Advanced Topics In Cybersecurity 2

Structural Attacks

Federico Dittaro, 12345773

Declaration

I declare that I am the sole author of this submission. I take full responsibility for all content. I confirm that I have either not used AI based tools at all, or, that I have only used them for the purposes of checking grammar and spelling.

1 Differential Cryptanalysis of a Single Round Cipher

1.1 Difference distribution table

The difference distribution table lists, for each input difference (rows) and output difference (column) the frequency by which it occurs. The maximum number that can be spotted in the table is 8. Since we are dealing with 3-bit numbers eight is also the theoretical maximum ($2^3 = 8$).
The first row correspond to the number of input pairs with input difference equal to zero. This means that the two input are the same and so the output difference will be zero. So, the total number of occurrences is position (0,0) is 8.
The first column correspond to the number of input pairs with output difference equal to zero. For the first row the result is 8 since we are back to the previous point while in all the other rows is 0 because a well-designed cipher should scramble inputs such that different inputs lead to different outputs.
Finally all the numbers in the table are even because for any input difference there exist two symmetric pairs. For example considering an input difference of 1, there exist (among all the other pairs) the input pair (0,1) and (1,0). This symmetry leads two the same result when the two numbers are XORED ($0 \oplus 1 = 1 \oplus 0$).
For any input difference the total number of pairs is 8, but since there exist a symmetric pair the total number of different pairs is 4 (which is also the highest number that can be spotted in the table for any row $\neq 0$).

1.2 Input-output difference to substitution box

The code uses as differential the pair (2, 4) which has a value of 4 in the DDT. This means that an input difference of 2 leads to an output difference of 4 four times. In this

case it is a good differential since it is the highest possible value that can be spotted in any row $\neq 0$. Choosing a good input-output pair it is very important for the attack complexity since that a good choice reduces the number of chosen plaintexts needed, narrows the space of possible keys faster and lowers the computational effort to recover the key.
In this case it has been used the value (2, 4) but other good input-output pairs are (2, 3), (5, 3), (5, 7), (7, 4), (7, 7).

1.3 Input-output pairs

In practice it is not possible to list all the pairs of plaintexts and ciphertexts that correspond to the chosen differential because the total number is enormous and therefore infeasible (the total number of possibilities is exponential w.r.t. the number of bits).
In a real-world scenario, instead of listing all possible pairs, the attacker relies on statistical analysis. They generate and analyze a manageable subset of plaintext pairs with the chosen input difference, observing the resulting ciphertext differences. By identifying patterns in the observed differences, the attacker determines which pairs satisfy the desired differential and uses this information to proceed with the attack.

2 Differential Cryptanalysis of a Two Round Cipher

In this case we are dealing with 4-bit numbers instead of 3-bit numbers and also the encryption function performs another substitution after applying the k1 key. So the encryption and decryption rules are:

$$Enc : c = S(S(m \oplus k_0) \oplus k_1)$$
$$Dec : m = S^{-1}(S^{-1}(c) \oplus k_1)) \oplus k_0$$

After adapting the code so that any call made to the 3-bit substitution box is now made to the 4-bit substitution box, and any call to an encryption function with a one-round cipher is now made to the encryption function of the two-round cipher, only one observation remains. This observation is that we can reduce ourselves to the previous case with only one substitution box by inverting the last SBox application. The box below shows the modified part of the code. (the entire one can be found in the file *diff_cryptanalysis_4bit.py*).

```
s_box = toy_ciphers.s_4b
    inv_s_box = [s_box.index(i) for i in range(len(s_box))]

    diff_dist_table = difference_distribution_table(s_box)
    print('Difference table:\n')
    pprint(diff_dist_table)
    print('')
    good_difference = (6,2)
    good_pairs = get_good_pairs(good_difference[0],good_difference[1],
       s_box)
    print('For the input-output difference',good_difference,'the good (
       u,v) pairs are:')
    pprint(good_pairs)
    print('')
    good_input=[]
    for m in range(16):
        m_d = m^6
        c =  toy_ciphers.encrypt_sub_2r(m)
        c = inv_s_box[c]
        c_d= toy_ciphers.encrypt_sub_2r(m_d)
        c_d = inv_s_box[c_d]
        if c^c_d ==2:
            good_input.append(((m,m_d),(c,c_d)))
    print('For the above input-output difference, the good input-output
        pairs ((m, m+delta_m),(c, c+delta_c)) are:')
    pprint(good_input)
    print('')
```

3 Differential Cryptanalysis of a two Round SPN Cipher

3.1 Permutation size

A small permutation, applied independently to multiple small parts of the cipher's state, can be sensible in some scenarios because it might not spread the influence of a bit change across the entire cipher state. This could weaken resistance to differential and linear cryptanalysis and therefore be vulnerable.
Also designing the permutation size to align with the output size of the substitution box could be sensible to attacks. It could allow differences to remain confined to certain parts of the state, making it easier for an attacker to trace differences round by round.

3.2 Extension of the encryption and decryption function

The implementation of the encryption function for a two round SPN encryption computes the following steps:

1. Divides the input in two 4-bit chunks;
2. Applies the SBox;
3. Applies k_0 to both chunks;
4. Merges again the two halves and applies the permutation step. Then it divides the result again;
5. Applies k_1 to both chunks;
6. Applies the Sbox and merges the two halves to give it as the final result

The decryption algorithm just applies the steps in the reverse order. The encryption code is reported in the box below (the whole code can be found inside the *toy_cipher.py* file).

```
def encrypt_sub_2r_spn(state):
    state = [(state >> 4) & 0xF, state & 0xF]

    state = [state[i] ^ k_0_8b[i] for i in range(len(state))]
    state = [s_4b[x] for x in state]
    binary_str = ''.join(f"{x:04b}" for x in state)
    permuted_str = ''.join(binary_str[i] for i in p_8b)
    state = [int(permuted_str[:4], 2), int(permuted_str[4:], 2)]

    state = [state[i] ^ k_1_8b[i] for i in range(len(state))]
    state = [s_4b[x] for x in state]

    return (state[0] << 4) | state[1]
```

2 round SPN Cipher

3.2.1 Corresponding differential attack

Differently from the previous case, here we are dealing with two SBoxes and there is a permutation layer before applying k_1. We can still reuse again most of the previous script. In fact, after adapting the script such that now the functions work with 8-bit we can reconduct ourselves to the previous case by inverting the SBox and the permutation layer. At the end of this process we are in the situation represented in the picture.

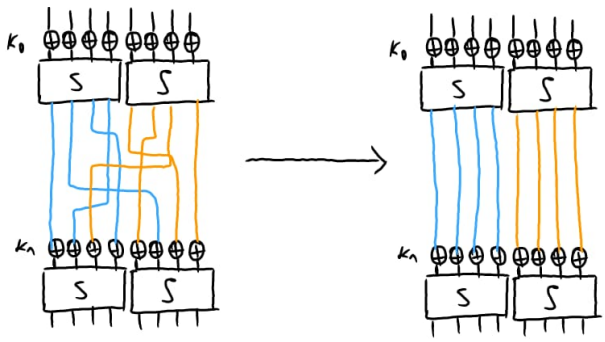

Figure 1: The left part represents the original encryption, the right one represent the situation after inverting the SBox and the permutation layer.

Now we can proceed by attacking the two SBoxes independently. To do such an attack we need four lists:

- k0_list_left which represents the first 4-bit of the key k_0;
- k0_list_right which represents the second 4-bit of the key k_0;
- k1_list_left which represents the first 4-bit of the key k_1;
- k1_list_right which represents the second 4-bit of the key k_1;

Now by trying the combinations between the right part and the left part we can find the original key. The box below represents the main parts of the code, the whole code can be found inside the file *diff_cryptanalysis_8bit.py*.

```
good_input_right=[]
    good_input_left=[]
    for m_right in range(16):
        m_d_right = m_right^6
        m_left = m_right << 4
        m_d_left = m_left^(6 << 4)
        c_right =  toy_ciphers.encrypt_sub_2r_spn(m_right)
        c_right = inverse_sbox_pbox(c_right)
        c_d_right = toy_ciphers.encrypt_sub_2r_spn(m_d_right)
        c_d_right = inverse_sbox_pbox(c_d_right)
        c_left = toy_ciphers.encrypt_sub_2r_spn(m_left)
        c_left = inverse_sbox_pbox(c_left)
        c_d_left = toy_ciphers.decrypt_sub_2r_spn(m_d_left)
        c_d_left = inverse_sbox_pbox(c_d_left)

        if c_right[1]^c_d_right[1] ==2:
            good_input_right.append(((m_right,m_d_right),(c_right[1],
                c_d_right[1])))
```

```
        if c_left[0]^c_d_left[0] ==2:
            good_input_left.append(((m_left >> 4,m_d_left >> 4),(c_left
                [0],c_d_left[0])))
    print('For the above input-output difference, the good input-output
        pairs ((m, m+delta_m),(c, c+delta_c)) are:')
    pprint(good_input_right)
    pprint(good_input_left)
    print('')

    k0_list_right: list[int] = [0]*len(good_pairs)
    k0_list_left: list[int] = [0]*len(good_pairs)
    k1_list_right: list[int] = [0]*len(good_pairs)
    k1_list_left: list[int] = [0]*len(good_pairs)

    for i in range(len(good_pairs)):
        k0_list_right[i]= good_pairs[i][0] ^ good_input_right[0][0][0]
        k0_list_left[i]= good_pairs[i][0] ^ good_input_left[0][0][0]
        k1_list_right[i]= good_pairs[i][1] ^ good_input_right[0][1][0]
        k1_list_left[i]= good_pairs[i][1] ^ good_input_left[0][1][0]

    print('Candidates for k0 left:', k0_list_left)
    print('Candidates for k0 right:', k0_list_right)
    print('Candidates for k1 left:', k1_list_left)
    print('Candidates for k1 right:', k1_list_right)
    message = random.randint(0, 15)
    for i in range(len(k0_list_left)):
        for j in range(len(k0_list_right)):

            cipher = toy_ciphers.encrypt_sub_2r_spn(message)
            challenge = [(message >> 4) & 0xF, message & 0xF]

            challenge[0] = challenge[0] ^ k0_list_left[i]
            challenge[1] = challenge[1] ^ k0_list_right[j]
            challenge = [s_box[x] for x in challenge]
            binary_str = ''.join(f"{x:04b}" for x in challenge)
            permuted_str = ''.join(binary_str[i] for i in p_box)
            challenge = [int(permuted_str[:4], 2), int(permuted_str
                [4:], 2)]

            challenge[0] = challenge[0] ^ k1_list_left[i]
            challenge[1] = challenge[1] ^ k1_list_right[j]
            challenge = [s_box[x] for x in challenge]
            challenge = (challenge[0] << 4) | challenge[1]

            if cipher == challenge:
                print('Message: ', message)
                print('Keys:', '[', k0_list_left[i], ',', k0_list_right
                    [j], ']', ' [', k1_list_left[i], ',', k1_list_right[
                    j], ']')
```

3.3 Extension of the toy example

The number of rounds in a cipher significantly affects the attack strategy and complexity. Methods like differential cryptanalysis become more challenging because, with each additional round, differences propagate further, making them harder to track deterministically. Additionally, observing statistical patterns in differences requires collecting significantly more data to identify statistically significant patterns or correlations, consequently increasing the data complexity of the attack.

We can implement a further extension of the *encrypt_sub_2r_spn* cipher such that it performs 4 rounds. The code of the encryption is reported in the box below. The decryption algorithm just applies the steps in the reverse order (the whole code can be found inside the *toy_cipher.py* file)..

```
def encrypt_sub_4r_spn(state):
    state = [(state >> 4) & 0xF, state & 0xF]

    state = [state[i] ^ k_0_8b[i] for i in range(len(state))]
    state = [s_4b[x] for x in state]
    binary_str = ''.join(f"{x:04b}" for x in state)
    permuted_str = ''.join(binary_str[i] for i in p_8b)
    state = [int(permuted_str[:4], 2), int(permuted_str[4:], 2)]

    state = [state[i] ^ k_1_8b[i] for i in range(len(state))]
    state = [s_4b[x] for x in state]
    binary_str = ''.join(f"{x:04b}" for x in state)
    permuted_str = ''.join(binary_str[i] for i in p_8b)
    state = [int(permuted_str[:4], 2), int(permuted_str[4:], 2)]

    state = [state[i] ^ k_2_8b[i] for i in range(len(state))]
    state = [s_4b[x] for x in state]
    binary_str = ''.join(f"{x:04b}" for x in state)
    permuted_str = ''.join(binary_str[i] for i in p_8b)
    state = [int(permuted_str[:4], 2), int(permuted_str[4:], 2)]

    state = [state[i] ^ k_3_8b[i] for i in range(len(state))]
    state = [s_4b[x] for x in state]

    return (state[0] << 4) | state[1]
```

4 Differential Fault Analysis

4.1 Alteration of a specific bit in R15

To alter a specific bit in R15 we can create a new function called *fault_encryption* which differently from the original DES encryption takes as input another parameter which is the bit to flip. This new function is very similar to the original one except for an if condition that verifies if the current round is the 15 and checks wether the bit corresponding to the given position is set to 1 or 0 and flips it. The box below shows this part of the code.

```
if round_num == 14:
            new_rpt[fault_bit] = '1' if new_rpt[fault_bit] == '0' else
                '0'
            print(f"Fault injected in the right half at round 15, bit {
                fault_bit}")
```

Alteration of a specific bit in R15

4.2 Two bits affection

To be sure that it is possible to affect two-bits in the expansion permutation we can just add a print statement in both encryptions to show the value of the variable *expanded_result* at round 16. Using the string "Example" as plaintext and bit 0 as fault-bit we can see that the bit in position 1 and position 47 are different:

- Original: 0011010111111110010111010110111101010101000010000;
- Faulted: 0111010111111110010111010110111101010101000010001;

By using the same plaintext and 1 as fault-bit we can see that there is just one-bit affection (in position two)

4.3 Fault attack implementation

The idea of this attack is to focus on a specific SBox in order to get some 6-bit key candidates for that specific part of the key. By looking at the AES implementation, the original 32 bits are divided in to the SBoxes as follows:

- **SBox 1**: takes as input the bits 0 to 6
- **SBox 2**: takes as input the bits 6 to 12

- **SBox 3**: takes as input the bits 12 to 18
- **SBox 4**: takes as input the bits 18 to 24
- **SBox 5**: takes as input the bits 24 to 30
- **SBox 6**: takes as input the bits 30 to 36
- **SBox 7**: takes as input the bits 36 to 42
- **SBox 8**: takes as input the bits 42 to 48

So the code uses "Example" as the reference plaintext and based on which SBox we are considering the number of the fault bit will be different. Then the code works as follows:

- Encrypts the plaintext with both the original and faulted encryption algorithm. This gives as output the ciphertexts "c" and "fault_c";
- For both ciphertexts performs the *reverse_cipher* function. This function inverts the permutation and the substitution box. It gives as output the pair *(input, output)* where input represents the bits before applying the SBox and output the bits after the appliance of the SBox;
- For all possible keys ($2^6 = 64$) applies again the substitution and checks whether the left part and the right part are the same. If so it means that we have a key candidate;
- To check whether the candidates contains the right 6-bits we can temporary print the actual keys used in the encryption.

This algorithm prints some key candidates for each 6-bit chunk. To retrieve the entire last-round key, we can run this script for every bit involved in a specific S-box and save the key candidates for each one. The candidate that appears most often is likely to represent the original part of the key. By applying this method to all the S-boxes, we can determine each 6-bit part of the key. Finally, we can merge all the solutions; the resulting output represents the last-round key.

4.3.1 Usage example

The code is implemented at the bottom of the *DES_cipher.py* file. We will now consider the attack to the first S-box and to the second S-box with "Example" as the reference plaintext.

- If we want to attack the first SBox we should use as fault bit one between 0 and 6, in the function *reverse_cipher* put the output range to [0:4] and the input range to [0:6] and finally in the function *apply_substitution* use the Sbox number 0. The script with the fault bit equal to zero gives us these key candidates: ['001100', '011100', '100010', '101100', '110010', '111100']. The last one is the one actually used in the last round of DES.

- If we want to attack the second SBox we should use as fault bit one between 6 and 12, in the function *reverse_cipher* put the output range to [4:8] and the input range to [6:12] and finally in the function *apply_substitution* use the Sbox number 1. The script with the fault bit equal to seven gives us these key candidates: ['000001', '000011', '000100', '000110', '001001', '001011']. The last one is the one actually used in the last round of DES.