A Modified DES and Triple DES Algorithm for Wireless Networks

[1] **A.K. Santra,** [2]**Nagarajan S**

[1]Professor and Dean, MCA Department, CARE School of Computer Applications, Tiruchirappalli, Tamil Nadu.
[2]Research Scholar, Bharathiar University, Coimbatore and Professor and Head, The Oxford College of Science, Bangalore, Karnataka.

Abstract
The application of technological and related procedures to safeguard the security of various documents while moving on the channel is an important responsibility in electronic data systems. This paper specifies the modification of the Data Encryption Standard (DES) and the Triple Data Encryption Algorithm (TDEA) which may organizations to protect sensitive data. Protection of data during transmission or while in storage may be necessary to maintain the confidentiality and integrity of the information represented by the data. The algorithms uniquely define the mathematical steps required to transform data into a cryptographic cipher and also to transform the cipher back to the original form. The Data Encryption Standard is being made available for use by various agencies within the context of a total security consisting of physical security procedures, good information management practices, and computer system/network access controls.

Key words:
computer security, data encryption standard, triple data encryption algorithm, Federal

1. ormation Processing Standard (FIPS); security.

DES is a ***block cipher***--meaning it operates on plaintext blocks of a given size (64-bits) and returns ciphertext 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 subkeys.
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:

Step 1: Create 16 subkeys, 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,

Manuscript received April 5, 2013
Manuscript revised April 20, 2013

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:

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.

C0 = 1111000011001100101010101111
D0 = 0101010101100110011110001111
C1 = 1110000110011001010101011111
D1 = 1010101011001100111100011110
C2 = 1100001100110010101010111111
D2 = 0101010110011001111000111101
C3 = 0000110011001010101011111111
D3 = 0101011001100111100011110101
C4 = 0011001100101010101111111100
D4 = 0101100110011110001111010101
C5 = 1100110010101010111111110000
D5 = 0110011001111000111101010101
C6 = 0011001010101011111111000011
D6 = 1001100111100011110101010101
C7 = 1100101010101111111100001100
D7 = 0110011110001111010101010110
C8 = 0010101010111111110000110011
D8 = 1001111000111101010101011001
C9 = 0101010101111111100001100110
D9 = 0011110001111010101010110011
C10 = 0101010111111110000110011001
D10 = 1110001111010101010110011000
C11 = 0101011111111000011001100101
D11 = 1000111101010101011001100011
C12 = 0101111111100001100110010101
D12 = 0011110101010101100110011111
C13 = 0111111110000110011001010101
D13 = 1111010101010110011001111100
C14 = 1111111000011001100101010101
D14 = 1101010101011001100111110001
C15 = 1111100001100110010101010111
D15 = 0101010101100110011111000111
C16 = 1111000011001100101010101111
D16 = 0101010101100110011110001111

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

K2 = 011110 011010 111011 011001 110110 111100 100111100101

K3 = 010101 011111 110010 001010 010000 101100 111110 011001

K4 = 011100 101010 110111 010110 110110 110011 010100 011101

K5 = 011111 001110 110000 000111 111010 110101 001110 101000

K6 = 011000 111010 010100 111110 010100 000111 101100 101111

K7 = 111011 001000 010010 110111 111101 100001 100010 111100

K8 = 111101 111000 101000 111010 110000 010011 101111 111011

K9 = 111000 001101 101111 101011 111011 011110 011110 000001

K10 = 101100 011111 001101 000111 101110 100100 011001 001111

K11 = 001000 010101 111111 010011 110111 101101 001110 000110

K12 = 011101 010111 000111 110101 100101 000110 011111 101001

K13 = 100101 111100 010111 010001 111110 101011 101001 000001

K14 = 010111 110100 001110 110111 111100 101110 011100 111010

K15 = 101111 111001 000110 001101 001111 010011 111100 001010

K16 = 110010 110011 110110 001011 000011 100001 011111 110101

So much for the subkeys. Now we look at the message itself.

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 = 000000010010 0011010001010110 011110001001101010111100110111101111

IP = 1100110000000000110011001111111111110000101010101111000010101010

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 **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 **E**(R_{n-1}) are the bits in positions 32, 1 and 2 of R_{n-1} while the last 2 bits of **E**(R_{n-1}) are the bits in positions 32 and 1.

Example: We calculate **E**(R_0) from R_0 as follows:

R_0 = 1111 0000 1010 1010 1111 0000 1010 1010
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 **E**(R_{n-1}) with the key K_n:

K_n + **E**(R_{n-1}).

Example: For K_1 , **E**(R_0), we have

K_1 = 000110 110000 001011 101111 111111 000111 000001 110010
E(R_0) = 011110 100001 010101 010101 011110 100001 010101 010101
K_1+**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 + **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

S1

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

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:

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

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

S1
Column Number

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:

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:
$K_1 + E(R_0)$ = 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 = P(S1(B1)S2(B2)...S8(B8))
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:
$\mathbf{S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)}$
= 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,

L16 = 0100 0011 0100 0010 0011 0010 0011 0100

R16 = 0000 1010 0100 1100 1101 1001 1001 0101
We reverse the order of these two blocks and apply the final permutation to

R16L16 = 00001010 01001100 11011001 1001010
1 01000011 01000010 00110010 00110100
IP-1 = 10000101 11101000 00010011 01010100 0000
1111 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, follwing the same steps as above, but reversing the order in which the sub keys are applied.

2. 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.

3. 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.

4.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 ciphertext. 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.) [10]

5. Weaknesses of the DES Algorithm

3.1 Key space size

In DES, the key consists in a 56-bit vector providing a key space K of 256 = 7.2058 × 1016 elements. In an exhaustive search known-plaintext attack, the cryptanalyst will obtain the solution after 255 or 3.6029 × 1016 trials, on average. In 1977, Diffie and Hellman[DH77] have shown that a special purpose multiple parallel processor consisting of 106 intergrated circuits, each one trying a key every 1μ s, could determine the key used in about 10 hours on average in a known-plaintext attack. The cost of such a multiple processor machine would have been around $50,000,000 in 1977 [Pfl89]. If such a machine was used 365 days a year, 24 hours a day, amortizing the price over the number of key solutions obtained, then the price per solution would have been about $20,000 per solution.
Diffie and Hellman argued that if the key length was increased from 56 to 64 bits, it would make the DES algorithm secure even for “intelligence agencies budgets...” [Sim92], while decreasing the key length from 56 to only 48 would make DES “vulnerable to attack by almost any reasonable sized organization” [Sim92]. The key length is thus a very critical parameter to the security of DES.

3.2 Complement property

Another possible weakness of DES lies in the complement property of the DES algorithm. Let M be a 64-bit plaintext message to be encrypted into a 64-bit ciphertext C using the 56-bit key K: C = DESK (M) . The complement property of DES [Pfl89] indicates that the bit-by-bit modulo-2 complement of the ciphertext C, i.e. C, can be obtained from the plaintext M and key K as: C = DESK (M)
C = DESK (M)
Since complementing the ciphertext vector C takes much less time than actually performing the DES encryption transformation, the exhaustive key search attack can be reduced almost by half.

3.3 DES weak keys

The DES algorithm generates from the 56-bit key K a set, or sequence, of 16 distinct 48-bit sub-keys which are then used in each round of substitution and permutation transformation of DES. However, if the left and right registers Ci and Di of the sub-key schedule calculation branch are filled with “0” or “1”, the sub-keys will be identical:
k1 = k2 = . . . = k16
The encryption and decryption processes being the same except for the order of sub-keys, when such weak keys are employed, enciphering a plaintext messages M twice will result in the original plaintext message [DP84]: 11
DESK [DESK (M)] = M
The weak keys of the DES are listed hereafter:

K1	=	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1
K2	=	FE	FE	FE	FE	FE	FE	FE	FE
K3	=	1F	1F	1F	1F	0E	0E	0E	0E
K4	=	E0	E0	E0	E0	F1	F1	F1	F1

3.4 DES semi-weak key pairs

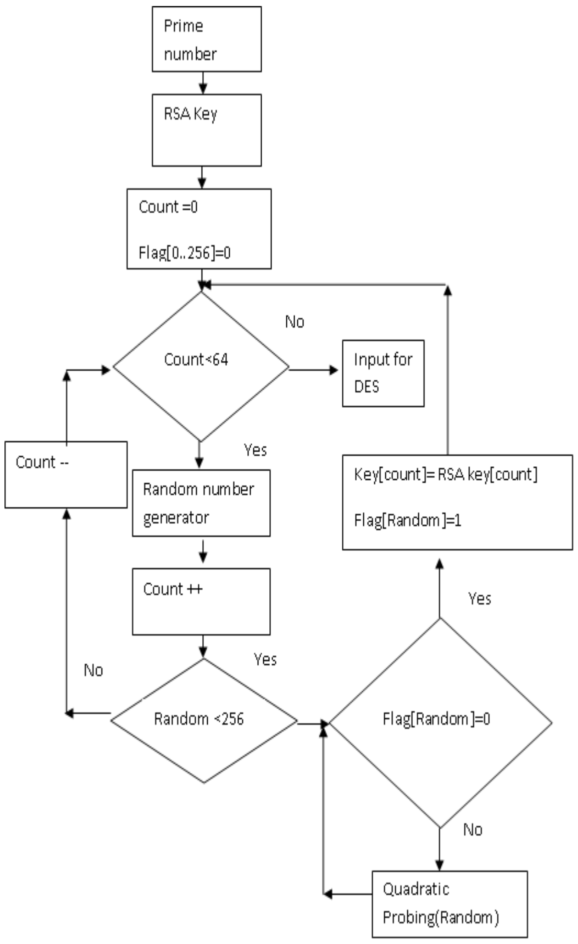
Another property observed in the DES algorithm is the existence of semi-weak pairs of keys for which the pattern of alternating zeroes and ones in the two sub-key registers Ci and Di . This results in the first key, say K1 , producing the sub-key sequence: k1 , k2 , . . ., k16 , while the second key of the pair, K2 , generates the inverse sub-key sequence: k16 , k15 , . . ., k1 . Thus the encryption of message M by key K1 followed by a second encryption with key K2 will give the original message

M: DESK2 [DESK1 (M)] = M

The semi-weak keys of the DES are [DP84]:

K1,1	=	0 1	FE	0 1	FE	0 1	FE	01	FE
K1,2	=	FE	0 1	FE	0 1	FE	01	FE	01
K2,1	=	1F	E0	1F	E0	0E	F1	0E	F1
K2,2	=	E0	1F	E0	1F	F1	0E	F1	0E
K3,1	=	01	E0	01	E0	01	F1	01	F1
K3,2	=	E0	01	E0	01	F1	01	F1	01
K4,1	=	1F	FE	1F	FE	0E	FE	0E	FE
K4,2	=	FE	1F	FE	1F	FE	0E	FE	0E
K5,1	=	01	1F	01	1F	01	0E	01	0E
K5,2	=	1F	01	1F	01	0E	01	0E	01
K6,1	=	E0	FE	E0	FE	F1	FE	F1	FE
K6,2	=	FE	E0	FE	E0	FE	F1	FE	F1

Proposed modification on DES and Tripple DES algorithm



The initial 64 bit key to be generated is generated using the RSA algorithm. The procedure is described in the following steps:

Step1: The 256 bit decimal output generated from the RSA is taken as the initial value to start with.

Step 2: Using random generator algorithm, randomly 64 bits are extracted from the 256 bit decimal output of the RSA which becomes the initial key for the DES and triple DES algorithms.

The proposed random generator algorithm is as shown in the above flow chart.

Jhbjhbvkjxkfksjxckzjxbckzjxckjzxckj

Theorem 1 (Fermat's Little Theorem) *If* p *is a prime number, and* a *is an integer such that* (a, p) = 1, *then* $a^{p-1} = 1(\mathrm{mod}\ p)$.

Proof: Consider the numbers (a · 1), (a · 2), . . . (a · (p − 1)), all modulo p. They are all different. If any of them were the same, say a · m = a · n(mod p), then a · (m − n) = 0(mod p) so m− n must be a multiple of p. But since all m and n are less than p, m = n. Thus a·1, a·2, . . . , a· (p−1) must be a rearrangement of 1, 2, . . . , (p−1). So modulo p, we have: $\prod_{i=1}^{p-1} i = \prod_{i=1}^{p-1} a.i = a^{p-1}\prod_{i=1}^{p-1} i$
so $a^{p-1} = 1(\mathrm{mod}\ p)$.

Theorem 2 (Fermat's Theorem Extension) If (a,m) = 1 then $a^{\Phi(m)} = 1(\mathrm{mod}\ m)$,where Φ(m) is the number of integers less than m that are relatively prime to m. The number m is not necessarily prime.

Proof: Same idea as above. Suppose Φ (m) = n. Then suppose that the n numbers less than m that are relatively prime to m are: $a_1, a_2, a_3, \ldots, a_n$. Then $a \cdot a_1, a \cdot a_2, \ldots, a \cdot a_n$ are also relatively prime to m, and must all be different, so they must just be a rearrangement of the $a_1, \ldots, a_n$ in some order. Thus: $\prod_{i=1}^{n} ai = \prod_{i=1}^{n} a.a_i = a^n\prod_{i=1}^{n} ai$
modulo m, so $a^n = 1(\mathrm{mod}\ m)$.

Theorem 3 (Chinese Remainder Theorem) *Let* p *and* q *be two numbers (not necessarily primes), but such that* (p, q) = 1. *Then if* a = b(mod p) *and* a = b(mod q) *we have* a = b(mod pq).

Proof: If a = b(mod p) then p divides (a − b). Similarly, q divides (a − b). But p and q are relatively prime, so pq divides (a − b). Consequently, a = b(mod pq). (This is a special case with only two factors of what is usually called the Chinese remainder theorem .)

3.1 Proof of the Main Result

Based on the theorems above, here is why the RSA encryption scheme works. Let p and q be two different (large) prime numbers, let $0 \le M < pq$ be a secret message1, let d be an integer (usually small) that is relatively prime to (p − 1)(q − 1), and let e be a number such that de = 1(mod (p − 1)(q − 1)). The encoded message is $C = M^e(\mathrm{mod}\ pq)$, so we need to show that the decoded message is given by$M = C^d(\mathrm{mod}\ pq)$.

Proof: Since de = 1(mod (p−1)(q −1)), de = 1+k(p−1)(q −1) for some integer k.
Thus: $C^d = M^{de} = M^{1+k(p-1)(q-1)} = M \cdot (M^{(p-1)(q-1)})^k$. If M is relatively prime to p,
then $M^{de} = M \cdot (M^{p-1})^{k(q-1)} = M \cdot (1)^{k(q-1)} = M(\text{mod } p)$
By the extension of Fermat's Theorem giving
$M^{p-1} = 1(\text{mod } p)$ followed by a multiplication of both sides byM. But ifM is not relatively prime to p, thenM is a multiple of p, so equation 1 still holds because both sides will be zero, modulo p.
By exactly the same reasoning, $M^{de} = M \cdot M^{q-1} = M(\text{mod } q)$
If we apply the Chinese remainder theorem to equations 1 and 2, we obtain the result we want: $M^{de} = M(\text{mod } pq)$.
Finally, given the integer d, we will need to be able to find another integer e such that
de = 1(mod (p−1)(q−1)). To do so we can use the extension of Fermat's theorem to get $d^{\Phi((p-1)(q-1))} = 1(\text{mod } (p-1)(q-1))$, so $d^{\Phi((p-1)(q-1))-1}(\text{mod } (p-1)(q-1))$
is a suitable value for e.

Mathematical proof of the RSA

8.1 Algorithm Key generation for RSA public-key encryption
Each entity creates an RSA public key and a corresponding private key. Each entity A should do the following:

1. Generate two large random (and distinct) primes *p* and *q*, each roughly the same size.
2. Compute *n* = *pq* and *Φ* = *(p — l)(q* — 1). (See Note 8.5.)
3. Select a random integer e, 1 < e < *Φ*, such that gcd(e, *Φ*) = 1.
4. Use the extended Euclidean algorithm to compute the unique integer *d,1 < d < Φ*, such that *ed* = 1 (mod *Φ).*
5. *A's* public key is (n, e); *A's* private key is *d.*

The integers e and *d* in RSA key generation are called the *encryption exponent* and the *decryption exponent,* respectively, while *n* is called the *modulus.*

RSA public-key encryption

B encrypts a message *m* for *A,* which *A* decrypts.

1. *Encryption. B* should do the following:

(a) Obtain A's authentic public key (n, e).

(b) Represent the message as an integer *m* in the interval [0, *n*—1].

(c) Compute c = m^e mod *n.*

(d) Send the ciphertext c to *A.*

2. Decryption. To recover plaintext *m* from c, *A* should do the following:

(a) Use the private key *d* to recover *m* = c^d mod *n.*
Proof that decryption works. Since *ed* = 1 (mod *Φ),* there exists an integer *k* such that *ed* = 1 + k *Φ.* Now, if *gcd(m,p)* = 1 then by Fermat's theorem m^{p-1} = 1 (mod *p).*

Raising both sides of this congruence to the power *k(q* — 1) and then multiplying both sides by *m* yields ml+k(P-l)(q-l) = *m* (mod *p)*

On the other hand, if gcd (m, p) = *p,* then this last congruence is again valid since each side is congruent to 0 modulo *p.* Hence, in all cases m^{ed} = *m* (mod *p).*By the same argument, m^{ed} =m(mod *q).* Finally, since *p* and *q* are distinct primes, it follows that m^{ed} = *m* (mod *n),* and, hence, $c^d = (m^e)^d$ = *m* (mod *n).*

(RSA encryption with artificially small parameters)
Key generation. Entity *A* chooses the primes *p* = 2357, *q* = 2551, and computes *n* = *pq* = 6012707 and *Φ* = *(p* —1)(q — 1) = 6007800. *A* chooses e = 3674911 and, using the extended Euclidean algorithm, finds *d* = 422191 such that *ed* = 1 (mod *<p). A's* public key is the pair (n = 6012707, e = 3674911), while A's private key is *d* = 422191. *Encryption.* To encrypt a message *m* = 5234673, *B* uses an algorithm for modular exponentiation (e.g., Algorithm 2.143) to compute c= m^e mod *n* = $5234673^{3674911}$ mod 6012707 = 3650502, and sends this to *A. Decryption.* To decrypt c, *A* computes c^d mod *n* = 3650502^{422191} mod 6012707 = 5234673. *(universal exponent)* The number λ = lcm(p — *l,q*— 1), sometimes called the *universal exponent* of *n,* may be used instead of *Φ=(p*—1)*(q*—1) in RSA key generation. Observe that λ is a proper divisor of *Φ.* Using λ can result in a smaller decryption exponent *d,* which may result in faster decryption. However, if *p* and *q* are chosen at random, then *gcd(p* —1, *q*— 1) is expected to be small, and consequently *Φ* and λ will be roughly of the same size.

Security of RSA

This subsection discusses various security issues related to RSA encryption. Various attacks which have been studied in the literature are presented, as well as appropriate measures to counteract these threats.

(i) Relation to factoring

The task faced by a passive adversary is that of recovering plaintext *m* from the corresponding ciphertext c, given the public information (n, e) of the intended receiver *A.* This is called the *RSA problem* (RSAP).There is no efficient algo-rithm known for this problem.

One possible approach which an adversary could employ to solving the RSA problem is to first factor *n*, and then compute *Φ* and *d* . Once *d* is obtained, the adversary can decrypt any ciphertext intended for *A*.
On the other hand, if an adversary could somehow compute *d*, then it could subsequently factor *n* efficiently as follows. First note that since *ed* = 1 (mod *Φ)*, there is an integer *k* such that *ed* — 1 = *k Φ*. Hence, a^{eti_1} = 1 (mod *n)* for all *a* Є Z*. Let *ed* — 1 = $2^s t$, where *t* is an odd integer. Then it can be shown that there exists an *i* Є [l,s] such that $a^{2i-1t} \neq \pm 1$ (mod *n)* anda^{2it} = 1 (mod *n)* for at least half of all $a \in Z_n$; if *a* and *i* are such integers then gcd(a—1, *n)* is a non-trivial factor of *n*. Thus the adversary simply needs to repeatedly select random *a* ЄZ* and check if an *i* Є [1, s] satisfying the above property exists; the expected number of trials before a non-trivial factor of *n* is obtained is 2. This discussion establishes the following.
The problem of computing the RSA decryption exponent *d* from the public key (n, e), and the problem of factoring *n*, are computationally equivalent. Then generating RSA keys, it is imperative that the primes *p* and *q* be selected in such a way that factoring *n* = *pq* is computationally infeasible.

(ii) Small encryption exponent e

In order to improve the efficiency of encryption, it is desirable to select a small encryption exponent e such as e = 3. A group of entities may all have the same encryption exponent e, however, each entity in the group must have its own distinct modulus. If an entity *A* wishes to send the same message *m* to three entities whose public moduli are n_1, n_2, n_3 and whose encryption exponents are e = 3, then *A* would send $Ci = m^3$ mod n; for *i* = 1,2,3. Since these moduli are most likely pairwise relatively prime, an eavesdropper observing c_1, C_2,C_3 can use Gauss's algorithm to find a solution *x*, $0 \leq x < n_1n_2n_3$, to the three congruences $x = c_1$ (mod n_1) *x* =C2(mod n_2) *x*=C3(mod n_3). Since $m^3 < n_1n_2n_3$, by the Chinese remainder theorem, it must be the case that x = m^3. Hence, by computing the integer cube root of x, the eavesdropper can recover the plaintext m. Thus a small encryption exponent such as e = 3 should not be used if the same message, or even the same message with known variations, is sent to many entities. Alternatively, to prevent against such an attack, a pseudorandomly generated bitstring of appropriate length should be appended to the plaintext message prior to encryption; the pseudorandom bit-string should be independently generated for each encryption. This process is sometimes referred to as *salting* the message.
Small encryption exponents are also a problem for small messages m, because if m< $n^1/^e$, then m can be recovered from the ciphertext c = m^e mod *n* simply by computing the integer e^{th} root of c; salting plaintext messages also circumvents this problem.

(iii) Forward search attack

If the message space is small or predictable, an adversary can decrypt a ciphertext c by simply encrypting all possible plaintext messages until c is obtained. Salting the message as described above is one simple method of preventing such an attack.

(iv) Small decryption exponent d

As was the case with the encryption exponent e, it may seem desirable to select a small decryption exponent *d* in order to improve the efficiency of decryption.^{x} However, if gcd *(p*—1, *q*—1) is small, as is typically the case, and if *d* has up to approximately one-quarter as many bits as the modulus *n*, then there is an efficient algorithm (referenced on page 313) for computing *d* from the public information (n, e). This algorithm cannot be extended to the case where *d* is approximately the same size as *n*. Hence, to avoid this attack, the decryption exponent *d* should be roughly the same size as *n*.

(v) Multiplicative properties

Let m_1 and m_2 be two plaintext messages, and let C_1 and C_2 be their respective RSA encryptions. Observe that $(m_1m_2)^e = m_1^e m_2^e$ = C1C2 (mod *n)*.In other words, the ciphertext corresponding to the plaintext $m = m_1m_2$ mod *n* is c = c_1c_2 mod n; this is sometimes referred to as the *homomorphic property* of RSA. This observation leads to the following *adaptive chosen-ciphertext attack* on RSA encryption.

Suppose that an active adversary wishes to decrypt a particular ciphertext c = m^e mod *n* intended for *A*. Suppose also that *A* will decrypt arbitrary ciphertext for the adversary, other than c itself. The adversary can conceal c by selecting a random integer *x* Є Z_n* and computing $c^- = cx^e$ mod *n*. Upon presentation of c^-, *A* will compute for the adversary $m^- = (c^-)^d$ mod n. Since $m^- = (c^-)^d = c^d (x^e)^d$ = mx (mod n), the adversary can then compute $m = m^- x^{-1}$ mod *n*.
This adaptive chosen-ciphertext attack should be circumvented in practice by imposing some structural constraints onplaintext messages. If a ciphertext c is decrypted to a message not possessing this structure, then c is rejected by the decryptor as being fraudulent. Now, if a plaintext message *m* has this (carefully chosen) structure, then with high probability *mx* mod *n* will not for *x*Є Z_n*. Thus the adaptive chosen-ciphertext attack described in the previous paragraph will fail because *A* will not decrypt c for the adversary.

(vi) Common modulus attack

The following discussion demonstrates why it is imperative for each entity to choose its own RSA modulus *n.* It is sometimes suggested that a central trusted authority should select a single RSA modulus *n,* and then distribute a distinct encryption/decryption exponent pair (e_i, d_i) to each entity in a network. However, as shown in (i) above, knowledge of any (e_i, d_i) pair allows for the factorization of the modulus *n,* and hence any entity could subsequently determine the decryption exponents of all other entities in the network. Also, if a single message were encrypted and sent to two or more entities in the network, then there is a technique by which an eavesdropper (any entity not in the network) could recover the message with high probability using only publicly available information.

(vii) Cycling attacks

Let $c = m^e$ mod n be a ciphertext. Let k be a positive integer such that $c^{ek}=c$(mod n); since encryption is a permutation on the message space {0,1,... , n — 1} such an integer k must exist. For the same reason it must be the case that $c^{k-1} = m$ (mod n). This observation leads to the following *cycling attack* on RSA encryption. An adversary computes c^e mod n, c^{e2} mod n, c^{e3} mod n,... until c is obtained for the first time. If c^{ek} mod n =c, then the previous number in the cycle, namely c^{ek-1} mod n, is equal to the plaintext *m. A generalized cycling attack* is to find the smallest positive integer u such that f =gcd(c^e— *c,n)* > 1. If c^e=c (mod p) and $c^e \neq c$ (mod q) then $f = p$. Similarly, if
$c^e \neq c$ (mod p) and $c^e = c$ (mod q (8.2) then 1 = q. In either case, n has been factored, and the adversary can recover d and then m. On the other hand, if both c =c (mod p) and c =c (mod q),(8.3) then $f = n$ and $c^e = c$ (mod n). In fact, u must be the smallest positive integer k for which $c^e = c$ (mod n). In this case, the basic cycling attack has succeeded and so 7n = c mod n can be computed efficiently. The generalized cycling attack usually terminates before the cycling attack does. For this reason, the generalized cycling attack can be viewed as being essentially an algorithm for factoring n. Since factoring n is assumed to be intractable, these cycling attacks do not pose a threat to the security of RSA encryption.

(viii) Message concealing

A plaintext message m, 0 < m < n—1, in the RSA public-key encryption scheme is said to be *unconcealed* if it encrypts to itself; that is, $m^e = m$ (mod n). There are always some messages which are unconcealed (for example m = 0, m = 1, and m = n—1). In fact, the number of unconcealed messages is exactly

[1 + gcd(e—*l* , *p* —1)] • [1 + gcd(e—1,*q*—1)].

Since *e — l,p—l* and *q — 1* are all even, the number of unconcealed messages is always at least 9. If p and q are random primes, and if e is chosen at random (or if e is chosen to be a small number such as e = 3or e = 2^{16} + l = 65537), then the proportion of messages which are unconcealed by RSA encryption will, in general, be negligibly small, and hence unconcealed messages do not pose a threat to the security of RSA encryption in practice.

RSA encryption in practice

There are numerous ways of speeding up RSA encryption and decryption in software and hardware implementations. Some of these techniques are covered in Chapter 14, including fast modular multiplication , fast modular exponentiation, and the use of the Chinese remainder theorem for faster decryption. Even with these improvements, RSA encryption/decryption is substantially slower than the commonly used symmetric-key encryption algorithms such as DES. In practice, RSA encryption is most commonly used for the transport of symmetric-key encryption algorithm keys and for the encryption of small data items.
The RSA cryptosystem has been patented in the U.S. and Canada. Several standards organizations have written, or are in the process of writing, standards that address the use of the RSA cryptosystem for encryption, digital signatures, and key establishment. For discussion of patent and standards issues related to RSA.
Note *(recommended size of modulus)* Given the latest progress in algorithms for factoring integers , a 512-bit modulus n provides only marginal security from concerted attack. As of 1996, in order to foil the powerful quadratic sieve and number field sieve factoring algorithms, a modulus n of at least 768 bits is recommended. For long-term security, 1024-bit or larger moduli should be used.
(selectingprimes)
(i)The primes p and q should be selected so that factoring $n = pq$ is computationally infeasible. The major restriction *onp* and q in order to avoid the elliptic curve factoring algorithm is that p and q should be about the same bitlength, and sufficiently large. For example, if a 1024-bit modulus n is to be used, then each of p and q should be about 512 bits in lengt.
(ii) Another restriction on the primes p and q is that the difference p—q should not be too small. If p—q is small, then $p \approx q$ and hence $p \approx \sqrt{n}$. Thus, n could be factored efficiently simply by trial division by all odd integers close to $\sqrt{n}$. *If* p and q are chosen at random, then p—q will be appropriately large with overwhelming probability.
(iii) In addition to these restrictions, many authors have recommended that p and q be strong primes. A prime p is said to be a *strong prime* if the following three conditions are satisfied:
(a) p—1 has a large prime factor, denoted r;
(b) p + 1 has a large prime factor; and

(c) *r*—1 has a large prime factor.

The reasonfor condition (a) is to foil Pollard's *p*—1 factoring algorithm which is efficient only if *n* has a prime factor *p* such that *p*—1 is smooth. Condition (b) foils the *p* + 1 factoring algorithm mentioned, which is efficient only if *n* has a prime factor *p* such that *p* + 1 is smooth. Finally, condition (c) ensures that the cycling attacks will fail.

If the prime *p* is randomly chosen and is sufficiently large, then both *p—1* and *p+1* can be expected to have large prime factors. In any case, while strong primes protect against the *p—1* and *p+1* factoring algorithms, they do not protect against their generalization. The latter is successful in factoring *n* if a randomly chosen number of the same size as *p* has only small prime factors. Additionally, it has been shown that the chances of a cycling attack succeeding are negligible if p and *q* are randomly chosen. Thus, strong primes offer little protection beyond that offered by random primes. Given the current state of knowledge of factoring algorithms, there is no compelling reason for requiring the use of strong primes in RSA key generation. On the other hand, they are no less secure than random primes, and require only minimal additional running time to compute; thus there is little real additional cost in using them.

(small encryption exponents)

If the encryption exponent e is chosen at random, then RSA encryption using the repeated square-and-multiply algorithm (Algorithm 2.143) takes *k* modular squarings and an expected *k/2* (less with optimizations) modular multiplications, where *k* is the bitlength of the modulus *n*. Encryption can be sped up by selecting e to be small and/or by selecting e with a small number of 1 's in its binary representation. The encryption exponent e = 3 is commonly used in practice; in this case, it is necessary that neither *p*—1 nor *q*—1 be divisible by 3. This results in a very fast encryp-tion operation since encryption only requires 1 modular multiplication and 1 modular squaring. Another encryption exponent used in practice is e = 2^{16} + 1 = 65537. This number has only two 1's in its binary representation, and so encryption using the repeated square-and-multiply algorithm requires only 16 modular squarings and 1 modular multiplication. The encryption exponent e = 2^{16} + 1 has the advantage over e = 3 in that it resists the kind of attack , since it is unlikely the same message will be sent to 2^{16}+1 recipients.

Conclusion

In this paper the proposal is to modify the DES algorithm to improve the encryption information exchanged between any two nodes on the network. In its present form it can be broken. By the proposed modification the purpose is to enhance the time to break so that with the timestamp for the transfer of the frame the information would have already reached the destination and action accordingly taken as needed. This enhances the performance of the DES algorithm to a large extent. It is very clear with the proof related to RSA with regard to the key generated given above. For future research on this, the inclusion of the knowledge of some of the other theorems of number theory can be use to further enhance the performance of the DES algorithm.

References

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

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

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

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

[5] 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.

[6] 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.

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

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

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

[10] J. Orlin Grabbe, The DES Algorithm Illustrated.