INDIAN INSTITUTE OF TECHNOLOGY INDORE

TERM PROJECT, SPRING SEMESTER 2019-2020.

CRYPTOGRAPHY AND NETWORK SECURITY (CS 417/617)

Persistent Fault Analysis Attack on DES

Authors:
Naman Singhal (160001038)
Bitan Paul (160001016)
Ishan Goel (160001023)

Instructor:
Dr. BODHISATWA MAZUMDAR

Department of Computer Science and Engineering

June 10, 2020

Contents

List of Figures

Chapter 1

Problem Statement

Persistent fault analysis (PFA) on an encryption system recovers the secret key of the system by mounting fault while the encryption algorithm is running and then perform a statistical analysis to exploit such faults. The fault persists in the algorithm over multiple encryptions or queries, whereas it disappears when the device reboots. An instance of persistent fault can be given as toggling of an S-box lookup table entry in an SPN block cipher implementation. The toggle in the S-box lookup table implementation can be reversed when the device is powered up again.

Consider the persistent fault analysis on block ciphers in [1]. The project involves mounting a key recovery attack on a C/C++ implementation of DES cipher. In the report submission, the following points are to be discussed in detail:

1) Implement a functionally correct version of DES block cipher in C/C++. The code must be provided in the report.

2) Mount PFA on each 6 x 4 S-box of the cipher. The strategy mentioned in Section 3.3 of [1] can be referred to. The attack algorithm or strategy must be mentioned in the report.

3) Report the residual key entry with the respect to the number of ciphertexts or queries for each attack. Is the attack performance invariant of the S-box mapping? A thorough analysis of the attack must be mentioned in the report

4) Report the complexity analysis of the attack.

Chapter 2

The Proposed Attack

2.1 DES

The Data Encryption Standard (DES) is a symmetric-key block cipher with a Feistel Structure. Describing the complete functioning of DES in detail is not within the scope of the report but we will be using the following terminology to refer to components of DES throughout this report.

1. The Plain Text is PT (64 bit) and The Cipher Text CT (64 bit).
2. The Initial Permutation is IP and Final Permutation is FP which are inverse of each other.
3. The Master Key is MK (56 bit) and The Round Keys are K_1 to K_{16} (each 48 bit).
4. For Each Round i, inputs are (L_{i-1}, R_{i-1}, K_i) and outputs are (L_i, R_i) (each 32 bit).
5. The Round Function is $f(R_{i-1}, K_i)$.
6. The Round Function is made of 4 steps: Expansion D-box ($Dbox_{expansion}$), XOR, S-box, Straight D-box ($Dbox_{straight}$).
7. Each S-box is represented is S_1 to S_8 with 6-bit input and 4-bit output.
8. Faulty Outputs and functions are represented with a $*$ such as CT^*, L_{16}^*, R_{16}^*, $f^*()$.

2.2 Fault Model

The assumed fault model is same as [1] and is as follows:

1) The adversary can inject faults before the encryption of a block cipher. Typically, these faults alter a stored algorithm constant (S-box).

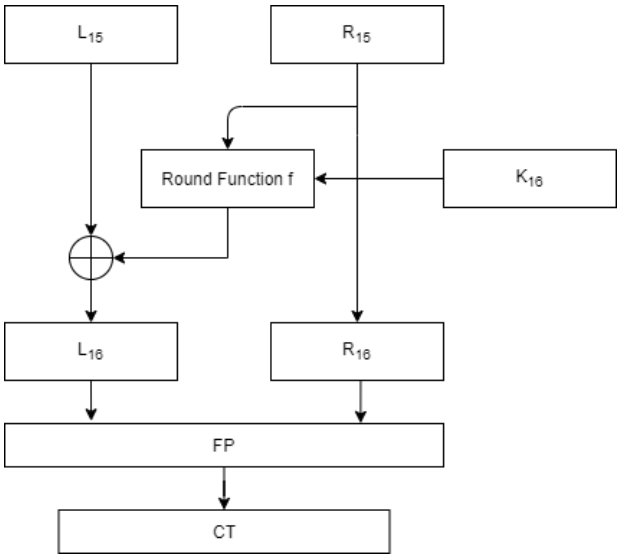
2) The injected faults are persistent, i.e., the affected constant stays faulty unless refreshed. Thus all iterations are computed with the faulty constant.

3) The adversary is capable of collecting multiple ciphertext outputs. Both with and without the Fault Injected

2.3 Formulation

In this section we build formulation for an attack where we inject a fault in any one of the S-box in the last round. We assume the fault is injected in S_1 at index v_1 without any loss of generality.

FIGURE 2.1: The last round of DES



For any Plain Text PT we have.

$$CT = DES\_ENC(PT)$$
$$CT^* = DES\_FAULTY\_ENC(PT)$$

Using IP as the inverse of FP

$$(L_{16}, R_{16}) = IP(CT)$$
$$(L^*_{16}, R^*_{16}) = IP(CT^*)$$

Using the properties of DES and that the fault is injected in the last round

$$\begin{aligned}
R_{15} &= R^*_{15} = R_{16} = R^*_{16} \\
L^*_{15} &= L_{15} \\
\Delta L_{16} &= L^*_{16} \oplus L_{16} \\
\Delta L_{16} &= L^*_{15} \oplus f^*(R^*_{15}, K_{16}) \oplus L_{15} \oplus f(R_{15}, K_{16}) \\
\Delta L_{16} &= f^*(R_{15}, K_{16}) \oplus f(R_{15}, K_{16}) \\
\Delta f_{16} &= f^*(R_{15}, K_{16}) \oplus f(R_{15}, K_{16})
\end{aligned}$$

$$\Delta f_{16} = \Delta L_{16}$$

So the change in the output of round function is equivalent to change in L_{16}.

Since the change in round function is only a change in the S-box we should calculate the inputs and outputs of the S_1.

$$
\begin{aligned}
f(R_{15}, K_{16}) &= Dbox_{straight}(SBox((Dbox_{expansion}(R_{15}) \oplus K_{16})) \\
SBox_{input} &= Dbox_{expansion}(R_{15}) \oplus K_{16} \qquad (2.1) \\
SBox_{output} &= Dbox^{-1}_{straight}(f(R_{15}, K_{16})) \\
\Delta SBox_{output} &= Dbox^{-1}_{straight}(\Delta f(R_{15}, K_{16})) \\
\Delta SBox_{output} &= Dbox^{-1}_{straight}(\Delta f_{16})) \\
\Delta SBox_{output} &= Dbox^{-1}_{straight}(\Delta L_{16})) \qquad (2.2)
\end{aligned}
$$

The SBox Input and Output can be split for 6-bit input and 4 bit output for each S_1 to S_8.

$$
\begin{aligned}
K_1 &= K_{16}[0...5] \\
A_1 &= Dbox_{expansion}(R_15)[0...5] \\
S1_{in} &= A_1 \oplus K_1 \\
B_1 &= Dbox^{-1}_{straight}(\Delta L_{16}))[0...3] \\
\Delta S1_{out} &= B_1
\end{aligned}
$$

Similarly we can split for all S_2 to S_8 according to need. Since we currently assumed fault is injected only in S_1, so only $\Delta S1_{out}$ can be non zero. We know the fault is injected in index v of S_1.

If $\Delta S1_{out} \neq 0$, then $S1_{in} = v$, so we have **Strategy 1**:

$$
\begin{aligned}
v &= A_1 \oplus K_1 \\
K_1 &= A_1 \oplus v
\end{aligned}
$$

Else $\Delta S1_{out} = 0$, then $S1_{in} \neq v$, so we have **Strategy 2**:

$$
\begin{aligned}
v &\neq A_1 \oplus K_1 \\
K_1 &\neq A_1 \oplus v
\end{aligned}
$$

As we can clearly see that the S-Box inputs and outputs can be independently split and this means we can mount a Persistent Fault in all S_1 to S_8 at the same time. Let the fault index for S_i be v_i. Then:

If $\Delta Si_{out} \neq 0$, then $Si_{in} = v$, we have **Strategy 1**:

$$K_i = A_i \oplus v_i \qquad (2.3)$$

Else $\Delta Si_{out} = 0$, then $Si_{in} \neq v$, we have **Strategy 2**:

$$K_i \neq A_i \oplus v_i \qquad (2.4)$$

2.4 Fault Attack 1 (Single S-Box)

Using the above Strategy 1 and 2, we can combine with of them to get the below algorithm, N as the number of Plain Text Queries, and i the S-box number.

1 Input $PT[0...N]$

2 Calculate $CT[0...N]$ using $PT[0...N]$

3 Inject Persistent Fault in S_i at index v_i

4 Repeat for each $PT[j]$ till K_i found:

4.1 Calculate CT^* using $PT[j]$

4.2 Calculate S_{in} and A_i using equation [2.1]

4.2 Calculate S_{out} and B_i using equation [2.2]

4.3 If $B_i \neq 0$: $K_i = A_i \oplus v_i$, K_i found **Strategy 1**

4.4 Else add $A_i \oplus v_i$ to set of discarded keys

4.5 If size of set of discarded keys is equal to 63, key found **Strategy 2**

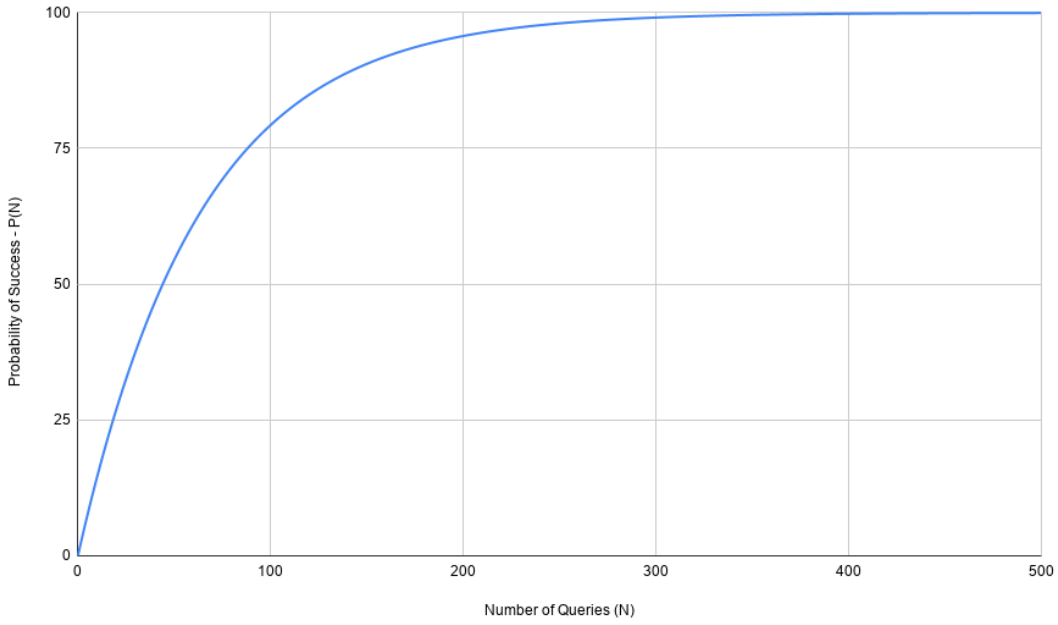
2.4.1 Attack Analysis

Each query uses a different plain text, using the property of diffusion the input of S_i can be assumed to be completely random. The input is 6-bit and the fault v_i is in one of these 64 possible indexes, so the probability of the input being v_i is $1/64$, i.e. the probability of us getting the key in the above algorithm in any query is $1/64$.

The probability of getting the key in N queries comes out to be:

$$P(N) = 1 - (1 - 1/64)^N$$
$$P(N) = 1 - (63/64)^N$$

FIGURE 2.2: $P(N)$ vs N (Attacking 1 S-Box)



The number of queries for an attack success probability of 95% comes out as 190. So for 200 queries we can get the key K_i with a probability of 95%

We can also attack the S-box one by one getting 6-bits of the Last Round Key for each S-box, getting the whole key in around 1600 queries and 8 different faults.

2.5 Fault Attack 2 (Attacking All S-Box)

The above algorithm can be easily expanded to injecting a fault in each of the S-box at the same time

1 Input $PT[0...N]$

2 Calculate $CT[0...N]$ using $PT[0...N]$

3 Inject Persistent Fault in S_i at index v_i for all i 1 to 8

4 Repeat for each $PT[j]$ till all K_i are found:

- 4.1 Calculate CT^* using $PT[j]$
- 4.2 Calculate S_{in} and all A_i using equation [2.1]
- 4.2 Calculate S_{out} and all B_i using equation [2.2]
- 4.3 For i 1 to 8
  - a If $B_i \neq 0$: $K_i = A_i \oplus v_i$, K_i found
  - b Else add $A_i \oplus v_i$ to SET_i of discarded keys
  - c If size of SET_i is equal to 63, K_i found

2.5.1 Attack Analysis

This attack is equivalent to 8 simultaneous above attacks, hence the probability of success is the product of the probability of success of all 8 attack. Hence by symmetry:

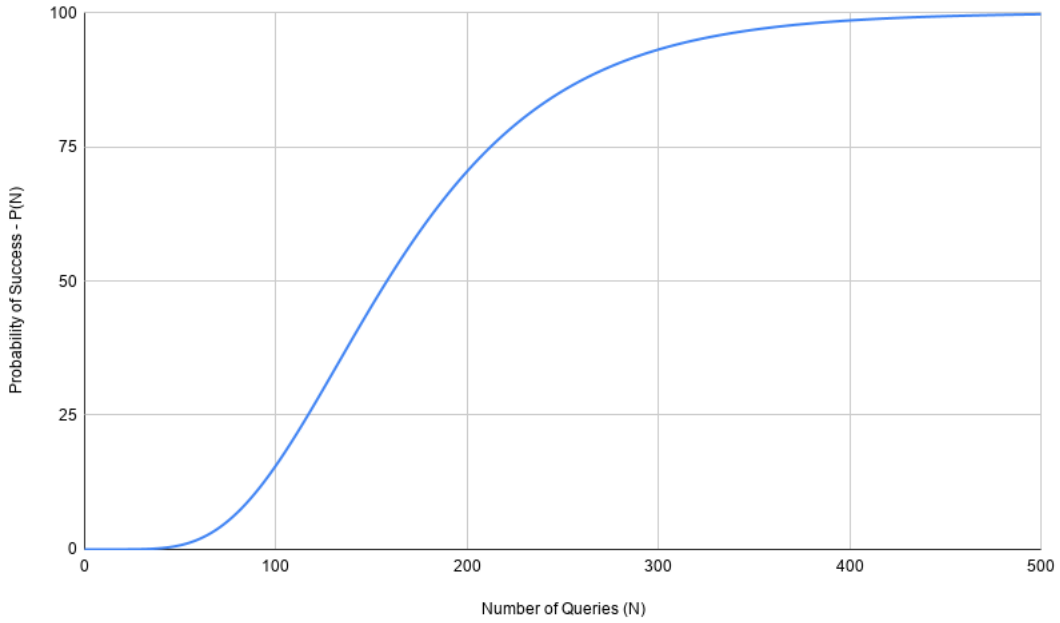
$$P(N) = (1 - (63/64)^N)^8$$

The number of queries for an attack success probability of 95% comes out as 320. So for 320 queries we can get the all keys K_i with a probability of 95%, which means we have the whole last round key.

2.6 Recovering the Master Key using the Last round key

DES has a simple key schedule algorithm with some rotational shifts and a compression box to reach to last round key. This means the bits of the last round can be easily one to one mapped to bit of the Master Key. So knowing the 48-bit last round key gives us 48 bits of the Master Key. A brute force approach is simple for the remaining 8 bits.

FIGURE 2.3: $P(N)$ vs N (Attack All S-Box)



2.7 Dependence of S-Box Mapping

Above, we can see that we haven't used the S-box mapping even once throughout the Attack or the Analysis. This is because the the mapping of the S-box creates no difference in the attack, the only thing that matters in this attack is that the mapping of exactly one of the index has changed because of the fault. If the size of the mapping changes from 6-bit, then only the attack's analysis and probability calculations will change accordingly.

We will see in the results how they the graphs for single fault attack comes out exactly the same for all S_1 to S_8.

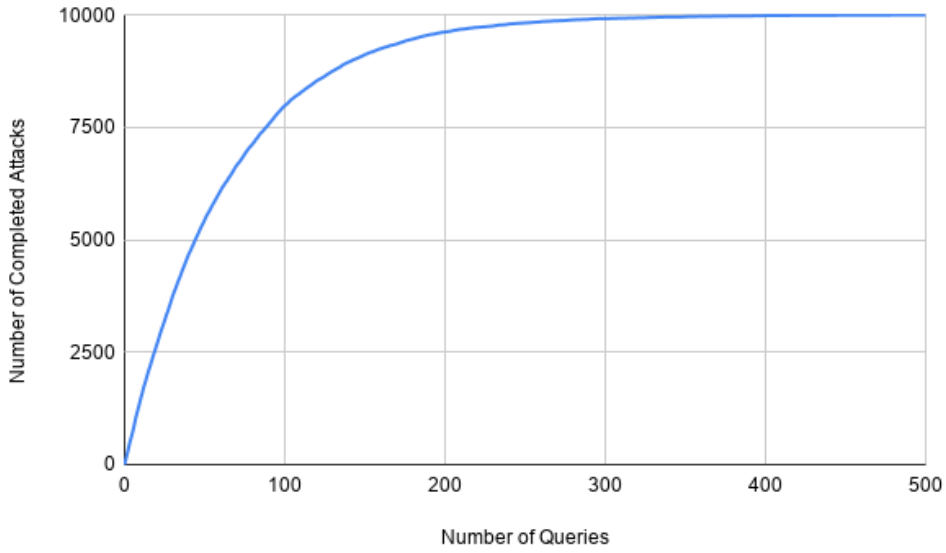
Chapter 3

Results

We generate these results using our implementation in C++ and over 10,000 attacks with randomly generated Master Key and random fault index(v_i).

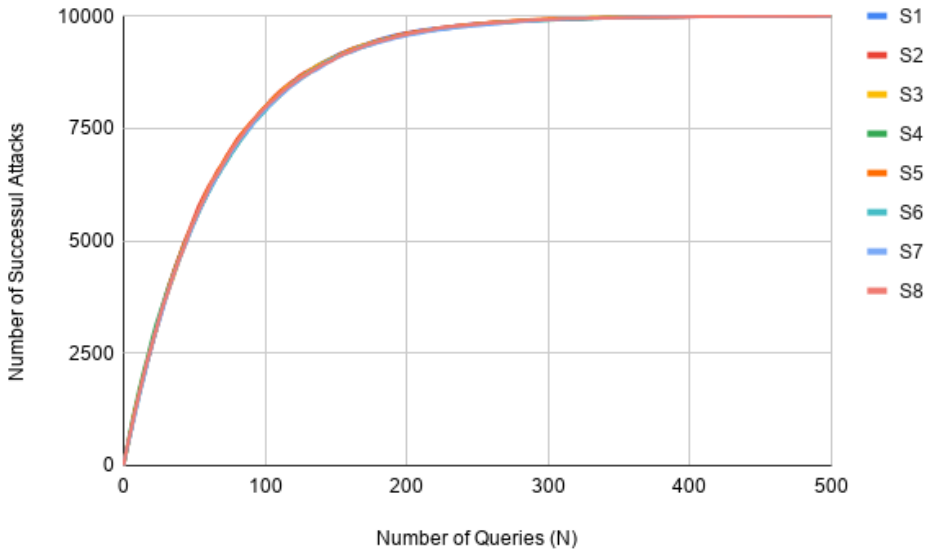
3.1 Attack 1 (Single S-Box)

FIGURE 3.1: Number of Completed Attacks vs N (Single S-Box)



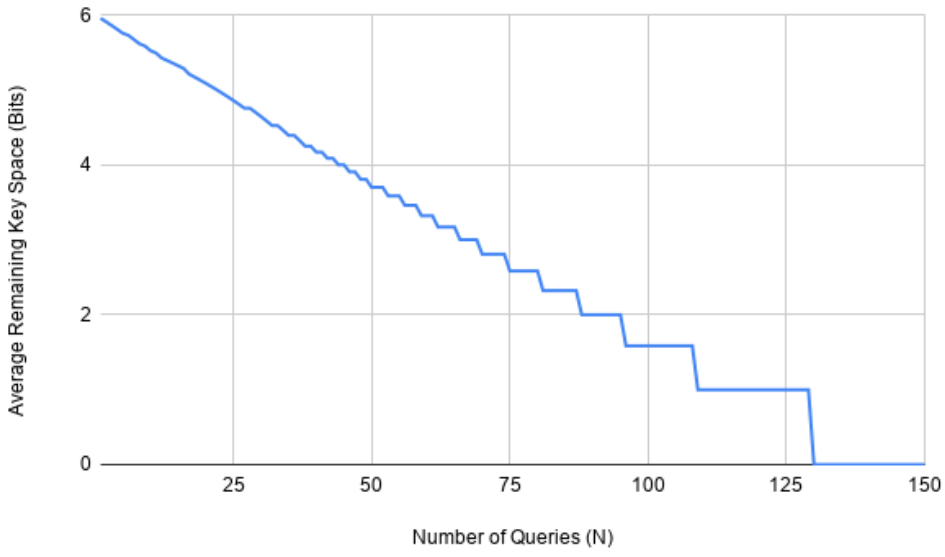
We can clearly see how this graph exactly matches graph [2.2] from the attack probability analysis. Also that the attack reaches 95% success by 200th query as stated theoretically.

FIGURE 3.2: Number of Completed Attacks vs N (Single S-Box)



The exact same results for all S_1 to S_8 shows the invariance of S-box mapping for the the attack.

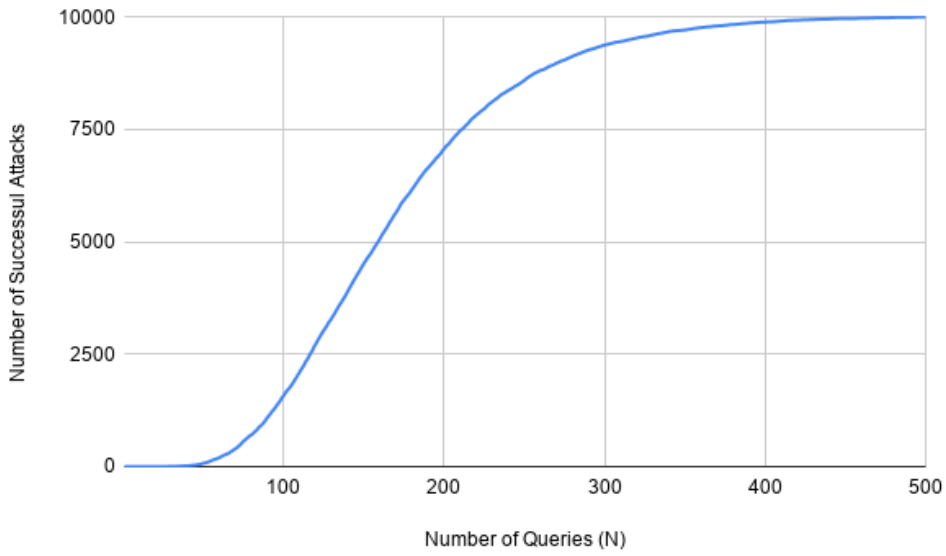
FIGURE 3.3: Key space reduction vs N (Single S-Box)



This graph shows the average key space reduction as we mount 10,000 attacks

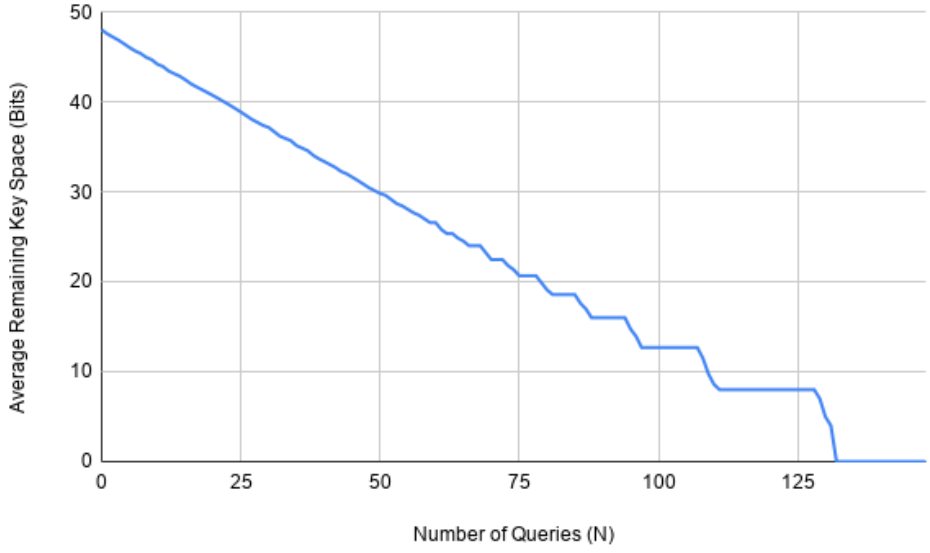
3.2 Attack 2 (All S-Box)

FIGURE 3.4: Number of Completed Attacks vs N (All S-Box)



We can clearly see how this graph exactly matches graph [2.3] from the attack probability analysis. Also that the attack reaches 95% success by 320th query as stated theoretically.

FIGURE 3.5: Key space reduction vs N (All S-Box)



This graph shows the average key space reduction as we mount 10,000 attacks

Chapter 4

Code

Files:

1. des.h : The DES Class header file
2. attack.cpp : Fault Injection and Complete Attack Procedure

Inputs: The programs automatically generated all inputs randomly (PTs, K). The 2 constants to set at the top of "attack.cpp" are:

1. N : Number of Queries for each attack
2. NA : Number of Attacks

Outputs: The program generates 4 output files which are all csv (comma separated values):

1. NumberOfSolved_Single.csv: It represents the number of attacks which has successfully completed till a particular query number. The file contains 8 rows for attacking each S-Box. Each row contains N values each representing the number of completed attack till the ith query.
2. NumberOfSolved_Multi.csv: It represents the number of attacks which has successfully completed till a particular query number. The file contains 1 row which is attacking all S-Box together. The row contains N values each representing the number of completed attack till the ith query.
3. KeySpace_Single.csv: It represents the average remaining Key Space after a particular query number. The file contains 8 row each attacking a S-Box. The row contains N values each representing the average remaining Key Space till the ith query.
4. KeySpace_Multi.csv: It represents the average remaining Key Space after a particular query number. The file contains 1 row which is attacking all S-Box together. The row contains N values each representing the average remaining Key Space till the ith query.

4.1 DES

The "des.h" header file contains all the constants and the DES class for use in the attack.

```
#ifndef DES_H
#define DES_H

#include <cstdint>
using namespace std;

#define ui64 uint64_t
#define ui32 uint32_t
```

```
#define ui8 uint8_t

// ------------------------------------------------------------------------
//-----KEY SCHEDULER UTILITIES-----
// Permuted Choice 1 Table [7*8]
static const char PC1[] =
    {
        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};
// Permuted Choice 2 Table [6*8]
static const char PC2[] =
    {
        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};
// Iteration Shift Array
static const char ITERATION_SHIFT[] =
    {
        1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1
        //  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
};
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
//----------DES ENCRYPTION UTILITIES----------
#define LB32_MASK 0x00000001
#define LB64_MASK 0x0000000000000001
#define L64_MASK 0x00000000ffffffff

// Initial Permutation Table [8*8]
static const char 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};

// Inverse Initial Permutation Table [8*8]
static const char FP[] =
    {
        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};

// Expansion table for Expansion D-BOX in f [6*8]
static const char EXPANSION[] =
    {
        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};

// The S-Box tables in f [8*16*4]
static const char SBOX[8][64] =
    {
        {// S1
         14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
         0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13},
        {// S2
         15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,
         3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
         0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
         13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9},
        {// S3
         10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
         13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
         13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
         1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12},
        {// S4
         7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
         13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
         10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
         3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14},
        {// S5
         2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
         14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
         4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
         11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3},
        {// S6
         12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,
         10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
         9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
         4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13},
        {// S7
         4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
         13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
         1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
         6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12},
        {// S8
         13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
         1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
         7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
         2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}};

// Post S-Box permutation in f [4*8]
static const char PBOX[] =
    {
        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};
// ---------------------------------------------------------------------------

/**
* Standart(Fault-free) DES class
* Contains the functions for DES Encryption and decryption using a user defined key.
*
* DES(ui64 key)                 DES Constructor
* des(ui64 inp, bool mode)  Driver function for DES Encryption (Mode: True-Decryption False-Encryption
* encrypt(ui64 inp)             DES encryption function(encrypt inp with object key)
* decrypt(ui64 inp)             DES decryption function(encrypt inp with object key)
* key_gen(ui64 key)             DES key generation algorithm
* i_perm(ui64 inp)              Initial Permutation Function
* f_perm(ui64 inp)              Final Permutation Function
* feistel_round(ui32 &Left, ui32 &Right, ui32 F)         Basic Fiestel Round
* des_func(ui32 Right, ui64 key)    Function "f"
* sub_key[16]                   Round Keys generated from key_gen algo (48 bits each)
*/
class DES
{
public:
    DES(ui64 key);
    ui64 des(ui64 inp, bool mode);
    ui64 encrypt(ui64 inp);
    ui64 decrypt(ui64 inp);

protected:
    void key_gen(ui64 key);
    ui64 i_perm(ui64 inp);
    ui64 f_perm(ui64 inp);
    void feistel_round(ui32 &Left, ui32 &Right, ui32 F);
    ui32 des_func(ui32 Right, ui64 key);
    ui64 sub_key[16];
};

//---------------------------------------------------------------------------
#pragma GCC optimize("unroll-loops") //Parallel execution of loops

//--------------------KEY GENERATION-----------------------------------
DES::DES(ui64 key)
{
    key_gen(key);
}

void DES::key_gen(ui64 key)
{
    //Setting "permuted_choice_1" as the first 56 bits of the key "key"
    ui64 permuted_choice_1 = 0;
    for (ui8 i = 0; i < 56; i++)
    {
        permuted_choice_1 <<= 1;
        permuted_choice_1 |= (key >> (64 - PC1[i])) & LB64_MASK;
    }

    // Splittong "permutation_choice_1 into two halves"
    ui32 C = (ui32)((permuted_choice_1 >> 28) & 0x000000000fffffff);
    ui32 D = (ui32)(permuted_choice_1 & 0x000000000fffffff);
```

```
    // Calculation of the 16 round keys
    for (ui8 i = 0; i < 16; i++)
    {
        // key schedule, shifting Ci and Di
        for (ui8 j = 0; j < ITERATION_SHIFT[i]; j++)
        {
            C = (0x0fffffff & (C << 1)) | (0x00000001 & (C >> 27));
            D = (0x0fffffff & (D << 1)) | (0x00000001 & (D >> 27));
        }

        ui64 permuted_choice_2 = (((ui64)C) << 28) | (ui64)D;

        sub_key[i] = 0; // 48 bits (2*24)
        for (ui8 j = 0; j < 48; j++)
        {
            sub_key[i] <<= 1;
            sub_key[i] |= (permuted_choice_2 >> (56 - PC2[j])) & LB64_MASK;
        }
    }
}

//-----------------------------\KEY GENERATION-----------------------------

//-----------------------------DES ENCRYPTION-----------------------------

void DES::feistel_round(ui32 &Left, ui32 &Right, ui32 F)
{
    ui32 foo = Right;
    Right = Left ^ F;
    Left = foo;
}

ui64 DES::i_perm(ui64 inp)
{
    // Initial permutation
    ui64 result = 0;
    for (ui8 i = 0; i < 64; i++)
    {
        result <<= 1;
        result |= (inp >> (64 - IP[i])) & LB64_MASK;
    }
    return result;
}

ui64 DES::f_perm(ui64 inp)
{
    // Final permutation
    ui64 result = 0;
    for (ui8 i = 0; i < 64; i++)
    {
        result <<= 1;
        result |= (inp >> (64 - FP[i])) & LB64_MASK;
    }
    return result;
}

ui32 DES::des_func(ui32 Right, ui64 key) // des_func(Right,k) function
{
    // applying expansion permutation to convert Right to 48-bits
    ui64 sbox_inp = 0;

    for (ui8 i = 0; i < 48; i++)
    {
```

```
        sbox_inp <<= 1;
        sbox_inp |= (ui64)((Right >> (32 - EXPANSION[i])) & LB32_MASK);
    }

    // XORing expanded Ri with Ki(round key)
    sbox_inp = sbox_inp ^ key;

    // Applying S-Boxes function and returning 32-bit data
    ui32 sbox_out = 0;
    for (ui8 i = 0; i < 8; i++)
    {
        // Finding Row no. from outer bits
        char row = (char)((sbox_inp & (0x0000840000000000 >> 6 * i)) >> (42 - 6 * i));
        row = (row >> 4) | (row & 0x01);

        // Finding Column no. from the middle 4 bits of input
        char column = (char)((sbox_inp & (0x0000780000000000 >> 6 * i)) >> (43 - 6 * i));

        sbox_out <<= 4;
        sbox_out |= (ui32)(SBOX[i][16 * row + column] & 0x0f);
    }

    // DES function straight DBox
    ui32 f_result = 0;
    for (ui8 i = 0; i < 32; i++)
    {
        f_result <<= 1;
        f_result |= (sbox_out >> (32 - PBOX[i])) & LB32_MASK;
    }

    return f_result;
}

ui64 DES::encrypt(ui64 inp)
{
    return des(inp, false); //Calling des function with mode "False" for encrypting "inp"
}

ui64 DES::decrypt(ui64 inp)
{
    return des(inp, true); //Calling des function with mode "True" for decrypting "inp"
}

ui64 DES::des(ui64 inp, bool mode) //Mode:: True-Decryption, False-Encryption
{
    // Applying initial permutation
    inp = i_perm(inp);
    // Splitting "inp" into two 32-bit parts
    ui32 Right = (ui32)(inp & L64_MASK);
    ui32 Left = (ui32)(inp >> 32) & L64_MASK;

    ui32 F;
    // 16 round Encryption
    for (ui8 i = 0; i < 16; i++)
    {
        if (mode)
            F = des_func(Right, sub_key[15 - i]);
        else
            F = des_func(Right, sub_key[i]);
        feistel_round(Left, Right, F);
    }

    // Swapping Left & R
```

```
    inp = (((ui64)Right) << 32) | (ui64)Left;
    // Applying final permutation
    return f_perm(inp);
}

//------------------------------\DES ENCRYPTION-------------------------------------

#endif
```

4.2 Attack

The "attack.cpp" header file contains the whole attack procedure.

```
#include "des.h"
#include <bits/stdc++.h>
#include <random>

// Number of inputs
#define N 500
// Number of attacks
#define NA 100


// Library Used to Generate Randon Numbers With a Uniform Distribuation
#define WORD_MASK (0xffffffffffffffffull) // 64 Bit Word Mask
std::mt19937 rng;
std::uniform_int_distribution<uint64_t> uni_dist(0x0ull, WORD_MASK);


ui64 v = 0x0000000000000000ull; // 48 bit representation of faulty indices in 8 S-Boxes.
ui8 L8_MASK = 0x80;

// Faulty SBOX values to used no the simulated attack
char SBOX_Faulty[8][64];

/**
* Fault injected DES class Derived from "DES" class
* Contains the functions for Faulty DES Encryption using a user defined key.
*
* DES_Faulty(ui64 key)            DES_Faulty Constructor
* des_Faulty(ui64 inp, bool mode) Driver function for faulty DES Encryption (Mode: True-Dec Fal
* encrypt_Faulty(ui64 inp)        Faulty DES encryption function(encrypt with key "key")
* des_func_Faulty(ui32 Right, ui64 key)     Function "f" that uses the faulty SBOX values
*/
class DES_Faulty : public DES
{
public:
    DES_Faulty(ui64 key);
    ui64 encrypt_Faulty(ui64 inp);
    ui64 des_Faulty(ui64 inp, bool mode);
    ui64 last_subkey();

protected:
    ui32 des_func_Faulty(ui32 Right, ui64 key);
};

DES_Faulty::DES_Faulty(ui64 key) : DES(key)
{
}

ui64 DES_Faulty::last_subkey()
```

```
{
    return sub_key[15];
}

ui64 DES_Faulty::encrypt_Faulty(ui64 inp)
{
    return des_Faulty(inp, false);
}

ui64 DES_Faulty::des_Faulty(ui64 inp, bool mode)
{
    // Applying initial permutation
    inp = i_perm(inp);

    // Splitting "inp" into two 32-bit parts
    ui32 Left = (ui32)(inp >> 32) & L64_MASK;
    ui32 Right = (ui32)(inp & L64_MASK);

    ui32 F;
    // 16 round Encryption
    for (ui8 i = 0; i < 16; i++)
    {
        if (i != 15)
        {
            if (mode)
                F = des_func(Right, sub_key[15 - i]);
            else
                F = des_func(Right, sub_key[i]);
        }
        else
        {
            if (mode)
                F = des_func_Faulty(Right, sub_key[15 - i]);
            else
                F = des_func_Faulty(Right, sub_key[i]);
        }

        feistel_round(Left, Right, F);
    }

    // Swapping Left & Right
    inp = (((ui64)Right) << 32) | (ui64)Left;
    // Applying final permutation
    return f_perm(inp);
}

ui32 DES_Faulty::des_func_Faulty(ui32 Right, ui64 key) // des_func(Right,k) function
{
    // applying expansion permutation to convert Right to 48-bits
    ui64 sbox_inp = 0;
    for (ui8 i = 0; i < 48; i++)
    {
        sbox_inp <<= 1;
        sbox_inp |= (ui64)((Right >> (32 - EXPANSION[i])) & LB32_MASK);
    }

    // XORing expanded Ri with Ki(round key)
    sbox_inp = sbox_inp ^ key;

    // Applying S-Boxes function and returning 32-bit data
    ui32 sbox_out = 0;
    for (ui8 i = 0; i < 8; i++)
    {
```

```
        // Finding Row no. from outer bits
        char row = (char)((sbox_inp & (0x0000840000000000 >> 6 * i)) >> (42 - 6 * i));
        row = (row >> 4) | (row & 0x01);

        // Finding Column no. from the middle 4 bits of input
        char column = (char)((sbox_inp & (0x0000780000000000 >> 6 * i)) >> (43 - 6 * i));

        sbox_out <<= 4;
        sbox_out |= (ui32)(SBOX_Faulty[i][16 * row + column] & 0x0f); //Using Fault injected S-
    }

    // DES function straight DBox
    ui32 f_result = 0;
    for (ui8 i = 0; i < 32; i++)
    {
        f_result <<= 1;
        f_result |= (sbox_out >> (32 - PBOX[i])) & LB32_MASK;
    }

    return f_result;
}

// A function to print the binary form of a 64 bit integer
int printBinary(ui64 n)
{
    for (int i = 0; i < 64; i++)
    {
        cout << ((n >> (63 - i)) & 1);
    }
    cout << endl;
    return 0;
}

/*
Performs a multi fault PFA on the given des_f comparing
with the correct ciphertext from des.
Faulty indices in SBOXs are given beforehand using mask and the remaining
keyspace at each input will be stored in analysis.
Returns the input index at which the attack is completed.
*/
int attack(DES des, DES_Faulty des_f, ui64 inputs[N], ui8 mask, int analysis[8][N])
{
    ui64 recovered_key = 0x0000000000000000;  // The recovered 48bit key from the attack

    ui8 key_pos = 0x00;  // Recovered key positions 6 bits at a time

    vector<ui64> keyspace[8];  // Vectors of impossible keys for every SBOX

    int n;  // Input index at which the attack is completed.

    // Persistent Fault Analysis iterating over the given inputs
    for (int i = 0; i < N; i++)
    {
        ui64 result = des.encrypt(inputs[i]);
        ui64 result_faulty = des_f.encrypt_Faulty(inputs[i]);

        // Reverse final permutation for ciphertexts
        ui64 before_fp_f = 0;
        ui64 before_fp = 0;
        for (ui8 i = 0; i < 64; i++)
        {
            before_fp <<= 1;
            before_fp_f <<= 1;
```

```
        before_fp |= (result >> (64 - IP[i])) & LB64_MASK;
        before_fp_f |= (result_faulty >> (64 - IP[i])) & LB64_MASK;
    }

    // Picking the last 32 bits of the permutaion inversed ciphertext
    ui32 a = before_fp & L64_MASK;
    // Applying expansion permutation to convert a to 48-bits
    ui64 a1 = 0;
    for (ui8 i = 0; i < 48; i++)
    {
        a1 <<= 1;
        a1 |= (ui64)((a >> (32 - EXPANSION[i])) & LB32_MASK);
    }

    // Indicator of the difference of outputs from the round function
    ui32 d_b1 = (before_fp ^ before_fp_f) >> 32;

    // Constant to reverse the straight d-box permutation in the round function
    static const char PBOXI[] =
        {
            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};
    // Reverse f straight d-box permutation
    ui32 d_b = 0;
    for (ui8 i = 0; i < 32; i++)
    {
        d_b <<= 1;
        d_b |= (d_b1 >> (32 - PBOXI[i])) & LB32_MASK;
    }
    ui64 KEY_MASK = 0x0000fc0000000000;
    ui32 B_MASK = 0xf0000000;
    // Iterating over every SBOX
    for (int j = 0; j < 8; j += 1)
    {
        // Performing differing SBOX outputs, already discovered key,
        // and checks of affected SBOX respectively (Strategy 1)
        if ((d_b & (B_MASK >> (j * 4))) && !(key_pos & (L8_MASK >> j)) && (mask & (L8_MASK >> j)))
        {
            analysis[j][i] = 1;  // The key is found
            recovered_key |= (KEY_MASK >> (j * 6)) & (a1 ^ v);
            // Recovered SBOX key appended to the master recovered_key

            key_pos |= (L8_MASK >> j);  // Indicator of discovered key at SBOX
            n = i;  // Input index of key recovery
            continue;
        }

        // Update the remaining keyspace at each input (Strategy 2)
        // Check for affected SBOX
        if (mask & (L8_MASK >> j))
        {
            // Check if key is already found
            if (analysis[j][((i == 0) ? 0 : (i - 1))] == 1)
            {
                analysis[j][i] = 1;
                continue;
            }
```

```
                // Update the list of impossible keys
                if (find(keyspace[j].begin(), keyspace[j].end(),
                (KEY_MASK >> (j * 6)) & (a1 ^ v)) == keyspace[j].end())
                {
                    keyspace[j].push_back((KEY_MASK >> (j * 6)) & (a1 ^ v));
                }
                analysis[j][i] = 64 - keyspace[j].size();
            }
        }
    }

    // Check if all faulty SBOX keys are recovered
    if (key_pos ^ mask)
        cout << "The full key has not been recovered. Please add more ciphertexts." << endl;

    cout << "Recovered key = ";
    printBinary(recovered_key);
    return n;
}

int main()
{
    // The Average Key space left after the queries (for analysis)
    ui64 anal_avg_single[8][N];
    ui64 anal_avg_multi[8][N];

    // Number of query at which we get the successful hit (for analysis)
    ui64 n_s[8][N];
    ui64 n_m[N];

    for (int i = 0; i < 8; i++)
        for (int j = 0; j < N; j++)
        {
            anal_avg_multi[i][j] = 0;
            anal_avg_single[i][j] = 0;
            n_s[i][j] = 0;
            n_m[j] = 0;
        }

    // Performing the attack NA times
    for (int a = 0; a < NA; a++)
    {
        // THE 56 BIT KEY
        ui64 key = uni_dist(rng) & 0xffffffffffffffull;
        DES des(key);
        DES_Faulty des_f(key);

        // Reset All Faulty SBOXs
        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 64; j++)
            {
                SBOX_Faulty[i][j] = SBOX[i][j];
            }
        }

        // Reset Remaining Key Space
        for (int i = 0; i < 8; i++)
            for (int j = 0; j < N; j++)
                analysis[i][j] = 64;

        int analysis[8][N];
```

```
        // Assigning random 64 bit inputs (Plain Text Bank)
        ui64 inputs[N];
        for (int i = 0; i < N; i++)
        {
            inputs[i] = uni_dist(rng);
        }

        cout << "\n\n**********ATTACK " << a + 1 << " **********
             << endl;



        // The key under attack
        cout << "Last Round Key:";
        printBinary(des_f.last_subkey());

        // For all possible single faults
        cout << "Single Faults----------" <<

        // Iterate through single faults in each S-Box
        for (int i = 0; i < 8; i++)
        {
            cout << "SBOX " << i + 1 << ":" << endl;

            // Mounting the Fault in the S-box randomly
            int i_t = uni_dist(rng) % 64;
            int t = uni_dist(rng) % 5 + 1;
            char row = (char)((i_t & 0x20) >> 4) | (i_t & 0x01);
            char column = (char)((i_t & 0x1e) >> 1);
            int index = 16 * row + column;  // Random index
            SBOX_Faulty[i][index] = (SBOX_Faulty[i][index] + t) % 16;
            // Introducing a random fault in a random index of ith SBOX

            v = ((ui64)i_t) << (48 - ((i + 1) * 6));  // Updating location of faulty index


            ui64 q_no = attack(des, des_f, inputs, L8_MASK >> i, analysis);  // Performing the attack
            n_s[i][q_no]++;

            SBOX_Faulty[i][index] = (SBOX_Faulty[i][index] - t + 16) % 16;  // Resetting the fault
        }
        for (int l = 0; l < 8; l++)
            for (int m = 0; m < N; m++)
                anal_avg_single[l][m] += analysis[l][m];

        // For multiple faults
        v = 0;
        cout << "Multiple Faults----------" <<

        // Mounting Fault in Each S-box
        for (int i = 0; i < 8; i++)
        {
            int i_t = uni_dist(rng) % 64;
            i_t = 63;
            int t = uni_dist(rng) % 5 + 1;

            char row = (char)((i_t & 0x20) >> 4) | (i_t & 0x01);
            char column = (char)((i_t & 0x1e) >> 1);
            int index = 16 * row + column;  // Random index
            SBOX_Faulty[i][index] = (SBOX_Faulty[i][index] + t) % 16;
            // Introducing a random fault in a random index of ith SBOX

            v |= ((ui64)i_t) << (48 - ((i + 1) * 6));
```

```
            // Updating location of faulty index

            for (int j = 0; j < N; j++)
                analysis[i][j] = 64;
        }
        ui64 q_no = attack(des, des_f, inputs, 0xff, analysis);  // Performing the attack
        n_m[q_no]++;

        for (int l = 0; l < 8; l++)
            for (int m = 0; m < N; m++)
                anal_avg_multi[l][m] += analysis[l][m];
    }

    // Preparing the analysis outputs

    // Average number of inputs for single faults
    float avg_s[8] = {0};
    // Average number of inputs for multiple faults
    float avg_m = 0;

    // Averaging analysis over all attacks
    for (int m = 0; m < N; m++)
    {
        avg_m += n_m[m] * (m + 1);

        for (int l = 0; l < 8; l++)
        {
            avg_s[l] += n_s[l][m] * (m + 1);
            anal_avg_single[l][m] /= NA;
            anal_avg_multi[l][m] /= NA;
        }
    }
    avg_m /= NA;

    for (int l = 0; l < 8; l++)
        avg_s[l] /= NA;

    // Exporting analysis of remaining keyspace to a .csv file for single faults
    ofstream out1("KeySpace_Single.csv");
    for (auto &row : anal_avg_single)
    {
        for (auto col : row)
            out1 << log2(col) << ',';
        out1 << '\n';
    }

    // Exporting analysis of remaining keyspace to a .csv file for multiple faults
    ofstream out2("KeySpace_Multi.csv");
    for (int i = 0; i < N; i++)
    {
        ui64 t = 1;
        for (int j = 0; j < 8; j++)
            t *= anal_avg_multi[j][i];
        out2 << log2(t) << ',';
    }

    // Exporting analysis of completed attacks to a .csv file for single faults
    ofstream out3("NumberOfSolved_Single.csv");
    for (auto &row : n_s)
    {
        ui64 t = 0;
        for (auto col : row)
        {
```

```
            t += col;
            out3 << t << ',';
        }
        out3 << '\n';
    }

    // Exporting analysis of completed attacks to a .csv file for multiple faults
    ofstream out4("NumberOfSolved_Multi.csv");
    ui64 t = 0;
    for (int i = 0; i < N; i++)
    {
        t += n_m[i];
        out4 << t << ',';
    }

    cout << "\n\nFor single faults, average querys required: " << endl;
    for (int i = 0; i < 8; i++)
        cout << "SBOX " << i + 1 << ": " << avg_s[i] << endl;
    cout << "For multiple faults, average queries required: " << avg_m << endl;

    return 0;
}
```

Bibliography

[1] Xinjie Zhao Shivam Bhasin Wei He Ruyi Ding Samiya Qureshi Fan Zhang Xiaoxuan Lou and Kui Ren. “Persistent fault analysis on block ciphers”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems*. 2018, 150–172.