Name: Hsuan-Chih Hsu

1) Please consider the following AES inputs and **compute the output after the first SubBytes operation.** Only use a pocket calculator and the table given in Figure 1. This is intended to make sure you fully understood the concept. (10 pts)

Plaintext = 00 00 00 00 00 00 C1 A5 51 F1 ED C0 FF EE B4 BE
Key = 00 00 01 02 03 04 DE CA F0 C0 FF EE 00 00 00 00

(Plaintext)

00	00	51	FF
00	00	F1	EE
00	C1	ED	B4
00	A5	C0	BE

(Key)

00	03	F0	00
00	04	C0	00
01	DE	FF	00
02	CA	EE	00

Firstly, using the Key Addition to compute the start of the round: Plaintext XOR with Key.

00	03	A1	FF
00	04	31	EE
01	1F	12	B4
02	6F	2E	BE

Finally, we can compute the output after the first SubBytes operation by using the given Look Up Table (LUT).
=> Answer

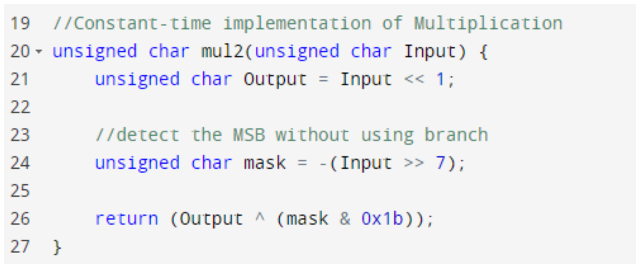
63	**7B**	**32**	**16**
63	**F2**	**C7**	**28**
7C	**C0**	**C9**	**8D**
77	**A8**	**31**	**AE**

2)

a) What is the type of problem here?
Ans: We should not let the Input containing sensitive and secure data be branched.

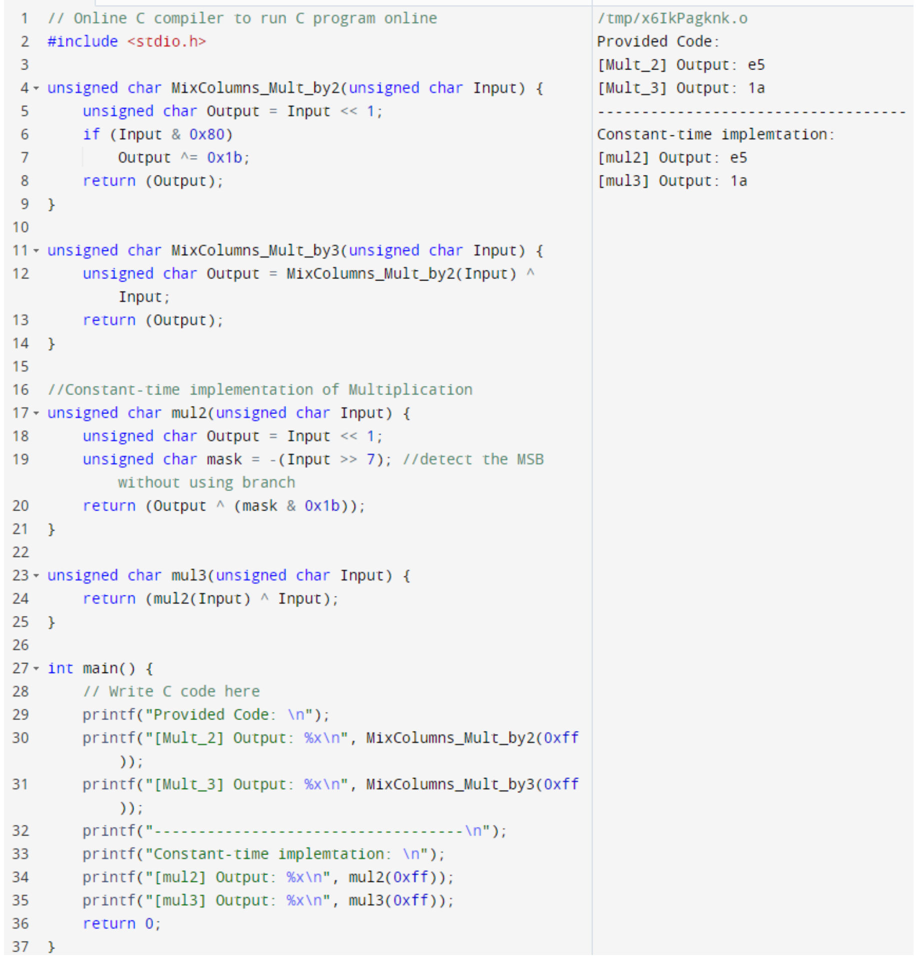
b) Identify the function(s) with undesired behavior.
Ans: “MixColumns_Mult_by2” uses the branch on the sensitive input, which causes the execution time of the function to really depend on the value of the input. This would also make it vulnerable to timing attacks.

c) Suggest a code fragment that solves the problem under idealized assumptions.
Ans:



```c
//Constant-time implementation of Multiplication
unsigned char mul2(unsigned char Input) {
    unsigned char Output = Input << 1;

    //detect the MSB without using branch
    unsigned char mask = -(Input >> 7);

    return (Output ^ (mask & 0x1b));
}
```

Compared to the given code, I have the same result from using my code.



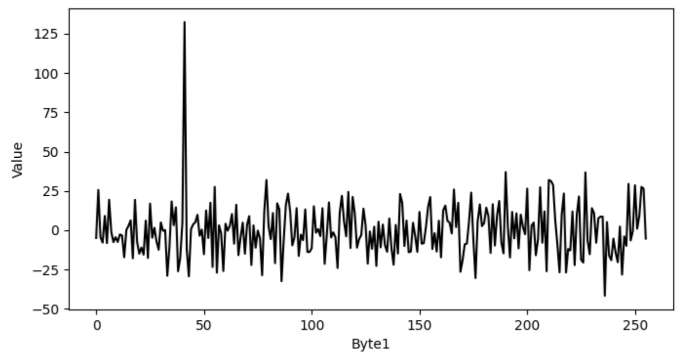
```c
// Online C compiler to run C program online
#include <stdio.h>

unsigned char MixColumns_Mult_by2(unsigned char Input) {
    unsigned char Output = Input << 1;
    if (Input & 0x80)
        Output ^= 0x1b;
    return (Output);
}

unsigned char MixColumns_Mult_by3(unsigned char Input) {
    unsigned char Output = MixColumns_Mult_by2(Input) ^ Input;
    return (Output);
}

//Constant-time implementation of Multiplication
unsigned char mul2(unsigned char Input) {
    unsigned char Output = Input << 1;
    unsigned char mask = -(Input >> 7); //detect the MSB without using branch
    return (Output ^ (mask & 0x1b));
}

unsigned char mul3(unsigned char Input) {
    return (mul2(Input) ^ Input);
}

int main() {
    // Write C code here
    printf("Provided Code: \n");
    printf("[Mult_2] Output: %x\n", MixColumns_Mult_by2(0xff));
    printf("[Mult_3] Output: %x\n", MixColumns_Mult_by3(0xff));
    printf("----------------------------------\n");
    printf("Constant-time implemtation: \n");
    printf("[mul2] Output: %x\n", mul2(0xff));
    printf("[mul3] Output: %x\n", mul3(0xff));
    return 0;
}
```

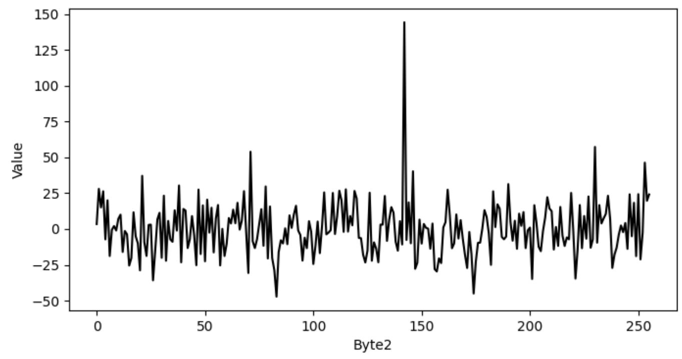
```
/tmp/x6IkPagknk.o
Provided Code:
[Mult_2] Output: e5
[Mult_3] Output: 1a
---------------------------------
Constant-time implemtation:
[mul2] Output: e5
[mul3] Output: 1a
```

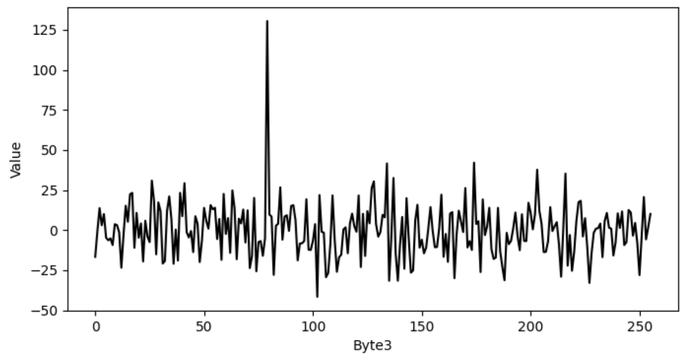
d) Is your solution processor independent? Please provide appropriate reasoning.
Ans: It is not independent. Because when we call the function MixColums_Mult_by3, it will also call the function MixColums_Mult_by2. Let’s say, 03a = 02a + a (a = Input). Overall, the method of timing attack is no longer available in these two functions.

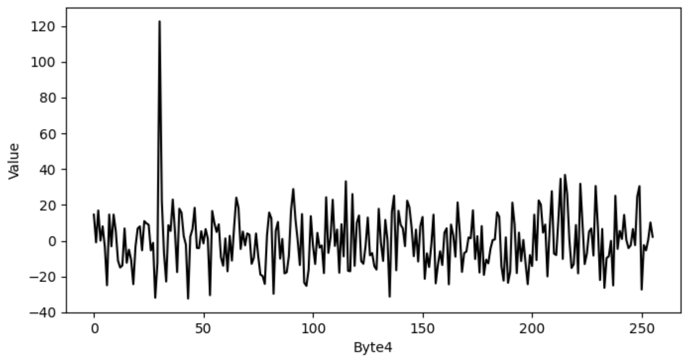
3)

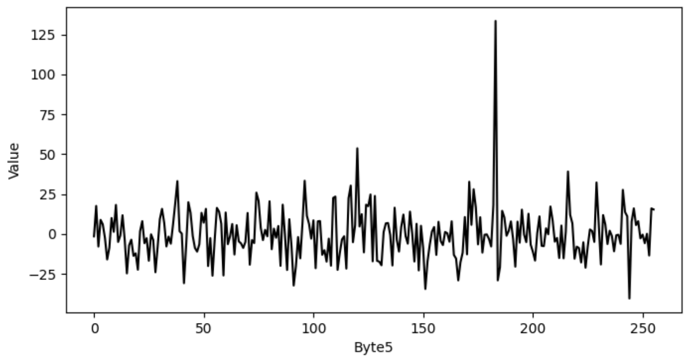
a) Recover the key. Please assume that the Most Significant Bit (MSB) is a good hypothesis. Provide figures to support that you found the correct key.
Key: [41, 142, 79, 30, 183, 104, 193, 19, 15, 246, 189, 223, 236, 119, 47, 176]

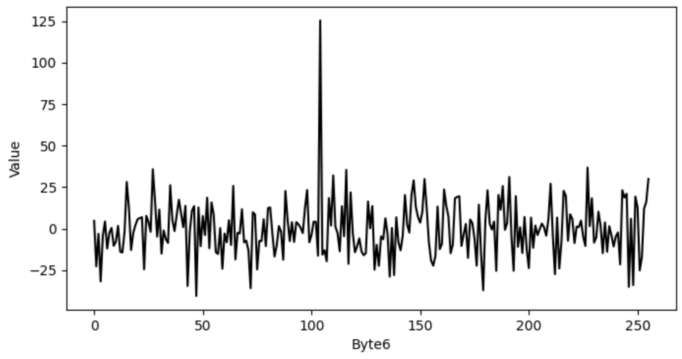


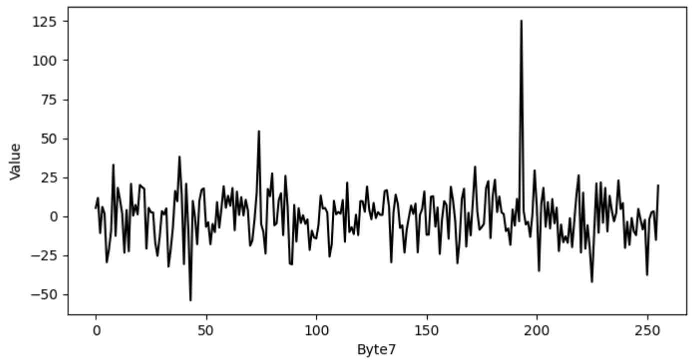


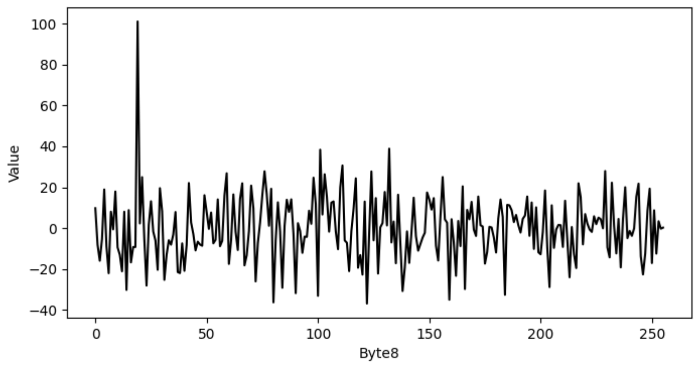


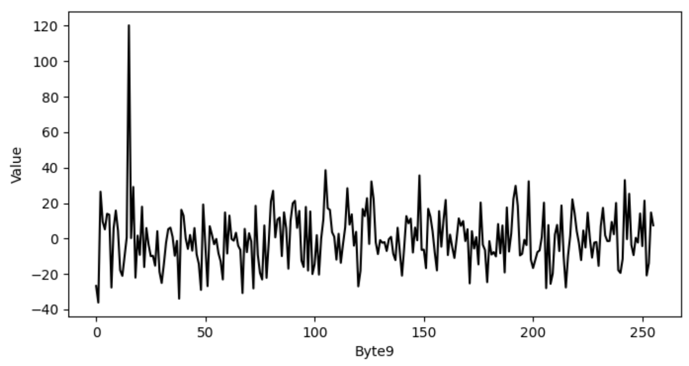


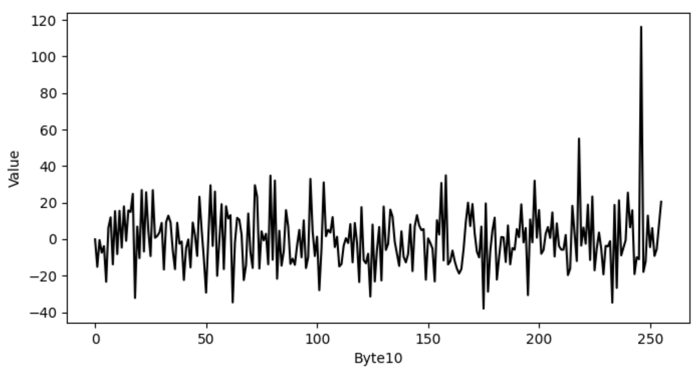


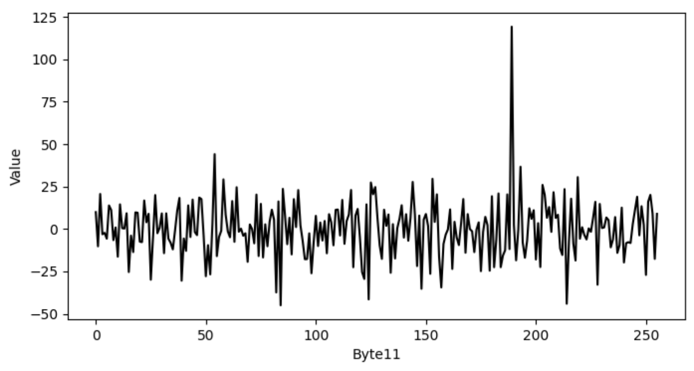


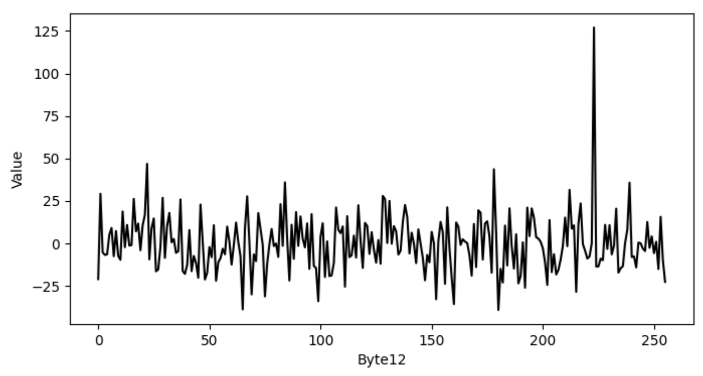


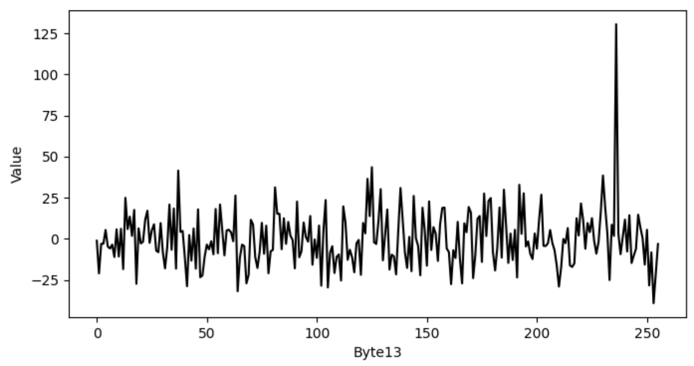


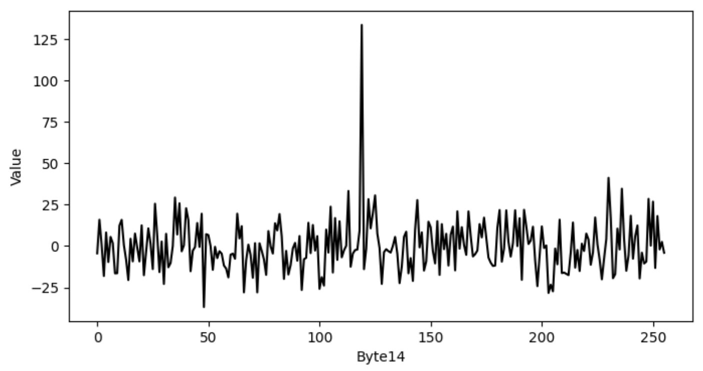

Value
Byte9

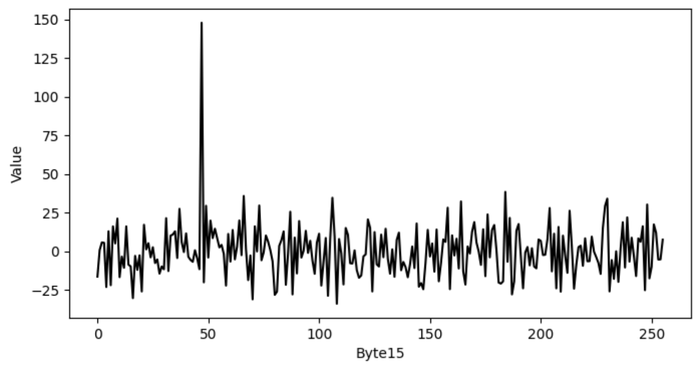

Value
Byte10

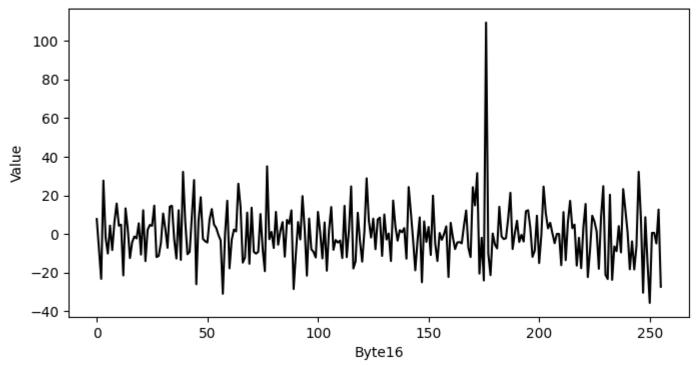

Value
Byte11


Value
Byte12


Value
Byte13

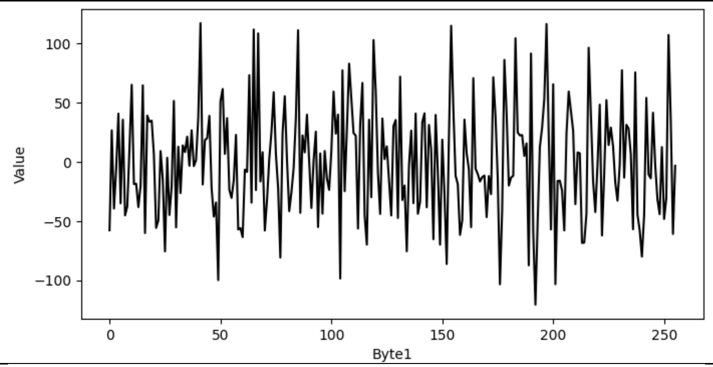

Value
Byte14

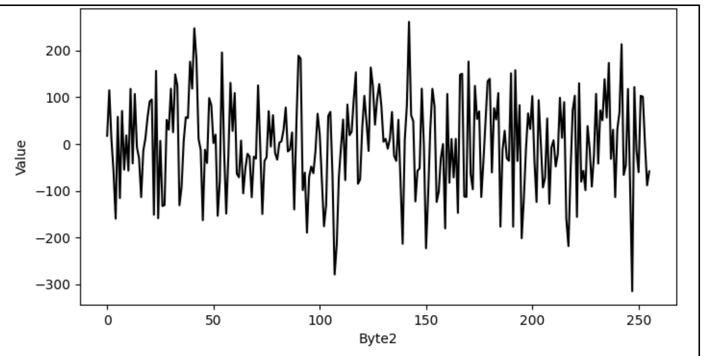

Value
Byte15

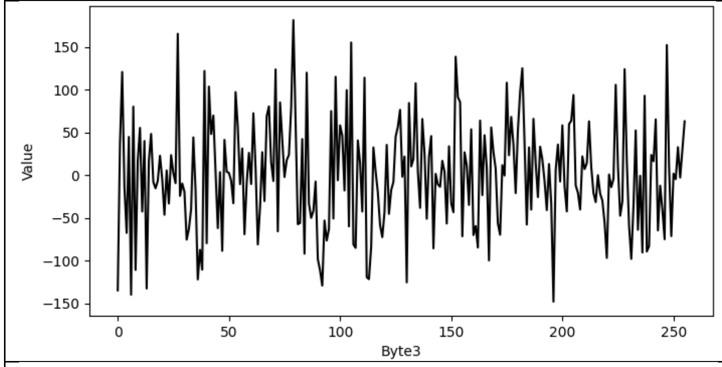

Value
Byte16

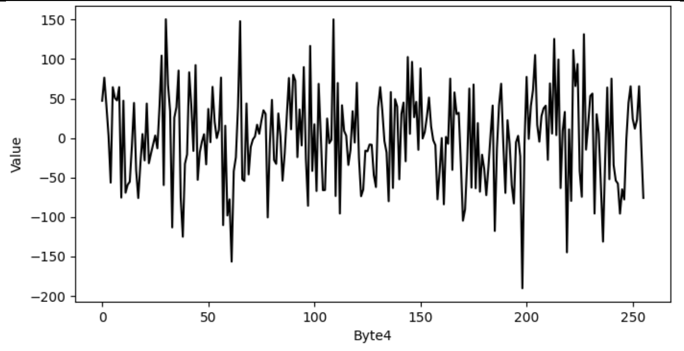
b) Check how many samples you need for each key byte. This can be done, e.g., in 1000 step increments and does not need to be an exact number.
Ans: Check if the key is found every 1000 step. If the key is found, break the loop and move on to the next key.
Key: [41, 142, 79, 30, 183, 104, 193, 19, 15, 246, 189, 223, 236, 119, 47, 176]
Step: [55000, 13000, 34000, 44000, 47000, 31000, 107000, 28000, 103000, 103000, 10000, 27000, 15000, 103000, 17000, 30000]

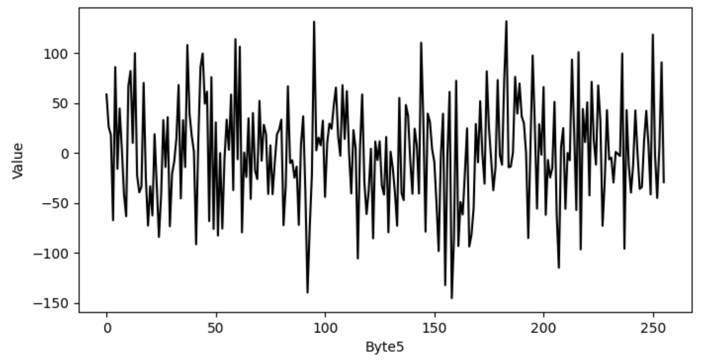
Key Byte	# of steps	Key Byte	# of steps
Byte 1: [41]	55000	Byte 9: [15]	103000
Byte 2: [142]	13000	Byte 10: [246]	103000
Byte 3: [79]	34000	Byte 11: [189]	10000
Byte 4: [30]	44000	Byte 12: [223]	27000
Byte 5: [183]	47000	Byte 13: [236]	15000
Byte 6: [104]	31000	Byte 14: [119]	103000
Byte 7: [193]	107000	Byte 15: [47]	17000
Byte 8: [19]	28000	Byte 16: [176]	30000

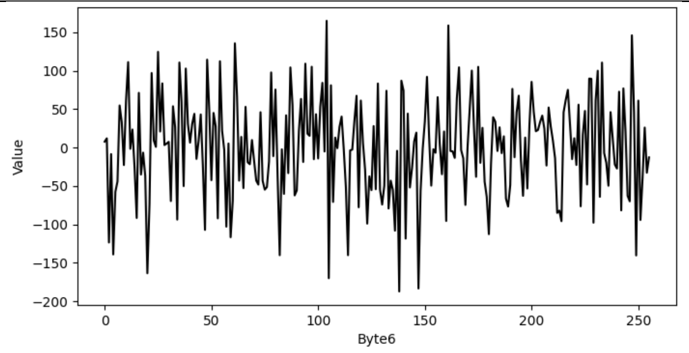


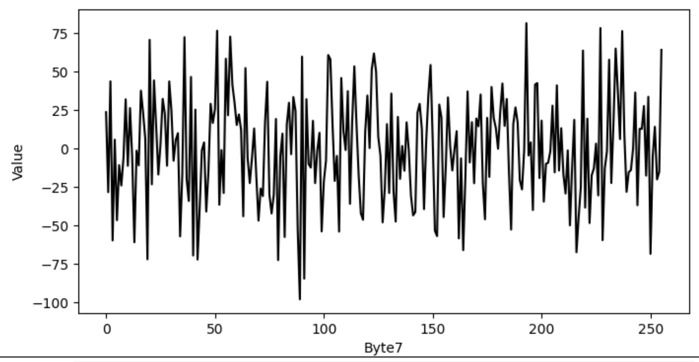
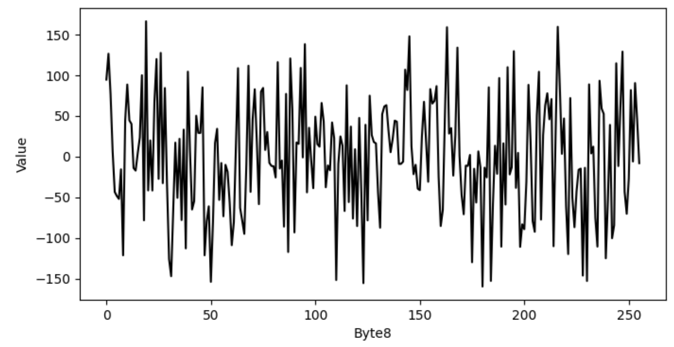
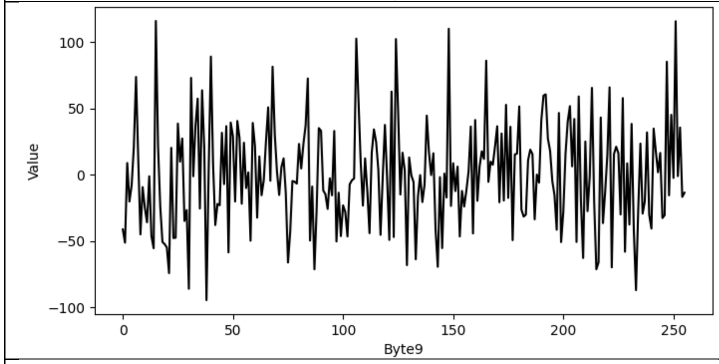
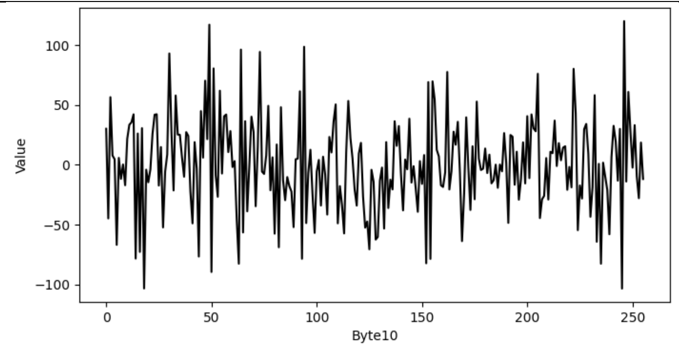
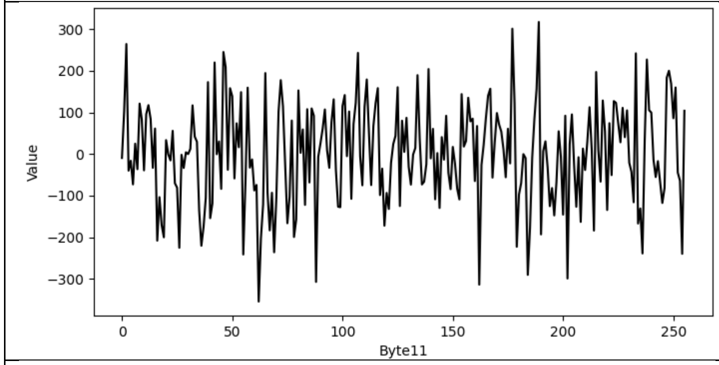
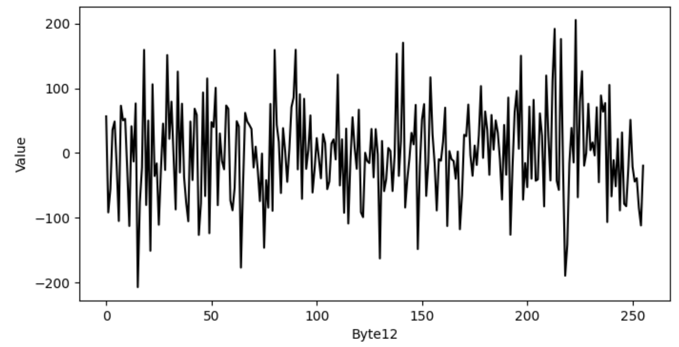
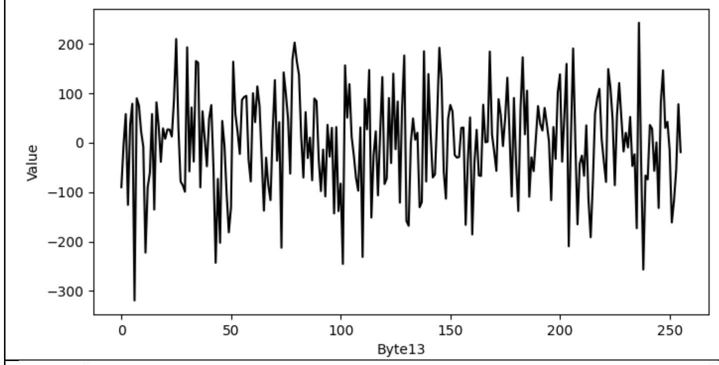
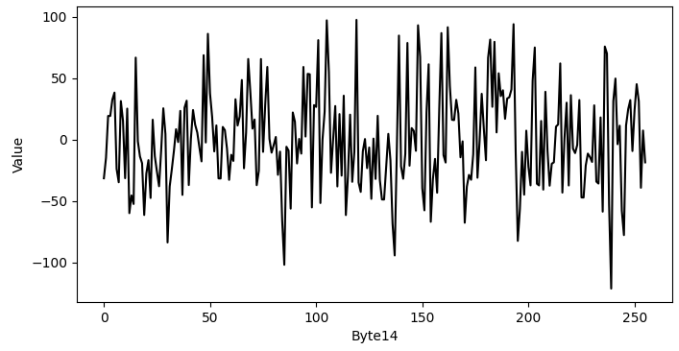
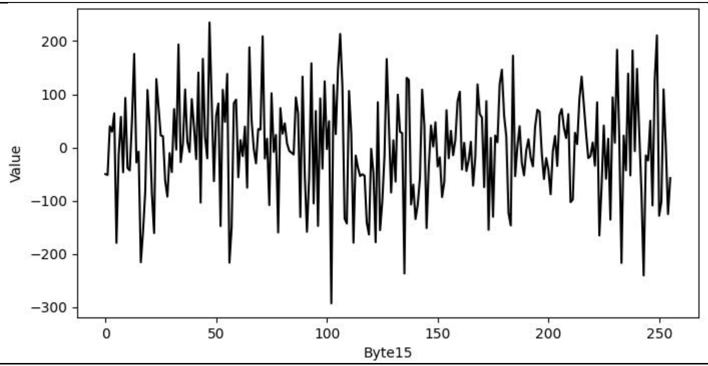
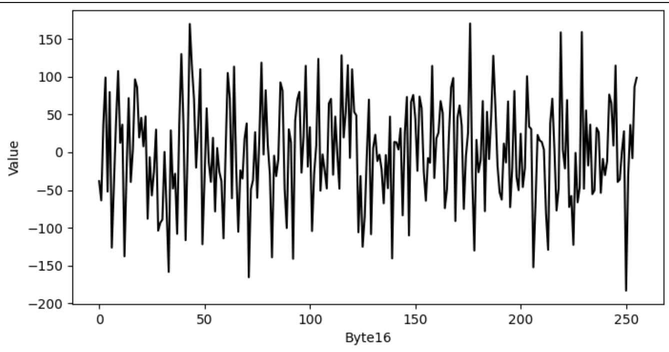


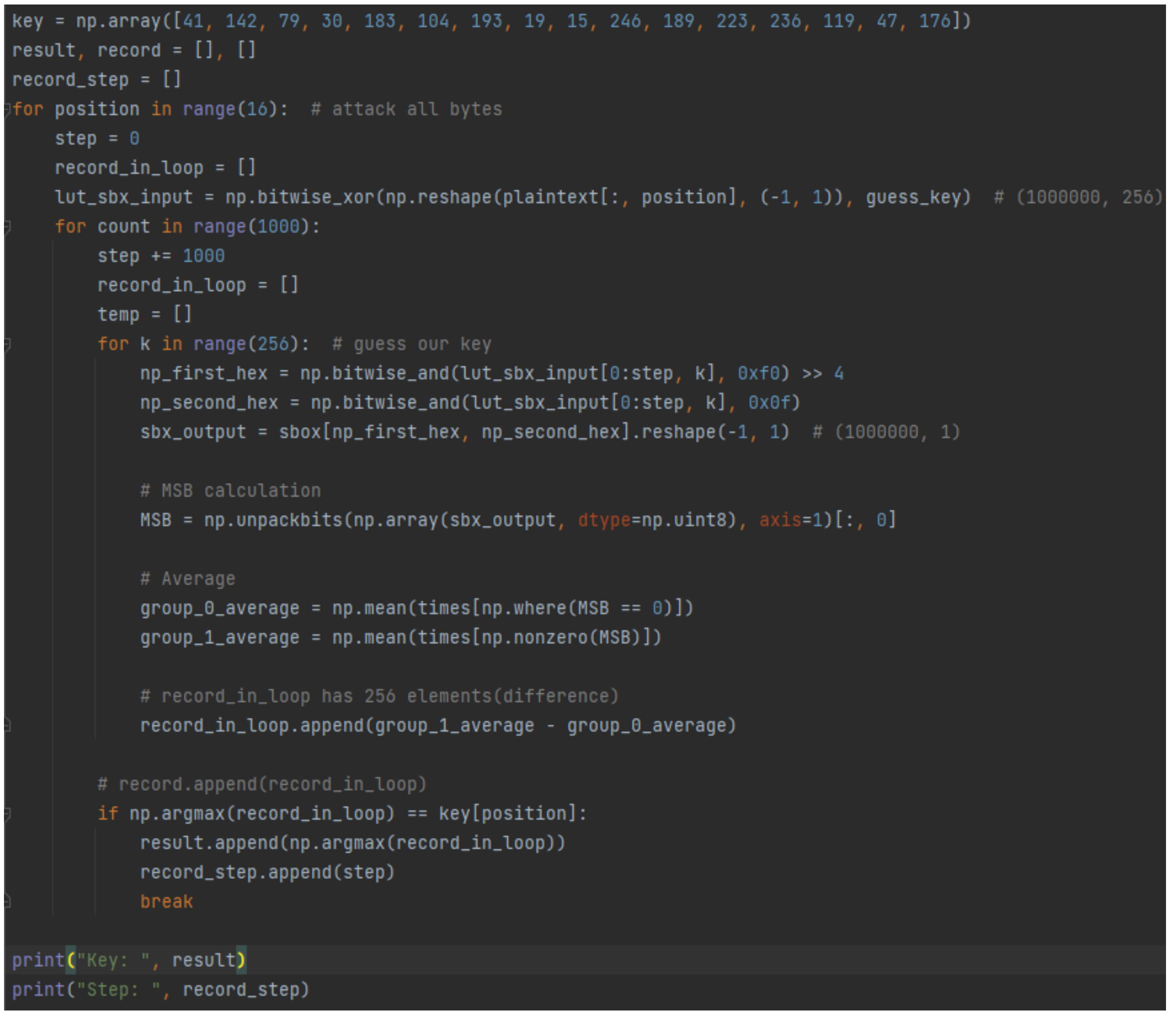




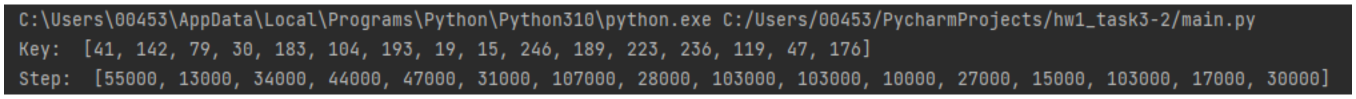





Value
Byte7

Value
Byte8

Value
Byte9

Value
Byte10

Value
Byte11

Value
Byte12

Value
Byte13

Value
Byte14

Value
Byte15

Value
Byte16



```python
key = np.array([41, 142, 79, 30, 183, 104, 193, 19, 15, 246, 189, 223, 236, 119, 47, 176])
result, record = [], []
record_step = []
for position in range(16):  # attack all bytes
    step = 0
    record_in_loop = []
    lut_sbx_input = np.bitwise_xor(np.reshape(plaintext[:, position], (-1, 1)), guess_key)  # (1000000, 256)
    for count in range(1000):
        step += 1000
        record_in_loop = []
        temp = []
        for k in range(256):  # guess our key
            np_first_hex = np.bitwise_and(lut_sbx_input[0:step, k], 0xf0) >> 4
            np_second_hex = np.bitwise_and(lut_sbx_input[0:step, k], 0x0f)
            sbx_output = sbox[np_first_hex, np_second_hex].reshape(-1, 1)  # (1000000, 1)

            # MSB calculation
            MSB = np.unpackbits(np.array(sbx_output, dtype=np.uint8), axis=1)[:, 0]

            # Average
            group_0_average = np.mean(times[np.where(MSB == 0)])
            group_1_average = np.mean(times[np.nonzero(MSB)])

            # record_in_loop has 256 elements(difference)
            record_in_loop.append(group_1_average - group_0_average)

        # record.append(record_in_loop)
        if np.argmax(record_in_loop) == key[position]:
            result.append(np.argmax(record_in_loop))
            record_step.append(step)
            break

print("Key: ", result)
print("Step: ", record_step)
```



```
C:\Users\00453\AppData\Local\Programs\Python\Python310\python.exe C:/Users/00453/PycharmProjects/hw1_task3-2/main.py
Key:  [41, 142, 79, 30, 183, 104, 193, 19, 15, 246, 189, 223, 236, 119, 47, 176]
Step:  [55000, 13000, 34000, 44000, 47000, 31000, 107000, 28000, 103000, 103000, 10000, 27000, 15000, 103000, 17000, 30000]
```

c) Optimize your code such that the overall attack time (without incremental steps) is well below 3 minutes. Please report your execution time in addition to your processor and memory specification. Include a note which operating system you used. If you needed to optimize your code to improve runtime, briefly include a note how you optimized.
Ans:

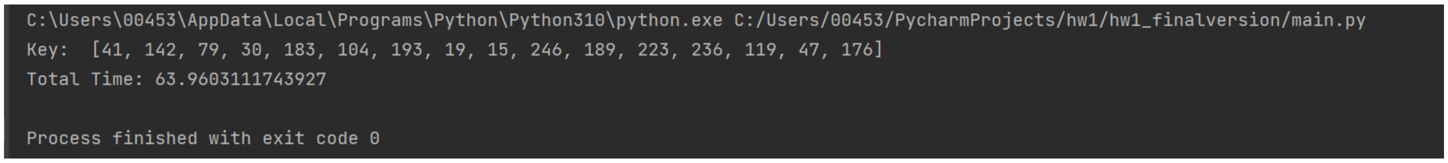
Execution Time:	63.96 seconds
Processor:	11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
Memory:	16.0 GB
Operating System:	Windows 10

Method:

Step 1: I start from constructing the 3 for-loop code structure and not using numpy in the biggest for loop. And it takes me 3-4 minutes to recovery one key byte.

Step 2: I discard using the biggest for loop and start to use numpy for large-data bitwise and arithmetic calculation. In details, 'lut_sbx_input' is a (1000000, 256) 2-D array, containing all the required data for sbox inputs. And this 2-D is only calculated once for each key byte, which optimizing the code. Similarly, 'msb' calculation, allocating and averaging the group are also using the similar numpy methods. Through this, the runtime is optimized, but still takes 3-4 minutes.

Final step: Because the previous step 2 almost reaches the requirement, I introduce the multiprocessing to make my code be processed in parallel. My computer has 4 core, which I can utilize 4 processes at the same time. By this, I can recover the key in 3 minutes, and the final runtime is around 63 seconds.



```
C:\Users\00453\AppData\Local\Programs\Python\Python310\python.exe C:/Users/00453/PycharmProjects/hw1/hw1_finalversion/main.py
Key:  [41, 142, 79, 30, 183, 104, 193, 19, 15, 246, 189, 223, 236, 119, 47, 176]
Total Time: 63.9603111743927

Process finished with exit code 0
```