards Institute (ANSI). ANSI X3.92, adopted in 1980, specified the use of the DES algorithm.

1.2 Preliminary Examples of DES

DES works on bits, or binary numbers--the 0s and 1s common to digital computers. Each group of four bits makes up a hexadecimal, or base 16, number. Binary "0001" is equal to the hexadecimal number "1", binary "1000" is equal to the hexadecimal number "8", "1001" is equal to the hexadecimal number "9", "1010" is equal to the hexadecimal number "A", and "1111" is equal to the hexadecimal number "F".

DES works by encrypting groups of 64 message bits, which is the same as 16 hexadecimal numbers. To do the encryption, DES uses "keys" where are also apparently 16 hexadecimal numbers long or *apparently* 64 bits long. However, every 8th key bit is ignored in the DES algorithm, so that the effective key size is 56 bits. But, in any case, 64 bits (16 hexadecimal digits) is the round number upon which DES is organized.

For example, if we take the plaintext message "8787878787878787", and encrypt it with the DES key "0E329232EA6D0D73", we end up with the cipher text "0000000000000000". If the cipher text is decrypted with the same secret DES key "0E329232EA6D0D73", the result is the original plaintext "8787878787878787".

This example is neat and orderly because our plaintext was exactly 64 bits long. The same would be true if the plaintext happened to be a multiple of 64 bits. But most messages will not fall into this category. They will not be an exact multiple of 64 bits (that is, an exact multiple of 16 hexadecimal numbers).

For example, take the message "Your lips are smoother than Vaseline". This plaintext message is 38 bytes (76 hexadecimal digits) long. So this message must be padded with some extra bytes at the tail end for the encryption. Once the encrypted message has been decrypted, these extra bytes are thrown away. There are, of course, different padding schemes--different ways to add extra bytes. Here we will just add 0s at the end, so that the total message is a multiple of 8 bytes (or 16 hexadecimal digits, or 64 bits).

The plaintext message "Your lips are smoother than vaseline" is, in hexadecimal,

"596F7572206C6970 732061726520736D 6F6F746865722074 68616E2076617365 6C696E650D0A".

(Note here that the first 72 hexadecimal digits represent the English message, while "0D" is hexadecimal for Carriage Return, and "0A" is hexadecimal for Line Feed, showing that the message file has terminated.) We then pad this message with some 0s on the end, to get a total of 80 hexadecimal digits:

"596F7572206C6970 732061726520736D 6F6F746865722074 68616E2076617365 6C696E650D0A0000".

If we then encrypt this plaintext message 64 bits (16 hexadecimal digits) at a time, using the same DES key "0E329232EA6D0D73" as before, we get the ciphertext:

"C0999FDDE378D7ED 727DA00BCA5A84EE 47F269A4D6438190 D52F78F5358499 828AC9B453E0E653".

This is the secret code that can be transmitted or stored. Decrypting the cipher text restores the original message "Your lips are smoother than Vaseline". (Think how much better off Bill Clinton would be today, if Monica Lewinsky had used encryption on her Pentagon computer!)

1.3 How DES Works in Detail

DES is a ***block cipher***--meaning it operates on plaintext blocks of a given size (64-bits) and returns cipher text blocks of the same size. Thus DES results in a ***permutation*** among the 2^64 (read this as: "2 to the 64th power") possible arrangements of 64 bits, each of which may be either 0 or 1. Each block of 64 bits is divided into two blocks of 32 bits each, a left half block **L** and a right half **R**. (This division is only used in certain operations.)

Example: Let **M** be the plain text message **M** = 0123456789ABCDEF, where **M** is in hexadecimal (base 16) format. Rewriting **M** in binary format, we get the 64-bit block of text:

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
L = 0000 0001 0010 0011 0100 0101 0110 0111
R = 1000 1001 1010 1011 1100 1101 1110 1111

The first bit of **M** is "0". The last bit is "1". We read from left to right.

DES operates on the 64-bit blocks using *key* sizes of 56- bits. The keys are actually stored as being 64 bits long, but every 8th bit in the key is not used (i.e. bits numbered 8, 16, 24, 32, 40, 48, 56, and 64). However, we will nevertheless number the bits from 1 to 64, going left to right, in the following calculations. But, as you will see, the eight bits just mentioned get eliminated when we create sub keys.

Example: Let **K** be the hexadecimal key **K** = 133457799BBCDFF1. This gives us as the binary key (setting 1 = 0001, 3 = 0011, etc., and grouping together every eight bits, of which the last one in each group will be unused):

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

The DES algorithm uses the following steps:

1.3.1 Step 1: Create 16 sub keys, each of which is 48-bits long.

The 64-bit key is permuted according to the following table, **PC-1**. Since the first entry in the table is "57", this means that the 57th bit of the original key **K** becomes the first bit of the permuted key **K**+. The 49th bit of the original key becomes the second bit of the permuted key. The 4th bit of the original key is the last bit of the permuted key. Note only 56 bits of the original key appear in the permuted key.

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

Example: From the original 64-bit key

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

We get the 56-bit permutation

K+ = 1111000 0110011 0010101 0101111 0101010 1011001 1001111 0001111

Next, split this key into left and right halves, C_0 and D_0, where each half has 28 bits.

Example: From the permuted key **K**+, we get

C_0 = 1111000 0110011 0010101 0101111
D_0 = 0101010 1011001 1001111 0001111

With C_0 and D_0 defined, we now create sixteen blocks C_n and D_n, $1 <= n <= 16$. Each pair of blocks C_n and D_n is formed from the previous pair C_{n-1} and D_{n-1}, respectively, for n = 1, 2, ..., 16, using the following schedule of "left shifts" of the previous block. To do a left shift, move each bit one place to the left, except for the first bit, which is cycled to the end of the block.

Iteration Number	Number of Left Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

This means, for example, C_3 and D_3 are obtained from C_2 and D_2, respectively, by two left shifts, and C_{16} and D_{16} are obtained from C_{15} and D_{15}, respectively, by one left shift. In all cases, by a single left shift is meant a rotation of the bits one place to the left, so that after one left shift the bits in the 28 positions are the bits that were previously in positions 2, 3,..., 28, 1.

Example: From original pair pair C_0 and D_0 we obtain:

C_0 = 1111000011001100101010101111
D_0 = 0101010101100110011110001111

C_1 = 1110000110011001010101011111
D_1 = 1010101011001100111100011110

C_2 = 1100001100110010101010111111
D_2 = 0101010110011001111000111101

C_3 = 0000110011001010101011111111
D_3 = 0101011001100111100011110101

C_4 = 0011001100101010101111111100
D_4 = 0101100110011110001111010101

C_5 = 1100110010101010111111110000
D_5 = 0110011001111000111101010101

C_6 = 0011001010101011111111000011
D_6 = 1001100111100011110101010101

C_7 = 1100101010101111111100001100
D_7 = 0110011110001111010101010110

C_8 = 0010101010111111110000110011
D_8 = 1001111000111101010101011001

C_9 = 0101010101111111100001100110
D_9 = 0011110001111010101010110011

C_{10} = 0101010111111110000110011001
D_{10} = 1111000111101010101011001100

C_{11} = 0101011111111000011001100101
D_{11} = 1100011110101010101100110011

C_{12} = 0101111111100001100110010101
D_{12} = 0001111010101010110011001111

C_{13} = 0111111110000110011001010101
D_{13} = 0111101010101011001100111100

C_{14} = 1111111000011001100101010101
D_{14} = 1110101010101100110011110001

C_{15} = 1111100001100110010101010111
D_{15} = 1010101010110011001111000111

C_{16} = 1111000011001100101010101111
D_{16} = 0101010101100110011110001111

We now form the keys K_n, for 1<=n<=16, by applying the following permutation table to each of the concatenated pairs C_nD_n. Each pair has 56 bits, but **PC-2** only uses 48 of these.

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

Therefore, the first bit of K_n is the 14th bit of C_nD_n, the second bit the 17th, and so on, ending with the 48th bit of K_n being the 32th bit of C_nD_n.

Example: For the first key we have C_1D_1 = 1110000 1100110 0101010 1011111 1010101 0110011 0011110 0011110

which, after we apply the permutation **PC-2**, becomes

K_1 = 000110 110000 001011 101111 111111 000111 000001 110010

For the other keys we have

K_2 = 011110 011010 111011 011001 110110 111100 100111 100101
K_3 = 010101 011111 110010 001010 010000 101100 111110 011001
K_4 = 011100 101010 110111 010110 110110 110011 010100 011101
K_5 = 011111 001110 110000 000111 111010 110101 001110 101000
K_6 = 011000 111010 010100 111110 010100 000111 101100 101111
K_7 = 111011 001000 010010 110111 111101 100001 100010 111100
K_8 = 111101 111000 101000 111010 110000 010011 101111 111011
K_9 = 111000 001101 101111 101011 111011 011110 011110 000001
K_{10} = 101100 011111 001101 000111 101110 100100 011001 001111
K_{11} = 001000 010101 111111 010011 110111 101101 001110 000110
K_{12} = 011101 010111 000111 110101 100101 000110 011111 101001
K_{13} = 100101 111100 010111 010001 111110 101011 101001 000001
K_{14} = 010111 110100 001110 110111 111100 101110 011100 111010
K_{15} = 101111 111001 000110 001101 001111 010011 111100 001010
K_{16} = 110010 110011 110110 001011 000011 100001 011111 110101

So much for the sub keys. Now we look at the message itself.

1.3.2 Step 2: Encode each 64-bit block of data

There is an *initial permutation* **IP** of the 64 bits of the message data **M**. This rearranges the bits according to the following table, where the entries in the table show the new arrangement of the bits from their initial order. The 58th bit of **M** becomes the first bit of **IP**. The 50th bit of **M** becomes the second bit of **IP**. The 7th bit of **M** is the last bit of **IP**.

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

Example: Applying the initial permutation to the block of text **M**, given previously, we get

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
IP = 1100 1100 0000 0000 1100 1100 1111 1111 1111 0000 1010 1010 1111 0000 1010 1010

Here the 58th bit of **M** is "1", which becomes the first bit of **IP**. The 50th bit of **M** is "1", which becomes the second bit of **IP**. The 7th bit of **M** is "0", which becomes the last bit of **IP**.

Next divide the permuted block **IP** into a left half L_0 of 32 bits, and a right half R_0 of 32 bits.

Example: From **IP**, we get L_0 and R_0

L_0 = 1100 1100 0000 0000 1100 1100 1111 1111
R_0 = 1111 0000 1010 1010 1111 0000 1010 1010

We now proceed through 16 iterations, for $1 <= n <= 16$, using a function f which operates on two blocks--a data block of 32 bits and a key K_n of 48 bits--to produce a block of 32 bits. **Let + denote XOR addition, (bit-by-bit addition modulo 2)**. Then for **n** going from 1 to 16 we calculate

$$L_n = R_{n-1}$$
$$R_n = L_{n-1} + f(R_{n-1}, K_n)$$

This results in a final block, for $n = 16$, of $L_{16}R_{16}$. That is in each iteration, we take the right 32 bits of the previous result and make them the left 32 bits of the current step. For

the right 32 bits in the current step, we XOR the left 32 bits of the previous step with the calculation ***f*** .

Example: For ***n*** = 1, we have

K_1 = 000110 110000 001011 101111 111111 000111 000001 110010
$L_1 = R_0$ = 1111 0000 1010 1010 1111 0000 1010 1010
$R_1 = L_0 + f(R_0, K_1)$

It remains to explain how the function ***f*** works. To calculate ***f***, we first expand each block R_{n-1} from 32 bits to 48 bits. This is done by using a selection table that repeats some of the bits in R_{n-1} . We'll call the use of this selection table the function **E**. Thus $\mathbf{E}(R_{n-1})$ has a 32 bit input block, and a 48 bit output block.

Let **E** be such that the 48 bits of its output, written as 8 blocks of 6 bits each, are obtained by selecting the bits in its inputs in order according to the following table:

E BIT-SELECTION TABLE

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

Thus the first three bits of $\mathbf{E}(R_{n-1})$ are the bits in positions 32, 1 and 2 of R_{n-1} while the last 2 bits of $\mathbf{E}(R_{n-1})$ are the bits in positions 32 and 1.

Example: We calculate $\mathbf{E}(R_0)$ from R_0 as follows:

R_0 = 1111 0000 1010 1010 1111 0000 1010 1010
$\mathbf{E}(R_0)$ = 011110 100001 010101 010101 011110 100001 010101 010101

(Note that each block of 4 original bits has been expanded to a block of 6 output bits.)

Next in the ***f*** calculation, we XOR the output $\mathbf{E}(R_{n-1})$ with the key K_n:

$$K_n + \mathbf{E}(R_{n-1}).$$

Example: For K_1 , $\mathbf{E}(R_0)$, we have

K_1 = 000110 110000 001011 101111 111111 000111 000001 110010
$\mathbf{E}(R_0)$ = 011110 100001 010101 010101 011110 100001 010101 010101
$K_1 + \mathbf{E}(R_0)$ = 011000 010001 011110 111010 100001 100110 010100 100111.

We have not yet finished calculating the function f . To this point we have expanded R_{n-1} from 32 bits to 48 bits, using the selection table, and XORed the result with the key K_n . We now have 48 bits, or eight groups of six bits. We now do something strange with each group of six bits: we use them as addresses in tables called "**S boxes**". Each group of six bits will give us an address in a different **S** box. Located at that address will be a 4 bit number. This 4 bit number will replace the original 6 bits. The net result is that the eight groups of 6 bits are transformed into eight groups of 4 bits (the 4-bit outputs from the **S** boxes) for 32 bits total.

Write the previous result, which is 48 bits, in the form:

$$K_n + \mathbf{E}(R_{n-1}) = B_1B_2B_3B_4B_5B_6B_7B_8,$$

where each B_i is a group of six bits. We now calculate

$$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$$

where $S_i(B_i)$ referres to the output of the i-th **S** box.

To repeat, each of the functions ***S1, S2,..., S8***, takes a 6-bit block as input and yields a 4-bit block as output. The table to determine S_1 is shown and explained below:

S1

Row No.	Column Number 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

If S_1 is the function defined in this table and B is a block of 6 bits, then $S_1(B)$ is determined as follows: The first and last bits of B represent in base 2 a number in the decimal range 0 to 3 (or binary 00 to 11). Let that number be i. The middle 4 bits of B represent in base 2 a number in the decimal range 0 to 15 (binary 0000 to 1111). Let that number be j. Look up in the table the number in the i-th row and j-th column. It is a number in the range 0 to 15 and is uniquely represented by a 4 bit block. That block is the output $S_1(B)$ of S_1 for the input B. For example, for input block B = 011011 the first bit is "0" and the last bit "1" giving 01 as the row. This is row 1. The middle four bits are "1101". This is the binary equivalent of decimal 13, so the column is column number 13. In row 1, column 13 appears 5. This determines the output; 5 is binary 0101, so that the output is 0101. Hence $S_1(011011) = 0101$.

The tables defining the functions $S_1,...,S_8$ are the following:

S1

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

S2

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

S3

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

S4

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

S5

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

S6

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

S7

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

S8

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

Example: For the first round, we obtain as the output of the eight **S** boxes:

$K_1 + \mathbf{E}(R_0)$ = 011000 010001 011110 111010 100001 100110 010100 100111.

$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$ = 0101 1100 1000 0010 1011 0101 1001 0111

The final stage in the calculation of f is to do a permutation **P** of the **S**-box output to obtain the final value of f:

$$f = \mathbf{P}(S_1(B_1)S_2(B_2)...S_8(B_8))$$

The permutation **P** is defined in the following table. **P** yields a 32-bit output from a 32-bit input by permuting the bits of the input block.

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

Example: From the output of the eight **S** boxes:

$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$ = 0101 1100 1000 0010 1011 0101 1001 0111

we get

$$f = 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011$$

$R_1 = L_0 + f(R_0 , K_1)$

= 1100 1100 0000 0000 1100 1100 1111 1111
+ 0010 0011 0100 1010 1010 1001 1011 1011
= 1110 1111 0100 1010 0110 0101 0100 0100

In the next round, we will have $L_2 = R_1$, which is the block we just calculated, and then we must calculate $R_2 = L_1 + f(R_1, K_2)$, and so on for 16 rounds. At the end of the sixteenth round we have the blocks L_{16} and R_{16}. We then ***reverse*** the order of the two blocks into the 64-bit block

$$R_{16}L_{16}$$

and apply a final permutation $\mathbf{IP^{-1}}$ as defined by the following table:

$\mathbf{IP^{-1}}$

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

That is, the output of the algorithm has bit 40 of the preoutput block as its first bit, bit 8 as its second bit, and so on, until bit 25 of the preoutput block is the last bit of the output.

Example: If we process all 16 blocks using the method defined previously, we get, on the 16th round,

$\boldsymbol{L_{16}}$ = 0100 0011 0100 0010 0011 0010 0011 0100
$\boldsymbol{R_{16}}$ = 0000 1010 0100 1100 1101 1001 1001 0101

We reverse the order of these two blocks and apply the final permutation to

$\boldsymbol{R_{16}L_{16}}$ = 00001010 01001100 11011001 10010101 01000011 01000010 00110010 00110100

$\boldsymbol{IP^{-1}}$ = 10000101 11101000 00010011 01010100 00001111 00001010 10110100 00000101

which in hexadecimal format is

85E813540F0AB405.

This is the encrypted form of **M** = 0123456789ABCDEF: namely, **C** = 85E813540F0AB405.

Decryption is simply the inverse of encryption, following the same steps as above, but reversing the order in which the sub keys are applied.

1.4 Summaries

Key schedule:
C[0]D[0] = PC1(key)
for 1 <= i <= 16
C[i] = LS[i](C[i-1])
D[i] = LS[i](D[i-1])
K[i] = PC2(C[i]D[i])

Encipherment:
L[0]R[0] = IP(plain block)
for 1 <= i <= 16
L[i] = R[i-1]
R[i] = L[i-1] xor f(R[i-1], K[i])
cipher block = FP(R[16]L[16])

Decipherment:
R[16]L[16] = IP(cipher block)
for 1 <= i <= 16
R[i-1] = L[i]
L[i-1] = R[i] xor f(L[i], K[i])
plain block = FP(L[0]R[0])

To encrypt or decrypt more than 64 bits there are four official modes (defined in FIPS PUB 81). One is to go through the above-described process for each block in succession. This is called Electronic Codebook (ECB) mode. A stronger method is to exclusive-or each plaintext block with the preceding cipher text block prior to encryption. (The first block is exclusive-or'ed with a secret 64-bit initialization vector (IV). This IV is generally a random value that is kept with the key.) This is called Cipher Block Chaining (CBC) mode. The other two modes are Output Feedback (OFB) and Cipher Feedback (CFB).

When it comes to padding the data block, there are several options. One is to simply append zeros. Two suggested by FIPS PUB 81 are, if the data is binary data, fill up the block with bits that are the opposite of the last bit of data, or, if the data is ASCII data, fill up the block with random characters and put the ASCII character for the number of pad characters in the last byte of the block

The DES algorithm can also be used to calculate cryptographic checksums up to 64 bits long (see FIPS PUB 113). If the number of data bits to be check summed is not a multiple of 64, the last data block should be padded with zeros. If the data is ASCII data, the most significant bit of each byte should be set to 0. The data is then encrypted in CBC mode with IV = 0. The most significant n bits (where 16 <= n <= 64, and n is a multiple of 8) of the final cipher text block are an n-bit checksum.

1.5 DES Modes of Operation

The DES algorithm turns a 64-bit message block **M** into a 64-bit cipher block **C**. If each 64-bit block is encrypted individually, then the mode of encryption is called ***Electronic Code Book*** (ECB) mode. There are two other modes of DES encryption, namely ***Chain Block Coding*** (CBC) and ***Cipher Feedback*** (CFB), which make each cipher block dependent on all the previous messages blocks through an initial XOR operation.

1.6 Cracking DES

Before DES was adopted as a national standard, during the period NBS was soliciting comments on the proposed algorithm, the creators of public key cryptography, Martin Hellman and Whitfield Diffie, registered some objections to the use of DES as an encryption algorithm. Hellman wrote: "Whit Diffie and I have become concerned that the proposed data encryption standard, while probably secure against commercial assault, may be extremely vulnerable to attack by an intelligence organization" (letter to NBS, October 22, 1975).

Diffie and Hellman then outlined a "brute force" attack on DES. (By "brute force" is meant that you try as many of the 2^56 possible keys as you have to before decrypting the ciphertext into a sensible plaintext message.) They proposed a special purpose "parallel computer using one million chips to try one million keys each" per second, and estimated the cost of such a machine at $20 million.

Fast forward to 1998. Under the direction of John Gilmore of the EFF, a team spent $220,000 and built a machine that can go through the entire 56-bit DES key space in an average of 4.5 days. On July 17, 1998, they announced they had cracked a 56-bit key in 56 hours. The computer, called Deep Crack, uses 27 boards each containing 64 chips, and is capable of testing 90 billion keys a second.

Despite this, as recently as June 8, 1998, Robert Litt, principal associate deputy attorney general at the Department of Justice, denied it was possible for the FBI to crack DES: "Let me put the technical problem in context: It took 14,000 Pentium computers working for four months to decrypt a single message . . . . We are not just talking FBI and NSA [needing massive computing power], we are talking about every police department."

Responded cryptograpy expert Bruce Schneier: " . . . the FBI is either incompetent or lying, or both." Schneier went on to say: "The only solution here is to pick an algorithm with a longer key; there isn't enough silicon in the galaxy or enough time before the sun burns out to brute- force triple-DES" (*Crypto-Gram*, Counterpane Systems, August 15, 1998).

1.7 Triple-DES

Triple-DES is just DES with two 56-bit keys applied. Given a plaintext message, the first key is used to DES- encrypt the message. The second key is used to DES-decrypt the encrypted message. (Since the second key is not the right key, this decryption just scrambles the data further.) The twice-scrambled message is then encrypted again with the first key to yield the final cipher text. This three-step procedure is called triple-DES.

Triple-DES is just DES done three times with two keys used in a particular order. (Triple-DES can also be done with three separate keys instead of only two. In either case the resultant key space is about 2^112.)

CODE

```
/------------------------------------------ encrypt.cpp ----------------------------------------/
#include"crypt0.h"


void main(int argc,char *argv[])
{
  clrscr();
  int a;
  if(argc!=3){printf("Format: c:>encrypt filename filename" ); exit(1); }

  if((meg=fopen(argv[1],"rb"))==NULL)
    {
     printf("\nCant open message file");
     getche();
     exit(1);
    }
  keyc();
  output=fopen(argv[2],"wb");
  printf("\n\n\n\tFile Encryption in process. Please Wait....");
  openinput();
  printf("\n\n\n\tFile encrypted in %s\n\n\n",argv[2]);
  fcloseall();
 }


void openinput(void)
{
  int a,flag=0,count;

  while(!feof(meg))
        {
                count=0;
                for(a=0;a<8;a++)    mesg[a]=0;

                for(a=0;a<8;a++)
                   {
                    fscanf(meg,"%c",&mes[a]);
                    if(feof(meg))      { flag=1;break; }
                    count++;
                    mesg[a]=mes[a];
                   }
                if(flag)    break;
                message();
        }
 if(flag && count)    message();
  fcloseall();
```

```
}
int bit2(int n,char *x)
{
int group,p,mask;
group=n/8;
p=n%8;
mask=pow(2,7-p);
return (  (x[group] & mask) ? 1:0);
}


void keyc(void)
{
char cipher[8];                                 //char cipher[8]="▯ 4Wy›¼ßñ";// for cipher
printf("   Enter The Key You Want\n   It should be not longer than  8 character: ");
for(int z=0;z<8;z++)    cipher[z]=getche();
printf("\n\n\tThankyou For entering chiperkey which is:  ");

for(z=0;z<8;z++)  printf("%c",cipher[z]);

char c[17][4];
char d[17][4];                                  // c&d for permutated 1/2 key
char kp[7]={0,0,0,0,0,0,0};
char kcd[17][7];
int a,i,x=0,y=0,b=0,v=0,block,p;

//cleaning c and d
for(a=0;a<17;a++)for(i=0;i<4;i++){c[a][i]=0;d[a][i]=0;}
for(a=0;a<17;a++)for(i=0;i<7;i++){kcd[a][i]=0;}


for(a=0;a<7;a++)
   {
   for(i=0;i<8;i++)
      {
      if(bit2(pc1[y][x]-1,cipher)==1)     kp[a]=kp[a]+pow(2,7-i);x++;
      if(x==7){x=0;y++;}
      }
    }
```

```
// for c0 and d0
b=0;v=7;

for(a=0;a<56;a++)
  {
     if( bit2(a,kp)==1)
          {
          if(a<28) {c[0][b]=c[0][b]+pow(2,v);}
          if(a>27){d[0][b]=d[0][b]+pow(2,v);}
          }
          v--;if(v==-1){v=7;b++;}
     if(a==27){v=7;b=0;}
   }
// c[0] and d[0] made

i=pow(2,7);v=pow(2,6);x=pow(2,4);y=pow(2,5);

for(int k=0;k<17;k++)
  {
  for(a=shift[k];a<28;a++)
    {
    block=(a-shift[k])/8;p=(a-shift[k])%8;b=pow(2,7-p);
    if(bit2(a,c[k])==1)  c[k+1][block]=c[k+1][block]+b;
    if(bit2(a,d[k])==1)  d[k+1][block]=d[k+1][block]+b;
     }// c
   if(shift[k]==1)
     {
     if(( (c[k][0] & i ) ? 1:0)==1) c[k+1][3]=c[k+1][3]+x;
     if(( (d[k][0] & i ) ? 1:0)==1) d[k+1][3]=d[k+1][3]+x;
     }
   if(shift[k]==2)
     {
         if(( (c[k][0] & i)  ? 1:0)==1)c[k+1][3]=c[k+1][3]+y;
         if(( (c[k][0] & v)  ? 1:0)==1)c[k+1][3]=c[k+1][3]+x;
         if(( (d[k][0] & i)  ? 1:0)==1)d[k+1][3]=d[k+1][3]+y;
         if(( (d[k][0] & v)  ? 1:0)==1)d[k+1][3]=d[k+1][3]+x;
     }
  }//c++
// adding c and d back together

v=7;
```

```
for(a=1;a<17;a++)
   {
   for(i=0;i<56;i++)
     {
         block=i/8;p=i%8;b=pow(2,7-p);
         if(i<28)if(bit2(i,c[a])==1)kcd[a][block]=kcd[a][block]+b;
         if(i>27)
           {
            if(i==28){v=0;}
            if(bit2(v,d[a])==1)kcd[a][block]=kcd[a][block]+b;
           }
         v++;
     }
   }// end of kcd loop

x=0;y=0;v=0;
for(v=1;v<17;v++)
  {
  x=0;y=0;
  for(a=0;a<6;a++)
    {
     for(i=0;i<8;i++)
           {
            if(bit2(pc2[y][x]-1,kcd[v])==1) key[v][a]=key[v][a]+pow(2,7-i);
            x++;
            if(x==6){x=0;y++;}
           }
    }
  }
}




void message(void)
{
int a,i,x=0,y=0,b=0,sx=0,t,sy=0,epb=0,scc=0,scc2=0;
char f[4]={0,0,0,0};
char l[17][4],r[17][4];
char ep[8];
char ipm[8];
int data[8];
int dat[8];
for(a=0;a<8;a++){ep[a]=0;ipm[a]=0;data[a]=0;}
for(a=0;a<17;a++)for(b=0;b<4;b++){l[a][b]=0;r[a][b]=0;}
```

```
for(a=0;a<8;a++)
    for(i=0;i<8;i++)
      {
       if(bit2(ip[y][x]-1,mesg)==1)
       ipm[a]=ipm[a]+pow(2,7-i);x++;
      }

// dividing now

for(a=0;a<8;a++)
   {
   if(a<4) l[0][a]=ipm[a];
   else r[0][a-4]=ipm[a];
   }

for(b=1;b<17;b++)
   {
   for(a=0;a<4;a++)l[b][a]=r[b-1][a];// l[n]=r[n-1]
   //making r[n]=l[n-1] + f( r[n-1], k[n])

   // step 1= making e(r[n-1]) into ep
   y=0;x=0;
   for(a=0;a<7;a++)ep[a]=0;// cleaning ep
   for(a=0;a<6;a++)
      {
      for(i=0;i<8;i++)
            {
             if(bit2(e[y][x]-1,r[b-1])==1)ep[a]=ep[a]+pow(2,7-i);
             x++;
             if(x==6) {x=0;y++;}
            }
      }

   // step 2= xor between ep and k[n] result saved in ep
   for(a=0;a<6;a++)
      {
      ep[a]=ep[a]^key[b][a];
      }
   // step 3= replacing  6 bit form ep into r[n] (4 bits)
   x=0;y=0;epb=1;scc=0;scc2=0;
   for(a=0;a<8;a++)
      {
      for(i=0;i<6;i++)
            {
            if(i==0)sx=0;
```

```
            if(i==1)sy=0;
            if(bit2(epb-1,ep)==1)
               {
               if(i==0)sx=2;
               if(i==5)sx=sx+1;
               if(i>0 || i<5)sy=sy+pow(2,4-i);
               }
            epb++;
            }
      if(epb==49)epb=0;
      if(scc%2==0)r[b][scc2]=s[a][sx][sy]<<4;
      if(scc%2==1){r[b][scc2]=r[b][scc2]+s[a][sx][sy];scc2++;}scc++;
      }

// step 4 = appling permutation p for completing f function result into f

   x=0;y=0;
   for(a=0;a<4;a++)f[a]=0;// cleaning f
   for(a=0;a<4;a++)
      {
      for(i=0;i<8;i++)
            {
             if(bit2(p[y][x]-1,r[b])==1)f[a]=f[a]+pow(2,7-i);
             x++;
             if(x==4) {x=0;y++;}
            }
      }
  //step 5= xor f (r[n]) with l[n-1] to find r[n]

 for(a=0;a<4;a++)   r[b][a]=f[a]^l[b-1][a];
  }

 //step 6= making r[16]+l[16] into mesg

  for(a=0;a<8;a++)mesg[a]=0;//cleaning mesg
  for(a=0;a<8;a++)
     {
     if(a<4)mesg[a]=r[16][a];
     else mesg[a]=l[16][a-4];
     }
```

```
//step 7= making final  encrypted data

  y=0;x=0;
  for(a=0;a<8;a++)data[a]=0;
  for(a=0;a<8;a++)
  for(i=0;i<8;i++)
      {
       if(bit2(ip1[y][x]-1,mesg)==1)
       data[a]=data[a]+pow(2,7-i);x++;
      }

for(a=0;a<8;a++)  {  fprintf(output,"%c",data[a]);}
}
```

```
/------------------------------------------ decrypt.cpp ----------------------------------------/

#include"crypt0.h"
void main(int argc,char *argv[])
{
  clrscr();
  int a;
  if(argc!=3){printf("Format: c:>decrypt filename filename" ); exit(1); }

  if((meg=fopen(argv[1],"rb"))==NULL)
    {
     printf("\nCant open encrypted file");
     getche();
     exit(1);
    }

  keyc();
  output=fopen(argv[2],"wb");
  printf("\n\n\n\tFile Decryption in process. Please Wait....");
  openinput();
  printf("\n\n\n\tFile decrypted in %s\n\n\n",argv[2]);
  fcloseall();

}

void openinput(void)
{
  int a,flag=0,count;

  while(!feof(meg)){
                    count=0;
                    for(a=0;a<8;a++)      mesg[a]=0;
                    for(a=0;a<8;a++)
                    {
                     fscanf(meg,"%c",&mes[a]);
                     if(feof(meg))   { flag=1;   break; }
                     count=1;
                     mesg[a]=mes[a];
                    }
                    if(flag)     break;
                    message();
                   }

 if(flag && count)    message();
 fcloseall();
}
```

```
int bit2(int n,char *x)
{
int group,p,mask;
group=n/8;
p=n%8;
mask=pow(2,7-p);
return (  (x[group] & mask) ? 1:0);
}


void keyc(void)
{
//char cipher[8]="▯ 4Wy›¼ßñ";// for cipher
char cipher[8]; printf("    Enter The Key You Want\n    It should be not longer than  8
character: ");
for(int z=0;z<8;z++)  cipher[z]=getche();

printf("\n\n\tThankyou For entering chiperkey which is:  ");
for(z=0;z<8;z++)  printf("%c",cipher[z]);

char c[17][4];
char d[17][4];// c&d for permutated 1/2 key
char kp[7]={0,0,0,0,0,0,0};
char kcd[17][7];
int a,i,x=0,y=0,b=0,v=0,block,p;

//cleaning c and d
for(a=0;a<17;a++)for(i=0;i<4;i++){c[a][i]=0;d[a][i]=0;}
for(a=0;a<17;a++)for(i=0;i<7;i++){kcd[a][i]=0;}
for(a=0;a<7;a++)
   {
   for(i=0;i<8;i++)
      {
      if(bit2(pc1[y][x]-1,cipher)==1)     kp[a]=kp[a]+pow(2,7-i);x++;
      if(x==7){x=0;y++;}
      }
    }
```

```
// for c0 and d0
b=0;v=7;
for(a=0;a<56;a++)
   {
      if( bit2(a,kp)==1)
           {
           if(a<28) {c[0][b]=c[0][b]+pow(2,v);}
           if(a>27){d[0][b]=d[0][b]+pow(2,v);}
           }
           v--;if(v==-1){v=7;b++;}
      if(a==27){v=7;b=0;}
    }

// c[0] and d[0] made
i=pow(2,7);v=pow(2,6);x=pow(2,4);y=pow(2,5);
for(int k=0;k<17;k++)
   {
   for(a=shift[k];a<28;a++)
      {
    block=(a-shift[k])/8;p=(a-shift[k])%8;b=pow(2,7-p);
    if(bit2(a,c[k])==1)  c[k+1][block]=c[k+1][block]+b;
    if(bit2(a,d[k])==1)  d[k+1][block]=d[k+1][block]+b;
     }// c
   if(shift[k]==1)
      {
      if(( (c[k][0] & i ) ? 1:0)==1) c[k+1][3]=c[k+1][3]+x;
      if(( (d[k][0] & i ) ? 1:0)==1) d[k+1][3]=d[k+1][3]+x;
      }
   if(shift[k]==2)
      {
          if(( (c[k][0] & i)  ? 1:0)==1)c[k+1][3]=c[k+1][3]+y;
          if(( (c[k][0] & v)  ? 1:0)==1)c[k+1][3]=c[k+1][3]+x;
          if(( (d[k][0] & i)  ? 1:0)==1)d[k+1][3]=d[k+1][3]+y;
          if(( (d[k][0] & v)  ? 1:0)==1)d[k+1][3]=d[k+1][3]+x;
      }
   }//c++
```

```
// adding c and d back together
v=7;
for(a=1;a<17;a++)
   {
   for(i=0;i<56;i++)
     {
         block=i/8;p=i%8;b=pow(2,7-p);
         if(i<28)if(bit2(i,c[a])==1)kcd[a][block]=kcd[a][block]+b;
         if(i>27)
           {
           if(i==28){v=0;}
           if(bit2(v,d[a])==1)kcd[a][block]=kcd[a][block]+b;
           }
         v++;
     }
   }// end of kcd loop

x=0;y=0;v=0;
for(v=1;v<17;v++)
  {
  x=0;y=0;
  for(a=0;a<6;a++)
    {
     for(i=0;i<8;i++)
           {
           if(bit2(pc2[y][x]-1,kcd[v])==1) key[v][a]=key[v][a]+pow(2,7-i);
           x++;
           if(x==6){x=0;y++;}
           }
    }
  }
}


void message(void)
{
int a,i,x=0,y=0,b=0,sx=0,t,sy=0,epb=0,scc=0,scc2=0;
char f[4]={0,0,0,0};
char l[17][4],r[17][4];
char ep[8];
char ipm[8];
int data[8];
int dat[8];
for(a=0;a<8;a++){ep[a]=0;ipm[a]=0;data[a]=0;}
for(a=0;a<17;a++)for(b=0;b<4;b++){l[a][b]=0;r[a][b]=0;}
```

```
for(a=0;a<8;a++)
    for(i=0;i<8;i++)
      {
       if(bit2(ip[y][x]-1,mesg)==1)
       ipm[a]=ipm[a]+pow(2,7-i);x++;
      }

// dividing now

for(a=0;a<8;a++)
   {
   if(a<4) l[0][a]=ipm[a];
   else r[0][a-4]=ipm[a];
   }

for(b=1;b<17;b++)
   {
   for(a=0;a<4;a++)l[b][a]=r[b-1][a];// l[n]=r[n-1]
   //making r[n]=l[n-1] + f( r[n-1], k[n])

   // step 1= making e(r[n-1]) into ep
   y=0;x=0;
   for(a=0;a<7;a++)ep[a]=0;// cleaning ep
   for(a=0;a<6;a++)
      {
      for(i=0;i<8;i++)
            {
             if(bit2(e[y][x]-1,r[b-1])==1)ep[a]=ep[a]+pow(2,7-i);
             x++;
             if(x==6) {x=0;y++;}
            }
      }

   // step 2= xor between ep and k[n] result saved in ep
   for(a=0;a<6;a++)
      {
      ep[a]=ep[a]^key[17-b][a];
      }

   // step 3= replacing  6 bit form ep into r[n] (4 bits)
   x=0;y=0;epb=1;scc=0;scc2=0;
   for(a=0;a<8;a++)
      {
      for(i=0;i<6;i++)
            {
            if(i==0)sx=0;
```

```
            if(i==1)sy=0;
            if(bit2(epb-1,ep)==1)
               {
               if(i==0)sx=2;
               if(i==5)sx=sx+1;
               if(i>0 || i<5)sy=sy+pow(2,4-i);
               }
            epb++;
            }
      if(epb==49)epb=0;
      if(scc%2==0)r[b][scc2]=s[a][sx][sy]<<4;
      if(scc%2==1){r[b][scc2]=r[b][scc2]+s[a][sx][sy];scc2++;}scc++;
      }

   // step 4 = applying permutation p for completing f function result into f
   x=0;y=0;
   for(a=0;a<4;a++)f[a]=0;// cleaning f
   for(a=0;a<4;a++)
      {
      for(i=0;i<8;i++)
            {
             if(bit2(p[y][x]-1,r[b])==1)f[a]=f[a]+pow(2,7-i);
             x++;
             if(x==4) {x=0;y++;}
            }
      }

   //step 5= xor f (r[n]) with l[n-1] to find r[n]
   for(a=0;a<4;a++)   r[b][a]=f[a]^l[b-1][a];
  }

   //step 6= making r[16]+l[16] into mesg
  for(a=0;a<8;a++)mesg[a]=0;//cleaning mesg
  for(a=0;a<8;a++)
     {     if(a<4)mesg[a]=r[16][a];     else mesg[a]=l[16][a-4];    }

  //step 7= making final  encrypted data
  y=0;x=0;
  for(a=0;a<8;a++)data[a]=0;
  for(a=0;a<8;a++)
       for(i=0;i<8;i++)
            {       if(bit2(ip1[y][x]-1,mesg)==1)       data[a]=data[a]+pow(2,7-i);x++;      }
for(a=0;a<8;a++)
   {  fprintf(output,"%c",data[a]);  }
}
/---------------------------------------------- Crypt0.h-----------------------------------------------/
```

```
#include<stdio.h>
#include<stdlib.H>
#include<string.h>
#include<conio.h>
#include<math.h>

void openinput(void);//opens files
int bit2(int,char *x);
//int bit2(int,int []);

void keyc(void);
void message(void);
int key[17][6];
int mes[8];
char mesg[8];

FILE *output;
FILE *meg;

int ip[8][8]={
                {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}
              };

int const e[8][6]={
                    {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[9][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[8][4]={
              {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}
             };

int ip1[8][8]={
                {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}
               };

int pc1[8][7]={
                 {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}
                };

int pc2[8][6]={
                 {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}
                };

int shift[16]={1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
```

References

"Cryptographic Algorithms for Protection of Computer Data during Transmission and Dormant Storage," Federal Register 38, No. 93 (May 15, 1973).

Data Encryption Standard, Federal Information Processing Standard (FIPS) Publication 46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C. (January 1977).

Carl H. Meyer and Stephen M. Matyas, Cryptography: A New Dimension in Computer Data Security, John Wiley & Sons, New York, 1982.

Dorthy Elizabeth Robling Denning, Cryptography and Data Security, Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

D.W. Davies and W.L. Price, Security for Computer Networks: An Introduction to Data Security in Teleprocessing and Electronics Funds Transfer, Second Edition, John Wiley & Sons, New York, 1984, 1989.

Miles E. Smid and Dennis K. Branstad, "The Data Encryption Standard: Past and Future," in Gustavus J. Simmons, ed., Contemporary Cryptography: The Science of Information Integrity, IEEE Press, 1992.

Douglas R. Stinson, Cryptography: Theory and Practice, CRC Press, Boca Raton, 1995.

Bruce Schneier, Applied Cryptography, Second Edition, John Wiley & Sons, New York, 1996.

Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, Handbook of Applied Cryptography, CRC Press, Boca Raton, 1997.