Université de Versailles – Saint-Quentin-en-Yvelines

CC Calcule securisé

Meribout Ahmed Yahia

22303132

September 23, 2024

Contents

1 Exercise 1 (fault attacks on DES)

Differential fault analysis (DFA) is a type of active side-channel attack in the field of cryptography, specifically cryptanalysis.

Taking a smartcard containing an embedded processor as an example, some unexpected environmental conditions it could experience include being subjected to high temperature, receiving unsupported supply voltage or current, being excessively overclocked, experiencing strong electric or magnetic fields, or even receiving ionizing radiation to influence the operation of the processor. When stressed like this, the processor may begin to output incorrect results due to physical data corruption [2].This can help cryptanalysts obtain information about the secret key.

In case of DES we have the following :

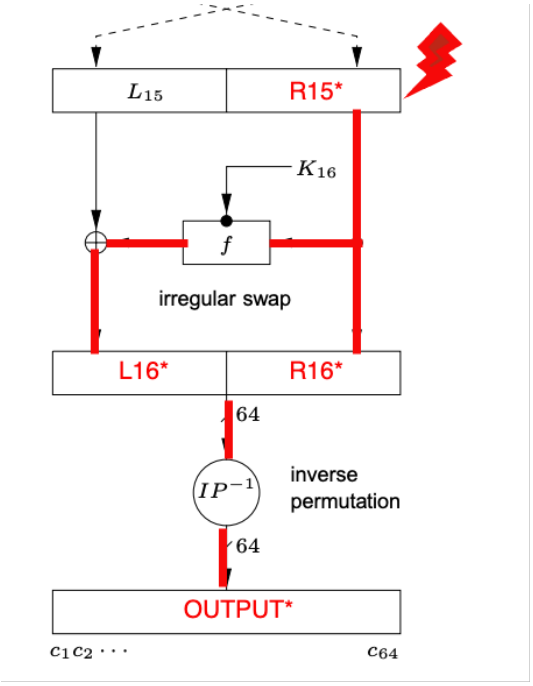

Figure 1: Making a fault in R_{15} of DES to get a diffrent output cypher [3]

As shown in Figure 1, introducing a fault in R_{15} of DES causes the fault to propagate to L_{16} and R_{16}, altering the output and resulting in a faulted ciphertext. The following demonstrations will illustrate the effectiveness of this attack.

2 Exercice 2 (finding K16)

From the last round we know:

$$l_{16} = f(k_{16} \oplus r_{15}) \oplus l_{15} \quad , \quad r_{16} = r_{15}$$

```
#Cyphertex without the fault
ctpermuted= permute(hex2bin(ct),initial_perm,64)
l = ctpermuted[0:32]
r = ctpermuted[32:64]
```

So with the fault in r_{15} we have

$$l^*_{16} = f(k_{16} \oplus r^*_{15}) \oplus l_{15} \quad , \quad r^*_{16} = r^*_{15}$$

```
#Cyphertext with the fault
dct = diffct[0]
dctpermuted= permute(hex2bin(dct),initial_perm,64)
dl16 = dctpermuted[0:32]
dr16 = dctpermuted[32:64]
```

If we xor l_{16} with l^*_{16} we get:

$$l_{16} \oplus l^*_{16} = f(k_{16} \oplus r_{15}) \oplus f(k_{16} \oplus r^*_{15})$$

$$l_{16} \oplus l^*_{16} = P(S(E(k_{16} \oplus r_{15}))) \oplus P(S(E(k_{16} \oplus r^*_{15})))$$

```
xor_l_dl = xor(l16,dl16)
```

We got rid of the l_{15} in the equation next we try to make the equation easy to solve:

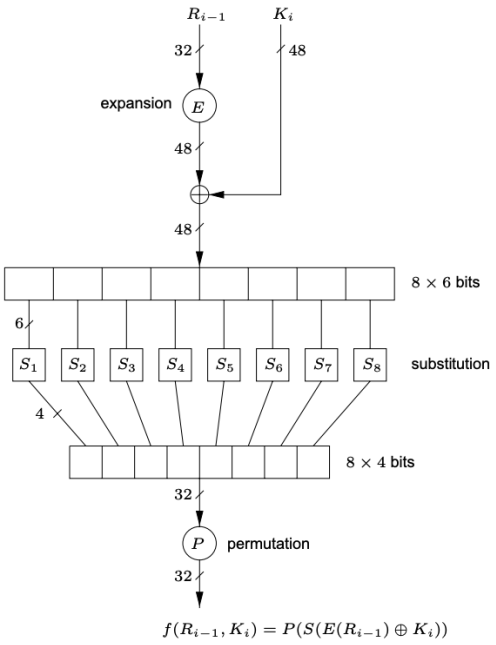

Figure 2: Figure showing f function in DES [3]

After applying the P^{-1} permutation to both sides, the equation becomes:

$$P^{-1}(l_{16} \oplus l_{16}^*) = S(E(k_{16} \oplus r_{15})) \oplus S(E(k_{16} \oplus r_{15}^*))$$

```
#Create the inverse of permutation P
inverse_per = [0] * len(per)
for i, num in enumerate(per):
    inverse_per[num - 1] = i + 1

# Permute the XOR result of left and right halves with the inverse
    permutation P
A = permute(xor_l_dl, inverse_per, 32)

# Permute the right half with the expansion permutation to get a 48-
    bit result
Er = permute(r16, exp_d, 48)
```

```
# Permute the faulty right half with the expansion permutation to
    get a 48-bit result
Edr = permute(dr16, exp_d, 48)
```

Now we get a series of 8 equations to solve :

$$P^{-1}(l_{16} \oplus l_{16}^*)_{\text{0 to 3 bit}} = (S(E(k_{16} \oplus r_{15})) \oplus S(E(k_{16} \oplus r_{15}^*)))_{\text{0 to 3 bit}} \quad (1)$$

$$P^{-1}(l_{16} \oplus l_{16}^*)_{\text{4 to 8 bit}} = (S(E(k_{16} \oplus r_{15})) \oplus S(E(k_{16} \oplus r_{15}^*)))_{\text{4 to 8 bit}} \quad (2)$$

$$\vdots$$

$$P^{-1}(l_{16} \oplus l_{16}^*)_{\text{23 to 27 bit}} = (S(E(k_{16} \oplus r_{15})) \oplus S(E(k_{16} \oplus r_{15}^*)))_{\text{23 to 27 bit}} \quad (7)$$

$$P^{-1}(l_{16} \oplus l_{16}^*)_{\text{28 to 31 bit}} = (S(E(k_{16} \oplus r_{15})) \oplus S(E(k_{16} \oplus r_{15}^*)))_{\text{28 to 31 bit}} \quad (8)$$

Each equation is related to 6 bits of the key and 6 bits of the result of the Expansion P-box. In our case, the unknown is K_{16}.

To find the key, we need to brute-force each equation to find all possible solutions. The cost of this brute-force approach is $2^6 \times 8$, which is manageable.

Since the S-boxes are not bijective, each equation will have multiple solutions, most of which are not part of the final correct key. To eliminate incorrect solutions, we test each potential solution against other faulted ciphertexts.

Instead of brute-forcing the solutions again, we test the possible solutions obtained from the first faulted ciphertexts.

The steps can be summarized as follows:

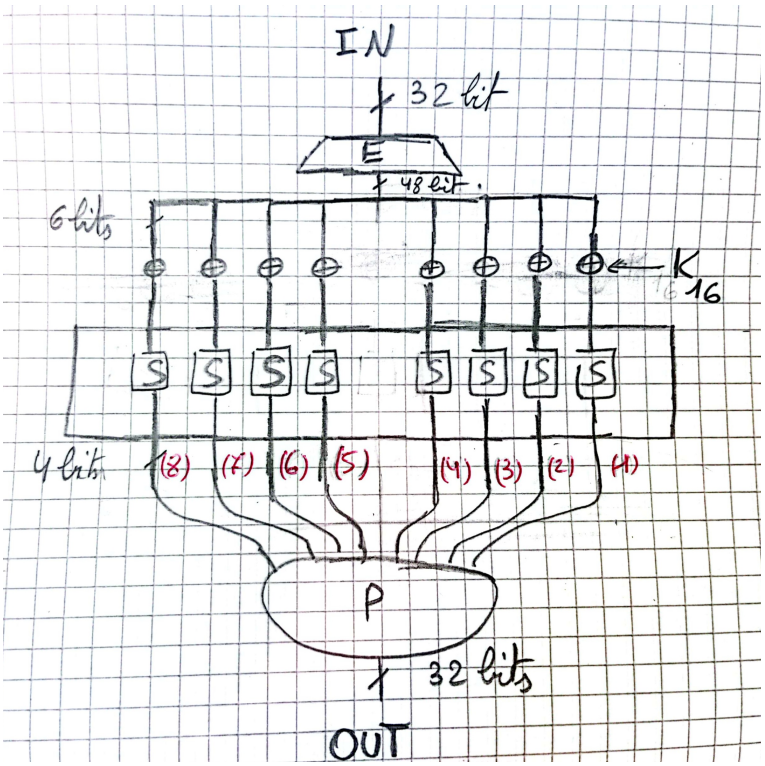

Figure 3: Drawing showing a visual representation for each equation

1. Brute-force each equation related to 6 bits of the key and the Expansion P-box.
2. Identify multiple solutions due to the non-bijective nature of the S-boxes.
3. Test these multiple solutions against other faulted ciphertexts.
4. Eliminate incorrect solutions and find the correct key bits.

And at the end of these steps we get the subkey K_{16}.

To translate what has been explained to code:

```
    # Split the result of permutation A into segments of 4 bits
    A_array = split_binary_into_segments(A, 4)

    # Split the result of permutation Er into segments of 6 bits
    Er_array = split_binary_into_segments(Er)
```

```
    # Split the result of permutation Edr into segments of 6 bits
    Edr_array = split_binary_into_segments(Edr)


     # Generate all possible combinations of 6 bits
    bits = generate_bit_combinations(6)

    for j in range(0, 8):
        for sk in bits:
            # Calculate the output of S-boxes with the faulted and non-
                faulted inputs
            B = xorthenSbox(Edr_array[j], sk, j)
            C = xorthenSbox(Er_array[j], sk, j)
            # XOR the outputs of the S-boxes
            x = xor(B, C)
            # Check if the result matches with the corresponding segment
                 of A
            if A_array[j] == x:
                # If there is a match, add the key to the list of
                    possible keys
                possible_keys[j].append(sk)
```

the result of this code gives us all the possible solutions which are many because of the S-box nature so we need to filter the solutions. as shown in the following output :

```
for i,line in enumerate(possible_keys):
    print("equation ", i+1, " : ", len(line), "solutions")

# Output
equation  1  :  10  solutions
equation  2  :  64  solutions
equation  3  :  64  solutions
equation  4  :  64  solutions
equation  5  :  10  solutions
equation  6  :  64  solutions
equation  7  :  64  solutions
equation  8  :  8  solutions
```

To filter out the bad solutions, we will utilize other faulty ciphers. We will attempt to associate each equation with the potential solutions we found earlier. If the potential solution satisfy the equation, we retain it and proceed to the next test. If not, we remove it from the pool of potential solutions. as shown in the following code:

```
    #Iterate over the different cyphertexts with faults
    for i in diffct:
        #Redo the initial steps with the new cyphertext
        dct = i
        dctpermuted= permute(hex2bin(dct),initial_perm,64)
        dl = dctpermuted[0:32]
        dr = dctpermuted[32:64]
        xor_l_dl = xor(l16,dl)
        A = permute(xor_l_dl,inverse_per,32)
        A_array = split_binary_into_segments(A,4)
        Er = permute(r16,exp_d,48)
        Edr = permute(dr,exp_d,48)
        Er_array = split_binary_into_segments(Er)
        Edr_array = split_binary_into_segments(Edr)

        #Iterate over the possible keys and remove the ones that don't
            match
        for j in range(0, 8):
            for s in possible_keys[j]:
                B = xorthenSbox(Edr_array[j],s,j)
                C = xorthenSbox(Er_array[j],s,j)
                x = xor(B,C)
                if A_array[j] != x:
                    #This solution doesnt satisfy the equation so we
                        remove it from the array
                    possible_keys[j].remove(s)
```

At the end of this code, the potential solutions will be narrowed down to one solution (6 bits of the key) per equation . By combining these solutions, we obtain the final round key K_{16}.

```
for i,line in enumerate(possible_keys):
    print("equation ", i+1, ": ", len(line), "solution after filtering"
        )

# Output
equation  1  :  1 solution after filtering
equation  2  :  1 solution after filtering
equation  3  :  1 solution after filtering
equation  4  :  1 solution after filtering
equation  5  :  1 solution after filtering
equation  6  :  1 solution after filtering
equation  7  :  1 solution after filtering
equation  8  :  1 solution after filtering
```

Now all we need to do to get the K16 is to combine these solutions into string of bits.

```
    # Join the inner arrays into strings
    inner_strings = [''.join(inner) for inner in possible_keys]

    # Join the inner strings into a final string
    k16 = ''.join(inner_strings)
```

in my case the result is the following

```
K16:  011100110000101011011110100111110010011110100111  ( 48  bits)
K16:  730ADE9F27A7  ( 12  hex)
```

2.

K16: 011100110000101011011110100111110010011110100111
K16: 730ADE9F27A7

3 Exercise 3 (find 64-bit key)

To find the full 56-bit key from the 48-bit round key K_{16}, we need to understand the key scheduling and permutation process in the Data Encryption Standard (DES).

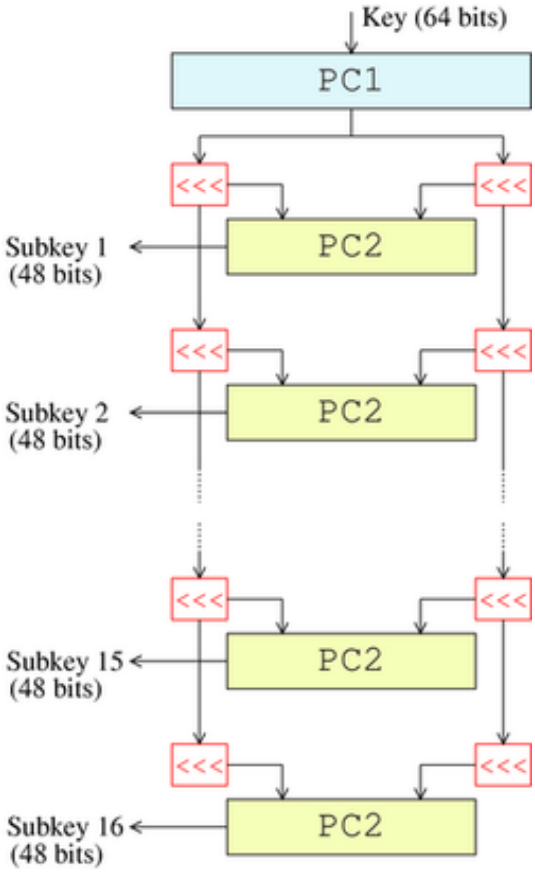

Figure 4: The key-schedule of DES [1]

Key Scheduling in DES

- **Initial Key** K: The original key K is a 64-bit key, but only 56 bits are used for the encryption process. The remaining 8 bits are used for parity checking and are not involved in the encryption.
- **Permuted Choice 1 (PC1)**: The 64-bit key K undergoes a permutation called Permuted Choice 1 (PC1). This permutation selects 56 bits from the 64-bit key according to a predefined table. The result of this permutation is a 56-bit key K'.

$$\text{PC1}(K) = K'$$

Round Key Generation

The 56-bit key K' is then split into two 28-bit halves, C_0 and D_0:

$$K' = (C_0, D_0)$$

For each of the 16 rounds in DES, the two halves C and D are subjected to left shifts (which vary per round), producing pairs C_n and D_n for $n = 1$ to 16.

The round keys K_n are generated from these pairs by applying another permutation, known as Permuted Choice 2 (PC2), which selects 48 bits from the combined 56 bits of C_n and D_n:

$$K_n = \text{PC2}(C_n, D_n)$$

In our case, the round key that we currently know is K_{16}. By substituting into the equation, we get:

$$K_{16} = \text{PC2}(C_{16}, D_{16})$$

Since this is the last round, the pairs (C_{16}, D_{16}) represent the state of the key halves after all the shifts have been applied throughout the 16 rounds. However, since we're at the end of the process, (C_{16}, D_{16}) effectively becomes the same as the initial key halves (C_0, D_0). This is because the shifts in DES effectively cycle back to the original state due to the nature of the key scheduling. Therefore, we can consider C_{16} and D_{16} to be equivalent to C_0 and D_0.

So, to form the 56-bit key K', we can directly combine C_{16} and D_{16}:

$$K' = (C_{16}, D_{16})$$

To get to this result we apply the inverse of the PC2 permutation to the round key K_{16} that we found

$$(C_{16}, D_{16}) = \text{PC2}^{-1}(K_{16})$$

```
    # Permute the key using inverse_pc2 permutation
    permuted_key = permuteArray(k16,inverse_pc2,56)
```

We get the following result :

```
permuted_key = ['0', '0', '1', '0', '0', '0', '1', '1', None, '0', '1',
    '1', '1', '0', '0', '0', '1', None, '0', '1', '0', None, '1', '1',
    None, '1', '1', '1', '1', '1', '0', '1', '1', '1', None, '1', '1',
    None, '1', '1', '1', '0', None, '0', '0', '1', '1', '0', '1', '0',
    '0', '0', '0', None, '1', '1']
```

As we can see the resulted K' isn't complete since PC2 gets 56 bits as inputs and 48 bits as output so the resulting array would have 8 bits missing. These missing bits need to be determined through a brute-force approach, comparing the resulting cipher to the original one. If a match is found, the 56-bit key is found.

```
    # Generate all possible combinations of the permuted key
    all_combinations = generate_combinations(permuted_key)

    # Initialize the final key variable
    final_key = ""

    # Iterate through each combination
    for combination in all_combinations:
        # Convert combination to string
        key = ''.join(combination)

        # Encrypt plaintext using current key
        cipher = DES(pt, key)

        # Check if ciphertext with the current key combination matches
            the ciphertext without fault
        if cipher == ct:
            # If match found, store the key and exit the loop
            final_key = key
            break
```

The result of this code is:

K' : 00100011101110001101001101111101111111111010011010000111
K' : 23B8D37DFFA687

Since now we have the 56-bit key, we can decrypt anything using that key as it's the most important one. However, to find the full 64-bit key, we need to apply the inverse of the PC1 permutation and then add the parity bits. The steps are as follows:

1. Apply the Inverse of PC1:

$$K = \text{PC1}^{-1}(K')$$

2. Separate Bits into Groups of 7: Divide the 56-bit key K' into 8 groups, each containing 7 bits:

$$\begin{aligned}&\{b_1, b_2, b_3, b_4, b_5, b_6, b_7\},\\&\{b_8, b_9, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}\},\\&\vdots\\&\{b_{50}, b_{51}, b_{52}, b_{53}, b_{54}, b_{55}, b_{56}\}\end{aligned}$$

3. Add Parity Bits: For each group of 7 bits, count the number of 1s:

- If the number of 1s is odd, add a 0 bit at the end.
- If the number of 1s is even, add a 1 bit at the end.

Each group should now have 8 bits.

4. Convert to Hexadecimal: Convert each 8-bit group to its hexadecimal representation to get the full 64-bit key:

$$\begin{aligned}&\{b_1, b_2, b_3, b_4, b_5, b_6, b_7, p_1\},\\&\{b_8, b_9, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}, p_2\},\\&\vdots\\&\{b_{50}, b_{51}, b_{52}, b_{53}, b_{54}, b_{55}, b_{56}, p_8\}\end{aligned}$$

Where p_i is the parity bit added to the i-th group.

Finally, convert the 64 bits to hexadecimal to get the full 64-bit key.

To translate this into code :

```
#Defining our add parity function
def add_parity_bits(key_56bit):
    if len(key_56bit) != 56:
        raise ValueError("Key must be 56 bits long")

    key_64bit = []
    for i in range(8):
        # Extract the 7 bits for this byte
        byte = key_56bit[i*7:(i+1)*7]

        # Calculate the parity bit (odd parity)
        parity_bit = '1' if byte.count('1') % 2 == 0 else '0'

        # Append the 7 bits and the parity bit to form the 8-bit byte
        key_64bit.append(byte + parity_bit)

    return ''.join(key_64bit)
```

```
    print("K' :-" , final_key)
    print("K' :-" , bin2hex(final_key))

    final_key = permuteArray(final_key ,inverse_pc1 ,64)
    final_key = ''.join(final_key)
    print("Key-Without-parity-bits:-" , bin2hex(final_key))
    final_key = add_parity_bits(final_key)
    print("Key-With-parity-bits-:-" , final_key)
    print("Key-With-parity-bits-:-" , bin2hex(final_key))
```

The final output is the following :

```
Key Without parity bits:  B36C4A777B8FB3
Key found :
   10110011101101100001001101001111011101101101110000111111001100111
Key found :  B3B6134F76DC3E67
```

To test the final result, we encrypt the plaintext with the found key using all the normal DES operations and we should get a match.

```
    #Test of the result :

    # getting 56 bit key from 64 bit using the parity bits
    key = permute(final_key , pc1, 56)
    #Encryption
    cypher = DES(pt,key)
    #Printing the result
    print("Cypher-:-" , cypher)
    print("Cypher-test-:-" , ct)

    if cypher == ct:
        print("The-key-is-correct")
```

and finally the output :

```
Key With parity bits :  B3B6134F76DC3E67
Cypher :  E1EC5D57AF996C74
Cypher test :  E1EC5D57AF996C74
The key is correct
```

2. **My encryption key is : B3B6134F76DC3E67.**

Total complexity to find the whole key is : $O(2^6 \times 8 + 2^8) \approx O(2^9)$

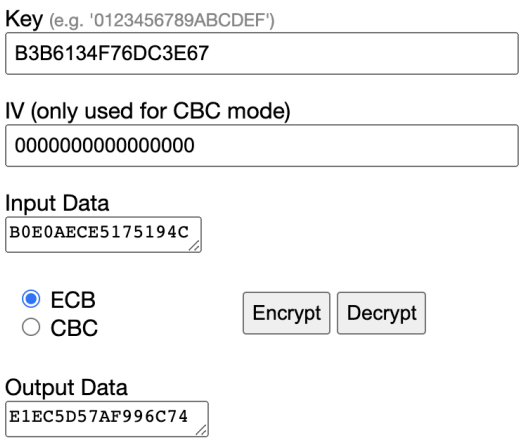

Figure 5: Screenshot showing that we got the correct key

4 Exercice 4:

4.1 Fault on the R_{14}

We start looking at equations :

$$
\begin{aligned}
L_{16} &= L_{15} \oplus F(K_{16} \oplus R_{15}) = R_{14} \oplus F(K_{16} \oplus R_{15}) \\
L^*_{16} &= L^*_{15} \oplus F(K_{16} \oplus R^*_{15}) = R^*_{14} \oplus F(K_{16} \oplus R^*_{15}) \\
R_{16} &= R_{15} = L_{14} \oplus F(K_{15} \oplus R_{14}) \\
R^*_{16} &= R^*_{15} = L_{14} \oplus F(K_{15} \oplus R^*_{14}) \\
R_{15} &= L_{14} \oplus F(K_{15} \oplus (L_{16} \oplus F(K_{16} \oplus R_{16}))) \\
R^*_{15} &= L_{14} \oplus F(K_{15} \oplus (L^*_{16} \oplus F(K_{16} \oplus R^*_{16}))) \\
R_{15} \oplus R^*_{15} &= F(K_{15} \oplus (L_{16} \oplus F(K_{16} \oplus R_{16}))) \oplus F(K_{15} \oplus (L^*_{16} \oplus F(K_{16} \oplus R^*_{16})))
\end{aligned}
$$

From the last equation we find the unknown values in red and the known values in green.

$$
\begin{aligned}
(R_{15} \oplus R^*_{15})_{\text{i to j bits}} = {} & F(K_{15} \oplus (L_{16} \oplus F(K_{16} \oplus R_{16})))_{\text{i to j bits}} \\
& \oplus F(K_{15} \oplus (L^*_{16} \oplus F(K_{16} \oplus R_{16})))_{\text{i to j bits}}
\end{aligned}
$$

To analyze the impact of a fault on each S-box, we consider all possible combinations of K_{16} and K_{15} for each S-box. Since each S-box operates on 6 bits, the total number of combinations is $2^6 \times 2^6 = 2^{12}$ for each S-box. Given that there are 8 S-boxes, the total number of combinations to test becomes $2^{12} \times 2^3 = 2^{15}$. This represents the complexity of finding all possible key solutions for each S-box.

However, after identifying potential key solutions, we need to eliminate false positives caused by other faults. Subsequently, brute-forcing the remaining 8 bits of the key is necessary. This final brute force step adds a complexity of $O(2^8)$.

Thus, the overall complexity of the attack, considering both the initial analysis and the subsequent brute force, is $O(2^{15} + 2^8)$, which simplifies to $O(2^{15})$. While this complexity represents an increase compared to attacks targeting R_{15}, which have a complexity of $O(2^9)$, it remains significantly lower than a brute force attack on the full 56-bit key..

4.2 Fault on the R14

We know :

$$R_{14} = L_{13} \oplus f(K_{14} \oplus R_{13})$$

By replacing we find

$$\begin{aligned} L_{16} &= L_{15} \oplus f(K_{16} \oplus R_{15}) \\ &= R_{14} \oplus f(K_{16} \oplus R_{15}) \\ &= L_{13} \oplus f(K_{14} \oplus R_{13})) \oplus f(K_{16} \oplus R_{15}) \end{aligned}$$

Next we do the following we find :

$$
\begin{aligned}
&(L_{16} \oplus L_{16}^{*})_{\text{i to j bits}} \\
&\quad = f(K_{14} \oplus R_{13})_{\text{i to j bits}} \\
&\qquad \oplus f(K_{16} \oplus R_{15})_{\text{i to j bits}} \\
&\qquad \oplus f(K_{14} \oplus R_{13}^{*})_{\text{i to j bits}} \\
&\qquad \oplus f(K_{16} \oplus R_{15}^{*})_{\text{i to j bits}}
\end{aligned}
$$

In this case, we have 4 unknowns, each represented by 6 bits. To explore all possible combinations, we need to consider 2^6 options for each unknown. Therefore, the total number of combinations is $(2^6)^4$. So for 1 equation, the complexity would be 2^{24}, and since we have 8 equations, the total cost is 2^{27}. Again, While this complexity represents an increase compared to attacks targeting specific rounds, such as $R_{15}$and R_{14} , it remains significantly lower than a brute force attack on the full 56-bit key, which would be $O(2^{56})$.

We can practically induce faults in the earlier rounds and still be able to find the complete key. However, eventually, the number of tests required to explore all possible combinations will exceed 2^{56}. At that point, it becomes more efficient to simply brute force the entire 56-bit key.

5 Exercise 5 :

There are several countermeasures that can be used to protect an algorithm against this type of attack.

Random Delay: If a high degree of precision is required the attack could be slowed to the point where an attacker will not believe the attack is possible. This applies to both hardware and software random delays.

However, since CPU cycles are available for other tasks, the impact

on performance is relatively minor. Thus, while there may be a slight downgrade in performance, the benefit of heightened security.

Redundancy: We can repeat the calculations multiple times, say 3 or more times, as a verification strategy, making it hard, if not almost impossible, for an attacker to induce an exact fault on all the calculations simultaneously.

However, this countermeasure significantly impacts performance since it takes at least twice as many CPU cycles to verify the output.

Shielding: We can shield the hardware that is performing the encryption physically, for example, using special materials to prevent electromagnetic interference and electrical injection. While this solution does not impact performance, it could be costly for manufacturers to implement.A possible solution for this is just to shield the last 5 rounds of DES since any fault earlier is just going to make the Key search harder for the attacker than just search for the 56-bit key

The previous solutions can be combined to form multiple lines of defense. This way, if an attacker finds a way to countermeasure one solution, there is still another layer of security in place.

Useful URLs

DES implementation

TU Berlin DES Page

DES Key Parity Bit Calculator

EMV Lab DES Calculator

References

[1] Data Encryption Standard - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Data_Encryption_Standard.

[2] Differential fault analysis - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Differential_fault_analysis.

[3] eCampus — ecampus.paris-saclay.fr. https://ecampus.paris-saclay.fr/pluginfile.php/2962412/mod_resource/content/0/DES.pdf.