

**School of Computer Science and Engineering**

**CE/CZ 4055 Cyber Physical System Security**

**Project Report**

Aleem Siddique (U1821548E)

<Insert Name Here>

Table of Contents

[CPA Technique 3](#_Toc100005822)

[Implementation of CPA 4](#_Toc100005823)

[Array of the specific byte of all plain texts 5](#_Toc100005824)

[Hypothetical power model 5](#_Toc100005825)

[Correlation between model trace of every possible value of key byte and the actual trace 6](#_Toc100005826)

[Sorting and getting the key byte with the largest correlation 6](#_Toc100005827)

[Plotting plot 1 7](#_Toc100005828)

[Plotting plot 2 7](#_Toc100005829)

[Multiprocessing 8](#_Toc100005830)

[Console GUI 8](#_Toc100005831)

[Experimental Results 10](#_Toc100005832)

[100 traces key decipher 10](#_Toc100005833)

[Plot 1: 10](#_Toc100005834)

[Plot 2: 13](#_Toc100005835)

[Countermeasures against side-channel attacks 16](#_Toc100005836)

# CPA Technique

# 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 on Jupyter Notebook to get a grasp of the plots since they can be executed block by block. We then shifted to .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.



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.



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 () |
| 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.



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.



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.



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.



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.



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.



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.



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

Text, scatter chart

Description automatically generated

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 multi-threading 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.



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 GUI

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.

Text

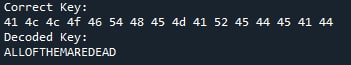
Description automatically generated with low confidence

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

Text

Description automatically generated

The file path of the trace file, the number of traces and whether the user will want to plot the graphs. This plotting of graph 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).



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

# 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.

Graphical user interface, text

Description automatically generated

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, out 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.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

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