CSI4108: Cryptanalysis Project

Danning Chen, 300234800
Matthieu Ramanitrera, 300264295

November 8, 2024

1 S-Box Design and Analysis

Our randomly generated s-box is represented by the table below

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
5	3	A	1	E	D	2	9	6	F	C	7	B	8	0	4

Table 1: S-box Representation

For the permutation portion of the rounds we used the permutation given in Heys' tutorial [1]

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	16

Table 2: Permutation

Initial analysis

We build the difference distribution table, which was generated using Python (refer to Listing 1 in the Appendix for the code). The difference distribution table is shown in Table 3.

To find the best differential characteristic, we used the highest probabilities found in the differential distribution table while minimizing the number of active s-boxes at each round. We used the following difference pairs of the S-box:

$$S_{12}: \ \Delta X = D \rightarrow \Delta Y = 1 \text{ with probability } \frac{4}{16}$$
$$S_{24}: \ \Delta X = 4 \rightarrow \Delta Y = 8 \text{ with probability } \frac{4}{16}$$
$$S_{31}: \ \Delta X = 1 \rightarrow \Delta Y = 3 \text{ with probability } \frac{4}{16}$$

		Output Difference															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
I	**0**	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
n	**1**	0	0	0	**4**	**2**	0	**2**	0	0	**2**	0	**6**	0	0	0	0
p	**2**	0	0	**2**	0	**2**	0	0	0	**2**	0	**2**	**2**	**4**	0	0	**2**
u	**3**	0	**2**	0	**2**	**2**	0	0	**2**	**2**	**2**	0	0	0	0	0	**4**
t	**4**	0	0	0	**2**	0	0	0	**2**	**4**	0	0	**2**	**2**	**2**	**2**	0
	5	0	0	0	**4**	**2**	0	0	**2**	**4**	0	0	0	0	**2**	**2**	0
D	**6**	0	0	0	0	**2**	0	**2**	**4**	0	0	**2**	**2**	**2**	0	0	**2**
i	**7**	0	**2**	**2**	0	**2**	0	0	**2**	0	0	0	0	**4**	0	0	**4**
f	**8**	0	0	**2**	**2**	0	**4**	**4**	0	0	0	0	0	**2**	**2**	0	0
f	**9**	0	0	0	0	0	**2**	**6**	0	0	**2**	**2**	0	0	**4**	0	0
e	**A**	0	**2**	0	0	**2**	0	0	0	0	**6**	0	0	**2**	0	**4**	0
r	**B**	0	0	**4**	0	0	**2**	0	**2**	0	0	**4**	0	0	**2**	0	**2**
e	**C**	0	0	**2**	0	0	**2**	0	0	**2**	0	**2**	**2**	0	0	**6**	0
n	**D**	0	**4**	0	0	0	**4**	0	0	**2**	0	0	**2**	0	**2**	**2**	0
c	**E**	0	**2**	**2**	0	**2**	**2**	**2**	**2**	0	**2**	**2**	0	0	0	0	0
e	**F**	0	**4**	**2**	**2**	0	0	0	0	0	**2**	**2**	0	0	**2**	0	**2**

Table 3: Difference Distribution Table

The input difference to the cipher is equivalent to the input difference to the first round (denoted ΔU_1) which is:

$$\Delta P = \Delta U_1 = [0000\ 1101\ 0000\ 0000] = (0D00)_{16}$$

The following is how this input difference propagates through the network for this specific differential characteristic where ΔU_i denotes the difference at the input of the i-th round S-boxes and ΔV_i the difference at the output of the i-th round S-boxes:

- Round 1: pick $(\Delta X, \Delta Y) = (D, 1)$ with probability $\frac{4}{16}$

$$\Delta U_1 = [0000\ 1101\ 0000\ 0000] = (0D00)_{16}$$
$$\Delta V_1 = [0000\ 0001\ 0000\ 0000] = (0100)_{16}$$

$$Probability = \frac{4}{16}$$

- Round 2: pick $(\Delta X, \Delta Y) = (4, 8)$ with probability $\frac{4}{16}$

$$\Delta U_2 = [0000\ 0000\ 0000\ 0100] = (0004)_{16}$$
$$\Delta V_2 = [0000\ 0000\ 0000\ 1000] = (0008)_{16}$$

$$Probability = \frac{4}{16} \times \frac{4}{16}$$

- Round 3: pick $(\Delta X, \Delta Y) = (1, 3)$ with probability $\frac{4}{16}$

$$\Delta U_3 = [0001\ 0000\ 0000\ 0000] = (1000)_{16}$$
$$\Delta V_3 = [0011\ 0000\ 0000\ 0000] = (3000)_{16}$$

$$Probability = \frac{4}{16} \times \frac{4}{16} \times \frac{4}{16}$$

- Input of Round 4 S-boxes:

$$\Delta U_4 = [0000\ 0000\ 1000\ 1000] = (0088)_{16} \tag{1}$$
$$Probability = \frac{4}{16} \times \frac{4}{16} \times \frac{4}{16} = \frac{1}{64} = 0.015625$$

Figure 1 is a visual representation of the propagation of the input difference through the network.

This differential characteristic has a probability of $\frac{1}{16} \approx 0.0156$ to occur which is relatively high. It allows us to try to guess the last two nibbles $K_{5,9}...K_{5,12}$ and $K_{5,13}...K_{5,16}$ of the fifth subkey used during the encryption process.

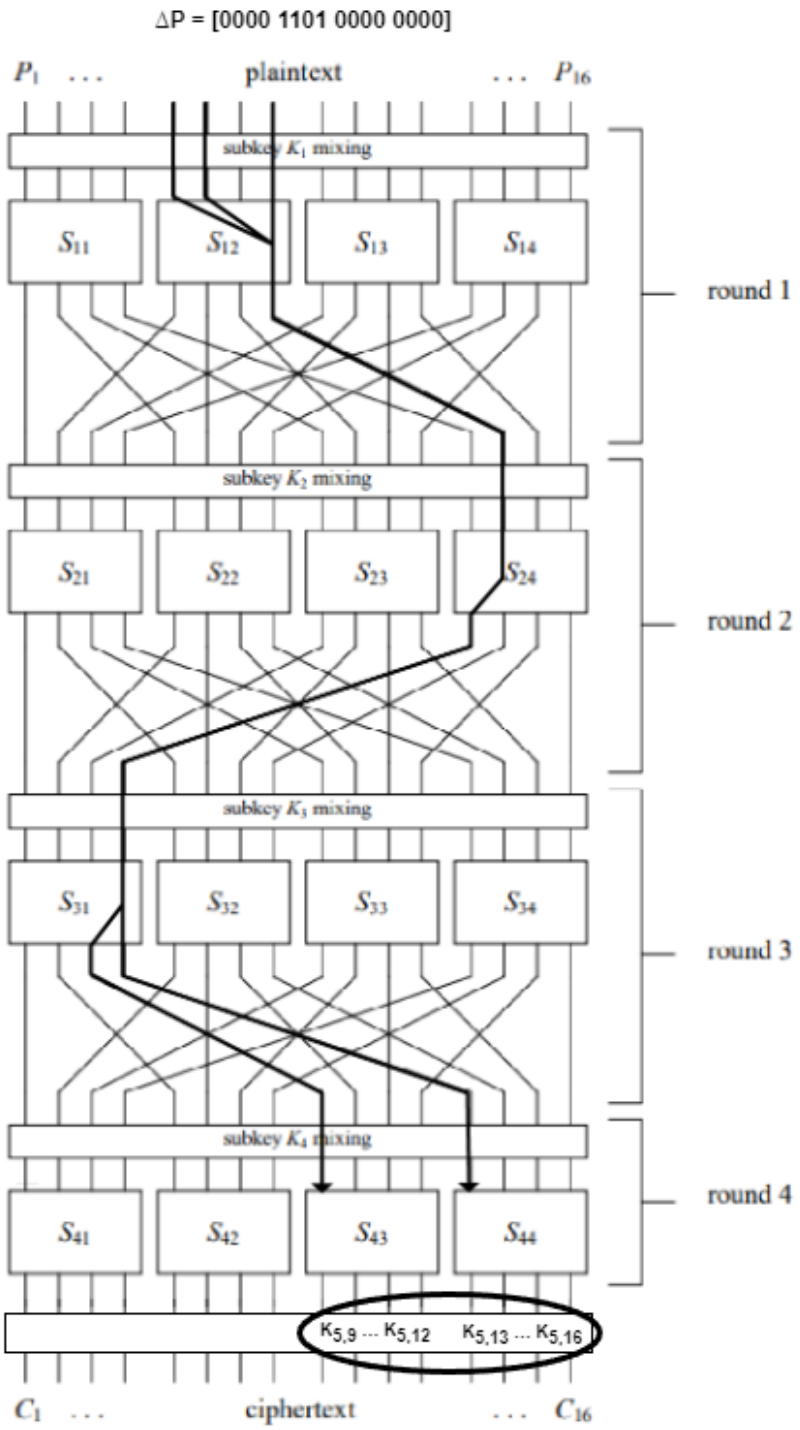

Figure 1: Chosen Differential Characteristic

2 Plaintext Generation and Encryption

Since differential cryptanalysis is a chosen plaintext attack, we can generate plaintext pairs with the input difference chosen in (1). Using Python, we each generated 5000 random 16-bit plaintexts and generated their respective plaintext pair by taking their exclusive-or with the desired input difference $\Delta P = [0000\ 1101\ 0000\ 0000] = (0D00)_{16}$. This gave us 10000 16-bit plaintext strings (5000 strings with their respective pair).

We then each ran the pairs through the Substitution-Permutation Network using a list of round keys not known by the other person and shared the output text files with each other to try to attack the cipher. The details of the attacks are presented in Section 3. The code for the implementation of the SPN can be found in the Appendix as Listing 2 as well as the code for plaintext pairs generation and encryption as Listing 3.

3 Differential attacks

3.1 Person 1: Danning and Person 2: Matthieu

The goal of the attack on this cipher was to try to guess bits of the last subkey used by Danning during the encryption of the plaintext pairs. The differential characteristic presented above affects the inputs of S-boxes S_{43} and S_{44} so I could try to recover bits $[K_{5,9} \cdots K_{5,16}]$ from the fifth subkey.

First, I filtered the ciphertext pairs and only kept the ones for which the difference between them for the first 8 bits $[K_{5,1} \cdots K_{5,8}]$ was zero. That is because pairs having a non-zero difference in those bits couldn't be right pairs given the desired differential characteristic.

I then ran a partial decryption of the rest of the ciphertext string pairs up until the input of the 4-th round S-boxes. I did so for all possible values of bits $[K_{5,9} \cdots K_{5,16}]$ for the last subkey and incremented the count for that combination when the desired $\Delta U_4 = (0088)_{16}$ occurred at the end. Table 4 presents a portion of the data obtained from the subkey counts. The

probability column indicate the estimated probability of occurence of right pairs for the candidate partial subkey as it was done in Heys' tutorial [1]. The probability was calculated by dividing the count by the total number of ciphertext pairs (here 5000).

In this case, I was expecting the probability of occurrence of right pairs to be about $p_D = 0.015625$ and found out the highest probability among all candidates given by the partial subkey value [9,5] was $p_D = 0.0214$ which is noticeably higher. It was revealed by Danning that this combination was indeed the one he used as the last byte of the final subkey so the attack was successful. The Python code for the subkey count and retrieval is given in Listing 4 of the Appendix.

Partial subkey [$K_{5,9}$ ... $K_{5,16}$]	**Probability**	**Partial subkey [$K_{5,9}$ ... $K_{5,16}$]**	**Probability**
8 C	0.0000	9 A	0.0000
8 D	0.0000	9 B	0.0000
8 E	0.0000	9 C	0.0000
8 F	0.0000	9 D	0.0000
9 0	0.0078	9 E	0.0020
9 1	0.0000	9 F	0.0000
9 2	0.0000	A 0	0.0042
9 3	0.0098	A 1	0.0000
9 4	0.0000	A 2	0.0000
9 5	**0.0214**	A 3	0.0042
9 6	0.0090	A 4	0.0000
9 7	0.0052	A 5	0.0076
9 8	0.0050	A 6	0.0042
9 9	0.0022	A 7	0.0022

Table 4: Experimental Results for Differential Attack

3.2 Person 1: Matthieu and Person 2: Danning

The attack is peformed in this process:

- Generate the guessed key

- Loop through all cipher pairs
- Key mix with the ciphertext pair, then you get the guessed ciphertext pair before 5th key mixing
- S-box substitution with the inverse s box, then you get the cipher pair before 4th substitution.
- Compare the difference of the cipher pair with the expected difference, increase the count of this guessed key if they are the same.
- Increase the guessed key and go to step 2 again

The python code used to guess the last byte of the final subkey can be found in Listing 5 of the Appendix. When run on Matthieu's ciphertex pairs text file, the program correcty guessed the last byte of the fifth subkey to be $(1E)_{16}$.

Appendix: Python Code

Generating the Difference Distribution Table

Here is the Python code used to generate the difference distribution table:

```
import pandas as pd

#S_Box
s_box = [0x5, 0x3, 0xA, 0x1, 0xE, 0xD, 0x2, 0x9,
         0x6, 0xF, 0xC, 0x7, 0xB, 0x8, 0x0, 0x4]

ddt = [[0 for _ in range(16)] for _ in range(16)]

# Compute the DDT
for x1 in range(16):
    for x2 in range(16):
        delta_x = x1 ^ x2
        delta_y = s_box[x1] ^ s_box[x2]
        ddt[delta_x][delta_y] += 1

# Convert to DataFrame
ddt_df = pd.DataFrame(ddt)
print(ddt_df)
```

Listing 1: Python code for Difference Distribution Table (ddt.py)

Substitution-Permutation Network

Here is a Python implementation of the Substitution-Permutation Network illustrated in Figure 1:

```
def s_box(input):
    #Applies the s-box to a 4-bit input given as a 4 bit
decimal integer.

    s_box_binary = [
        [0, 1, 0, 1],  # 0x5
        [0, 0, 1, 1],  # 0x3
        [1, 0, 1, 0],  # 0xA
```

```
            [0, 0, 0, 1],  # 0x1
            [1, 1, 1, 0],  # 0xE
            [1, 1, 0, 1],  # 0xD
            [0, 0, 1, 0],  # 0x2
            [1, 0, 0, 1],  # 0x9
            [0, 1, 1, 0],  # 0x6
            [1, 1, 1, 1],  # 0xF
            [1, 1, 0, 0],  # 0xC
            [0, 1, 1, 1],  # 0x7
            [1, 0, 1, 1],  # 0xB
            [1, 0, 0, 0],  # 0x8
            [0, 0, 0, 0],  # 0x0
            [0, 1, 0, 0]   # 0x4
        ]

        return s_box_binary[input]

    def permute(input):
        # Applies the permutation step to a 16-bit binary
    integer given as input.

        permutation_table = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6,
    10, 14, 3, 7, 11, 15]
        combined_input = [item for sublist in input for item
    in sublist]
        permuted = [0] * 16
        for i in range(16):
            permuted[permutation_table[i]] = combined_input[i
    ]

        permuted_as_int = []
        for i in range(0, len(permuted), 4):
            permuted_as_int.append(binary_to_int(permuted[i:i
    +4]))

        return permuted_as_int

    def key_mix(input, round_key):
        # Mixes the round key to the input given as a list of
     4 4-bit decimal integers

        divided_key = [(round_key >> 12) & 0xF,
                       (round_key >> 8) & 0xF,
                       (round_key >> 4) & 0xF,
                       round_key & 0xF]
```

```
        output = []
        for i in range(4):
            output.append(input[i] ^ divided_key[i])

        return output

    def spn(plaintext, round_keys):
        # Implements a 4-round SPN cipher

        current_input = process_plaintext(plaintext)
        for i in range(3):
            mixed = key_mix(current_input, round_keys[i])
            subbed = []
            for nibble in mixed:
                subbed.append(s_box(nibble))
            current_input = permute(subbed)

        final_sub = []
        for nibble in key_mix(current_input, round_keys[3]):
            final_sub.append(binary_to_int(s_box(nibble)))

        ciphertext = process_ciphertext(key_mix(final_sub,
    round_keys[4]))
        return ciphertext


    def binary_to_int(binary):
        # Helper function that converts a 4-bit binary
    integer to hexadecimal

        return int("".join(map(str, binary)), 2)

    def process_plaintext(plaintext):
        # Helper function to convert the 16-bit hexadecimal
    string into a list of 4 decimal integers

        nibbles = [(plaintext >> (4 * i)) & 0xF for i in
    range(4)]

        # The nibbles are in reverse order, so reverse them
        nibbles.reverse()
        return nibbles

    def process_ciphertext(ciphertext):
```

```
        # Helper function to convert a list of 4 decimal
    integers into a hexadecimal string

        hex_value = sum(nibble << (4 * i) for i, nibble in
    enumerate(reversed(ciphertext)))

        # Convert the integer to hexadecimal
        return hex(hex_value)
```

Listing 2: Python code for SPN (spn.py)

Plaintext generation and encryption

Here is the Python code used to generate and encrypt the plaintext pairs:

```
    import random
    from spn import*

    def generate_pairs(input_difference, num_pairs=5000):
        ciphertext_pairs = []

        #Change the keys
        keys = [0x0000, 0x0000, 0x0000, 0x0000, 0x0000]

        for _ in range(num_pairs):
            # Generate a random plaintext
            plaintext1 = random.randint(0, 0xFFFF)
            # Calculate the second plaintext using the input
    difference
            plaintext2 = plaintext1 ^ input_difference

            # Encrypt both plaintexts
            ciphertext1 = spn(plaintext1, keys)
            ciphertext2 = spn(plaintext2, keys)

            # Store the plaintext pair and their
    corresponding ciphertexts
            ciphertext_pairs.append((ciphertext1, ciphertext2
    ))

        return ciphertext_pairs
```

```
    def main():
        pairs = generate_pairs(0x0d00)

        # Output as a text file
        with open("ciphertext_pairs.txt", "w") as output_file
    :
            for pair in pairs:
                output_file.write(f"{pair[0]}, {pair[1]}\n")

    if __name__ == "__main__":
        main()
```

Listing 3: Python code for plaintext pairs generation and encryption (generate.py)

Differential attack

Here is the Python code used for the subkey count and retrieval of Danning's partial subkey:

```
    import pandas as pd
    from spn import *
    def first_two_nibbles_equal(c1, c2):
        # Checks if two first nibbles (8 bits) of two strings
    are equal
        first_two_nibbles_1 = (c1 >> 8)
        first_two_nibbles_2 = (c2 >> 8)

        # Check if their values are equal
        return first_two_nibbles_1 == first_two_nibbles_2

    def reverse_sbox(input):
        # Reverse the S-box substitution
        rev_s_box = [14, 3, 6, 1, 15, 0, 8, 11, 13, 7, 2, 12,
    10, 5, 4, 9]
        nibbles = [(input >> (4 * i)) & 0xF for i in range(4)
    ]
        nibbles.reverse()
        output = 0
        for nibble in nibbles:
            output = (output << 4) | rev_s_box[nibble]
```

```
    return output

def main():
    # Opening the ciphertext pairs file
    with open("ciphertext_pairs.txt") as f:
        pairs = []
        for line in f:
            values = line.strip().split(',')
            if len(values) == 2:
                pair = (int(values[0].strip(), 16), int(
values[1].strip(), 16))
                pairs.append(pair)

    # Desired ciphertext difference in hex
    difference = 0x0088

    # Find pairs that have 0 difference in the first 8
bits
    possible_pairs = []
    for c1, c2 in pairs:
        if first_two_nibbles_equal(c1, c2):
            possible_pairs.append((c1, c2))

    key_count = [0] * 256
    for i in range(256):
        for c1, c2 in possible_pairs:
            # Reverse key mixing
            reversed_c1 = c1 ^ i
            reversed_c2 = c2 ^ i
            # Check difference at the entrance of the
fourth round
            if reverse_sbox(reversed_c1) ^ reverse_sbox(
reversed_c2) == difference:
                key_count[i] += 1   # increment the count
 for that value

    probabilities = [count / 5000 for count in key_count]
    data = {
        'Partial subkey (hex)': [hex(i) for i in range
(256)],
        'Probability': probabilities
    }
    df = pd.DataFrame(data)
```

```
        # Options to display the DataFrame
        pd.set_option('display.max_rows', None)
        pd.set_option('display.max_columns', None)
        pd.set_option('display.float_format', '{:.4f}'.format
    )
        # Display the DataFrame
        print(df.to_string(index=False))

        # The predicted value is the one with the highest
    count
        predicted_byte = hex(key_count.index(max(key_count)))
        print(f"Predicted byte for fifth round key: {
    predicted_byte}")

    if __name__ == "__main__":
        main()
```

Listing 4: Python code for subkey count and retrieval by Matthieu

Here is Danning's Python code for the partial subkey retrieval:

```
    from collections import Counter

    TARGET_OUTPUT_DIFF = 0x0088

    def read_ciphertext_pairs(filename):

        ciphertext_pairs = []
        with open(filename, 'r') as file:
            for line in file:

                parts = line.strip().split(',')

                cipher_1 = f"{int(parts[0].strip(), 16):04x}"
                cipher_2 = f"{int(parts[1].strip(), 16):04x}"

                cipher_1_digits = [int(digit, 16) for digit
    in cipher_1]
                cipher_2_digits = [int(digit, 16) for digit
    in cipher_2]


                ciphertext_pairs.append((cipher_1_digits,
    cipher_2_digits ))
```

```
        return ciphertext_pairs

    def inverse_s_box(output):
        inverse_s_box_binary = [
        [1, 1, 1, 0],  # 0x0
        [0, 0, 1, 1],  # 0x1
        [0, 1, 1, 0],  # 0x2
        [0, 0, 0, 1],  # 0x3
        [1, 1, 1, 1],  # 0x4
        [0, 0, 0, 0],  # 0x5
        [1, 0, 0, 0],  # 0x6
        [1, 0, 1, 1],  # 0x7
        [1, 1, 0, 1],  # 0x8
        [0, 1, 1, 1],  # 0x9
        [0, 0, 1, 0],  # 0xA
        [1, 1, 0, 0],  # 0xB
        [1, 0, 1, 0],  # 0xC
        [0, 1, 0, 1],  # 0xD
        [0, 1, 0, 0],  # 0xE
        [1, 0, 0, 1]   # 0xF
    ]
        return inverse_s_box_binary[output]

    def process_plaintext(plaintext):

        nibbles = [(plaintext >> (4 * i)) & 0xF for i in
    range(4)]

        nibbles.reverse()
        return nibbles

    def binary_to_int(binary):
        return int("".join(map(str, binary)), 2)

    def partial_decrypt(input, subkey_guess):

        #key mixing with guessed subkey
        mixed = []
        for i in range(4):
            mixed.append(input[i] ^ subkey_guess[i])


        #undo the s_box substitution in round 4
        cipher_R4 = []
```

```
        for i in range(4):
            cipher_R4.append(binary_to_int(inverse_s_box(
    mixed[i])))

        return process_ciphertext(cipher_R4)

def process_ciphertext(ciphertext):
    hex_value = sum(nibble << (4 * i) for i, nibble in
    enumerate(reversed(ciphertext)))
    return hex(hex_value)

def generate_subkey_hex_digits(byte_value):

    assert 0 <= byte_value <= 0xFF, "Subkey byte must be
    between 0 and 255."
    high_nibble = (byte_value >> 4) & 0xF
    low_nibble = byte_value & 0xF
    subkey_hex_digits = [0, 0, high_nibble, low_nibble]

    return subkey_hex_digits

    def differential_attack(ciphertext_pairs, num_trials=1):

        #counter to track the frequency of each subkey guess
        subkey_guess_counts = Counter()

        for _ in range(num_trials):

            #loop through each ciphertext pair
            for cipher_1, cipher_2 in ciphertext_pairs:
                #possible values of the last byte of the 5th
    subkey
                for byte_value in range(0x00, 0x100):

                    #generate the subkey
                    subkey_guess_bits =
    generate_subkey_hex_digits(byte_value)

                    #partially decrypt cipher_1 and cipher_2
    using the subkey guess
                    partial_decrypt_cipher_1 =
    partial_decrypt(cipher_1, subkey_guess_bits)
                    partial_decrypt_cipher_2 =
    partial_decrypt(cipher_2, subkey_guess_bits)
```

```
                #check if the partial decryption result
matches the target output difference
                if (int(partial_decrypt_cipher_1, 16) ^
int(partial_decrypt_cipher_2, 16)) == TARGET_OUTPUT_DIFF:
                    subkey_guess_counts[byte_value] += 1


    fifth_subkey = subkey_guess_counts.most_common(1)
[0][0]
    print(f"Last byte of the 5th subkey: {fifth_subkey
:02X}")
    return


 ciphertext_pairs = read_ciphertext_pairs("
ciphertext_pairs.txt")
 fifth_subkey  = differential_attack(ciphertext_pairs)
```

Listing 5: Python code subkey retrieval by Danning

References

[1] Howard M. Heys, A Tutorial on Linear and Differential Cryptanalysis