School of Computer Science and Engineering

CE/CZ 4055 Cyber Physical System Security

Project Report

Aleem Siddique (U1821548E)

Tan Mei Shuang Ethel (U1922103C)

Fatin Farihah Binte Miswan (U1921369E)

Png Yao Wei, Samuel (U1922139J)

Table of Contents

Introduction

A side-channel attack is the exploitation of the physical implementations of hardware or a system where implicit information flow is leaked unintentionally outside of its specified policy. Given its generally non-invasive nature in extracting knowledge of confidential data, there has been a growing number of known side-channel attack vectors [1,2]. This report will discuss one of the most common attacks used to uncover cryptographic keys through the analysis of the power consumption of a cryptographic device.

Power Analysis is a passive form of side-channel attack. It is carried out by studying the power consumption of a cryptographic hardware device by gathering information leaks from the operations carried out by the device. Different inputs will result in varying power usage fluctuations which are the traces. These traces can then be analysed to reveal confidential data [3].

Correlation Power Analysis is the attack that takes in known inputs and outputs and observes the behaviour of the device over the numerous operations carried out. This is done to gather many power traces. A statistical method is then used to find out the key that is most likely used by correlating the hypothetical power and the real power consumption [1].

This report will be focussing on the steps taken to perform the Correlation Power Analysis and elaborate on the implementation of it. Countermeasures of power analysis attack would also be discussed.

CPA Technique

The aim of Correlation Power Analysis (CPA) is to find a correlation between a predicted output and the actual power output. If a power model is accurate, a strong correlation between the two outputs should be established. By accumulating a large number of traces, it will be possible to correctly derive the secret key.

The general CPA Technique relies on a few key implementations and models to successfully derive the secret key. The details of such implementations will be briefly elaborated below.

The substitution box (S-Box) is a fixed lookup table based on the Advanced Encryption Standard (AES) algorithm that is used to analyse the correlation between the hypothetical and the real power usage. If a byte of the plaintext is known, we can derive a hypothetical power model consumption based on Hamming Weight. A correlation matrix based on Pearson's Correlation Coefficient determines the relationship between the power model created with the Hamming Weight and the actual power consumption to establish a relationship. By iteratively executing the technique for each byte of the key at a time we will extract its subkeys. Collectively the subkeys will then recover the secret key.

The CPA Technique pseudocode is as follows.

```
for each byte_value in range {
        Construct hypothetical power model matrix
        Construct correlation matrix using Pearson's Correlation
        Extract the value with the best correlation to determine subkey
}
```

Implementation of CPA

We used a Python program for our CPA implementation since they are the easiest in handling datasets and drawing graphs. We first implemented it on Jupyter Notebook to get a grasp of the plots since they can be executed block by block. We then shifted to a .py file to implement a simple GUI for users to control certain variables.

Since the goal is to find out where the substitution box(S-box) operation is occurring, we used a Python list to create the standard S-box for AES128 cipher.

```
Sbox = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16,)
```

Figure 1: S-box implementation

An S-box used in AES128 is a 16x16 matrix called the Rijndael S-box. The values are already predetermined, hence the easy creation of the S-box matrix.

Since we are deciphering the secret key byte by byte, we created a function where it will return only the given deciphered byte key.

```
def getByteKey(args):
    cycle, data_arr, no_of_possible_values_of_key_byte, no_of_traces,
no_of_power_trace, power_model_matrix, actual_power_model_matrix, plot_graph = args
```

Figure 2: Arguments that can be passed through the getByteKey function

The arguments that can be passed through to the function are as follows:

Argument	Purpose
cycle	The key byte selected to decipher
data_arr	The whole power trace dataset
no_of_possible_values_of_key_byte	S-box size ($16 * 16 = 256$)
no_of_traces	The number of traces used to decipher the key
no_of_power_trace	The period of power trace collected (default to 2500)
power_model_matrix	Empty matrix based on the possible key bytes

actual_power_model_matrix	Power Matrix to
plot_graph	Variable for user to choose to plot a graph or otherwise

Array of the specific byte of all plain texts

The first thing to do was to create an array of the plain text bytes based on the key byte chosen. For example, if key byte 0 was chosen, only plain texts of key byte 0 is put into the array.

```
plaintext_bytes = []
    # Array containing the specific byte of all the plain texts
    for i in range(0,len(data_arr)):
        plaintext_bytes.append(int(data_arr[i][cycle:cycle+2],16))
```

Cycle as mentioned earlier, represents the key byte chosen. If cycle is 0, data_arr[i][0:2] is picked and appended to the array.

Hypothetical power model

Next, we were to construct the hypothetical power model for us to be able to compare with the actual power model.

```
# Hypothetical power model for all values of k
# - Power model matrix
for key_byte_guess in range(0,no_of_possible_values_of_key_byte):
    k = key_byte_guess

    leaky_sbox_output_value_array = []

    # Fill up the leaky_sbox_output_value_array with Sbox(x xor k)
    for byte in plaintext_bytes:
        byte_now = int(byte ^ k)
        leaky_sbox_output_value_array.append(Sbox[byte_now])

    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
```

To create the power model, we fill up the leaky s-box with the respective values based on ‘x xor k’. We then append the number of ‘1’s of the leaky s-box to the hamming weight array. This will create a 256*256 power model matrix array. To calculate the number of ones in a byte, a simple function ‘hw’ was used.

```
# Calculate the number of ones in a byte
def hw(int_no):
    count = 0
    for i in format(int_no, "08b"):
        if i == '1':
            count += 1
    return count
```

As mentioned earlier, when there is a one switch, this will require a higher current during the conversion, and this will be used to compare with the actual power model.

Correlation between model trace of every possible value of key byte and the actual trace

A huge 256*2500 correlation matrix was created using Pearson's correlation.

```
for key_byte_guess in range(0, no_of_possible_values_of_key_byte):
    correlation_values = []
    model_trace = power_model_matrix[key_byte_guess]
    for i in range(0,no_of_power_trace):
        power_trace = actual_power_model_matrix[i]
        corr_value = stats.pearsonr(power_trace, model_trace)
        correlation_values.append(corr_value[0])
    correlation_matrix.append(correlation_values)
```

This code required the longest computation time due to the sheer number of the power trace (2500 traces). Hence, getting the correlation matrix for 1 key byte takes a few minutes. This, however, will be solved later to increase the speed of this method.

Since the correlation matrix is a 256*2500 matrix, we require the best correlation value for each column (key byte) which will then create an array with the length of 256 key bytes.

```
best_correlation_values = []
x_index = []
# get the largest correlation value in every column of the correlation
matrix
for i in range(0, no_of_possible_values_of_key_byte):
    x_index.append(i)
    x = np.where(correlation_matrix[i] == np.amax(correlation_matrix[i]))
    best_correlation_values.append(correlation_matrix[i][x[0][0]])
```

The function of np.where helps to locate the index in the correlation matrix to determine the power trace of the key byte with the highest correlation value. For example, 2500 correlation values per key byte, only the best correlation value is selected and appended to the best correlation values. Creating an array of length 256, to represent all the possible key bytes.

Sorting and getting the key byte with the largest correlation

Using numpy’s argsort, the correlation values are sorted from descending order.

```
sorting_order = np.argsort(best_correlation_values)
sorting_order = sorting_order[::-1]
key = sorting_order[0]
return key
```

Therefore, the value at position 0 in sorting order corresponds to the correct key with the highest correlation value. The value of the key byte is then returned.

Plotting plot 1

The 1st plot requires to plot the correlation of all possible key bytes for 100 traces. The correct key byte will be highlighted in red.

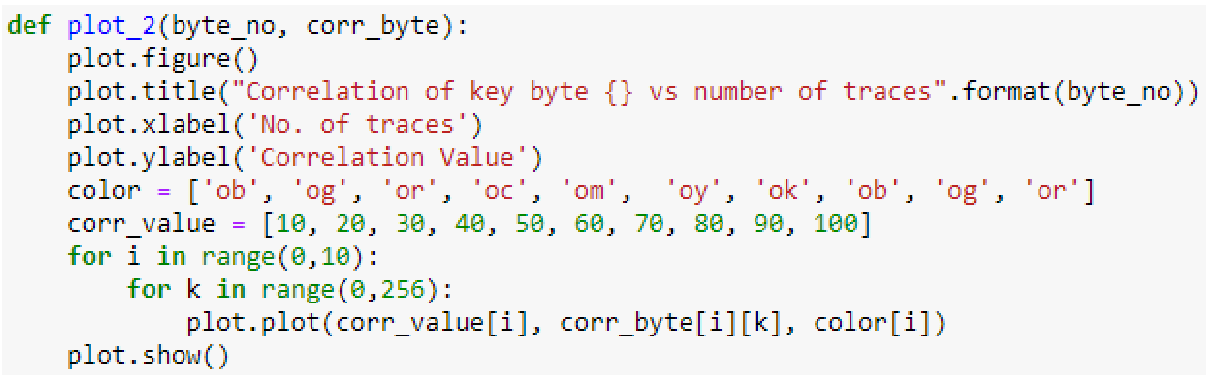
To plot this, it is as simple as plotting the best correlation values (the array with length 256) against the possible key bytes.

```
# Prints the graph for each key byte
best_pos = np.where(best_correlation_values == correlation_matrix[x1[0]][y1[0]])
if (plot_graph):
    plot.figure()
    plot.title("Correct Key Byte : {}".format(sorting_order[0]))
    plot.xlabel('Value of key byte')
    plot.ylabel('Correlation Value')
    plot.plot(x_index, best_correlation_values)
    plot.plot(x1[0], best_correlation_values[best_pos[0][0]],'r*')
    plot.show()
```

To highlight the correct key byte, we had to find the position of the correct key byte. This was done using the np.where function to find the correct index.

Plotting plot 2

The 2nd plot requires to plot the correlation of the correct key byte vs the number of traces. Since this required a bit of manipulation to the code, we used Jupyter Notebook instead and changed the return function for the getByteKey. Instead of returning the correct key, the best correlation values were returned instead.



```
def plot_2(byte_no, corr_byte):
    plot.figure()
    plot.title("Correlation of key byte {} vs number of traces".format(byte_no))
    plot.xlabel('No. of traces')
    plot.ylabel('Correlation Value')
    color = ['ob', 'og', 'or', 'oc', 'om',  'oy', 'ok', 'ob', 'og', 'or']
    corr_value = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
    for i in range(0,10):
        for k in range(0,256):
            plot.plot(corr_value[i], corr_byte[i][k], color[i])
    plot.show()
```

Plotting the graph is quite simple since the 256 correlation values were returned and they were plotted according to the number of traces. In total, the program ran 160 times for 10 different number of traces for each of the 16 key bytes. The colour alternates for each number of traces so that we can differentiate between the different number of traces.

Multiprocessing

As mentioned earlier, the process during the calculation of the correlation values takes the most amount of time. Since a total of 256*2500 values had to be calculated. To speed up the process, we had to think of different ways. Either multithreading or multiprocessing.

Multiprocessing allowed the process to be spread across the number of cores therefore, decreasing the overall time required to calculate the correlation values.

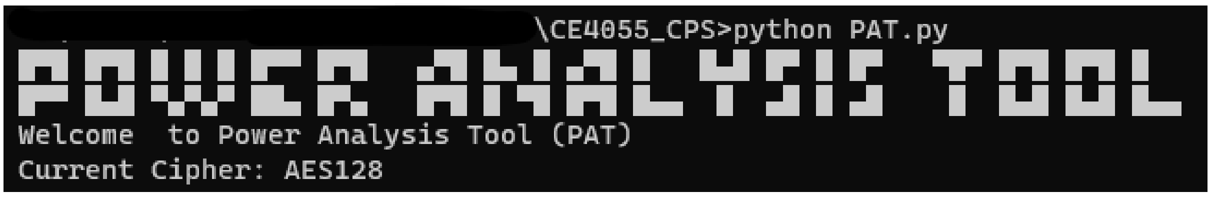
```
# Define function to run mutiple processors and pool the results together
def run_multiprocessing(func, i, n_processors):
    with Pool(processes=n_processors) as pool:
        return pool.map(func, i)


actual_key = run_multiprocessing(getByteKey, key_bytes, n_processors)
```

The key_bytes variable is an array that contains all the necessary argument values for the getByteKey to be run 16 times for all 16 key bytes. Multiprocessing will then stitch and append the resultant values to actual_key which can then be printed. This improved the overall timing for the whole 16-byte key from requiring 10 mins to less than 2 mins. A huge performance upgrades.

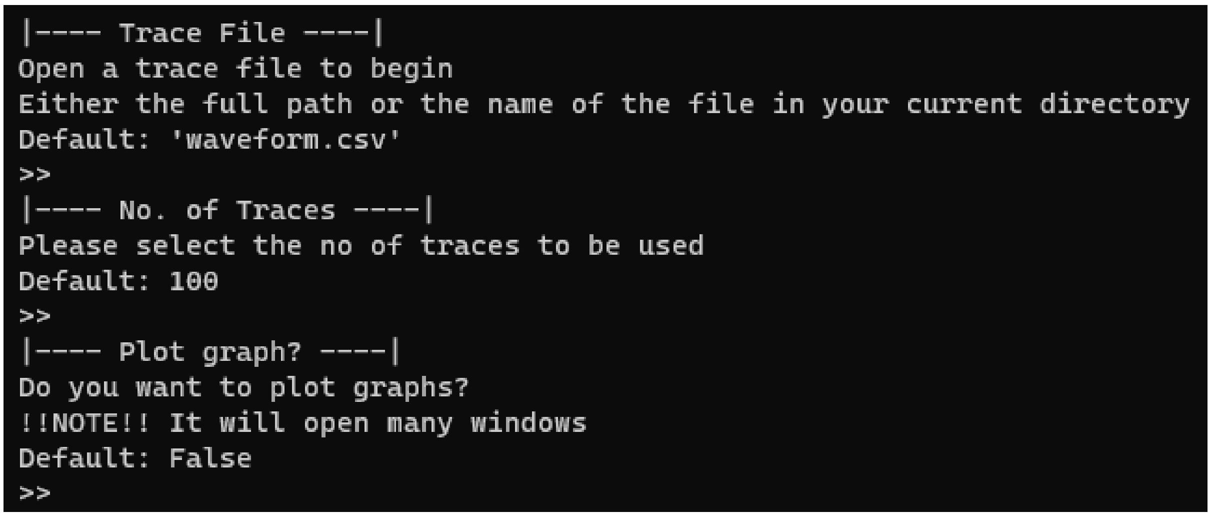
Console Program

We decided to go with the easy console GUI. The file can be run the same as running other scripts. Users will be greeted with this.



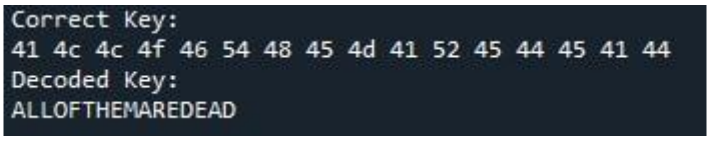
```
[redacted]\CE4055_CPS>python PAT.py
POWER ANALYSIS TOOL
Welcome  to Power Analysis Tool (PAT)
Current Cipher: AES128
```

There are 3 required inputs for the users to use the script.



```
|---- Trace File ----|
Open a trace file to begin
Either the full path or the name of the file in your current directory
Default: 'waveform.csv'
>>
|---- No. of Traces ----|
Please select the no of traces to be used
Default: 100
>>
|---- Plot graph? ----|
Do you want to plot graphs?
!!NOTE!! It will open many windows
Default: False
>>
```

The file path of the trace file, the number of traces and whether the user will want to plot the graphs. This plotting of graphs is only for plot 1 as plot 2 requires changes to the backbone of the code. Overall, if done successfully, the user will be greeted with the correct key bytes and the decoded key (if there is).

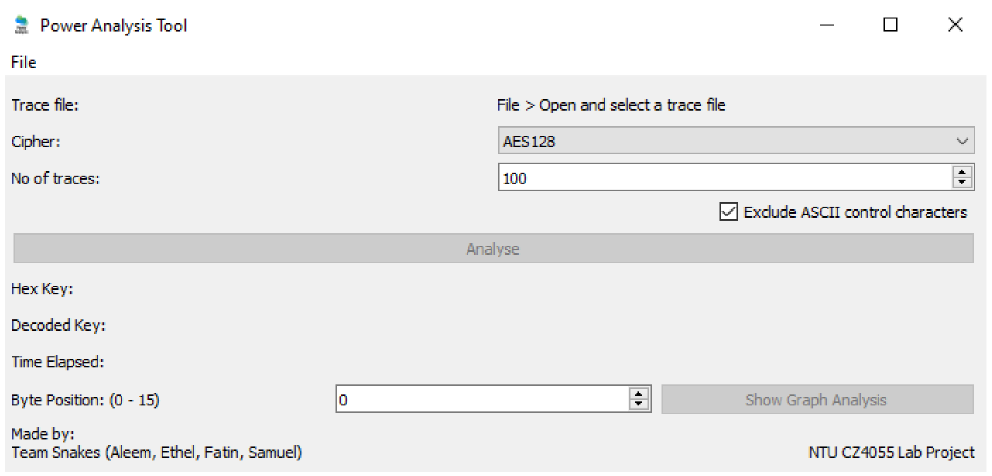


```
Correct Key:
41 4c 4c 4f 46 54 48 45 4d 41 52 45 44 45 41 44
Decoded Key:
ALLOFTHEMAREDEAD
```

In our case, we had a message hidden within the key. A reference to the popular Netflix show, All of Us Are Dead.

GUI Program

On top of the console program, we've also built a GUI with PyQT5 for a much cleaner experience for users. The graphical interface look as follows :


Power Analysis Tool

File

Trace file: File > Open and select a trace file

Cipher: AES128

No of traces: 100

☑ Exclude ASCII control characters

Analyse

Hex Key:

Decoded Key:

Time Elapsed:

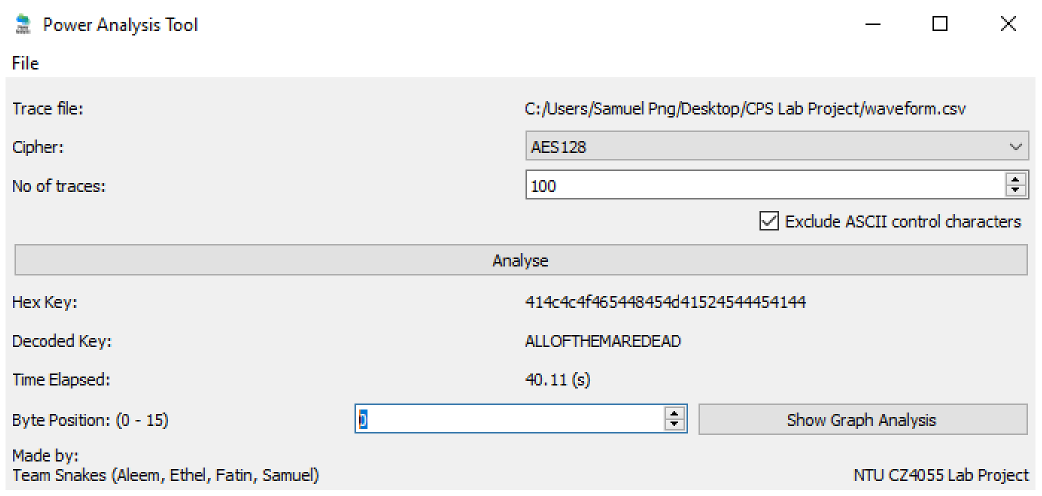
Byte Position: (0 - 15) 0

Show Graph Analysis

Made by:
Team Snakes (Aleem, Ethel, Fatin, Samuel)

NTU CZ4055 Lab Project

Similar to the console gui, the user needs to provide the file path of the trace file, number of traces but also has a new option of excluding ASCII control characters. If excluding ASCII control characters is selected, only ASCII characters will be included, and the guessing range will be 32 to 127 as opposed to 0 to 255. This is done so that we can cut down on the processing time since some key byte guesses are control characters that will unlikely be the correct key byte. We can then press on the “Analyse” button to begin.


Power Analysis Tool

File

Trace file: C:/Users/Samuel Png/Desktop/CPS Lab Project/waveform.csv

Cipher: AES128

No of traces: 100

☑ Exclude ASCII control characters

Analyse

Hex Key: 414c4c4f465448454d41524544454144

Decoded Key: ALLOFTHEMAREDEAD

Time Elapsed: 40.11 (s)

Byte Position: (0 - 15) 0

Show Graph Analysis

Made by:
Team Snakes (Aleem, Ethel, Fatin, Samuel)

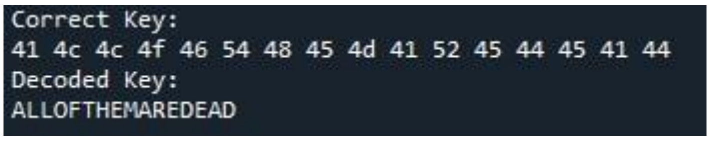
NTU CZ4055 Lab Project

After the analysis process is done, we will be able to see the decoded key in the lower part of the GUI as seen above. Graphical analysis of each key byte guess and their correlation can be viewed by specifying the byte position and pressing on the “Show Graph Analysis” button.

Experimental Results

100 traces key decipher

For the number of traces used is 100, the correct secret key is deciphered. The decimal key byte was converted to hexadecimal.

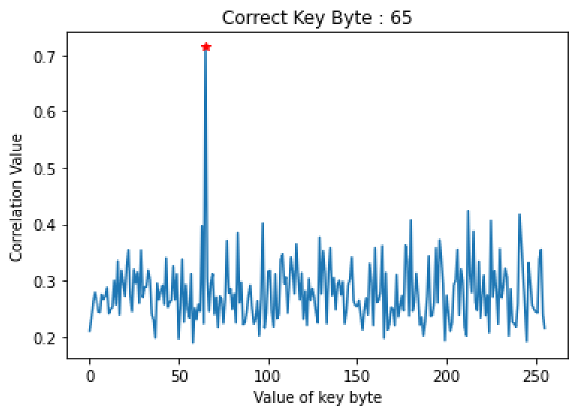
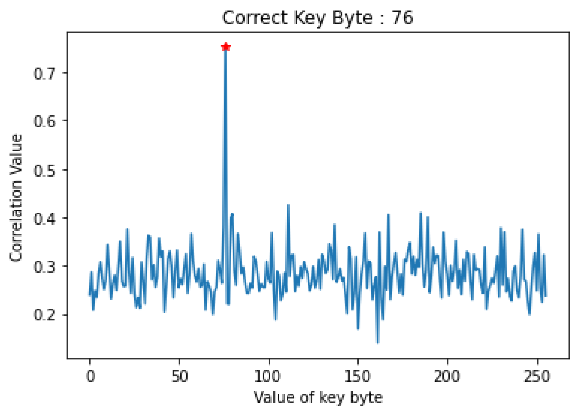
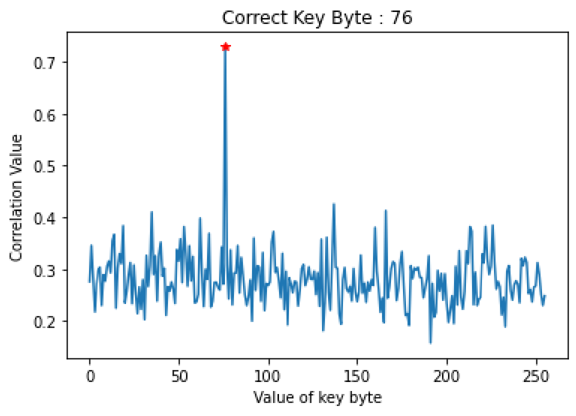
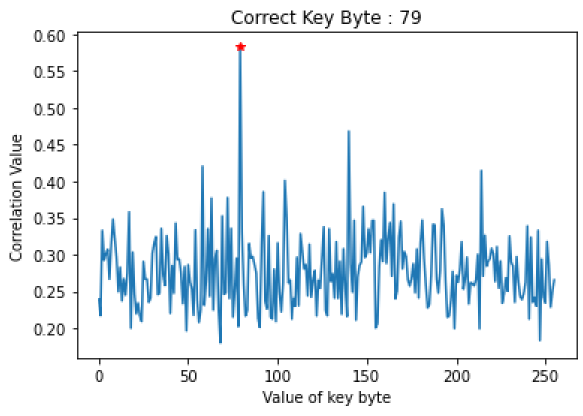


```
Correct Key:
41 4c 4c 4f 46 54 48 45 4d 41 52 45 44 45 41 44
Decoded Key:
ALLOFTHEMAREDEAD
```

Converting the hexadecimal key to characters results in the decoded secret key that was ‘ALLOFTHEMAREDEAD’.

Plot 1

For plot 1, our objective is to plot for “Correlation value vs the value of the key byte”. The correct key byte with the highest correlation value is marked with a red star. All 16 plots for the 16 bytes are plotted below.

 Figure 3: key byte 0	 Figure 4: key byte 1
 Figure 5: key byte 2	 Figure 6: key byte 3

Even though the main correlation matrix has a size of 256 x 2500, only the best correlation of each 2500 columns were picked, leaving only with 256 values.

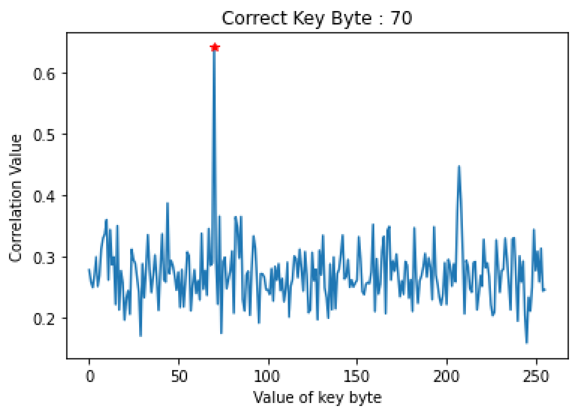


Figure 7: key byte 4

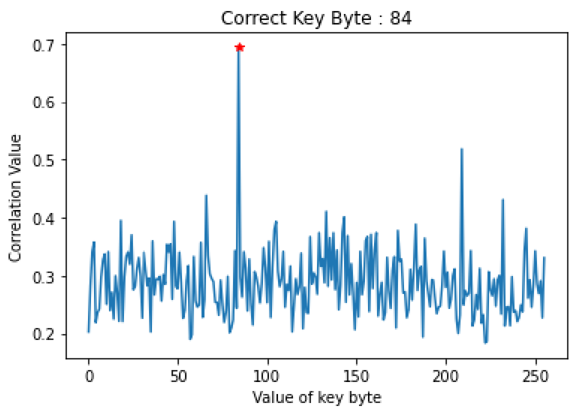


Figure 8: key byte 5

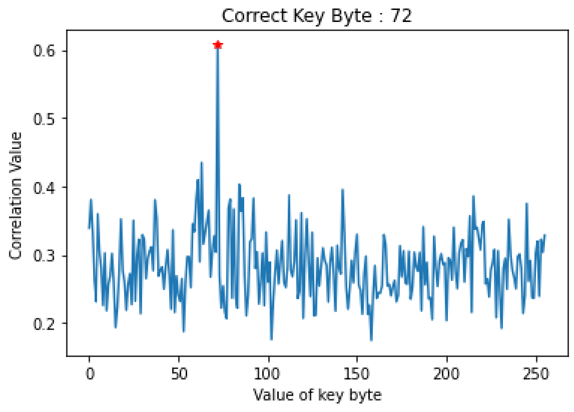


Figure 9: key byte 6

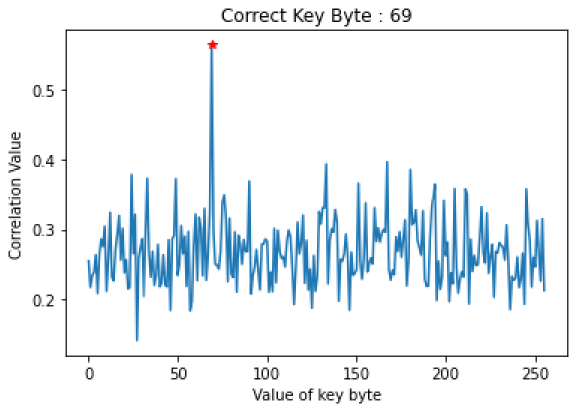


Figure 10: key byte 7

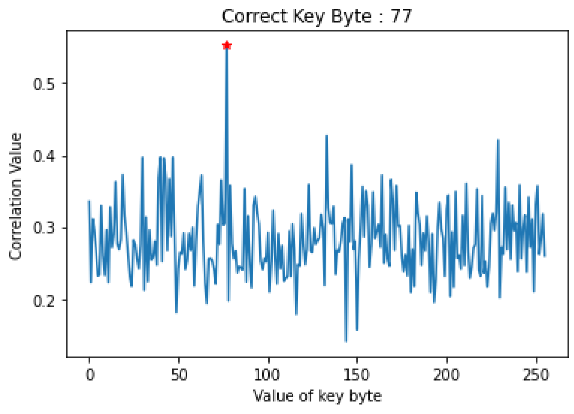


Figure 11: key byte 8

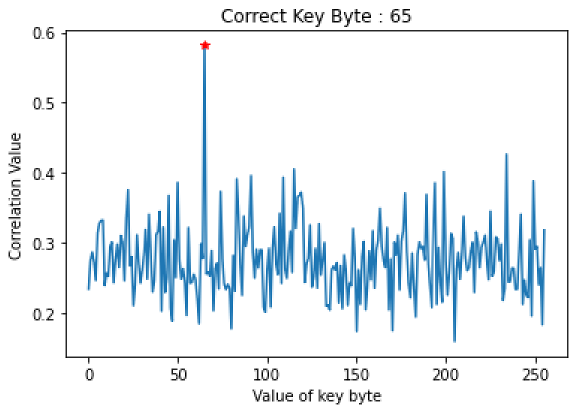


Figure 12: key byte 9

Some of the correlation distribution has only 1 distinct peak but some key bytes, e.g., key byte 5 in Figure 8 has about 3 peaks but only 1 peak has the highest correlation value. Due to the number of distinct peaks in key byte 5, this will be obvious in plot 2, where the correct key byte does not easily emerge based on the number of traces used.

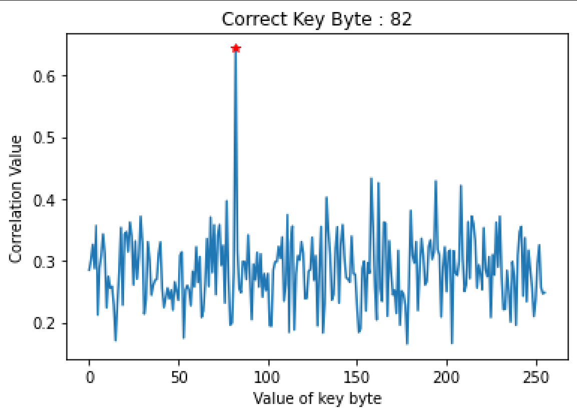


Figure 13: key byte 10

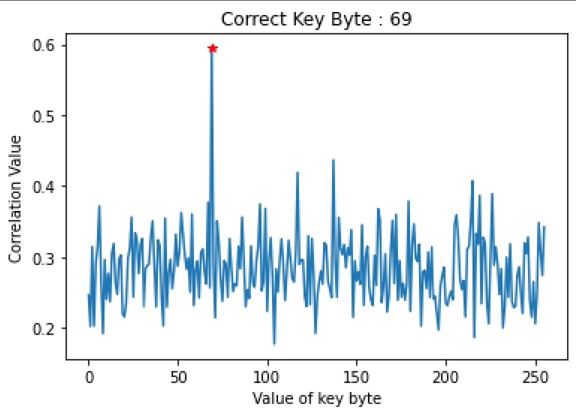


Figure 14: key byte 11

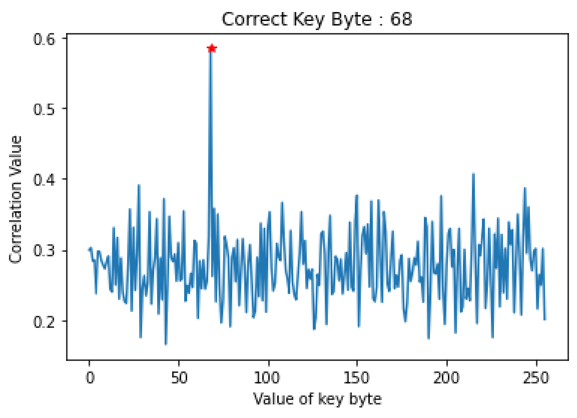


Figure 15: key byte 12

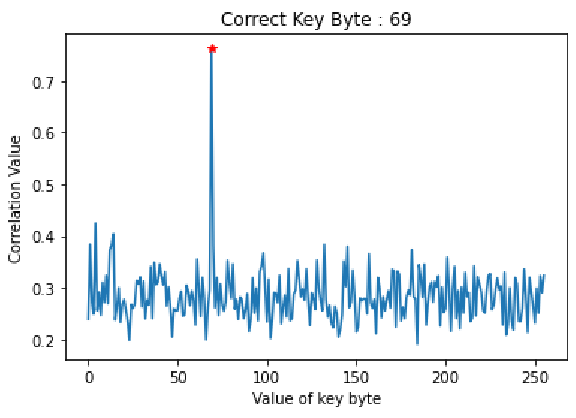


Figure 16: key byte 13

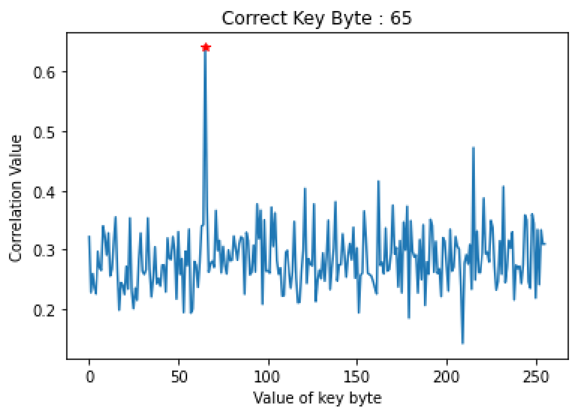


Figure 17: key byte 14

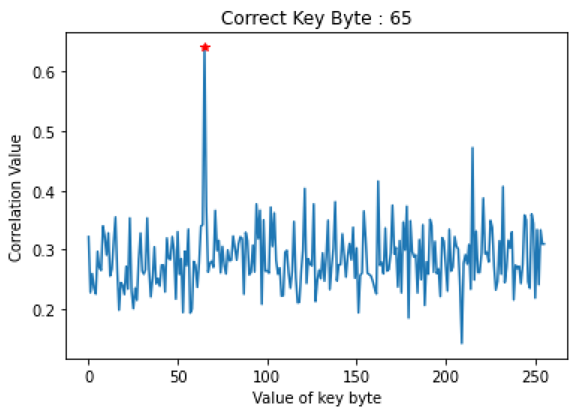
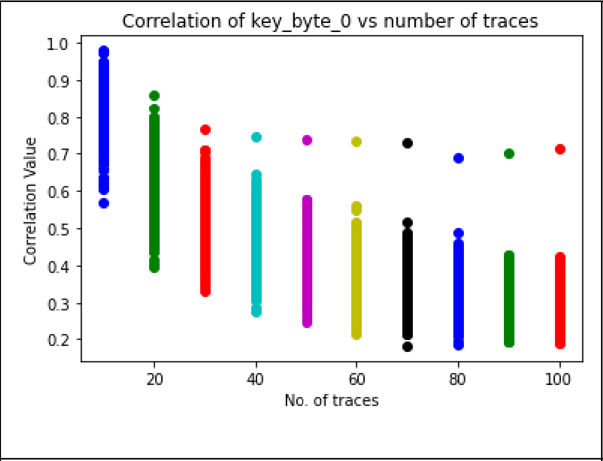


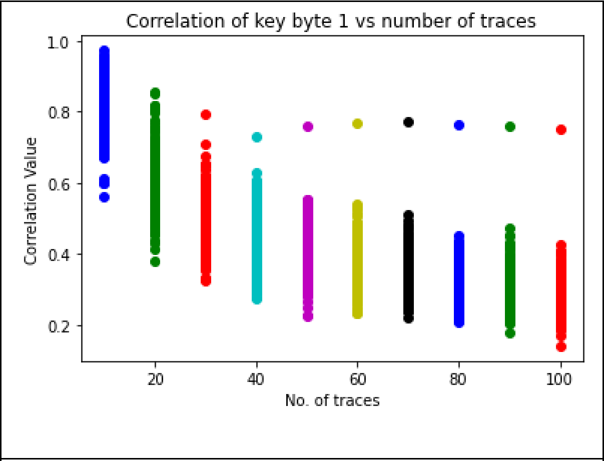
Figure 18: key byte 15

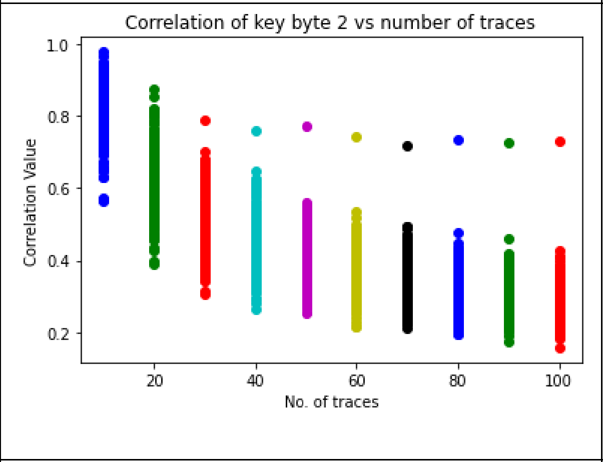
With 100 traces, all the possible key bytes were plotted correctly, without problems.

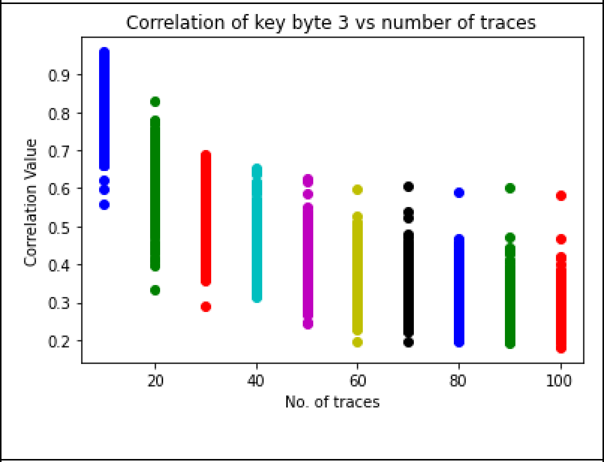
Plot 2

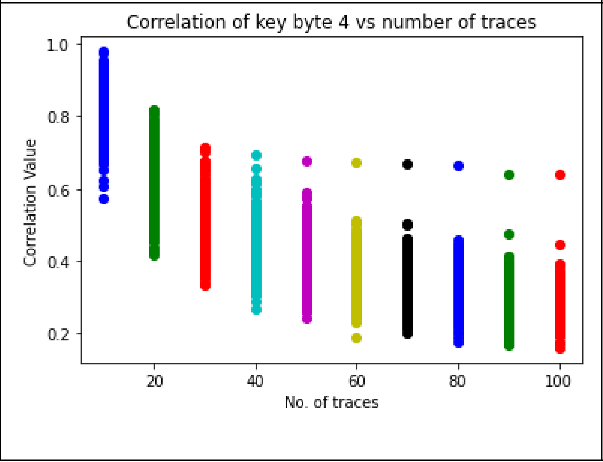
For plot 2, our task was to plot the correlation of the correct key byte vs the number of traces. We were to observe the plot where the correlation for the correct key byte emerges from correlation of all the other wrong key bytes.

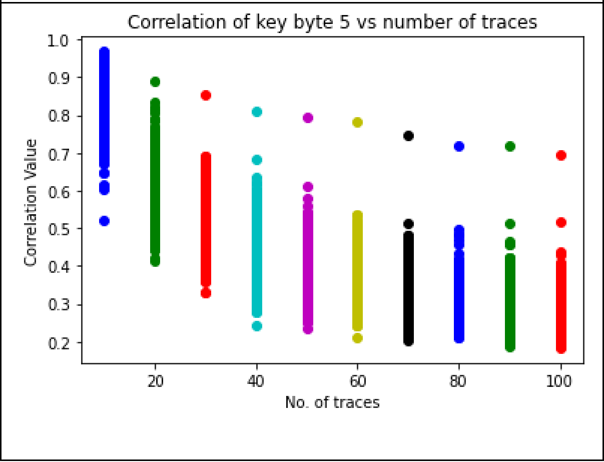


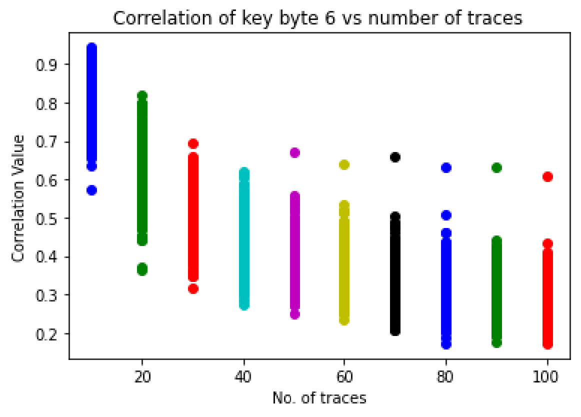


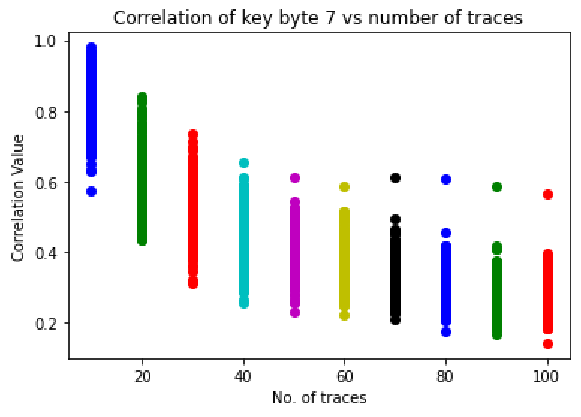


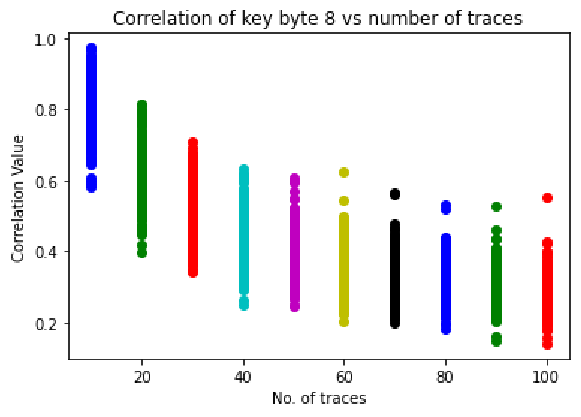


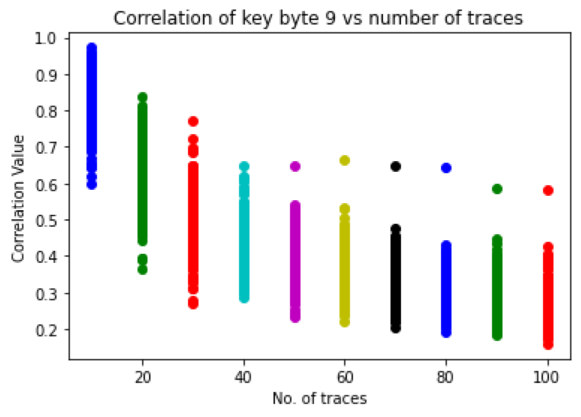


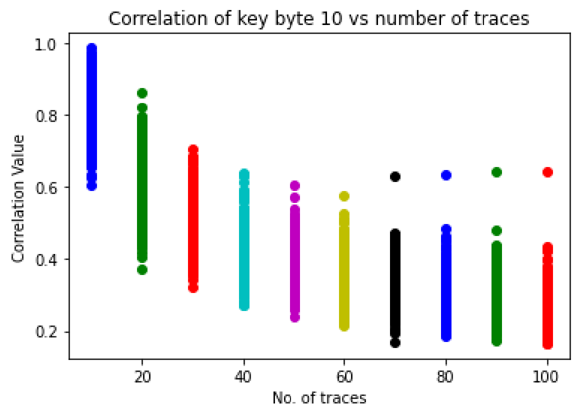


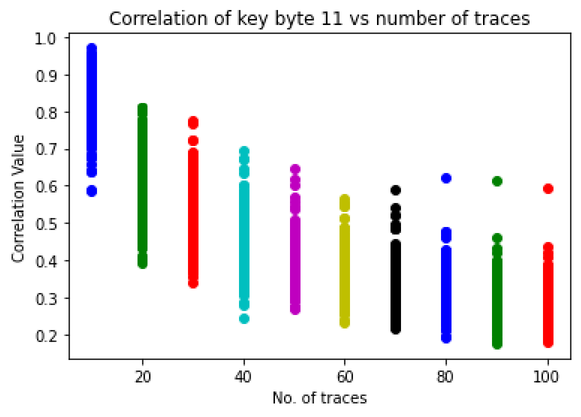

Correlation of key byte 6 vs number of traces
Correlation Value
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
20
40
60
80
100
No. of traces

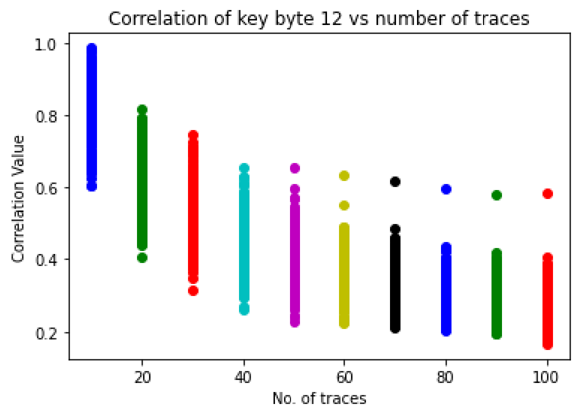

Correlation of key byte 7 vs number of traces
Correlation Value
0.2
0.4
0.6
0.8
1.0
20
40
60
80
100
No. of traces

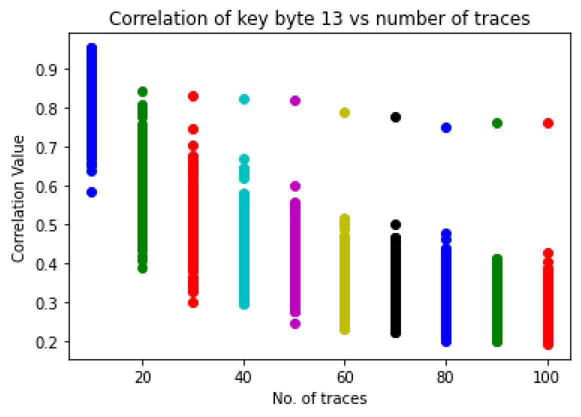

Correlation of key byte 8 vs number of traces
Correlation Value
0.2
0.4
0.6
0.8
1.0
20
40
60
80
100
No. of traces

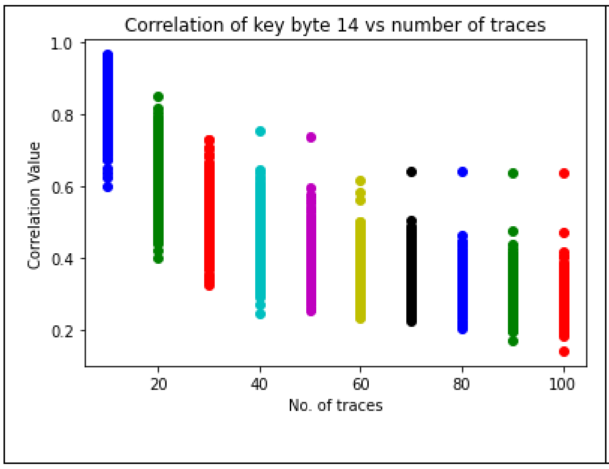

Correlation of key byte 9 vs number of traces
Correlation Value
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
20
40
60
80
100
No. of traces

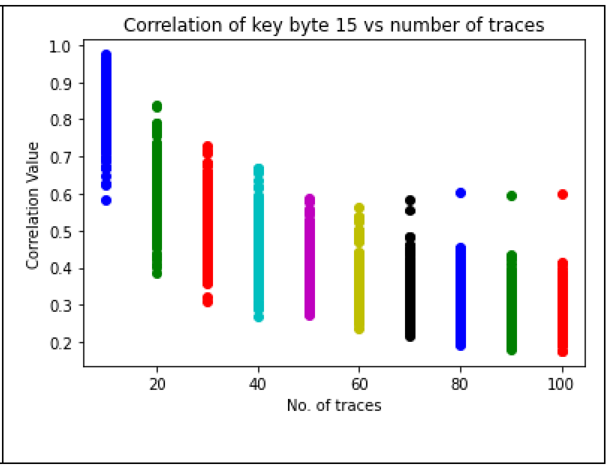

Correlation of key byte 10 vs number of traces
Correlation Value
0.2
0.4
0.6
0.8
1.0
20
40
60
80
100
No. of traces


Correlation of key byte 11 vs number of traces
Correlation Value
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
20
40
60
80
100
No. of traces


Correlation of key byte 12 vs number of traces
Correlation Value
0.2
0.4
0.6
0.8
1.0
20
40
60
80
100
No. of traces


Correlation of key byte 13 vs number of traces
Correlation Value
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
20
40
60
80
100
No. of traces





As seen from the plots above, most correct key byte emerges after 60 traces. However, for some, we can see it requires almost 80 traces for it to correctly separate the correlation values of the correct key byte. This is true when we looked at the deciphered key, where some of the key byte values remained correct till the number of traces get reduced beyond the threshold.

Actual secret key: '41 4c 4c 4f 46 54 48 45 4d 41 52 45 44 45 41 44'

No. of traces	**Deciphered correct key byte**	**Wrong key bytes**
100	41 4c 4c 4f 46 54 48 45 4d 41 52 45 44 45 41 44	None
90	41 4c 4c 4f 46 54 48 45 4d 41 52 45 44 45 41 44	None
80	41 4c 4c 4f 46 54 48 45 85 41 52 45 44 45 41 44	8th
70	41 4c 4c 4f 46 54 48 45 85 41 52 45 44 45 41 44	8th
60	41 4c 4c 4f 46 54 48 45 85 41 52 45 44 45 41 fa	8th & 15th
50	41 4c 4c 4f 46 54 48 45 85 41 52 89 44 45 b1 44	8th, 11th, 14th
40	41 4c 4c 4f cf 54 fa 45 9b 41 75 89 44 45 b1 1f	6 wrongs
30	41 4c 4c e3 fe 54 e8 8a 84 cc 29 89 22 45 b1 1f	11 wrongs
20	e9 bf 3c 9f 42 2c a5 37 6d cc e7 d7 ef 45 f9 73	15 wrongs
10	e7 fb 5c 36 9f 68 cc 0 d4 28 9c bc 6e ad 9 2c	ALL wrong

As seen from the table above, as expected, less traces result in greater inaccuracy.

Countermeasures Against Side Channel Attacks

The countermeasures to be taken against the many side-channel attacks vary depending on the type of attack being carried out, therefore this literature survey focuses on countermeasures against power analysis side channel attacks.

Increasing the difficulty in differentiating the signal traces

The first type of countermeasures aims to increase the difficulty in differentiating the signal in the traces. The constant exponentiation time countermeasure does this by ensuring that the results of each exponentiation operation are only shown after taking the same amount of time[4]. Alternatively, the hiding technique changes the power consumption in the time and amplitude domains through adding dummy operations, reshuffling operations, or adding random noise[4,5,6]. Both methods cause performance to degrade, but the hiding technique has better performance than the constant exponentiation time technique[4]. However, with enough traces, the secret key can still be discovered[5]. Therefore, countermeasures of this type cannot completely prevent side channel attacks.

Eliminate the relationship between the leaked power consumption & key

The second type of countermeasures aim to eliminate the relationship between the leaked power consumption and the secret key. Two countermeasures that do this are blinding and masking.

Firstly, blinding involves multiplying the ciphertext with a random integer before performing an exponentiation operation to increase the difficulty in deducing the correct ciphertext[4,7]. However, this technique not only causes significant performance degradation but also can be attacked[7].

Secondly, masking involves multiplying intermediate values with a mask to randomise them so that the power consumption will have no relation to the actual plaintext[5,6,8]. While this method has proven to be more effective than countermeasures in the first type[5], unintentional interactions with the hardware can still lead to a leakage of information[9] and performance may still degrade[6]. Therefore, researchers have improved on masking by creating methods to ensure and verify near perfect masking[8,9].

Overall, while countermeasures to side channel attacks exist, performance degradation is often a trade-off for them. Additionally, new side channel attacks will continue to emerge at an overwhelming rate[10]. Therefore, the sensitivity of the data to be protected and impact of performance degradation should be considered in order to determine the appropriate amount and combination of countermeasures to apply to the data. Vigilance is also important.

Citations

[1] L. Owen, B. William J, and D.Carson. "Power analysis attacks on the AES-128 S-box using differential power analysis (DPA) and correlation power analysis (CPA)." Journal of Cyber Security Technology 1.2, 2017,pp. 88-107, doi.org/10.1080/23742917.2016.1231523.

[2] G. Joy Persial, M. Prabhu, and R. Shanmugalakshmi. "Side channel attack-survey." Int J Adva Sci Res Rev 1.4 , 2011, pp. 54-57.

[3] H. Gamaarachchi and H. Ganegoda. "Power analysis based side channel attack." arXiv preprint arXiv:1801.00932 ,2018 , doi.org/10.48550/arXiv.1801.00932.

[4] M. Devi and A. Majumder, "Side-Channel Attack in Internet of Things: A Survey," in Applications of Internet of Things, Singapore, J. K. Mandal, S. Mukhopadhyay, and A. Roy, Eds., 2021// 2021: Springer Singapore, pp. 213-222.

[5] P. Arpaia, F. Bonavolontà, A. Cioffi and N. Moccaldi, "Power Measurement-Based Vulnerability Assessment of IoT Medical Devices at Varying Countermeasures for Cybersecurity," in IEEE Transactions on Instrumentation and Measurement, vol. 70, pp. 1-9, 2021, Art no. 5503209, doi: 10.1109/TIM.2021.3088491.

[6] M. Zhao and G. E. Suh, "FPGA-Based Remote Power Side-Channel Attacks," 2018 IEEE Symposium on Security and Privacy (SP), 2018, pp. 229-244, doi: 10.1109/SP.2018.00049.

[7] R. Kumar et al., "A Time-/Frequency-Domain Side-Channel Attack Resistant AES-128 and RSA-4K Crypto-Processor in 14-nm CMOS," in IEEE Journal of Solid-State Circuits, vol. 56, no. 4, pp. 1141-1151, April 2021, doi: 10.1109/JSSC.2021.3052146.

[8] H. Eldib, C. Wang, and P. Schaumont, "Formal Verification of Software Countermeasures against Side-Channel Attacks," ACM Trans. Softw. Eng. Methodol., vol. 24, no. 2, p. Article 11, 2014, doi: 10.1145/2685616.

[9] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards automatic elimination of power-analysis leakage in ciphers," arXiv preprint arXiv:1912.05183, 2019.

[10] R. Spreitzer, V. Moonsamy, T. Korak and S. Mangard, "Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices," in IEEE Communications Surveys & Tutorials, vol. 20, no. 1, pp. 465-488, Firstquarter 2018, doi: 10.1109/COMST.2017.2779824.