Differential Cryptanalytic Attacks of 3-Round DES*

Wenbo Yang
Email: http://solrex.org
The State Key Laboratory of Information Security
Chinese Academy of Sciences, Beijing, P.R. China

Oct 25, 2007

Abstract

This lab report decribes how to do differential cryptanalytic attacking to 3-round DES(Data Encryption Standard[1]). We provide the whole process and source code to attacking 3-round DES with DC.

1 Introduction

Differential cryptanalysis is a chosen plaintext attack, which tracks the difference of plaintext pairs across all the rounds to find useful information of the subkey. The first using of it to attack DES is introduced by Biham and Sharmir[2] at 1990. The full attacking method to reduced-round DES can be found in [3].

Here we will not provide the background of differential cryptanalysis of DES, which you can refer to [3] for more information. The main content of this L.R. is about the implementation details and the results of attacking on DES with 3 rounds.

Partly following [2], we introduce the notations bellow:

n_x**:** An hexadecimal number is denoted by a subscript x (i.e., $10_x = 16$).

$X_i, X_i^*, X_i^{'}$**:** At any intermediate point during the encryption of pairs of messages, X_i and X_i^* are the correspoding intermediate values after the i round executions of the algorithm, and $X_i^{'}$ is defined to be $X_i^{'} = X_i \oplus X_i^*$.

$(L_i, R_i), (L_i^*, R_i^*), (L_i^{'}, R_i^{'})$**:** The left and right halves of the intermediate value after the i round execution are denoted by L_i and R_i respectively, and $X_i = L_i \parallel R_i$. So as to $(L_i^*, R_i^*), (L_i^{'}, R_i^{'})$.

K_i**:** The subkey used in i round execution is denoted by K_i.

$F(X, K_i)$**:** The F function.

$P(X)$**:** The P permutation.

$E(X)$**:** The E expansion.

$IP(X)$**:** The initial permutation. The existence of IP and IP^{-1} is ignored in this L.R.

P**:** The plaintext(after the known initial permutation IP) is denoted by P. P^* is the other plaintext in the pair and $P^{'} = P \oplus P^*$ is the plaintexts XOR.

T**:** The ciphertexts of the corresponding plaintexts P, P^*(before the inverse initial permutation IP^{-1}) are denoted by T and T^*. $T^{'} = T \oplus T^*$ is the ciphertexts XOR.

*This is a lab report. If you want the LaTeX source code of this article, please refer http://share.solrex.org/mywork/dc_des_lr.tar.gz

Sn**:** The S boxes.

$Sn_{Ei}, Sn_{Ki}, Sn_{Ii}, Sn_{Oi}$**:** The input of Sn in round i is denoted by Sn_{Ii} for $i \in \{1_x, 2_x, \ldots, f_x\}$. The output of Sn in round i is denoted by Sn_{Oi}. The value of the six subkey bits entering the S box Sn is denoted by Sn_{Ki} and the value of the six input bits of the expanded data ($E(R_i)$) which are XORed with Sn_{Ki} to form Sn_{Ei}. The S box number n and the round marker i are optional. For example $S1_{E1}$ denotes the first six bits of $E(R_1)$. $S1_{K1}$ denotes the first six bits of the subkey $K1$. $S1_{I1}$ denotes the input of the S box S1 which is $S1_{I1} = S1_{E1} \oplus S1_{K1}$. $S1_{O1}$ denotes the output of S1 which is $S1_{O1} = S1(S1_{I1})$.

$S_{Ei}, S_{Ki}, S_{Ii}, S_{Oi}$**:** $S_{Ei} = S1_{Ei} \parallel S2_{Ei} \parallel \cdots \parallel S8_{Ei}$, so as the other three.

2 3-rounds DES Attacking

In DES reduced to 3 rounds, we can deduce following equations which is the foundation of 3-R attack:

$$\begin{aligned} R_3 &= L_2 \oplus F(R_2, K_3) \\ &= R_1 \oplus F(R_2, K_3) \\ &= L_0 \oplus F(R_0, K_1) \oplus F(R_2, K_3) \end{aligned} \tag{1}$$

and samely

$$R_3^* = L_0^* \oplus F(R_0^*, K_1^*) \oplus F(R_2^*, K_3^*) \tag{2}$$

So that

$$\begin{aligned} R_3^{'} &= R_3 \oplus R_3^* \\ &= L_0^{'} \oplus f(R_0, K_1) \oplus f(R_0^*, K_1) \oplus f(R_2, K_3) \oplus f(R_2^*, K_3) \end{aligned} \tag{3}$$

We now choose plaintext pair (L_0, R_0) and (L_0^*, R_0^*) to satisfy $R_0 = R_0^*$, i.e. $R_0^{'} = R_0 \oplus R_0^* = 00\cdots0$, then we obtain

$$R_3^{'} = L_0^{'} \oplus F(R_2, K_3) \oplus F(R_2^*, K_3)$$

i.e.

$$F(R_2, K_3) \oplus F(R_2^*, K_3) = R_3^{'} \oplus L_0^{'} \tag{4}$$

Since

$$F(R_2, K_3) = P(S_{O3}), F(R_2^*, K_3) = P(S_{O3}^*)$$

we get

$$P(S_{O3}) \oplus P(S_{O3}^*) = R_3^{'} \oplus L_0^{'}$$

Because XOR stays valid after $P(X)$, and $P(X)$ is invertible, so

$$S_{O3}^{'} = S_{O3} \oplus S_{O3}^* = P^{-1}(R_3^{'} \oplus L_0^{'}) \tag{5}$$

hence we get S-boxes output XOR $S_{O3}^{'}$ of the 3rd round.

We also have

$$R_2 = L_3, R_2^* = L_3^*$$

so the S-boxes input XOR $S_{I3}^{'}$ of the 3rd round can be deduced by

$$\begin{aligned} S_{I3}^{'} &= S_{I3} \oplus S_{I3}^* \\ &= E(R_2) \oplus E(R_2^*) \\ &= E(L_3) \oplus E(L_3^*) \end{aligned} \tag{6}$$

Here we got the S-boxes input and output XOR of the 3rd round, then we can use them to compute the subkey K_3 with method introduced in Example 7 of [2].

Example[1] Assume we have 3 pairs of chosen plaintext and ciphertext encrypted with the same key as bellow(From [4], encryption using 3 round, no initial and final permutation, no last interchanging of left and right):

Plaintext	Ciphtertext
748502cd38451097	03C70306D8A09F10
3874756438451097	78560a0960e6d4cb
486911026acdff31	78560a0960e6d4cb
375bd31f6acdff31	134f7915ac253457
357418da013fec86	d8a31b2f28bbc5cf
12549847013fec86	0f317ac2b23cb944

Step 1: Get input of S box in 3rd round, which we get:

Pair	Input of S_{I3}
1	00000000 01111110 00001110 10000000 01101000 00001100
1	10111111 00000010 10101100 00000101 01000000 01010010
2	10100000 10111111 11110100 00010101 00000010 11110110
2	10001010 01101010 01011110 10111111 00101000 10101010
3	11101111 00010101 00000110 10001111 01101001 01011111
3	00000101 11101001 10100010 10111111 01010110 00000100

Step 2: Get input and output difference of S box in 3rd round, which we get:

Pair	Input Diff of S_{I3}
1	10111111 01111100 10100010 10000101 00101000 01011110
2	00101010 11010101 10101010 10101010 00101010 01011100
3	11101010 11111100 10100100 00110000 00111111 01011011

Pair	Output Diff of S_{I3}
1	10010110 01011101 01011011 01100111
2	10011100 10011100 00011111 01010110
3	11010101 01110101 11011011 00101011

Step 3: Calculate the subkey with J matrix introduce in [4]:

S box	J Matrix
1	1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
	0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0
	0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 3

[1] We copy this example from [4] because it is easy to check whether our algorithm is right or not. You can choose other pairs of course.

	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
2	0 0 0 1 0 3 0 0 1 0 0 1 0 0 0 0
	0 1 0 0 0 2 0 0 0 0 0 0 1 0 0 0
	0 0 0 0 0 1 0 0 1 0 1 0 0 0 1 0
	0 0 1 1 0 0 0 0 1 0 1 0 2 0 0 0
3	0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0
	0 0 0 3 0 0 0 0 0 0 0 0 0 0 1 1
	0 2 0 0 0 0 0 0 0 0 0 0 1 1 0 0
	0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0
4	3 1 0 0 0 0 0 0 0 0 2 2 0 0 0 0
	0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 1
	1 1 1 0 1 0 0 0 0 1 1 1 0 0 1 0
	00 0 0 0 1 1 0 0 0 0 0 0 0 0 2 1
5	0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
	0 0 0 0 2 0 0 0 3 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 2 0 0 0 0 0 0 1 0 0 0 0 2 0
6	1 0 0 1 1 0 0 3 0 0 0 0 1 0 0 1
	0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0
	1 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0
7	0 0 2 1 0 1 0 3 0 0 0 1 1 0 0 0
	0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1
	0 0 2 0 0 0 2 0 0 0 0 1 2 1 1 0
	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1
8	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1
	0 3 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0

So the i which $J[i] = 3$ gives us the subkey(48 bits) of the 3rd round:

```
10111100 01010100 11000000
01100000 01110001 11110001
```

Step 4: Generate the key schedule intermediate halves(56 bits) with subkey(48 bits) by reverse PC-2 permute[1]:

```
10001101 01100100 00100001 0100
00111100 00000100 11010001 1001
```

Where bits `9 18 22 25 35 38 43 54` were undetermined.

Then we got the key(64 bits without check sum) by right shift 4 bits and reverse PC-1 permute[1]:

```
00011010 01100010 01001000 10001000
01010010 00000000 11101100 01000110
```

Where bits: `26 19 52 57 46 22 45 55` were undetermined.

Here we got a 64 bits key with 8 bits undetermined. So it is easily to implement a exhaustive key search which requires only $2^8 = 256$ encryptions.

3 Conclusion

In this L.R, we implemented a real DC attack to 3 round DES. Base on this method, attacking more round DES can be realized by the intruction of [2].

4 C Source Code

Bellow is the entire sourcode of DES codec and DC attacking to 3-Round DES. If you want an ASCII C source file copy, please refer http://share.solrex.org/mywork/dc_des_lr.tar.gz.

```
/*
 *********************************************************************
 *       Filename:  des.c
 *
 *    Description:  DES encryption and decryption implementation, with
 *                  DC(differential cryptanalysis) attacking to 3-round DES.
 *
 *        Version:  1.0
 *        Created:  10/25/2007 04:50:01 PM CST
 *       Revision:  none
 *       Compiler:  gcc 4.1.3
 *
 *       The DES de/encryption algorithm
 *         Author: Jim Gillogly, May 1977
 *       Modified: 8/84 by Jim Gillogly and Lauren Weinstein to compile with
 *                 post-1977 C compilers and systems
 *
 *       Modified and added DC attacking algorithm, 10/25/2007
 *         Author:  Wenbo Yang <http://solrex.cn>
 *        Company:  the State Key Laboratory of Information Security
 *                  CAS, Beijing, China.
 *
 * This program is now officially in the public domain, and is available for
 * any non-profit use as long as the authorship line is retained.
 *********************************************************************
 */

/* Permutation algorithm:
 *        The permutation is defined by its effect on each of the 16 nibbles
 *        of the 64-bit input.  For each nibble we give an 8-byte bit array
 *        that has the bits in the input nibble distributed correctly.  The
 *        complete permutation involves ORing the 16 sets of 8 bytes designated
 *        by the 16 input nibbles.  Uses 16*16*8 = 2K bytes of storage for
 *        each 64-bit permutation.  32-bit permutations (P) and expansion (E)
 *        are done similarly, but using bytes instead of nibbles.
 *        Should be able to use long ints, adding the masks, at a
 *        later pass.  Tradeoff: can speed 64-bit perms up at cost of slowing
 *        down expansion or contraction operations by using 8K tables here and
 *        decreasing the size of the other tables.
 * The compressions are pre-computed in 12-bit chunks, combining 2 of the
 *        6->4 bit compressions.
 * The key schedule is also precomputed. */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* structure to store plaintext and ciphertext blocks in pair(DC) */
struct text_pair{
  unsigned char first[8];
  unsigned char second[8];
};
/* structure to store S box input in pair(DC) */
struct sin_pair{
  unsigned char first[6];
  unsigned char second[6];
```

```
};

/* structure to store S box output in pair(DC) */
struct sout_pair{
  unsigned char first[4];
  unsigned char second[4];
};

/* Some useful macros used as function arguments(DC) */
enum {
  NO_IP = 0,
  HAVE_IP = 1,
  NO_LAST_SWAP = 0,
  LAST_SWAP = 1,
  BIN_M = 0,
  HEX_M = 1,
  DEC_M = 2,
};

/* identical, final and  inital permutations */
unsigned char Iperm[16][16][8];
unsigned char iperm[16][16][8];
unsigned char fperm[16][16][8];

unsigned char s[4][4096];      /* S1 thru S8 precomputed */
unsigned char p32[4][256][4];  /* for permuting 32-bit f output */
unsigned char rp32[4][256][4]; /* for reverse permuting 32-bit f output */
unsigned char kn[16][6];       /* key selections */
unsigned char J[8][64];        /* J matrix to determine key(DC) */

/* Const string for printing binary numers. */
static const unsigned char *bin[16] = {
  "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
  "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"};

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

static const unsigned char ip[64] = {  /* initial permutation P */
  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 };

static const unsigned char fp[64] = {  /* final permutation F */
  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 };

static const unsigned char pc1[56] = { /* PC-1 (key) */
  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 };

static const unsigned char rpc1[64] = {/* reverse PC-1 (key, DC) */
   8, 16, 24, 56, 52, 44, 36,  0,
   7, 15, 23, 55, 51, 43, 35,  0,
   6, 14, 22, 54, 50, 42, 34,  0,
   5, 13, 21, 53, 49, 41, 33,  0,
   4, 12, 20, 28, 48, 40, 32,  0,
   3, 11, 19, 27, 47, 39, 31,  0,
   2, 10, 18, 26, 46, 38, 30,  0,
   1, 9,  17, 25, 45, 37, 29,  0 };

static const unsigned char totrot[16] = {/* number left rotations of PC-1 */
  1,2,4,6,8,10,12,14,15,17,19,21,23,25,27,28 };

static unsigned char pc1m[56];     /* place to modify pc1 into */
static unsigned char pcr[56];      /* place to rotate pc1 into */

static const unsigned char pc2[48] = { /* PC-2 (key)  */
  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 };

static const unsigned char rpc2[56] = { /* reverse PC-2 (key, DC) */
   5, 24,  7, 16,  6, 10, 20, 18,
   0, 12,  3, 15, 23,  1,  9, 19,
   2,  0, 14, 22, 11,  0, 13,  4,
   0, 17, 21,  8, 47, 31, 27, 48,
  35, 41,  0, 46, 28,  0, 39, 32,
  25, 44,  0, 37, 34, 43, 29, 36,
  38, 45, 33, 26, 42,  0, 30, 40 };

static const unsigned char si[8][64] = {   /* 48->32 bit compression tables*/
         /* S[1]       */
  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,
         /* S[2]       */
  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,
         /* S[3]       */
  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,
         /* S[4]       */
   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,
         /* S[5]       */
   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,
         /* S[6]       */
  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,
```

```
          /* S[7]       */
   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,
          /* S[8]       */
  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  };

static const unsigned char p32i[32] = { /* P in F function */
  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 };

static const unsigned char rp32i[32] = {   /* reverse P in F(DC) */
   9, 17, 23, 31,
  13, 28,  2, 18,
  24, 16, 30,  6,
  26, 20, 10,  1,
   8, 14, 25,  3,
   4, 29, 11, 19,
  32, 12, 22,  7,
   5, 27, 15, 21 };

static int bytebit[8] = {      /* bit 0 is left-most in byte */
  0x80,0x40,0x20,0x10,0x8,0x4,0x2,0x1 };

static int nibblebit[4] = { 0x8,0x4,0x2,0x1 };

/* permute inblock with perm, result into outblock,64 bits */
void permute(unsigned char *inblock,
             unsigned char perm[16][16][8],/* 2K bytes defining perm.  */
             unsigned char *outblock)
{
  register int i,j;
  register char *ib, *ob, *p, *q;

  for(i=0, ob=outblock; i<8; i++) *ob++ = 0;   /* clear output block    */
  ib = inblock;
  for (j = 0; j < 16; j += 2, ib++) {  /* for each input nibble  */
    ob = outblock;
    p = perm[j][(*ib >> 4) & 0xf];
    q = perm[j + 1][*ib & 0xf];
    for (i = 0; i < 8; i++)    /* and each output byte  */
      *ob++ |= *p++ | *q++;    /* OR the masks together*/
  }
}

/* 32-bit permutation at end of the f crypto function*/
void perm32(unsigned char inblock[4],
            unsigned char perm[4][256][4],
            unsigned char outblock[4])
{
  register int j;
  register char *ib, *ob, *q;

  ob = outblock;       /* clear output block */
  *ob++ = 0; *ob++ = 0; *ob++ = 0; *ob++ = 0;
  ib = inblock;        /* ptr to 1st byte of input */
  for (j=0; j<4; j++, ib++) {  /* for each input byte */
    q = perm[j][*ib & 0xff];
    ob = outblock;    /* and each output byte */
    *ob++ |= *q++;    /* OR the 16 masks together */
    *ob++ |= *q++;
```

```
    *ob++ |= *q++;
    *ob++ |= *q++;
  }
}

/* 32 to 48 bits with E oper, right is 32, bigright 48  */
void expand(unsigned char right[4], unsigned char bigright[6])
{
  register char *bb, *r, r0, r1, r2, r3;

  bb = bigright;
  r = right; r0 = *r++; r1 = *r++; r2 = *r++; r3 = *r++;
  *bb++ = ((r3 & 0x1) << 7) |  /* 32             */
    ((r0 & 0xf8) >> 1) |       /* 1 2 3 4 5      */
    ((r0 & 0x18) >> 3);        /* 4 5            */
  *bb++ = ((r0 & 0x7) << 5) |  /* 6 7 8          */
    ((r1 & 0x80) >> 3) |       /* 9              */
    ((r0 & 0x1) << 3) |        /* 8              */
    ((r1 & 0xe0) >> 5);        /* 9 10 11        */
  *bb++ = ((r1 & 0x18) << 3) | /* 12 13          */
    ((r1 & 0x1f) << 1) |       /* 12 13 14 15 16 */
    ((r2 & 0x80) >> 7);        /* 17             */
  *bb++ = ((r1 & 0x1) << 7) |  /* 16             */
    ((r2 & 0xf8) >> 1) |       /* 17 18 19 20 21 */
    ((r2 & 0x18) >> 3);        /* 20 21          */
  *bb++ = ((r2 & 0x7) << 5) |  /* 22 23 24       */
    ((r3 & 0x80) >> 3) |       /* 25             */
    ((r2 & 0x1) << 3) |        /* 24             */
    ((r3 & 0xe0) >> 5);        /* 25 26 27       */
  *bb++ = ((r3 & 0x18) << 3) | /* 28 29          */
    ((r3 & 0x1f) << 1) |       /* 28 29 30 31 32 */
    ((r0 & 0x80) >> 7);        /* 1              */
}

/* contract f from 48 to 32 bits, using 12-bit pieces into bytes */
void contract(unsigned char in48[6], unsigned char out32[4])
{
  register char *c;
  register char *i;
  register int i0, i1, i2, i3, i4, i5;

  i = in48;
  i0 = *i++; i1 = *i++; i2 = *i++; i3 = *i++; i4 = *i++; i5 = *i++;
  c = out32;      /* do output a byte at a time   */
  *c++ = s[0][0xfff & ((i0 << 4) | ((i1 >> 4) & 0xf ))];
  *c++ = s[1][0xfff & ((i1 << 8) | ( i2  & 0xff ))];
  *c++ = s[2][0xfff & ((i3 << 4) | ((i4 >> 4) & 0xf ))];
  *c++ = s[3][0xfff & ((i4 << 8) | ( i5  & 0xff ))];
}

/* critical cryptographic trans, num: index number of this iter */
void f(unsigned char right[4],int num, unsigned char fret[4])
{
  register char *kb, *rb, *bb;  /* ptr to key selection &c  */
  char bigright[6];    /* right expanded to 48 bits  */
  char result[6];      /* expand(R) XOR keyselect[num] */
  char preout[4];      /* result of 32-bit permutation */

  kb = kn[num];        /* fast version of iteration  */
  bb = bigright;
  rb = result;
  expand(right,bb);    /* expand to 48 bits    */
  *rb++ = *bb++ ^ *kb++;       /* expanded R XOR chunk of key  */
  *rb++ = *bb++ ^ *kb++;
  *rb++ = *bb++ ^ *kb++;
  *rb++ = *bb++ ^ *kb++;
  *rb++ = *bb++ ^ *kb++;
  *rb++ = *bb++ ^ *kb++;
  contract(result,preout);     /* use S fns to get 32 bits  */
  perm32(preout, p32, fret);   /* and do final 32-bit perm  */
}
```

```
/* 1 churning operation, num: i.e. the num-th one */
void iter(int num, unsigned char inblock[8], unsigned char outblock[8])
{
  char fret[4];        /* return from f(R[i-1],key)  */
  register char *ib, *ob, *fb;

  ob = outblock; ib = &inblock[4];
  f(ib, num, fret);    /* the primary transformation   */
  *ob++ = *ib++;       /* L[i] = R[i-1]    */
  *ob++ = *ib++;
  *ob++ = *ib++;
  *ob++ = *ib++;
  ib = inblock; fb = fret;  /* R[i]=L[i] XOR f(R[i-1],key)  */
  *ob++ = *ib++ ^ *fb++;
  *ob++ = *ib++ ^ *fb++;
  *ob++ = *ib++ ^ *fb++;
  *ob++ = *ib++ ^ *fb++;
}

/* initialize a perm array, p: 64-bit, either init or final */
void perminit(unsigned char perm[16][16][8], const char p[64])
{
  register int l, j, k;
  int i,m;

  for (i=0; i<16; i++)         /* each input nibble position */
    for (j=0; j<16; j++)       /* all possible input nibbles */
      for (k=0; k<8; k++)      /* each byte of the mask */
        perm[i][j][k]=0;       /* clear permutation array */
  for (i=0; i<16; i++)         /* each input nibble position */
    for (j = 0; j < 16; j++)   /* each possible input nibble */
      for (k = 0; k < 64; k++){/* each output bit position  */
        l = p[k] - 1;          /* where does this bit come from*/
        if ((l >> 2) != i)     /* does it come from input posn?*/
          continue;            /* if not, bit k is 0 */
        if (!(j & nibblebit[l & 3]))
          continue;            /* any such bit in input? */
        m = k & 0x7;           /* which bit is this in the byte*/
        perm[i][j][k>>3] |= bytebit[m];
      }
}

/* 64 bits (will use only 56)   */
void kinit(unsigned char key[8])        /* initialize key schedule array*/
{
  register int i,j,l;
  int m;

  for (j=0; j<56; j++){        /* convert pc1 to bits of key   */
    l=pc1[j]-1;                /* integer bit location    */
    m = l & 07;                /* find bit      */
  /* find which key byte l is in and which bit of that byte, and store 1-bit
   * result */
    pc1m[j]=(key[l>>3] & bytebit[m]) ? 1 : 0;
  }
  for (i=0; i<16; i++)         /* for each key sched section */
    for (j=0; j<6; j++)        /* and each byte of the kn */
      kn[i][j]=0;              /* clear it for accumulation */
  for (i=0; i<16; i++){        /* key chunk for each iteration */
    for (j=0; j<56; j++)       /* rotate pc1 the right amount */
      /* rotate left and right halves independently */
      pcr[j] = pc1m[(l=j+totrot[i])<(j<28? 28 : 56) ? l: l-28];
    for (j=0; j<48; j++)       /* select bits individually  */
      if (pcr[pc2[j]-1]){      /* check bit that goes to kn[j] */
        l= j & 0x7;
        kn[i][j>>3] |= bytebit[l]; /* mask it in if it's there */
      }
  }
}
```

```
/* 1 compression value for sinit*/
int getcomp(int k,int v)
{
  register int i,j;            /* correspond to i and j in FIPS */

  i=((v&0x20)>>4)|(v&1);        /* first and last bits make row */
  j=(v&0x1f)>>1;                /* middle 4 bits are column  */
  return (int) si[k][(i<<4)+j];/* result is ith row, jth col */
}

/* initialize s1-s8 arrays    */
void sinit()
{
  register int i,j;

  for (i=0; i<4; i++)       /* each 12-bit position */
    for (j=0; j<4096; j++)  /* each possible 12-bit value */
      /* store 2 compressions per char*/
      s[i][j]=(getcomp(i*2,j>>6)<<4) | (0xf&getcomp(i*2+1,j&0x3f));
}

/* initialize 32-bit permutation */
void p32init(unsigned char perm[4][256][4], const unsigned char p[32])
{
  register int l, j, k;
  int i,m;

  for (i=0; i<4; i++)       /* each input byte position */
    for (j=0; j<256; j++)   /* all possible input bytes */
      for (k=0; k<4; k++)   /* each byte of the mask */
        perm[i][j][k]=0;    /* clear permutation array */
  for (i=0; i<4; i++)       /* each input byte position */
    for (j=0; j<256; j++)   /* each possible input byte */
      for (k=0; k<32; k++){    /* each output bit position */
        l=p[k]-1;              /* invert this bit (0-31) */
        if ((l>>3)!=i)         /* does it come from input posn?*/
          continue;            /* if not, bit k is 0 */
        if (!(j&bytebit[l&0x7]))
          continue;            /* any such bit in input? */
        m = k & 0x7;           /* which bit is it? */
        perm[i][j][k>>3] |= bytebit[m];
      }
}

/* initialize all des arrays  */
void desinit(unsigned char *key)
{
  perminit(Iperm,I);        /* identical permutation */
  perminit(iperm,ip);       /* initial permutation */
  perminit(fperm,fp);       /* final permutation */
  kinit(key);               /* key schedule */
  sinit();                  /* compression functions */
  p32init(p32, p32i);       /* 32-bit permutation in f */
  p32init(rp32, rp32i);     /* reverse 32-bit permutation in f */
}

/* encrypt 64-bit inblock  */
void endes(unsigned char *inblock, unsigned char *outblock,
           int round, int have_ip, int last_swap)
{
  char iters[17][8];        /* workspace for each iteration */
  char swap[8];             /* place to interchange L and R */
  register int i;
  register char *s;

  if(have_ip == HAVE_IP)
    permute(inblock,iperm,iters[0]);   /* apply initial permutation */
  else
    permute(inblock,Iperm,iters[0]);   /* don't apply initial permutation */

  /* don't re-copy to save space  */
```

```
  for (i=0; i<round; i++)      /* 16 churning operations  */
    iter(i,iters[i],iters[i+1]);

  s = swap;
  if(last_swap == LAST_SWAP) {              /* do last swap left and right */
    /* interchange left and right. */
    for(i=4;i<8;i++) *s++ = iters[round][i];
    for(i=0;i<4;i++) *s++ = iters[round][i];
  } else {                     /* don't do last swap left and right */
    for(i=0;i<8;i++) *s++ = iters[round][i];
  }

  if(have_ip == HAVE_IP)
    permute(swap,fperm,outblock);   /* apply final permutation  */
  else
    permute(swap,Iperm,outblock);   /* don't apply final permutation  */
}

/* decrypt 64-bit inblock  */
void dedes(unsigned char *inblock, unsigned char *outblock)
{
  char iters[17][8];    /* workspace for each iteration */
  char swap[8];        /* place to interchange L and R */
  register int i;
  register char *s, *t;

  permute(inblock,iperm,iters[0]); /* apply initial permutation  */
  for (i=0; i<16; i++)             /* 16 churning operations  */
    iter(15-i,iters[i],iters[i+1]);
  /* reverse order from encrypting */
  s = swap; t = &iters[16][4];     /* interchange left    */
  *s++ = *t++; *s++ = *t++; *s++ = *t++; *s++ = *t++;
  t = &iters[16][0];               /* and right      */
  *s++ = *t++; *s++ = *t++; *s++ = *t++; *s++ = *t++;
  permute(swap,fperm,outblock);    /* apply final permutation  */
}

/* End of DES algorithm */

void print_array(unsigned char *in, int length, int mode)
{
  int i;
  if(mode == BIN_M){ /* print array in binary */
    for (i=0;i<length;i++) {
      if (i%8==0) {
        if (i!=0) printf("\n");
        printf("\t");
      }
      printf("%s%s ", bin[in[i]>>4], bin[in[i]&0xf]);
    }
  } else if(mode == HEX_M){    /* print array in hex */
    for (i=0;i<length;i++) {
      if (i%8==0) {
        if (i!=0) printf("\n");
        printf("\t");
      }
      printf("%02x", in[i]);
    }
  } else if(mode == DEC_M){    /* print array in decimal */
    for (i=0;i<length;i++) {
      if (i%16==0) {
        if (i!=0) printf("\n");
        printf("\t");
      }
      printf("%d ", in[i]);
    }
  }
 printf("\n");

}
```

```
void print_des(unsigned char *in, unsigned char *out, int mode)
{
  printf("\tP: "); print_array(in, 8, mode);
  printf("\tC: "); print_array(out, 8, mode);
}

/* End of helper functions */

/* Test the J matrix, here we calculate the value with whole 8 S boxes,
 * not one by one. */
void test_j(unsigned char sindiff[6], unsigned char soutdiff[4],
            unsigned char sin[6])
{
  register int i,j;
  unsigned char in[2][6], out[2][4], outdiff[4];
  unsigned char result[4];
  unsigned char inS[8];

  /* get single S box's input from 48bits array to 8 char. */
  for(i=0;i<2;i++) {           /* inS[0] store input to S1, etc. */
    inS[4*i] = (sin[3*i]>>2) & 0x3f;
    inS[4*i+1] = ((sin[3*i]<<4) | (sin[3*i+1]>>4)) & 0x3f;
    inS[4*i+2] = ((sin[3*i+1]<<2) | (sin[3*i+2]>>6)) & 0x3f;
    inS[4*i+3] = sin[3*i+2] & 0x3f;
  }
  for(i=0;i<64;i++) {          /* test input from 0 to 63 to every S box */
    for(j=0;j<6;j++)           /* clean input array(48bits) to store */
      in[0][j] = 0;            /* first part of a difference pair */
    for(j=0;j<2;j++) {         /* store i(6 bits) to items in input array */
      in[0][3*j] |= (i << 2) & 0xfc;
      in[0][3*j] |= (i >> 4) & 0x03;
      in[0][3*j+1] |= (i << 4) & 0xf0;
      in[0][3*j+1] |= (i >> 2) & 0x0f;
      in[0][3*j+2] |= (i << 6) & 0xc0;
      in[0][3*j+2] |= i & 0x3f;
    }
    for(j=0;j<6;j++)           /* XORed 1st part to get 2nd part in pair */
      in[1][j] = in[0][j] ^ sindiff[j];
    contract(in[0], out[0]);   /* get 1st part's S box output */
    contract(in[1], out[1]);   /* get 2nd part's S box output */
    for(j=0;j<4;j++) {
      /* calculate S box's output diff of pair */
      outdiff[j] = out[0][j] ^ out[1][j];
      /* XORed with the known outputdiff */
      result[j] = outdiff[j] ^ soutdiff[j];
      /* if Sx's outdiff equals soutdiff, the 4 bits in result array should
       * be 0, so Jx array correspoding counter++. */
      if(((result[j]>>4)&0x0f) == 0) J[2*j][i^inS[2*j]]++;
      if((result[j]&0x0f) == 0) J[2*j+1][i^inS[2*j+1]]++;
    }
  }
}

/* get the 64 bits key from 48-bits subkey. */
void get_key(unsigned char sub_key[6], unsigned char key[8],
             unsigned char unknown[8], int round)
{
  register int i,j;
  unsigned char *p = unknown;
  unsigned char *r_sub_key;

  /* convert the subkey to a 48 item array */
  for (j=0; j<48; j++)
    pc1m[j]=(sub_key[j>>3] & bytebit[j&0x7]) ? 1 : 0;
  for(i=0;i<8;i++) key[i] = 0; /* clear key to store 56 bits halves */
  j = 0;
  for(i=0;i<56;i++)            /* do reverse PC-2 permutation */
    if(rpc2[i]!=0){            /* PC-2 permute a 48 bits array to 56 bits */
      if(pc1m[rpc2[i]-1])      /* so there will be 8 bits undetermined */
        key[i>>3] |= bytebit[i&0x7];
    } else {
```

```
      unknown[j++] = i+1;      /* store the unkown bit's location */
    }
  /* convert the 56 bits halves to a 56 bits array */
  for(j=0; j<56; j++)
    pc1m[j]=(key[j>>3] & bytebit[j&0x7]) ? 1 : 0;
  /* rotate the array with right amount  */
  for(j=0; j<56; j++)
    pcr[j] = pc1m[(i=j-totrot[round-1]) >= (j<28 ? 0 : 28) ? i : 28+i];
  for(i=0;i<8;i++) key[i] = 0;     /* clear key again to store 64 bits key */
  for(i=0;i<64;i++)            /* do reverse PC-1 permutation */
    if(rpc1[i]!=0)
      if(pcr[rpc1[i]-1])
        key[i>>3] |= bytebit[i&0x7];
  /* get the unknown bits change while permute from 56 bits to 64 bits */
  for(i=0;i<8;i++) {
    /* rotate */
    unknown[i] = ((j=unknown[i]+totrot[round-1]) <= (unknown[i]<28 ? 28 : 56))
                 ? j : j-28;
    unknown[i] = pc1[unknown[i]-1];
  }
}

void crack_3round(struct text_pair plain[3],
                  struct text_pair cipher[3])
{
  register int i, j;
  /* Store cracked 3 round subkey, the key, the unkown bits. */
  unsigned char sub_key3[6], key[8], unknown_bit[8];
  /* The input and output diff of chosen pairs. */
  unsigned char inDiff[3][8], outDiff[3][8];
  unsigned char T[4];

 /* Round 3 S box input, output and diffs. */
  struct sin_pair S3in[3];
  struct sout_pair S3out[3];
  unsigned char S3inDiff[3][6];
  unsigned char S3outDiff[3][4];

  /* ---------------- DC start. ------------------*/
  printf("\nDifferencial Cryptanalysis start...\n");
  for(i=0;i<3;i++){ /* Got 3rd round S box input. */
    expand(cipher[i].first, S3in[i].first);
    expand(cipher[i].second, S3in[i].second);
    printf("Pair %d 3rd round S box input:\n", i);
    print_array(S3in[i].first, 6, BIN_M);
    print_array(S3in[i].second, 6, BIN_M);
  }
  printf("\n");
  for (i=0;i<8;i++)        /* Initial J matrix. */
    for(j=0;j<64;j++) J[i][j] = 0;
  for(i=0;i<3;i++) {       /* */
    for(j=0;j<6;j++)       /* input diff of S box in 3rd round */
      S3inDiff[i][j] = S3in[i].first[j] ^ S3in[i].second[j];
    /* input and output diff of chosen pairs, actually we only need the
     * left 4 bytes of input and right 4 bytes of output. */
    for(j=0;j<8;j++) {
      inDiff[i][j] = plain[i].first[j] ^ plain[i].second[j];
      outDiff[i][j] = cipher[i].first[j] ^ cipher[i].second[j];
    }
    /* L0 XOR R3 */
    for(j=0;j<4;j++) {
      T[j] = inDiff[i][j] ^ outDiff[i][j+4];
    }
    /* use L0 XOR R3 to get S box output diff of 3rd round */
    perm32(T, rp32, S3outDiff[i]);
    printf("The 3rd-round S box input diff of pair %d:\n", i);
    print_array(S3inDiff[i], 6, BIN_M);
    printf("The 3rd round S box output diff of pair %d:\n", i);
    print_array(S3outDiff[i], 4, BIN_M);
    /* test J matrix use S box input and output diff */
    test_j(S3inDiff[i], S3outDiff[i], S3in[i].first);
```

```
  }
  /* Print the J matrix. */
  for (i=0;i<8;i++) {
    printf("J%d: \n",i+1);
    for(j=0;j<64;j++) {
      if (j%16==0) {
        if (j!=0) printf("\n");
        printf("\t");
      }
      printf("%d ",J[i][j]);
      /* use key to temporarily store subkey in 8 chars for each S box */
      if(J[i][j]==3) key[i] = j;
    }
    printf("\n");
  }
  for(i=0;i<6;i++) sub_key3[i] = 0;
  /* generate the sub key(48 bits) of 3rd round from the stored 64 bits. */
  for(j=0;j<2;j++) {
    sub_key3[3*j] |= (key[4*j] << 2) & 0xfc;
    sub_key3[3*j] |= (key[4*j+1] >> 4) & 0x03;
    sub_key3[3*j+1] |= (key[4*j+1] << 4) & 0xf0;
    sub_key3[3*j+1] |= (key[4*j+2] >> 2) & 0x0f;
    sub_key3[3*j+2] |= (key[4*j+2] << 6) & 0xc0;
    sub_key3[3*j+2] |= key[4*j+3] & 0x3f;
  }
  printf("Sub key of 3rd round(48 bits):\n");
  print_array(sub_key3, 6, BIN_M);

  /* get the key from sub key of 3rd round */
  get_key(sub_key3, key, unknown_bit, 3);

  printf("Cracked key of the encryption(64 bits):\n");
  print_array(key, 8, BIN_M);
  printf("The unknown bit number of the cracked key:\n");
  print_array(unknown_bit, 8, DEC_M);
}

/* End of DC algorithm */

int main(int argc, char *argv[])
{
  /* key for encryption */
  unsigned char key[8] = {
    0x1a, 0x62, 0x4c, 0x89, 0x52, 0x0d, 0xec, 0x46
  };
  /* Chosen plaintext pairs. */
  struct text_pair in[3] = { {
      {0x74, 0x85, 0x02, 0xcd, 0x38, 0x45, 0x10, 0x97},
      {0x38, 0x74, 0x75, 0x64, 0x38, 0x45, 0x10, 0x97}
    }, {
      {0x48, 0x69, 0x11, 0x02, 0x6a, 0xcd, 0xff, 0x31},
      {0x37, 0x5b, 0xd3, 0x1f, 0x6a, 0xcd, 0xff, 0x31}
    }, {
      {0x35, 0x74, 0x18, 0xda, 0x01, 0x3f, 0xec, 0x86},
      {0x12, 0x54, 0x98, 0x47, 0x01, 0x3f, 0xec, 0x86}
    }
  };
  struct text_pair out[3]; /* Store ciphertext. */

  int i,j;
  desinit(key);    /* set up tables for DES  */
  printf("KEY: "); print_array(key, 8, HEX_M);
  printf("\nEncryption start...\n");
  for(i=0;i<3;i++){ /* encryption */
    endes(in[i].first, out[i].first, 3, NO_IP, NO_LAST_SWAP);
    endes(in[i].second, out[i].second, 3, NO_IP, NO_LAST_SWAP);
    printf("Pair %d encryption result:\n", i);
    print_des(in[i].first,out[i].first, HEX_M);
    print_des(in[i].first,out[i].second, HEX_M);
  }
  crack_3round(in, out);
```

```
  return 0;
}
```

References

[1] *Data Encryption Standard*, NIST, FIPS PUB 46-3, pp. 19-21, 1999

[2] Eli Biham, Adi Shamir, *Differential Cryptanalysis of DES-like Cryptosystems*, Journal of Cryptology, Vol. 4, No. 1, pp.3-72, 1991. The extended abstract appears in Advances in cryptology, proceeding of CRYPTO'90, pp. 2-21, 1990.

[3] Eli Biham, Adi Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, New York:Springer-Verlag, ISBN: 0-387-97930-1, pp. 33-69, 1993

[4] Dengguo Feng, *Cryptanalysis*, Beijing:Tsinghua Pub, ISBN: 7-302-03976-3, pp. 17-22, 1993