Persistent Fault Analysis of DES block cipher

Kumar Abhinav, Saptarshi Ghosh, Mohit Nathrani, Daanish Mahajan
January 17, 2022

1 DES Block Cipher

- DES block cipher is a feistel cipher that works on plain texts of block size of 64-bits, and key size 64 bits (56 + 8 parity bits). Thus, 56-bits is the effective key length. The general structure of DES block cipher is shown below.
- There are 16 rounds. The last round is different in the sense that there is no swap in the last round (round 16). The key generation algorithm is responsible for generating the 16 round keys of size 48-bit, which form input to the function F.
- The Feistel function (F) is depicted in figure 1. It operates on half block of 32-bit. The round key (48-bits) is the other input to F. The expansion box is responsible for expanding the 32-bit input to 48-bit by permuting and duplicating few entries, which are then XOR'ed with the round key. The data so obtained is divided into 8 nibbles of 6-bit each. The 6×4 S-Box then maps the 6-bit input to 4-bit output.
- The S-box contains 16×4 mappings. Thus each unique 4-bit output occurs 4 times in the S-Box. A single fault introduced in the S-Box changes the output distribution, in the sense that one 4-bit output occurs 5 times in the S-Box, while the replaced value occurs 3 times. Given a large no. of plain text queries, it can be seen that the faulty value will occur maximum no. of times, while the replaced value will occur the minimum no. of times.
- Finally, the output of all S-box is combined and passed through a linear permutation.
- DES block cipher has been implemented for the little endian architecture and has been verified using various test cases.

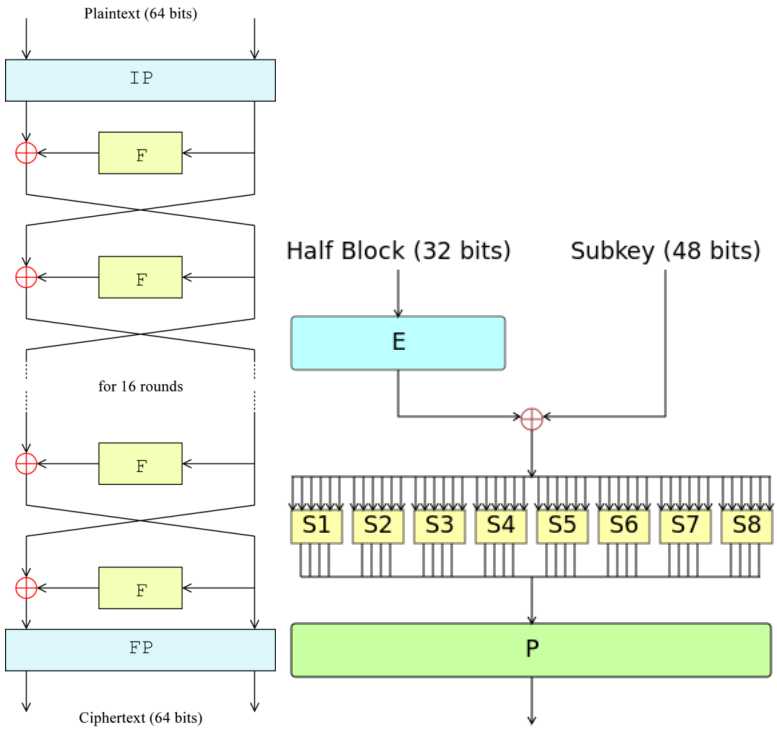

Figure 1: DES Block Cipher (Feistel Structure and Feistel Function F)

2 Persistent Fault Attack

- Persistent Fault Analysis refers to the class of Attacks where the fault introduced in the system stays (is persisted) throughout until it is restarted. Such faults can change the ciphertext distribution which can be used to recover the key.
- Given, the change in the distribution as implied by the fault introduced in the S-Box, we try to recover the last round key (48-bits) in DES block cipher.
- Consider the scenario where a single fault is introduced in one of the S-box S_i. Since the same S-Box is used in all rounds, the faulty S-box is used in all rounds. The attacker also has access to the correct cipher texts and faulty cipher texts encrypted using the same key.
- Attack has been implemented by introducing single fault in one of the S-Box and comparing the correct cipher text with the faulty one. If the correct and faulty cipher text do not match, it means the faulty value was encountered at some point. Also, given the feistel structure, the lower 32 significant bits ($R - 15$) for input to last round is the same as lower significant 32-bits of cipher text (after removing the Final Permutation FP).
- Since, we limit our attack only to the last round, but the fault persists throughout all rounds, we only consider those correct and faulty cipher texts pairs which have same R_{16} and differ in L_{16}. (Note : Correct and faulty pairs are generated from the same Plain Text)
- Since, we also know the faulty value in the S-Box (which can also be statistically profiled for its maximum occurrence across all cipher texts), we compare the the mismatch in the L_{16} with the faulty value to get those pairs which arise due to access to the faulty value.
- To recover the key, consider the the last Round function with fault in S-Box S_i. We know the faulty value in the S-Box and the corresponding index where it occurs in the S_i, i.e we know the 6-bit input to the S-box S_i. We also know $R_{15} == R_{16}$. Passing the 32-bit R_{15} through the expansion box, we get the 48-bit output.The relation between key, expanded half block (R_{15}) and input to 8 S-boxes is denoted by

$$\text{48-bit Input to S-box} = key \oplus E(R_{15})$$

Reversing, we know the 6-bits input of faulty S-box S_i. We take 6-bits of the known 48-bits of expanded R_{15}. From this we can recover the 6-bits of the last round key.

$$\text{6-bits of key} = \text{6-bit Input corresponding to } S_i \oplus \text{6-bits of } E(R_{15}) \text{ corresponding to i}$$

This shows the attack strategy to recover 6-bits of the key with one fault in any one of the S-Box. The attack strategy is independent of the fault value introduced. The pseudo-code is mentioned below.

Algorithm 1 Persistent Fault Analysis of DES Block Cipher

1: $PT[i]$: i^{th} plain text
2: $CT[i]$: i^{th} cipher text with correct s-box
3: $FT[i]$: i^{th} cipher text with faulty s-box
4: $key[i]$: i^{th} key map
5: $F_{xor}[i]$: xor of original and fault value for i^{th} s-box
6: N : Total No. of Plain Texts

7: **for** $i = 0$ to $N - 1$ **do** ▷ **Correct Encryption**
8: $CT[i] \leftarrow Enc(PT[i], KEY)$
9: **end for**

10: **for** $u = 0$ to 7 **do**
11: $idx \leftarrow rand(64)$ ▷ Introduce a fault in S-Box S_u at idx
12: $nval \leftarrow S[u][idx]$
13: **while** $nval == S[u][idx]$ **do** ▷ Make sure the faulty value is not the original value
14: $nval \leftarrow rand(16)$
15: **end while**
16: $F_{xor}[u] \leftarrow S[u][idx] \oplus nval$
17: $F_{xor}[u] \leftarrow F_{xor}[u] \ll (7 - u) * 4$
18: $F_{xor}[u] \leftarrow P(F_{xor}[u])$
19: $F_{xor}[u] \leftarrow F_{xor}[u] \ll 32$
20: $oval \leftarrow S[u][idx]$
21: $S[u][idx] \leftarrow nval$

22: **for** $i = 0$ to $N - 1$ **do** ▷ **Faulty Encryption**
23: $FT[i] \leftarrow Enc(PT[i], KEY)$
24: **end for**

25: Initialize $\text{key}_{cnt}[\cdot][\cdot] = 0$ ▷ Store recovered key along with its count

26: **for** $n = 0$ to $N - 1$ **do** ▷ **PFA Main Analysis**
27: $C_{ct} \leftarrow \text{IP}(CT[i])$ ▷ Reverse the Final Permutation
28: $F_{ct} \leftarrow \text{IP}(FT[i])$
29: xordiff $\leftarrow C_{ct} \oplus F_{ct}$
30: $right \leftarrow$ xordiff $\wedge ((1 \ll 32) - 1)$
31: **if** $right == 0$ && xordiff **then** ▷ consider pair if R_{16} are same and diff exist in L_{16}
32: **if** xordiff $== F_{xor}[u]$ **then** ▷ check if mismatch corresponds to the faulty value
33: $right \leftarrow C_{ct} \wedge ((1 \ll 32) - 1)$
34: $E_{val} \leftarrow E(right)$
35: $E_{val} \leftarrow (E_{val} \gg (7 - u) * 6) \wedge (0\text{x}3F)$ ▷ Get 6-bits of R_{15} corresponding to the S-Box S_u
36: $rbits \leftarrow E_{val} \oplus idx$ ▷ Possible 6-bits of key
37: $key_{cnt}[u][rbits] \leftarrow key_{cnt}[u][rbits] + 1$ ▷ Maintain count of current key
38: **end if**
39: **end if**
40: **end for**
41: Sort the $key_{cnt}[u][\cdot]$ by its count in decreasing order.
42: Possible key is the one with max count in $key_{cnt}[u][\cdot]$ ▷ **Output**
43: $S[u][idx] \leftarrow oval$ ▷ Restore original value
44: **end for**

The same can be extended for DES having single faults in all S-boxes simultaneously as well, although the probability that the pairs of correct and faulty cipher texts satisfy the above criteria (Line 31 and 32 in pseudo-code) reduces. Please refer to DES.cpp for the encryption and key generation algorithm. The DES-gen.cpp generates random plain texts. DES-analyze.cpp takes the correct and faulty encryption outputs and tries to recover the key

given fault in one S-box at a time.

For multiple-faults at same time, DES-multi-analyze.cpp uses the same approach to recover the whole 48 bits of key, 6 bits at a time. Multi-faults require only one pair of correct cipher text and faulty cipher text for each of the N plain texts.

3 Complexity analysis

- The attack is s-box fault invariant and total of less than 100 cipher texts is required to find the key for the corresponding s-box. Taking N = 10^6, the following table depicts how many times for each s-box, condition in line 32 in pseudocode is satisfied.

Table 1

S-box number	Count
1	12452
2	12243
3	12388
4	12457
5	12315
6	12535
7	12299
8	12365

- Satisfying line 32 is not enough for recovering the key since there are chances of getting faulty outputs but for our case we never encountered such a situation because of its low probability and could get the correct key as the only output. Low probability corresponds to the fact that considering fault was met in previous rounds, there is a high chance for multiple s-boxes to get affected at the end. But since we check for a single fault meaning that there is a high probability for fault to be met in the last round.
- With the above mentioned result, the attack has turned out to be a $0/1$ attack revealing all the queried bits after a fixed number of ciphertexts.
- With faults in all the S-box simultaneously, the following table depicts how many times the condition was satisfied for each s-box. As seen, the no. of times it satisfies the constraints is lower as expected.

Table 2

S-box number	Count
1	1043
2	1061
3	1089
4	1098
5	1117
6	1083
7	1064
8	946

4 Results

4.1 Single Fault at a time

Refer to code in "./codes/single/". An example output is as follows :

```
$ ./PFA.sh
Compiling
1
Running
No of plaintext:1000000
Avg encryption time:16.33 us
No of plaintext:1000000
Avg encryption time:16.24 us
Last round key:-
01011110010011000101001000001110010111010000000110
Likely keys (out of 12452 guesses):-
010111XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 100%
2
Running
No of plaintext:1000000
Avg encryption time:16.43 us
No of plaintext:1000000
Avg encryption time:16.93 us
Last round key:-
01011110010011000101001000001110010111010000000110
Likely keys (out of 12243 guesses):-
XXXXXX100100XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 100%
3
Running
No of plaintext:1000000
Avg encryption time:16.09 us
No of plaintext:1000000
Avg encryption time:16.10 us
Last round key:-
01011110010011000101001000001110010111010000000110
Likely keys (out of 12388 guesses):-
XXXXXXXXXXXX110001XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 100%
4
Running
No of plaintext:1000000
Avg encryption time:16.09 us
No of plaintext:1000000
Avg encryption time:16.12 us
Last round key:-
01011110010011000101001000001110010111010000000110
Likely keys (out of 12457 guesses):-
XXXXXXXXXXXXXXXXXX010010XXXXXXXXXXXXXXXXXXXXXXXX 100%
5
Running
No of plaintext:1000000
Avg encryption time:16.19 us
No of plaintext:1000000
Avg encryption time:16.13 us
Last round key:-
```

```
01011110010011000101001000001110010111010000000110
Likely keys (out of 12315 guesses):-
XXXXXXXXXXXXXXXXXXXXXXXX000011XXXXXXXXXXXXXXXXXX 100%
6
Running
No of plaintext:1000000
Avg encryption time:16.21 us
No of plaintext:1000000
Avg encryption time:16.09 us
Last round key:-
01011110010011000101001000001110010111010000000110
Likely keys (out of 12535 guesses):-
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX100101XXXXXXXXXXXX 100%
7
Running
No of plaintext:1000000
Avg encryption time:16.10 us
No of plaintext:1000000
Avg encryption time:16.16 us
Last round key:-
01011110010011000101001000001110010111010000000110
Likely keys (out of 12299 guesses):-
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX101000XXXXXX 100%
8
Running
No of plaintext:1000000
Avg encryption time:16.09 us
No of plaintext:1000000
Avg encryption time:16.16 us
Last round key:-
01011110010011000101001000001110010111010000000110
Likely keys (out of 12365 guesses):-
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX000110 100%
```

4.2 Multiple Faults simultaneously

Refer to code in "./codes/multi/". An example output is as follows :

```
 $ ./PFA-multi.sh
Compiling
Running
Master Key : 00000001000000010000000100000001101001111011001110100001110000001
No of plaintext:1000000
Avg encryption time:6.62 us
No of plaintext:1000000
Avg encryption time:6.72 us
Last round key:-
00001010000001010000110100100001000000000000001
Likely keys (out of 1043 guesses):-
000010XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 100%
Likely keys (out of 1061 guesses):-
XXXXXX100000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 100%
Likely keys (out of 1089 guesses):-
XXXXXXXXXXXX010100XXXXXXXXXXXXXXXXXXXXXXXXXXXX 100%
```

```
Likely keys (out of 1098 guesses):-
XXXXXXXXXXXXXXXXXX001101XXXXXXXXXXXXXXXXXXXXXXXX 100%
Likely keys (out of 1117 guesses):-
XXXXXXXXXXXXXXXXXXXXXXXX001000XXXXXXXXXXXXXXXXXX 100%
Likely keys (out of 1083 guesses):-
XXXXXXXXXXXXXXXXXXXXXXXXXXXX010000XXXXXXXXXXXX 100%
Likely keys (out of 1064 guesses):-
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX000000XXXXXX 100%
Likely keys (out of 946 guesses):-
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX000001 100%
Possible Last Round Key :
0000101000000101000011010010000100000000000000001
```

Thus, we successfully demonstrated Persistent Fault Attack (PFA) in case of DES Block cipher.

By:- Saptarshi Ghosh | 160002052
Kumar Abhinav | 160001032
Mohit Nathrani | 160002030
Daanish Mahajan | 160004010

References

- Fan Zhang, Xiaoxuan Lou, Xinjie Zhao, Shivam Bhasin, Wei He, Ruyi Ding, Samiya Qureshi, and Kui Ren. Persistent fault analysis on block ciphers. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 150–172, 2018.
- Images : “https://en.wikipedia.org/wiki/Data_Encryption_Standard”

Code

The relevant code section (C++) are given below. For the complete codes, refer to the ./codes/ folder. For brevity, only main DES encryption and fault analysis code is mentioned here.

4.2.1 DES.h

```
#include <bits/stdc++.h>

/* Type declarations */
using word_64 = uint64_t;
using word_56 = uint64_t;
using word_48 = uint64_t;
using word_32 = uint32_t;

/* Constants */
constexpr word_48 key_strip = (1ll<<28)-1;
constexpr int NUM_ROUNDS = 16;

/* Defines */
#define rot_32(a) (((a)<<32)|((a)>>32))
#define rot_28(a, x) ((((a)<<(x))|((a)>>(28-(x))))&key_strip)

/* Data */

// Tables adjusted for little endian architecture
```

```
uint8_t IPt[] = {   // Initial Permutation
     6, 14, 22, 30, 38, 46, 54, 62,
     4, 12, 20, 28, 36, 44, 52, 60,
     2, 10, 18, 26, 34, 42, 50, 58,
     0,  8, 16, 24, 32, 40, 48, 56,
     7, 15, 23, 31, 39, 47, 55, 63,
     5, 13, 21, 29, 37, 45, 53, 61,
     3, 11, 19, 27, 35, 43, 51, 59,
     1,  9, 17, 25, 33, 41, 49, 57,
};

uint8_t FPt[] = {   // Final Permutation
    24, 56, 16, 48,  8, 40,  0, 32,
    25, 57, 17, 49,  9, 41,  1, 33,
    26, 58, 18, 50, 10, 42,  2, 34,
    27, 59, 19, 51, 11, 43,  3, 35,
    28, 60, 20, 52, 12, 44,  4, 36,
    29, 61, 21, 53, 13, 45,  5, 37,
    30, 62, 22, 54, 14, 46,  6, 38,
    31, 63, 23, 55, 15, 47,  7, 39,
};

uint8_t Et[] = {    // Expansion Box
     0, 31, 30, 29, 28, 27, 28, 27,
    26, 25, 24, 23, 24, 23, 22, 21,
    20, 19, 20, 19, 18, 17, 16, 15,
    16, 15, 14, 13, 12, 11, 12, 11,
    10,  9,  8,  7,  8,  7,  6,  5,
     4,  3,  4,  3,  2,  1,  0, 31,
};

uint8_t Pt[] = {    // Feistel Permutation
    16, 25, 12, 11,  3, 20,  4, 15,
    31, 17,  9,  6, 27, 14,  1, 22,
    30, 24,  8, 18,  0,  5, 29, 23,
    13, 19,  2, 26, 10, 21, 28,  7,
};

uint8_t PC1t[] = {  // Permuted Choice 1
     7, 15, 23, 31, 39, 47, 55, 63,
     6, 14, 22, 30, 38, 46, 54, 62,
     5, 13, 21, 29, 37, 45, 53, 61,
     4, 12, 20, 28,  1,  9, 17, 25,
    33, 41, 49, 57,  2, 10, 18, 26,
    34, 42, 50, 58,  3, 11, 19, 27,
    35, 43, 51, 59, 36, 44, 52, 60,
};

uint8_t PC2t[] = {  // Permuted Choice 2
    42, 39, 45, 32, 55, 51, 53, 28,
    41, 50, 35, 46, 33, 37, 44, 52,
    30, 48, 40, 49, 29, 36, 43, 54,
    15,  4, 25, 19,  9,  1, 26, 16,
```

```
     5, 11, 23,  8, 12,  7, 17,  0,
    22,  3, 10, 14,  6, 20, 27, 24,
};

uint8_t key_rot[] = { 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1 };

uint8_t St[8][64] = {   // S-box
    /* S1 */
    14,  0,  4, 15, 13,  7,  1,  4,
     2, 14, 15,  2, 11, 13,  8,  1,
     3, 10, 10,  6,  6, 12, 12, 11,
     5,  9,  9,  5,  0,  3,  7,  8,
     4, 15,  1, 12, 14,  8,  8,  2,
    13,  4,  6,  9,  2,  1, 11,  7,
    15,  5, 12, 11,  9,  3,  7, 14,
     3, 10, 10,  0,  5,  6,  0, 13,
    /* S2 */
    15,  3,  1, 13,  8,  4, 14,  7,
     6, 15, 11,  2,  3,  8,  4, 14,
     9, 12,  7,  0,  2,  1, 13, 10,
    12,  6,  0,  9,  5, 11, 10,  5,
     0, 13, 14,  8,  7, 10, 11,  1,
    10,  3,  4, 15, 13,  4,  1,  2,
     5, 11,  8,  6, 12,  7,  6, 12,
     9,  0,  3,  5,  2, 14, 15,  9,
    /* S3 */
    10, 13,  0,  7,  9,  0, 14,  9,
     6,  3,  3,  4, 15,  6,  5, 10,
     1,  2, 13,  8, 12,  5,  7, 14,
    11, 12,  4, 11,  2, 15,  8,  1,
    13,  1,  6, 10,  4, 13,  9,  0,
     8,  6, 15,  9,  3,  8,  0,  7,
    11,  4,  1, 15,  2, 14, 12,  3,
     5, 11, 10,  5, 14,  2,  7, 12,
    /* S4 */
     7, 13, 13,  8, 14, 11,  3,  5,
     0,  6,  6, 15,  9,  0, 10,  3,
     1,  4,  2,  7,  8,  2,  5, 12,
    11,  1, 12, 10,  4, 14, 15,  9,
    10,  3,  6, 15,  9,  0,  0,  6,
    12, 10, 11,  1,  7, 13, 13,  8,
    15,  9,  1,  4,  3,  5, 14, 11,
     5, 12,  2,  7,  8,  2,  4, 14,
    /* S5 */
     2, 14, 12, 11,  4,  2,  1, 12,
     7,  4, 10,  7, 11, 13,  6,  1,
     8,  5,  5,  0,  3, 15, 15, 10,
    13,  3,  0,  9, 14,  8,  9,  6,
     4, 11,  2,  8,  1, 12, 11,  7,
    10,  1, 13, 14,  7,  2,  8, 13,
    15,  6,  9, 15, 12,  0,  5,  9,
     6, 10,  3,  4,  0,  5, 14,  3,
    /* S6 */
    12, 10,  1, 15, 10,  4, 15,  2,
```

```
     9,  7,  2, 12,  6,  9,  8,  5,
     0,  6, 13,  1,  3, 13,  4, 14,
    14,  0,  7, 11,  5,  3, 11,  8,
     9,  4, 14,  3, 15,  2,  5, 12,
     2,  9,  8,  5, 12, 15,  3, 10,
     7, 11,  0, 14,  4,  1, 10,  7,
     1,  6, 13,  0, 11,  8,  6, 13,
    /* S7 */
     4, 13, 11,  0,  2, 11, 14,  7,
    15,  4,  0,  9,  8,  1, 13, 10,
     3, 14, 12,  3,  9,  5,  7, 12,
     5,  2, 10, 15,  6,  8,  1,  6,
     1,  6,  4, 11, 11, 13, 13,  8,
    12,  1,  3,  4,  7, 10, 14,  7,
    10,  9, 15,  5,  6,  0,  8, 15,
     0, 14,  5,  2,  9,  3,  2, 12,
    /* S8 */
    13,  1,  2, 15,  8, 13,  4,  8,
     6, 10, 15,  3, 11,  7,  1,  4,
    10, 12,  9,  5,  3,  6, 14, 11,
     5,  0,  0, 14, 12,  9,  7,  2,
     7,  2, 11,  1,  4, 14,  1,  7,
     9,  4, 12, 10, 14,  8,  2, 13,
     0, 15,  6, 12, 10,  9, 13,  0,
    15,  3,  3,  5,  5,  6,  8, 11,
};

word_48 subkeys[NUM_ROUNDS];

/* Functions */

static inline word_48 rot_56(word_48 a, word_48 x) {
    word_48 left = a >> 28, right = a & key_strip;
    left = rot_28(left, x);
    right = rot_28(right, x);
    return (left << 28) | right;
}

static inline word_56 PC1(const word_64 in) {
    word_56 ret = 0;
    for (int32_t i = 0; i < 56; i++) {
        ret <<= 1;
        ret |= (in >> PC1t[i]) & 1;
    }
    return ret;
}

static inline word_48 PC2(word_56 in) {
    word_48 ret = 0;
    for (int32_t i = 0; i < 48; i++) {
        ret <<= 1;
        ret |= (in >> PC2t[i]) & 1;
    }
    return ret;
```

```
}

static inline void key_schedule(word_64 key, bool decrypt) {
    key = PC1(key);  // remove parity bits
    int inc = 1 - 2 * decrypt, sk_idx = (decrypt) * (NUM_ROUNDS - 1);
    for (int i = 0; i < NUM_ROUNDS; i++, sk_idx += inc) {
        key = rot_56(key, key_rot[i]);
        subkeys[sk_idx] = PC2(key);
    }
}

static inline word_48 E(const word_32 pt) {
    word_48 ret = 0;
    for (int32_t i = 0; i < 48; i++) {
        ret <<= 1;
        ret |= (pt >> Et[i]) & 1;
    }
    return ret;
}

static inline word_32 S(word_48 in) {
    word_32 ret = 0;
    for (int i = 0, j = 42; i < 8; i++, j -= 6) {
        ret <<= 4;
        ret |= St[i][(in >> j) & 0x3F];
    }
    return ret;
}

static inline word_32 P(const word_32 in) {
    word_32 ret = 0;
    for (int32_t i = 0; i < 32; i++) {
        ret <<= 1;
        ret |= (in >> Pt[i]) & 1;
    }
    return ret;
}

static inline word_32 F(word_48 key, word_32 right) {
    word_48 r_enc = E(right);
    word_48 tmp = key ^ r_enc;
    word_32 tmp_2 = S(tmp);
    return P(tmp_2);
}

static inline word_64 DES_round(const word_48 subkey, word_64 pt) {
    word_32 left = pt >> 32, right = pt & 0xFFFFFFFF;
    word_32 xor_with = F(subkey, right);
    left ^= xor_with;
    return left | (((word_64)right) << 32);
}
static inline word_64 DES_round_no_rot(const word_48 subkey, word_64 pt) {
    word_32 left = pt >> 32, right = pt & 0xFFFFFFFF;
    word_32 xor_with = F(subkey, right);
```

```
    left ^= xor_with;
    return (((word_64)left) << 32) | right;
}

static inline word_64 IP(const word_64 in) {
    word_64 ret = 0;
    for (int32_t i = 0; i < 64; i++) {
        ret <<= 1;
        ret |= (in >> IPt[i]) & 1;
    }
    return ret;
}

static inline word_64 FP(const word_64 in) {
    word_64 ret = 0;
    for (int32_t i = 0; i < 64; i++) {
        ret <<= 1;
        ret |= (in >> FPt[i]) & 1;
    }
    return ret;
}

static inline word_64 DES_encrypt(const word_64 pt, const word_64 key, const bool decrypt)
{
    key_schedule(key, decrypt);  // initialize subkeys
    word_64 tmp = IP(pt);
    for (int i = 0; i + 1 < NUM_ROUNDS; i++) {
        tmp = DES_round(subkeys[i], tmp);
    }
    tmp = DES_round_no_rot(subkeys[NUM_ROUNDS - 1], tmp);
    return FP(tmp);
}
```

4.2.2 DES-analyse.cpp

```
#include <bits/stdc++.h>
#include "DES.h"
using namespace std;

string c_file, f_file;
int sbox_num, sbox_idx;
word_64 fault_xor;

/**
 * If the fault occurs at last round,
 *
 * R16 == R15 (Feistel property) [after removing final permutation]
 * L16 in correct and faulty output will differ in 1 position corresponding to the faulty S
 */
bool check_valid_faulty(word_64 correct, word_64 faulty) {
    word_64 xordiff = correct ^ faulty;
    word_32 right = xordiff & ((1ll << 32) - 1);
    if (right == 0 && xordiff) {
        int shiftby = (7 - sbox_num) * 4;
        word_64 verify = fault_xor << shiftby;
```

```
            verify = P(verify);
            verify <<= 32;
            return xordiff == verify;
        }
        return false;
    }

    /**
     * Recover 6 - key bits corresponding to a single fault in sbox_num
     */
    int key_bits(word_64 enc) {
        word_32 right = enc & ((1ll << 32) - 1);
        word_48 E_val = E(right);
        int shiftby = (7 - sbox_num) * 6;
        E_val >>= shiftby;
        E_val &= 0x3F;            // 6 bits corresponding to the fault
        return E_val ^ sbox_idx;  // 6 bit input to s_box = E_val ^ key bits;
    }

    void arg_eval(int argc, char **argv) {
        if (argc != 6) {
            cout << "Usage: " << argv[0] << " <correct filename> <faulty filename> <S-box num>
            idx> <S-box fault value>" << endl;
            exit(1);
        }
        c_file = argv[1];
        f_file = argv[2];
        sbox_num = atoi(argv[3]) - 1;  // 0-indexed
        sbox_idx = atoi(argv[4]);
        int fault_val = atoi(argv[5]);
        fault_xor = fault_val ^ St[sbox_num][sbox_idx];  // This will be used to verify error
    }

    /** Print 6-bits of the recovered key **/
    string formatted(word_48 subkey, int sbox_num) {
        string ans;
        for (int i = 0; i < sbox_num * 6; i++) ans.push_back('X');
        string tmp;
        for (int i = 0; i < 6; i++) {
            tmp.push_back(subkey % 2 + '0');
            subkey /= 2;
        }
        reverse(tmp.begin(), tmp.end());
        ans += tmp;
        for (int i = (sbox_num + 1) * 6; i < 48; i++) ans.push_back('X');
        return ans;
    }

    int32_t main(int argc, char **argv) {
        ios::sync_with_stdio(false);
        cin.tie(0);
        arg_eval(argc, argv);

        ifstream cf(c_file), ff(f_file);
```

```
    map<word_48, int> key_cnt;

    int tot_cnt = 0;
    while (cf) {
        word_64 correct, faulty;
        cf >> correct;
        ff >> faulty;
        correct = IP(correct);  //remove last permutation
        faulty = IP(faulty);
        if (check_valid_faulty(correct, faulty)) {
            key_cnt[key_bits(correct)]++;
            tot_cnt++;
        }
    }

    vector<pair<int, word_48>> v;
    for (auto &i : key_cnt) {
        v.push_back({i.second, i.first});
    }

    sort(v.rbegin(), v.rend());
    cout << "Likely keys (out of " << tot_cnt << " guesses):- \n";

    for (auto &i : v) {
        float percent = 100. * i.first / tot_cnt;
        cout << formatted(i.second, sbox_num) << " " << percent << "%\n";
    }
    return 0;
}
```

4.2.3 DES-multi-analyze.cpp

```
#include <bits/stdc++.h>
#include "DES.h"
using namespace std;

string c_file, f_file;
int sbox_num, sbox_idx;
word_64 fault_xor;
vector<vector<word_48>> key(8);

/**
 * If the fault occurs at last round,
 *
 * R16 == R15 (Feistel property) [after removing final permutation]
 * L16 in correct and faulty output will differ in 1 position corresponding to the faulty S
 */
bool check_valid_faulty(word_64 correct, word_64 faulty) {
    word_64 xordiff = correct ^ faulty;
    word_32 right = xordiff & ((1ll << 32) - 1);
    if (right == 0 && xordiff) {
        int shiftby = (7 - sbox_num) * 4;
        word_64 verify = fault_xor << shiftby;
        verify = P(verify);
        verify <<= 32;
```

```
        return xordiff == verify;
    }
    return false;
}

/**
 * Recover 6 - key bits corresponding to a single fault in sbox_num
 */
int key_bits(word_64 enc) {
    word_32 right = enc & ((1ll << 32) - 1);
    word_48 E_val = E(right);
    int shiftby = (7 - sbox_num) * 6;
    E_val >>= shiftby;
    E_val &= 0x3F;            // 6 bits corresponding to the fault
    return E_val ^ sbox_idx;  // 6 bit input to s_box = E_val ^ key bits;
}

void arg_eval(int argc, char **argv, int idx) {
    if (argc != 27) {
        cout << "Usage: " << argv[0] << " <correct filename> <faulty filename> <S-box num>
        idx> <S-box fault value> ..." << endl;
        cout << "Enter <S-box num> <S-box fault idx> <S-box fault value> for each S-box" <<
        exit(1);
    }
    c_file = argv[1];
    f_file = argv[2];
    sbox_num = atoi(argv[3 + 3 * idx]) - 1;         // 0-indexed
    sbox_idx = atoi(argv[4 + 3 * idx]);
    int fault_val = atoi(argv[5 + 3 * idx]);
    fault_xor = fault_val ^ St[sbox_num][sbox_idx];  // This will be used to verify error
}

/** Print 6-bits of the recovered key **/
string formatted(word_48 subkey, int sbox_num) {
    string ans;
    for (int i = 0; i < sbox_num * 6; i++) ans.push_back('X');
    string tmp;
    for (int i = 0; i < 6; i++) {
        tmp.push_back(subkey % 2 + '0');
        subkey /= 2;
    }
    reverse(tmp.begin(), tmp.end());
    ans += tmp;
    for (int i = (sbox_num + 1) * 6; i < 48; i++) ans.push_back('X');
    return ans;
}

/** Recover 6-bits of key at a time **/
void recover_key_nibble(int argc, char **argv, int idx) {
    arg_eval(argc, argv, idx);

    ifstream cf(c_file, ios::in | ios::binary), ff(f_file, ios::in | ios::binary);
    map<word_48, int> key_cnt;
    int tot_cnt = 0;
```

```
    while (cf) {
        word_64 correct, faulty;
        cf >> correct;
        ff >> faulty;
        correct = IP(correct);  // Remove last permutation
        faulty = IP(faulty);

        if (check_valid_faulty(correct, faulty)) {
            key_cnt[key_bits(correct)]++;
            tot_cnt++;
        }
    }

    vector<pair<int, word_48>> v;
    for (auto &i : key_cnt) {
        v.push_back({i.second, i.first});
    }

    sort(v.rbegin(), v.rend());
    cout << "Likely keys (out of " << tot_cnt << " guesses):- \n";
    for (auto &i : v) {
        float percent = 100. * i.first / tot_cnt;
        cout << formatted(i.second, sbox_num) << " " << percent << "%\n";
        key[sbox_num].push_back(i.second);
    }
}

int32_t main(int argc, char **argv) {
    for (int i = 0; i < 8; ++i) {
        recover_key_nibble(argc, argv, i);
    }

    cout << "Possible Last Round Key : \n";
    for (int i1 = 0; i1 < key[0].size(); ++i1)
    for (int i2 = 0; i2 < key[1].size(); ++i2)
    for (int i3 = 0; i3 < key[2].size(); ++i3)
    for (int i4 = 0; i4 < key[3].size(); ++i4)
    for (int i5 = 0; i5 < key[4].size(); ++i5)
    for (int i6 = 0; i6 < key[5].size(); ++i6)
    for (int i7 = 0; i7 < key[6].size(); ++i7)
    for (int i8 = 0; i8 < key[7].size(); ++i8) {
        cout << bitset<6>(key[0][i1])
                << bitset<6>(key[1][i2])
                << bitset<6>(key[2][i3])
                << bitset<6>(key[3][i4])
                << bitset<6>(key[4][i5])
                << bitset<6>(key[5][i6])
                << bitset<6>(key[6][i7])
                << bitset<6>(key[7][i8]);
    }

    return 0;
}
```