SC4015 Cyber Physical Security

Group Project Report

Don Lim Zhan Chen U2121972J

Tay Chee Yong U2121657K

Yeoh Wei Yang U2121112A

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

2024/2025

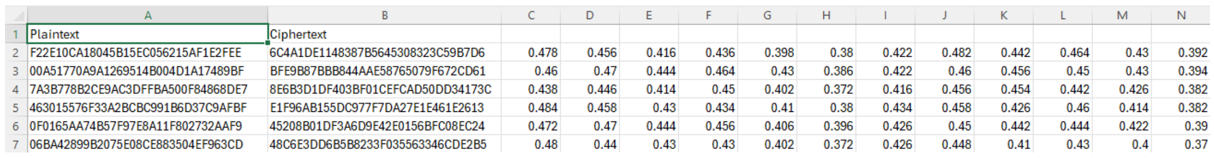
Table of Contents

1. Introduction

Given a set of power traces (waveform.csv), we are required to implement Correlation Power Analysis (CPA) on the given set of traces to extract the first byte of the key. This project is built on the “cpa_implementation.py” Python script that we used in the lab.

1.1 Brief overview of the CSV file

Shown below is the CSV file given to us. The first column is the plaintext, and the next column is its corresponding ciphertext. Accompanying it is its corresponding power trace when the MCU is performing the AES encryption. It contains 2500 data points, ranging from 0 to 1. In total, we have 110 samples.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Plaintext	Ciphertext												
2	F22E10CA18045B15EC056215AF1E2FEE	6C4A1DE1148387B5645308323C59B7D6	0.478	0.456	0.416	0.436	0.398	0.38	0.422	0.482	0.442	0.464	0.43	0.392
3	00A51770A9A1269514B004D1A17489BF	BFE9B87BBB844AAE58765079F672CD61	0.46	0.47	0.444	0.464	0.43	0.386	0.422	0.46	0.456	0.45	0.43	0.394
4	7A3B778B2CE9AC3DFFBA500F84868DE7	8E6B3D1DF403BF01CEFCAD50DD34173C	0.438	0.446	0.414	0.45	0.402	0.372	0.416	0.456	0.454	0.442	0.426	0.382
5	463015576F33A2BCBC991B6D37C9AFBF	E1F96AB155DC977F7DA27E1E461E2613	0.484	0.458	0.43	0.434	0.41	0.38	0.434	0.458	0.426	0.46	0.414	0.382
6	0F0165AA74B57F97E8A11F802732AAF9	45208B01DF3A6D9E42E0156BFC08EC24	0.472	0.47	0.444	0.456	0.406	0.396	0.426	0.45	0.442	0.444	0.422	0.39
7	06BA42899B2075E08CE883504EF963CD	48C6E3DD6B5B8233F035563346CDE2B5	0.48	0.44	0.43	0.43	0.402	0.372	0.426	0.448	0.41	0.43	0.4	0.37

Figure 1. CSV file. An additional row is added to denote the plaintext and ciphertext

1.2 Brief Introduction to AES Algorithm

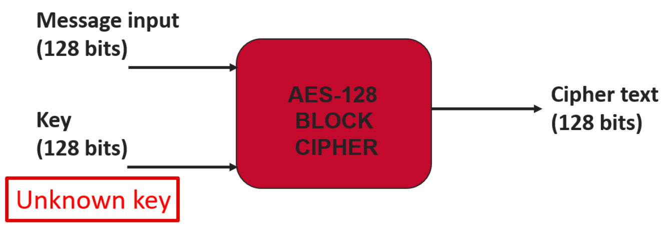


Figure 2. Basic overview of AES

The Advanced Encryption Standard (AES) is a symmetric block cipher that is used to encrypt and protect sensitive information. It takes in 2 inputs: the plaintext message and the secret key. In our lab, we can send plaintext to the MCU, which does the AES encryption, and receive the ciphertext as the output. However, the key is kept as a secret. The objective of this project is to extract this secret key via Correlation Power Analysis.

1.3 Underlying Principle of the Attack

The AES algorithm itself is secure and is computationally infeasible to recover the secret key. However, its implementation on hardware opens up a new attack vector: Power Analysis Side-Channel Attack. This approach relies on leakage models-When the device is at rest, the data lines between the memory and its internal registers are precharged to logic 1. When data transfer occurs, some data lines will be pulled low to logic 0. This change in the state of the data line causes the power consumption of the device to change. The power consumption of the device is, hence, proportional to the number of 0 bits. Therefore, we can use a Hamming Weight model, which measures the number of “1” bits remaining on the data line to determine the power consumption of the device. This model will be used for our attack in Correlation Power Analysis (CPA) to recover the secret key of the MCU.

2. Correlation Power Analysis

Correlation Power Analysis (CPA) is a side-channel attack that analyzes the power consumption of the cryptographic device when it is performing the AES encryption. The power traces obtained will then be used to analyze the correlation between potential secret keys and the power consumption of the device. This additional information obtained allows us to mount the attack and recover the secret key.

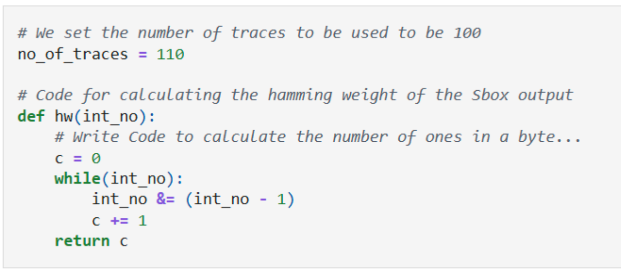
Firstly, the attacker causes the victim to encrypt several known plaintexts. During the encryption process, the power trace of the device, when it is performing the AES encryption, is also recorded. The corresponding ciphertext is also recorded. At this stage, we had obtained the plaintext-ciphertext pair along with the corresponding power trace. For this project, this information is given to us in the file “waveform.csv”.

Next, we look into obtaining a model for the victim’s power consumption. Usually, S-Box will be implemented as a lookup table: it will map the input values to the output values. The procedure is as such:

- The bus lines connecting the registers and memory are precharged to logic 1
- Data is first moved from the register to the memory over the bus (input). As such, certain bits will be changed from logic 1 to logic 0.
- The memory processes the information. During this time, the bus is precharged to logic 1 once again.
- When data is ready, data is then sent from the memory to the register (output of Sbox is sent to register). This also changes certain bits from logic 1 to 0.
- The number of bits that are being switched to 0 is, therefore, proportional to the power consumption of the device.

As such, a Hamming Weight model is used. Here, we define the Hamming Weight as the total number of 1 bits in the value.

$$\text{HammingWeight}(s) = \#\{s_i \in s : s_i = 1\}$$



```python
# We set the number of traces to be used to be 100
no_of_traces = 110

# Code for calculating the hamming weight of the Sbox output
def hw(int_no):
    # Write Code to calculate the number of ones in a byte...
    c = 0
    while(int_no):
        int_no &= (int_no - 1)
        c += 1
    return c
```

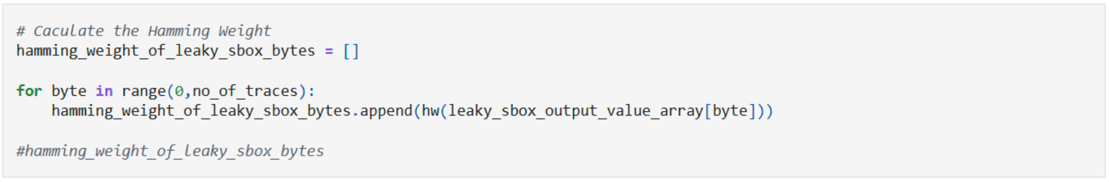
Figure 3. Function to calculate Hamming Weight

Lastly, we built a model trace for every possible key byte (0x00 to 0xFF). The steps are as follows:

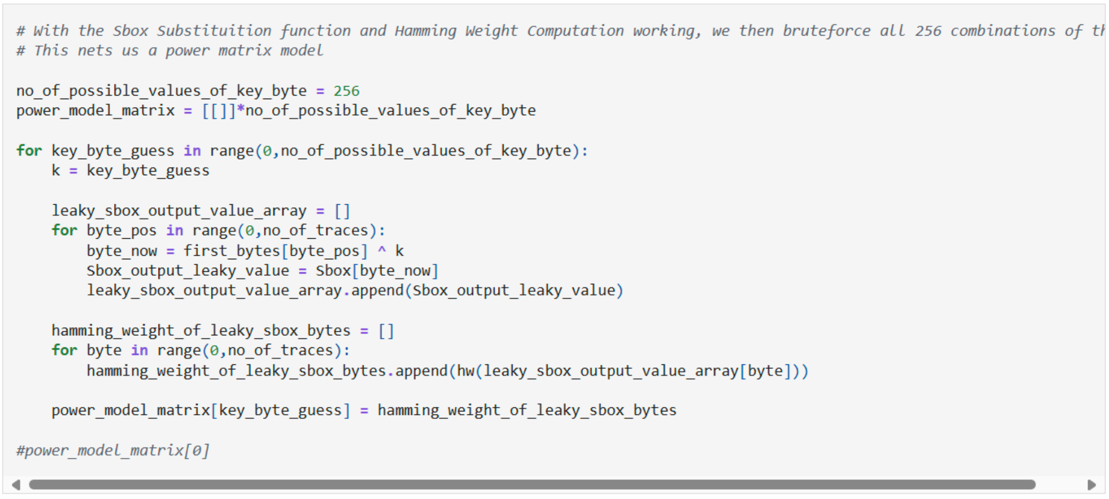
1. Taking the first byte of the plaintext and XOR-ing it with every possible key.
2. Find the Sbox output of the above step

```
for byte_pos in range(0,no_of_traces):
    byte_now = first_bytes[byte_pos] ^ k
    Sbox_output_leaky_value = Sbox[byte_now]
    leaky_sbox_output_value_array.append(Sbox_output_leaky_value)
```

3. Apply the Hamming Weight Model to obtain the Power Matrix Model

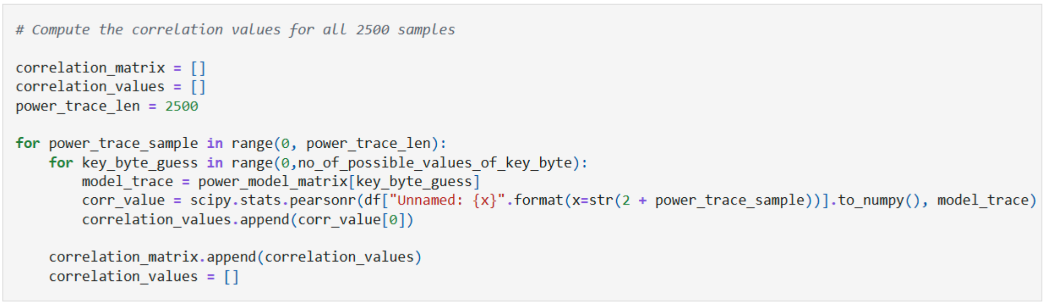


```
# Caculate the Hamming Weight
hamming_weight_of_leaky_sbox_bytes = []

for byte in range(0,no_of_traces):
    hamming_weight_of_leaky_sbox_bytes.append(hw(leaky_sbox_output_value_array[byte]))

#hamming_weight_of_leaky_sbox_bytes
```



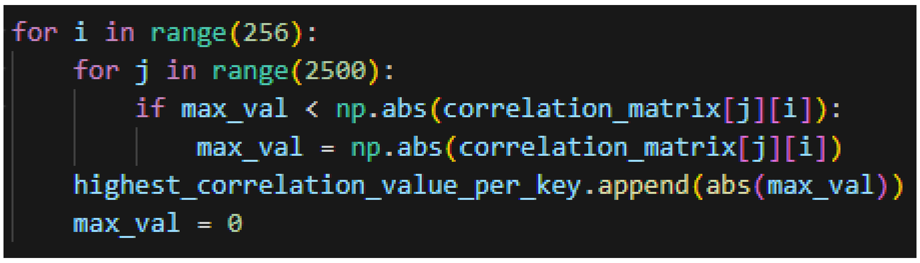
```
# With the Sbox Substituition function and Hamming Weight Computation working, we then bruteforce all 256 combinations of th
# This nets us a power matrix model

no_of_possible_values_of_key_byte = 256
power_model_matrix = [[]]*no_of_possible_values_of_key_byte

for key_byte_guess in range(0,no_of_possible_values_of_key_byte):
    k = key_byte_guess

    leaky_sbox_output_value_array = []
    for byte_pos in range(0,no_of_traces):
        byte_now = first_bytes[byte_pos] ^ k
        Sbox_output_leaky_value = Sbox[byte_now]
        leaky_sbox_output_value_array.append(Sbox_output_leaky_value)

    hamming_weight_of_leaky_sbox_bytes = []
    for byte in range(0,no_of_traces):
        hamming_weight_of_leaky_sbox_bytes.append(hw(leaky_sbox_output_value_array[byte]))

    power_model_matrix[key_byte_guess] = hamming_weight_of_leaky_sbox_bytes

#power_model_matrix[0]
```

4. We then match the model of every possible key byte against the actual power traces to obtain the 256x2500 correlation matrix.



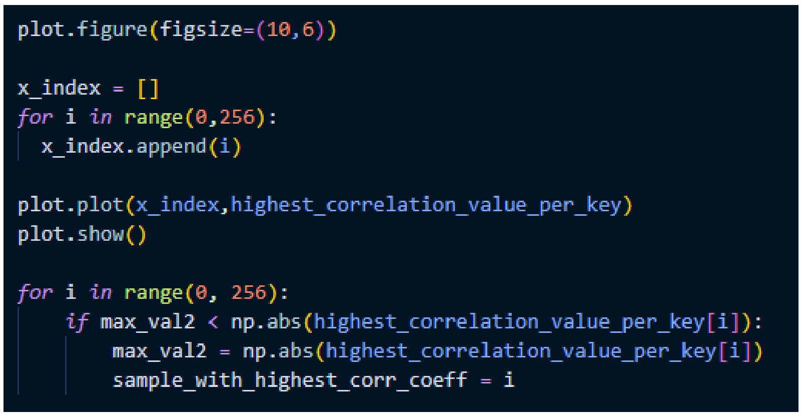
```
# Compute the correlation values for all 2500 samples

correlation_matrix = []
correlation_values = []
power_trace_len = 2500

for power_trace_sample in range(0, power_trace_len):
    for key_byte_guess in range(0,no_of_possible_values_of_key_byte):
        model_trace = power_model_matrix[key_byte_guess]
        corr_value = scipy.stats.pearsonr(df["Unnamed: {x}".format(x=str(2 + power_trace_sample))].to_numpy(), model_trace)
        correlation_values.append(corr_value[0])

    correlation_matrix.append(correlation_values)
    correlation_values = []
```

5. For each possible key, we find the highest correlation coefficient and append it to an array.



```
for i in range(256):
    for j in range(2500):
        if max_val < np.abs(correlation_matrix[j][i]):
            max_val = np.abs(correlation_matrix[j][i])
    highest_correlation_value_per_key.append(abs(max_val))
    max_val = 0
```

6. We then plot the graph of the correlation value per key against the key values. The key with the highest correlation coefficient has the highest probability of being the secret key.



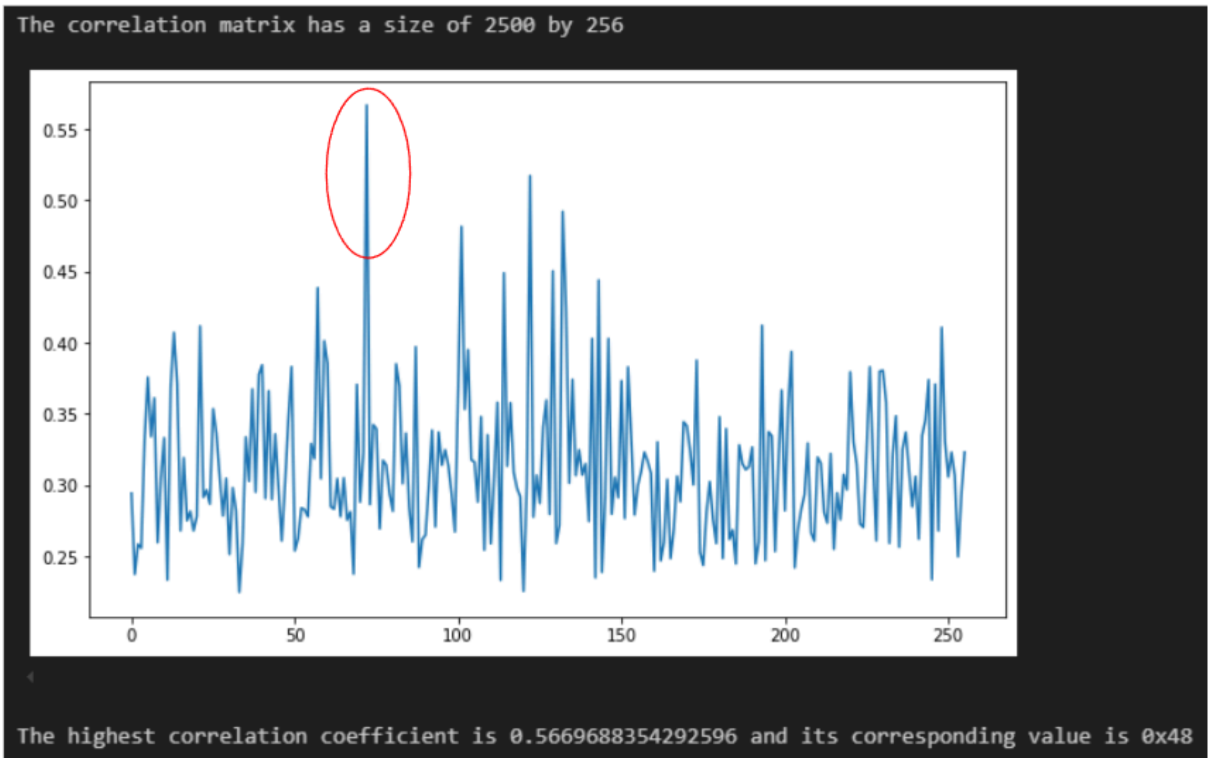
```
plot.figure(figsize=(10,6))

x_index = []
for i in range(0,256):
  x_index.append(i)

plot.plot(x_index,highest_correlation_value_per_key)
plot.show()

for i in range(0, 256):
    if max_val2 < np.abs(highest_correlation_value_per_key[i]):
        max_val2 = np.abs(highest_correlation_value_per_key[i])
        sample_with_highest_corr_coeff = i
```

7. Repeat for the remaining 15 plaintext bytes to recover the full 16-byte AES key.

3. Experimental Results

Plot 1: Value of key byte (x-axis) against correlation value (y-axis)

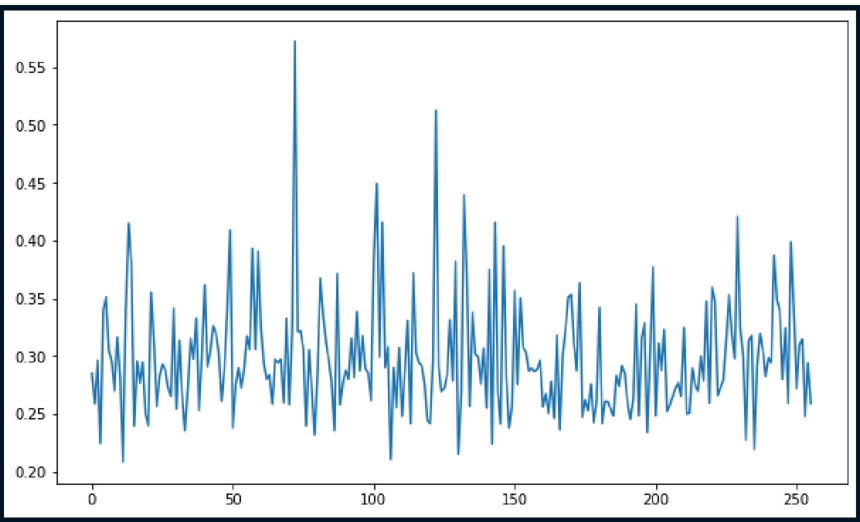
Key of first byte used: 0x48



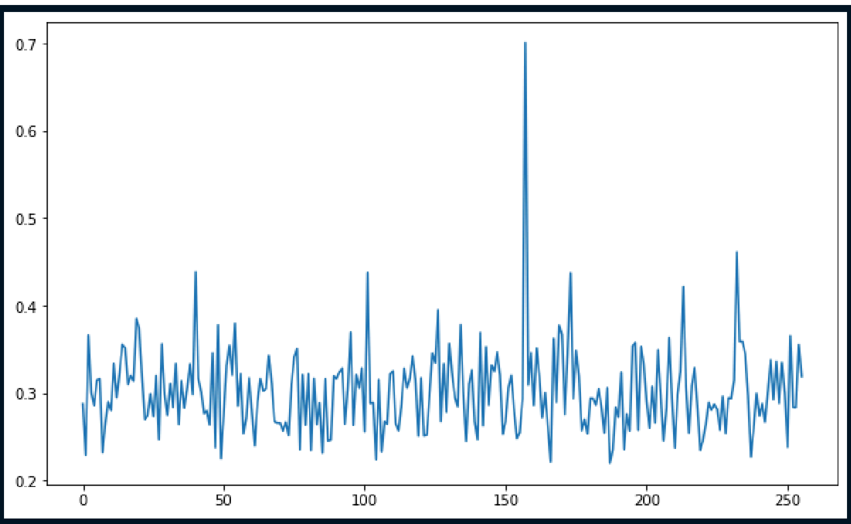
```
The correlation matrix has a size of 2500 by 256
```

```
The highest correlation coefficient is 0.5669688354292596 and its corresponding value is 0x48
```

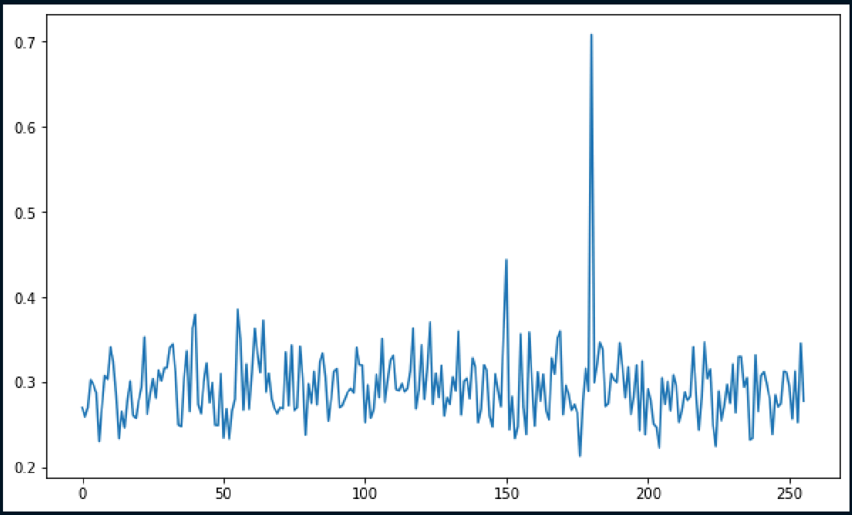
Plots 2-17. (CPA_key_extraction.ipynb) Full key extraction
Key used in AES:0x 48 9d b4 b3 f3 17 29 61 cc 2b cb 4e d2 e2 8e b7



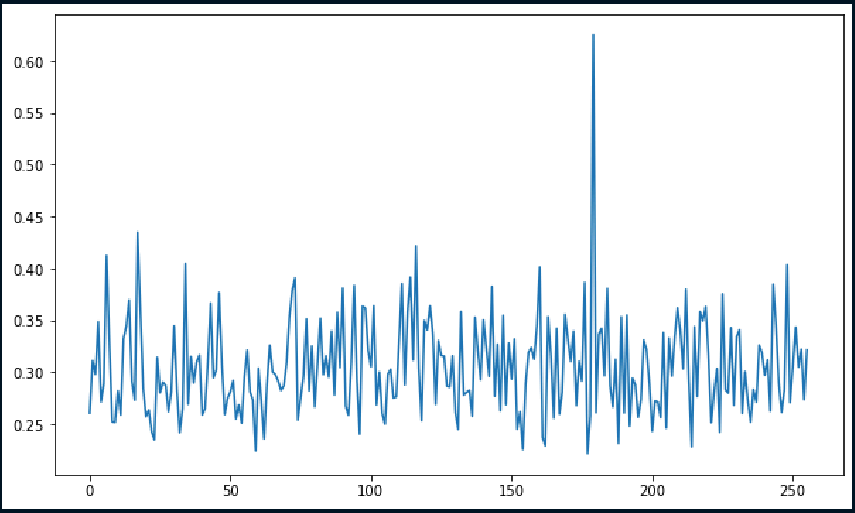
```
The highest correlation coefficient for key byte 0 is 0.5725063314393235 and its corresponding value is 0x48
```



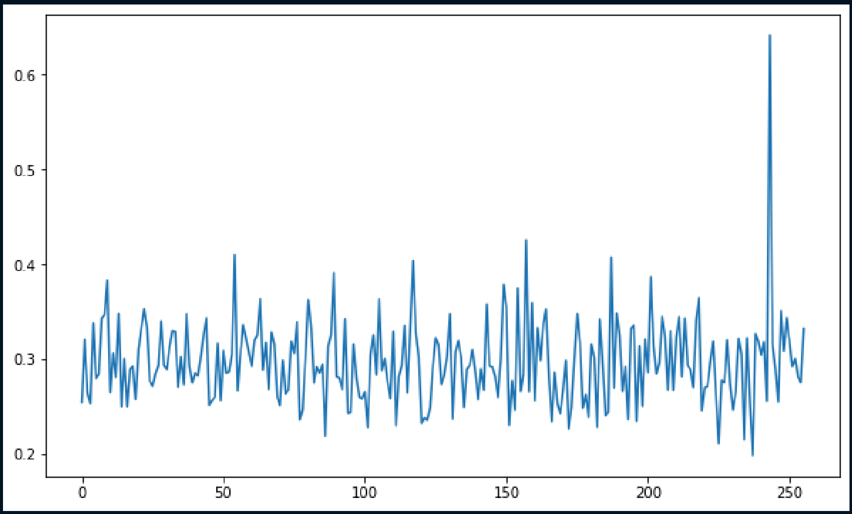
```
The highest correlation coefficient for key byte 1 is 0.7014632953761881 and its corresponding value is 0x9d
```



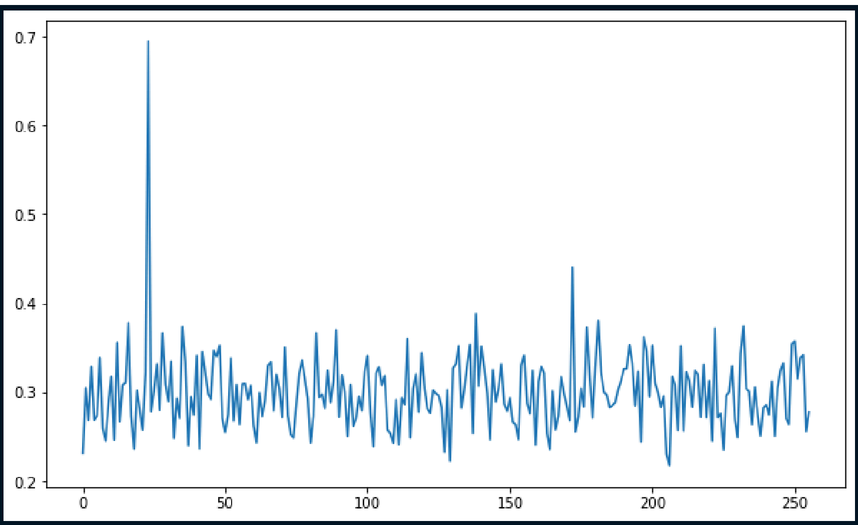
```
The highest correlation coefficient for key byte 2 is 0.7081874276805081 and its corresponding value is 0xb4
```



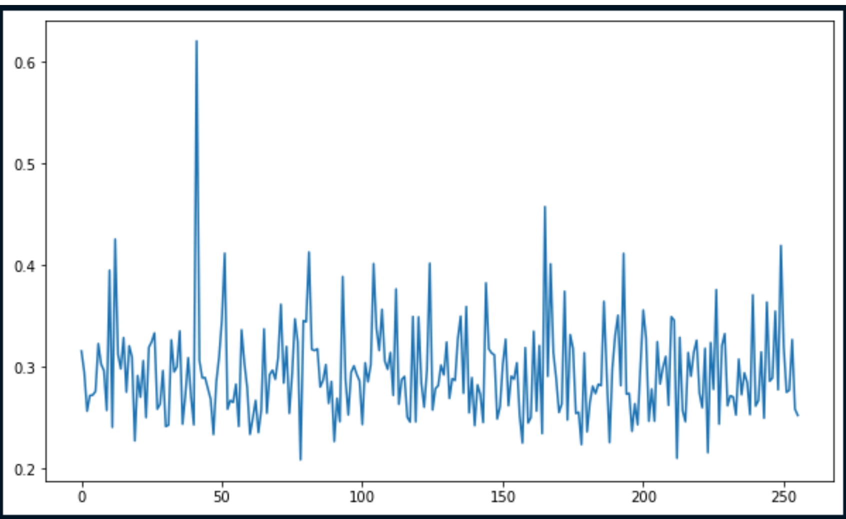
```
The highest correlation coefficient for key byte 3 is 0.6251981824507069 and its corresponding value is 0xb3
```



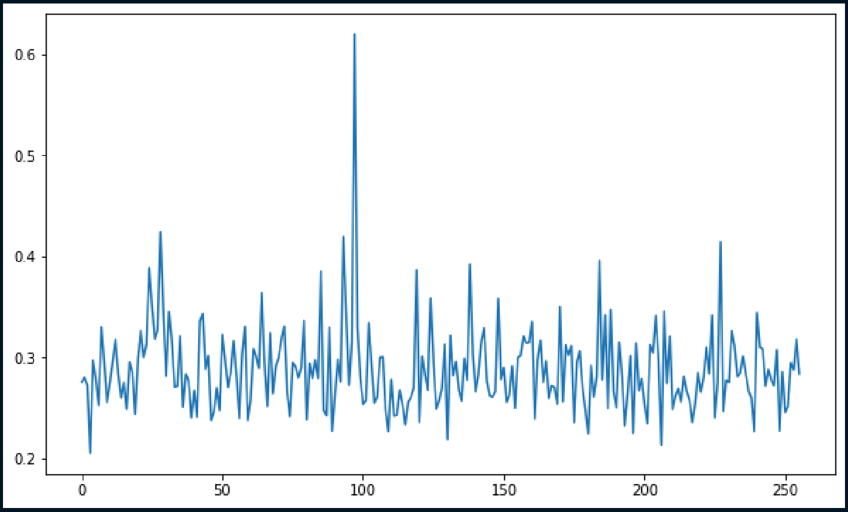
```
The highest correlation coefficient for key byte 4 is 0.6413699822039651 and its corresponding value is 0xf3
```



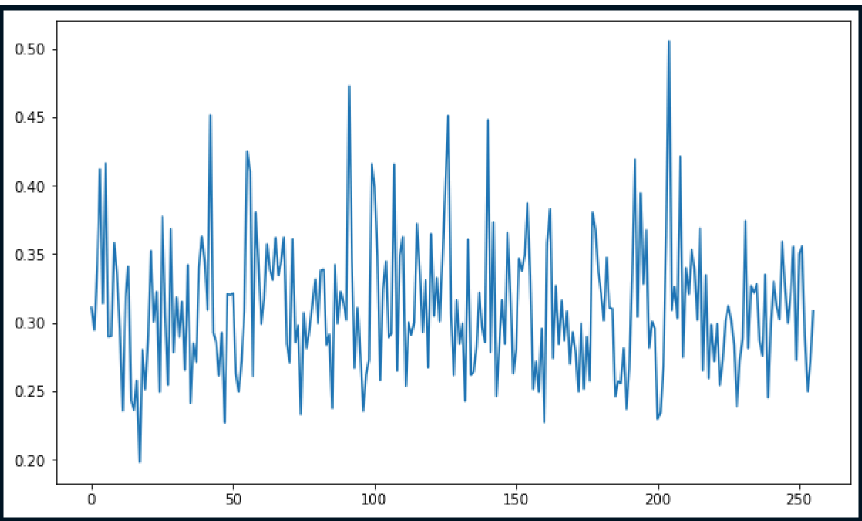
```
The highest correlation coefficient for key byte 5 is 0.694502001889467 and its corresponding value is 0x17
```



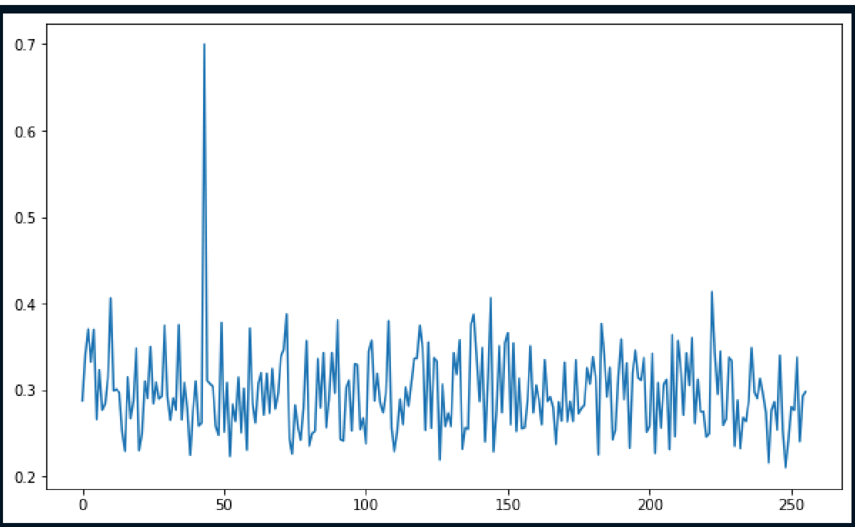
```
The highest correlation coefficient for key byte 6 is 0.6207291305454032 and its corresponding value is 0x29
```



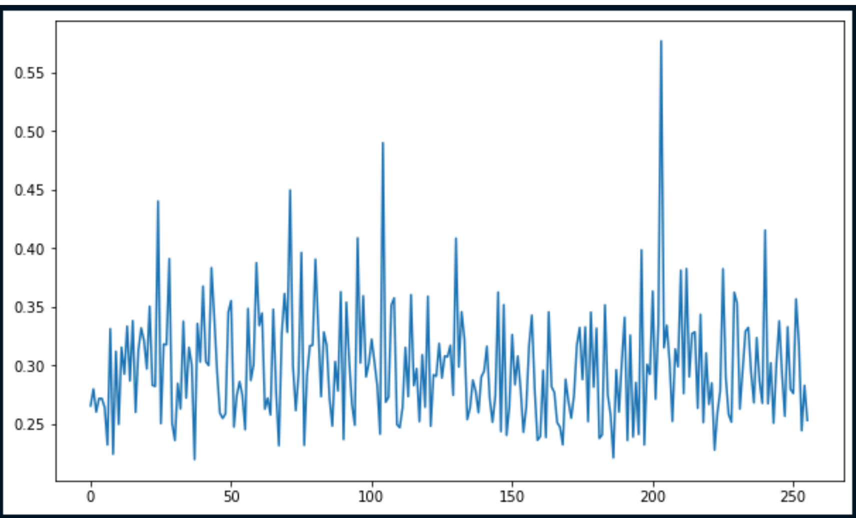
```
The highest correlation coefficient for key byte 7 is 0.6197514051006909 and its corresponding value is 0x61
```



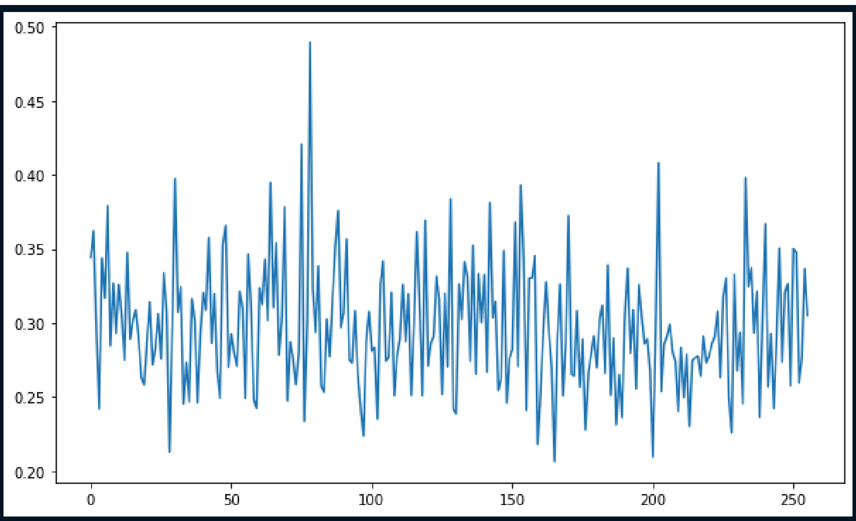
```
The highest correlation coefficient for key byte 8 is 0.5052991941460788 and its corresponding value is 0xcc
```



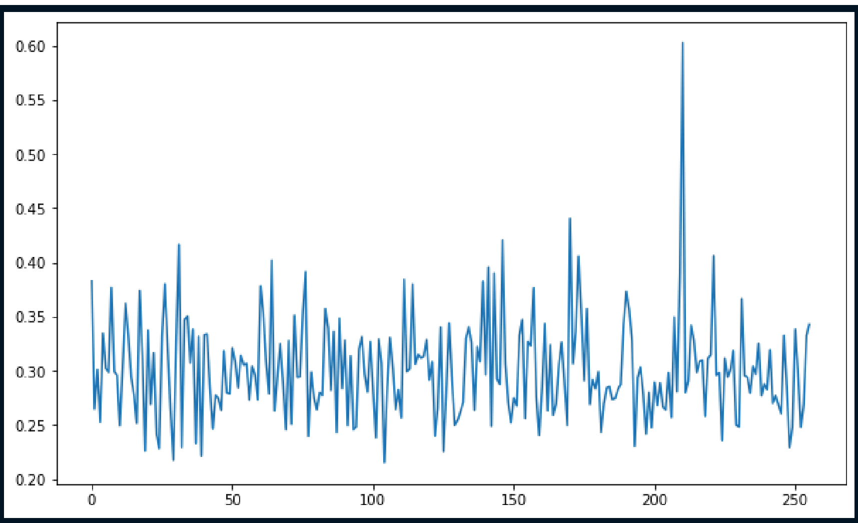
```
The highest correlation coefficient for key byte 9 is 0.6998306497288418 and its corresponding value is 0x2b
```



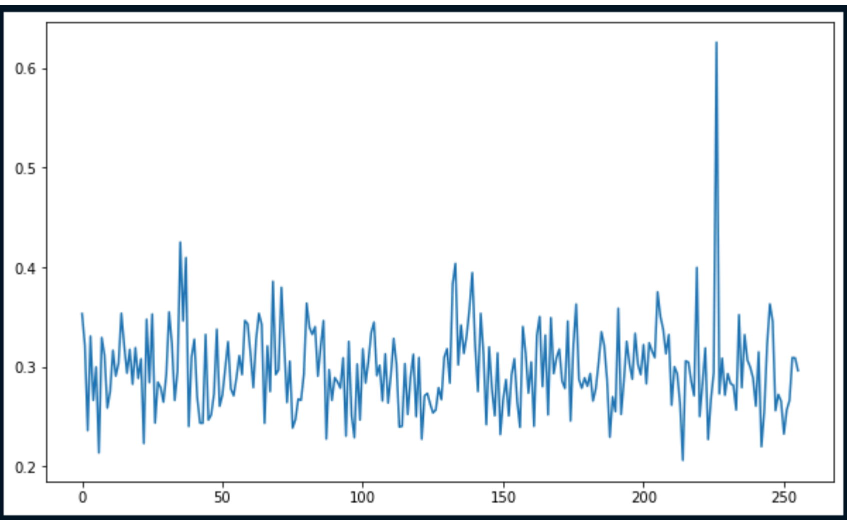
```
The highest correlation coefficient for key byte 10 is 0.5768388838813265 and its corresponding value is 0xcb
```



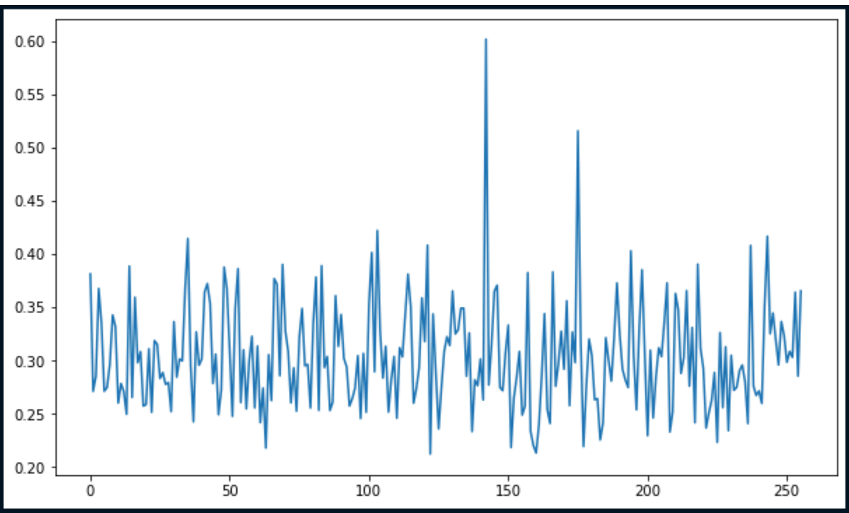
```
The highest correlation coefficient for key byte 11 is 0.4894392024137063 and its corresponding value is 0x4e
```



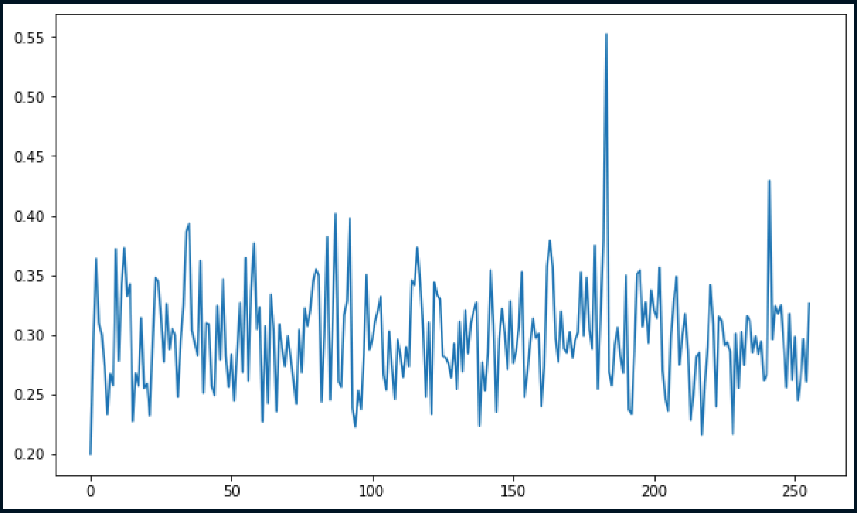
```
The highest correlation coefficient for key byte 12 is 0.6026759167842686 and its corresponding value is 0xd2
```



```
The highest correlation coefficient for key byte 13 is 0.6260018299062153 and its corresponding value is 0xe2
```



```
The highest correlation coefficient for key byte 14 is 0.6015484563777957 and its corresponding value is 0x8e
```



```
The highest correlation coefficient for key byte 15 is 0.5522224562587085 and its corresponding value is 0xb7
```