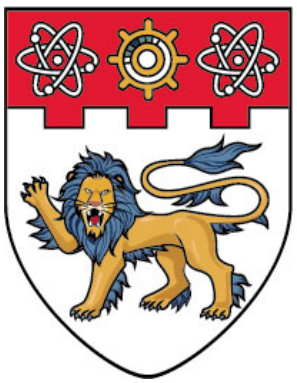

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE4055: Cyber Physical System Security

Correlation Power Analysis Project Report

Name: Tong Qi Long (U1822422C)

Lab Group: CE4

Table of Contents

1. Introduction

The Correlation Power Analysis (CPA) attack on the AES-128 algorithm works based on the Divide and Conquer strategy where 1 byte of the secret key is retrieved at a time and it is repeated 16 times for the whole key. The attack is done at the output of S-Box. Consider A0 is the first byte of the plaintext and K0 is the first byte of the key. The first byte Y0 of SBOX's output is given as Y0 = SBOX(A0⊕K0). For all 256 possible values of K0 (0 to 255), Y0 is calculated for the different plaintexts.

Hamming Weight Power Model can be used to obtain Hamming weight of Y0. The Hamming weight is the number of non-zeroes in a sequence. In 100 traces, a model trace matrix (100x256) is created for each byte of K. An actual power trace matrix (2500x100) is also created where 2500 is the number of points in one trace and 100 is the number of traces. Pearson correlation coefficient equation can be used to calculate correlation between the model trace matrix and the actual trace matrix. A new correlation matrix (256 x 2500) is then obtained. The location(row) with the highest correlation value in the correlation matrix is the correct key byte.

Given a set of power traces (in a csv file format as captured in the lab), a Correlation Power Analysis (CPA) program is created in this project to implement CPA on the given traces to extract the key. In this project report, the implementation of CPA in the program will be explained and the plots showing the correlation values of key bytes will be shown. Lastly, the countermeasures against side-channel attacks will be discussed.

2. Implementation of Correlation Power Analysis

The programming language chosen to implement Correlation Power Analysis (CPA) on the given set of traces to extract the key is Java. Java is chosen as there are many computations involved to obtain the correlation matrix due to large number of samples points and Java is a compiled language which takes less time to perform these computations.

The pseudocode to implement CPA is shown in Figure 1.

```
initialize actualPowerMatrix[][] array from CSV file
initialize plainText[] array from CSV file

for each keyByte 1 to 16:
        for keyGuess 0 to 255:
                set i to 0
                for plainText[] 0 to noOfPowerTraces:
                        set j to 0
                        sBoxValue = sBox[plainText XOR keyGuess]
                        hammingPowerMatrix[j][i] = getHammingWeight(sBoxValue)
                        increment j by 1
                increment i by 1
        for i 0 to 255:
                colOfHammingPowerMatrix[i] = getHammingPowerMatrixAtColumn(i)
                for j 0 to noOfPointsIn1Trace:
                        colOfActualPowerMatrix[j] = getActualPowerMatrixAtColumn(j)
                        corMatrix[i][j] = calculatePearsonCoefficient(colOfActualPowerMatrix,colOfHammingPowerMatrix)
        actualKey[keyByte-1] = getKeyGuessWithHighestCorrelation()
```

Figure 1

In the pseudocode from Figure 1, the array *actualPowerMatrix[][]* and *plainText[]* array are first initialized using the data from the CSV file at the start of the program. The array *actualPowerMatrix[][]* stores 2500 samples for each power traces and the array *plainText[]* stores the plaintext for each power traces. Since there are 16 key bytes used for encryption in the lab, CPA is performed on each key byte obtain all the 16 keys. The *hammingPowerMatirx[][]* is an array which stores 256 hamming weight values for each traces. The variable *noOfPowerTraces* is the number of power traces to perform CPA on and the variable *noOfPointsIn1Trace* is the number of points in each power trace.

The implementation of the CPA pseudocode in the actual Java program is organised into various functions such as *initializeActualPowerMatrixFromCSV()*,

hammingPowerMatrixCreation(), *correlationMatrixCreation()* and *getKeyGuessWithHighestCorrelation()*. The main program of CPA is shown in Figure 2. Each function in the main program will be explained in the next few sections.

```
public CPASecretKeyRecovery() {

    initializeActualPowerMatrixFromCSV("waveform.csv");
    int count = 0;
    for(int i = 1; i<=16;i++) {
        for(int noOfTraces = 10; noOfTraces<=100;noOfTraces+=10) {
            noOfPowerTraces = noOfTraces;
            hammingPowerMatrixCreation(i);
            correlationMatrixCreation();
            recoveredKey[i-1] = getKeyGuessWithHighestCorrelation(count);
            count++;
        }
        if(i == 1) {
            System.out.println(i + " byte of 16 Keys recovered...");
        }else {
            System.out.println(i + " bytes of 16 Keys recovered...");
        }
    }
    System.out.println("");
    printRecoveredKey();
    System.out.println("");
    exportToCSV("plot1Data",plot1Data);
    exportToCSV("plot2Data",plot2Data);
    System.out.println("");
    hammingDistanceFeature();
    System.out.print("Program exit ...");
}
```

Figure 2

2.1 Initialization of Actual Power Matrix

The actual power matrix contains all the samples points (2500) of each traces collected in the lab. It is first initialized at the start of the program by reading the CSV file which contains samples points of all the traces. The implementation to read the CSV file and store the data of traces to the actual power matrix is shown in Figure 3.

```
public void initializeActualPowerMatrixFromCSV(String fileName) {

    sizeOfActualPowerMatrixFromCSV(fileName);
    try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
        String row;
        try {
            int j=0;
            while ((row = br.readLine()) != null) {
                String[] data = row.split(",");
                plainText[j] = data[0];
                for(int i = 2; i<noOfPointsInATrace;i++) {
                    actualPowerMatrix[j][i-2] = Float.parseFloat(data[i]);
                }
                j++;
            }
            System.out.println("Successful Import of Data from waveform.csv");
            System.out.println("Key Recovery in Progress ...");
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(0);
        }
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
    }

}
```

Figure 3

2.2 Initialization of Hamming Power Matrix

Hamming Power Matrix contains the hamming weight value of sBox[plainText XOR keyGuess]. It is declared as a 2D array [row][column] with row representing the number of traces and column representing the number of key guess, 255. sBox is an array which stores the lookup result of S-Box as shown in Figure 4. The function getHammingWeight() computes the numbers of "1" in the lookup result of S-Box as shown in Figure 5. The implementation to initialize Hamming Power Matrix is shown in Figure 6.

```
public static int[] 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 4

```
public int getHammingWeight(int sBoxValue) {

    return Integer.bitCount(sBoxValue);
}
```

Figure 5

```
public void hammingPowerMatrixCreation(int num) {

    int sBoxValue;
    hammingPowerMatrix = new float[noOfPowerTraces][256];

    for(int i = 0 ; i<noOfPowerTraces;i++) {
        int plainTextOneByte =  Integer.parseInt(plainText[i].substring(2*(num-1),2*num),16);
        for(int keyGuess = 0; keyGuess<256;keyGuess+=1) {
            sBoxValue = sBox[plainTextOneByte ^ keyGuess];
            hammingPowerMatrix[i][keyGuess] = getHammingWeight(sBoxValue);
        }
    }
}
```

Figure 6

2.3 Calculation of Pearson Correlation Coefficient

The Pearson Correlation Coefficient equation is used to calculate correlation between the Actual Power Matrix and Hamming Power Matrix. The formula of Pearson Correlation Coefficient is given below. Pearson's correlation coefficient returns a value between -1 and 1. If the correlation coefficient is 1, it indicates a strong positive relationship. If the correlation coefficient is -1, it indicates a strong negative relationship.

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{\left[n\sum x^2 - (\sum x)^2\right]\left[n\sum y^2 - (\sum y)^2\right]}}$$

Notations:

N = Quantity of information

∑x = Total of the first variable value

∑y= Total of the second variable value

∑xy = Sum of first and second value

$\sum x^2$ = Sum of the Squares of the first value

$\sum y^2$ = Sum of the Squares of the first value

In CPA, x represents the values in Actual Power Matrix, y represents the values in Hamming Power Matrix and n represents the number of traces. The implementation to calculate Person Correlation Coefficient is shown in Figure 7.

```
private float calculatePearsonCoefficient(float[] x, float[] y) {
    float sumX = 0;
    float sumY = 0;
    float sumXY = 0;
    float sumXSquare = 0;
    float sumYSquare = 0;
    int n = noOfPowerTraces;

    for (int i = 0; i < n; i++)
    {
      sumX += x[i];
      sumY += y[i];
      sumXY += x[i] * y[i];
      sumXSquare += x[i] * x[i];
      sumYSquare += y[i] * y[i];
    }

    return  (float) ((n * sumXY - sumX * sumY)/ Math.sqrt((n * sumXSquare - sumX * sumX) * (n * sumYSquare - sumY * sumY)));
}
```

Figure 7

2.4 Initialization of Correlation Matrix

Correlation Matrix contains the Pearson Correlation Coefficient between Actual Power Matrix and Hamming Power Matrix. It is declared as a 2D array [i][j] with i representing the number of key guess, 255 and j representing the number of points in a trace. The implementation to initialize Correlation Matrix is shown in Figure 8.

```
public void correlationMatrixCreation() {
    float columnFromActualPowerMatrix[] = new float[noOfPowerTraces];
    float columnFromHammingPowerMatrix[] = new float[noOfPowerTraces];

    for(int j = 0; j< 255;j++) {

        columnFromHammingPowerMatrix = getHammingPowerMatrixAtColumn(j);

        for(int i = 0 ;i<noOfPointsInATrace;i++) {

            columnFromActualPowerMatrix = getActualPowerMatrixAtColumn(i);
            correlationMatrix[j][i] = calculatePearsonCoefficient(columnFromActualPowerMatrix,columnFromHammingPowerMatrix);
        }
    }
}
```

Figure 8

```
public float[] getHammingPowerMatrixAtColumn(int i) {

    float columnFromHammingPowerlMatrix[] = new float[noOfPowerTraces];
    for(int j = 0; j<noOfPowerTraces;j++)
        columnFromHammingPowerlMatrix[j] = (float) (hammingPowerMatrix[j][i]) ;
    return columnFromHammingPowerlMatrix;
}
```

Figure 9

```
public float[] getActualPowerMatrixAtColumn(int i) {

    float columnFromActualPowerMatrix[] = new float[noOfPowerTraces];
    for(int j = 0; j<noOfPowerTraces;j++)
        columnFromActualPowerMatrix[j] = (float) (actualPowerMatrix[j][i]) ;
    return columnFromActualPowerMatrix;
}
```

Figure 10

2.5 Key Recovery from Correlation Matrix

Key can be obtained by finding the row at which the highest correlation value is presented. The row index is the actual key byte. The implementation to obtain all the 16 key bytes is shown in Figure 11.

```
public String getKeyGuessWithHighestCorrelation(int rowCount) {
    float highestCor = -1;
    int highestCorByte = 0;
    for (int keyGuess = 0; keyGuess < 256; keyGuess++) {
        float rowHighestCor = -1;
        for (int i = 0; i < noOfPointsInATrace; i++) {
            if(correlationMatrix[keyGuess][i]>rowHighestCor) {
                rowHighestCor = correlationMatrix[keyGuess][i];
            }
            if (correlationMatrix[keyGuess][i] > highestCor)
            {
                highestCor = correlationMatrix[keyGuess][i];
                highestCorByte = keyGuess;
            }
        }
        if(noOfPowerTraces == 100) {
            plot1Data[(rowCount+1)/10 -1][keyGuess] = String.valueOf(rowHighestCor);
        }
        plot2Data[rowCount][keyGuess] = String.valueOf(rowHighestCor);
    }
    return Integer.toHexString(highestCorByte);

}
```

Figure 11

All the 16 key bytes will appear on the console after successful execution of the program as shown in Figure 12.

```
No of Power Traces: 100 No of Points in a Trace: 2500
Successful Import of Data from waveform.csv
Key Recovery in Progress ...
1 byte of 16 Keys recovered...
2 bytes of 16 Keys recovered...
3 bytes of 16 Keys recovered...
4 bytes of 16 Keys recovered...
5 bytes of 16 Keys recovered...
6 bytes of 16 Keys recovered...
7 bytes of 16 Keys recovered...
8 bytes of 16 Keys recovered...
9 bytes of 16 Keys recovered...
10 bytes of 16 Keys recovered...
11 bytes of 16 Keys recovered...
12 bytes of 16 Keys recovered...
13 bytes of 16 Keys recovered...
14 bytes of 16 Keys recovered...
15 bytes of 16 Keys recovered...
16 bytes of 16 Keys recovered...

Recovered Key: 43 59 42 45 52 50 48 59 53 49 43 41 4c 53 59 53
```

Figure 12

2.6 Additional Feature: Hamming Distance Calculator

After each successful recovery key with the program, the user can check for correctness by entering the correct key used for encryption. The program will compare and output the difference in hamming weights between the correct key used for encryption and the recovered key from the CPA program known as hamming distance. The CPA program will also output the total number of bits in which the recovered key is different from the correct key. This feature is similar to the Hamming Distance Checker tool in the Power Analysis Tool used at the lab. The implementation of the Hamming Distance Calculator is shown in Figure 13. The sample output of the Hamming Distance Calculator is shown in Figure 14.

```
public void hammingDistanceCalculator(String userInput, String arr[]) {

    String str = "";
    String input = userInput.replaceAll(" ","");
    int val1,val2;
    int sumUnmatchedBits=0;
    int j = 0;
    for(int i =0; i<arr.length;i++) {
        val1 = getHammingWeight(Integer.parseInt(input.substring(j, j+1),16)^Integer.parseInt(arr[i].substring(0, 1),16));
        sumUnmatchedBits +=val1;
        str = str + val1 + "";
        val2 = getHammingWeight(Integer.parseInt(input.substring(j+1, j+2),16)^Integer.parseInt(arr[i].substring(1, 2),16));
        sumUnmatchedBits +=val2;
        str = str + val2 + " ";
        j+=2;
    }
    System.out.println("Distance      :" + str);
    System.out.println("Sum (unmatched bits): " + sumUnmatchedBits);

}
```

Figure 13

```
/********** Hamming Distance Calculator **********/
Enter 0 to exit
Enter original Key (16bytes): 43 F9 42 45 52 50 F8 59 53 49 43 41 4c 53 59 53
HexA(obtained): 43 F9 42 45 52 50 F8 59 53 49 43 41 4c 53 59 53
HexB(original): 43 59 42 45 52 50 48 59 53 49 43 41 4c 53 59 53
Distance      :00 20 00 00 00 00 30 00 00 00 00 00 00 00 00 00
Sum (unmatched bits): 5
Program exit ...
```

Figure 14

3. Experimental Results

In the CPA program, after each successful execution to recover key bytes, two CSV files (plot1Data.csv and plot2Data.csv) will be outputted. plot1Data.csv contains the highest correlation value of each possible 256 key bytes for all the 16 key bytes. It will be used as an input to the python script (plotCPA.py) to output Plot 1. plot2Data.csv contains the highest correlation value of each possible 256 key bytes for the increasing number of traces (10, 20, 30, ..., 100) for all the 16 key bytes. It will be used as an input to the python script (plotCPA.py) to output Plot 2. In Lab 1 and 2, the 16 bytes key used for encryption is “43 59 42 45 52 50 48 59 53 49 43 41 4c 53 59 53”.

3.1 Plot 1 – Correlation of Possible Key Bytes

In 100 traces, the correlation of all possible key bytes is plotted and the correlation of the correct key byte is highlighted in red. The correct key byte has the highest correlation value. Plot 1 for all the 16 bytes of the key is shown in Figure 15.

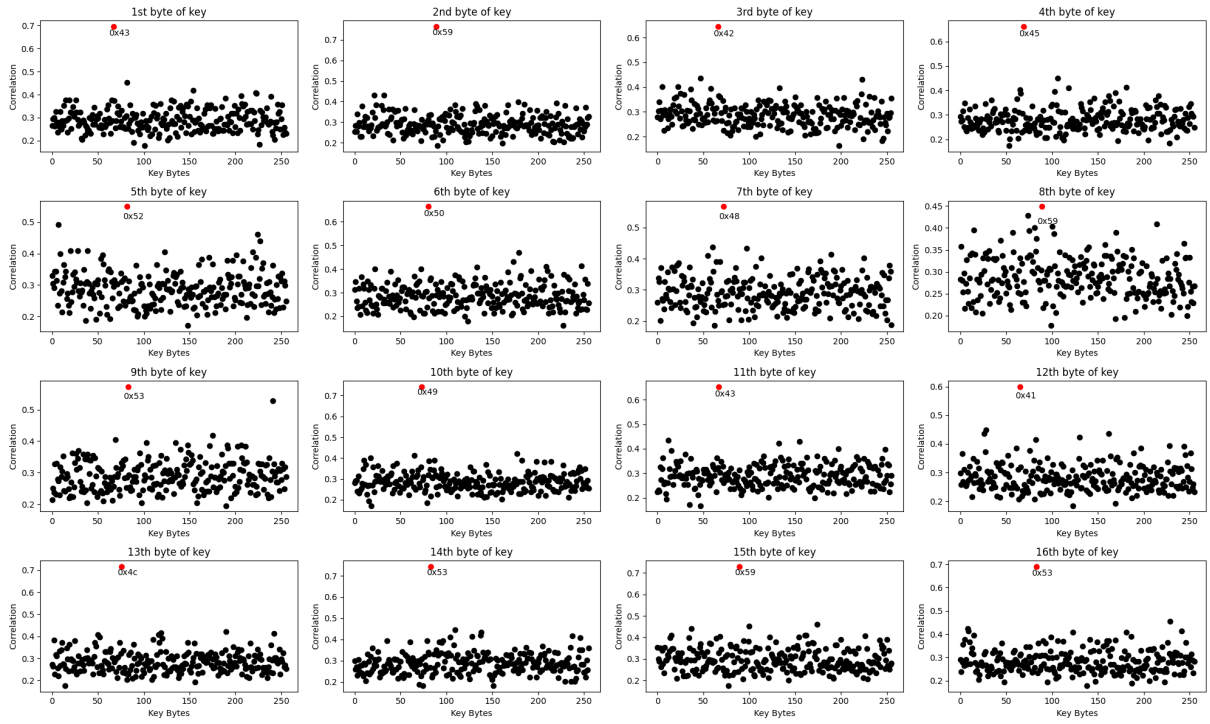

Figure 15

3.2 Plot 2 – Correlation of Correct Key Byte vs Number of Traces

CPA is implemented with the different number of traces from 10 to 100 in steps of 10. The highest correlation value for all 256 possible key bytes is plotted for the different number of traces. Generally, the correct key byte will start emerging as the number of traces increases. Plot 2 for all the 16 bytes of the key is shown in Figure 16.

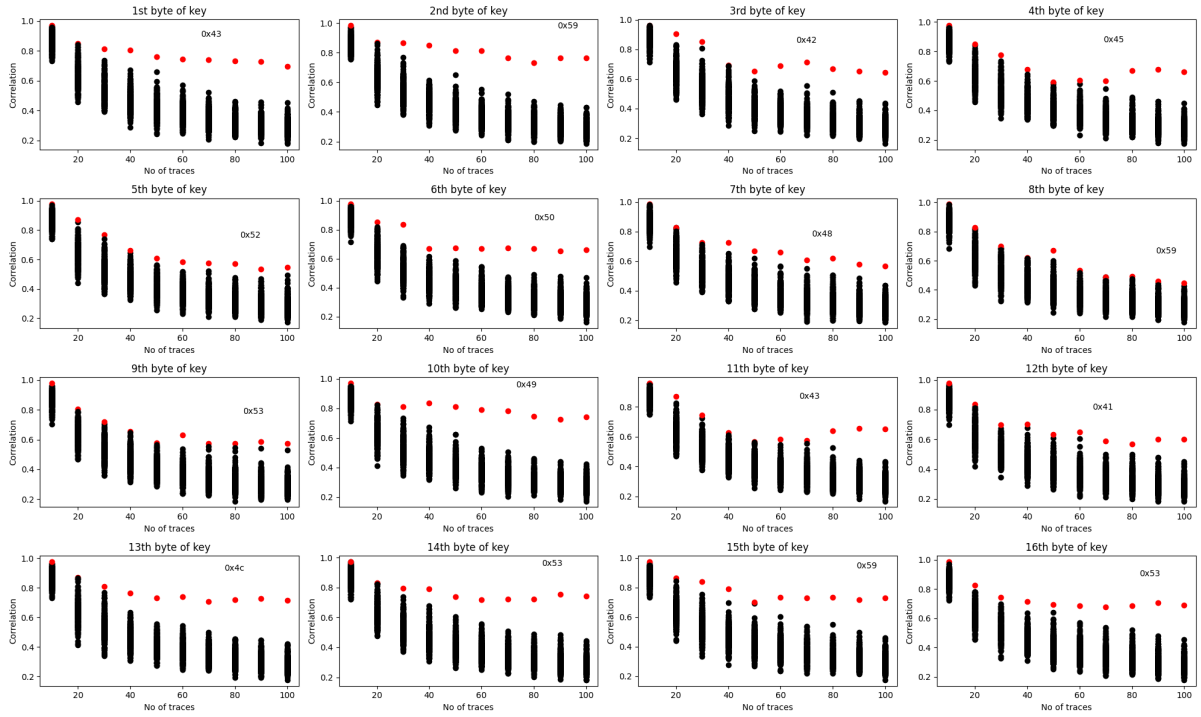

Figure 16

4. Countermeasures against Side-Channel Attacks

There are various software countermeasures against CPA. One of the software countermeasures is random instruction injection. NOP instructions can be inserted randomly during the encryption process, destroying the alignment of power traces which makes the attack difficult. However, NOP instructions will have a distinguishable power consumption pattern as it does not perform any operation, making it difficult to deter CPA attacks. Instead of inserting NOP instructions, real instructions can be inserted randomly, making the power consumption undistinguishable from other encryption codes. However, real instructions will require additional power consumption.

Another software countermeasure is randomly shuffling Sbox operations. Block cipher such as AES consists of 16 independent Sbox operations in each round and hence the sequence at which the Sbox operations are carried out can be randomised. When all the 16 Sboxes operations are randomised, the alignment of power traces is damaged, making it more difficult to perform CPA. This method is more efficient in terms of time and power than random instruction injection as no extra instruction is injected. Randomly shuffling Sboxes can also be introduced in random instruction injection as a means to provide extra security against side-channel with no extra time or power consumption.

Random instruction injection and randomly shuffling the Sboxes software countermeasures depends on the randomness. Random numbers are generated using pseudo-random generators (PRNG) in software. PRNG generates a sequence of random numbers based on an initialization value known as the seed. However, if the seed is static, the sequence of random numbers generated won't be truly random. A true hardware random generator can be used to provide the seed using natural phenomena which will be truly random.

Hardware countermeasures at the circuit level and microarchitectural level can be used against side-channel attacks. One of the circuit-level hardware countermeasures is introducing noise to the power line. With additional noise added to the device, the signal-to-noise ratio decreases, making the attack difficult. Another technique is filtering the power line. When filtering is applied, the power consumption patterns are attenuated. The skipping clock pulses technique can also be used. When the clock signal is passed through a filter which skips clock pulses randomly, the alignment of the power traces is destroyed, making the attack difficult.

One of the hardware countermeasures at the microarchitecture level is implementing dual-rail logic to achieve constant power consumption for every clock cycle. In this technique, CPA is not able to be performed as there will be no correlation between the power consumption of the device and the cryptographic operation. However, with constant power consumption in each operation, more power is consumed over time.

5. Conclusion

In this project, a Correlation Power Analysis (CPA) program is developed to implement CPA on the given set of traces obtained in Lab 1 and 2 to extract the key. With the aid of Hamming Distance Calculator in the CPA program, users can also check for correctness of the extracted key from the program by entering the correct key used for encryption. From the experimental results, one can infer that the accuracy of the recovered key increases with the increasing number of traces used to perform CPA on. However, computation times increase with the increasing size of power traces. With various countermeasures against side-channel attacks such as CPA, it will make it more difficult for attackers to perform attacks. More traces are needed for the attackers to acquire to perform CPA on which are time-consuming and costly.

6. References

[1] H. Gamaarachchi and H. Ganegoda, “Power Analysis Based Side Channel Attack,” University of Peradeniya, 2018.

[2] O. Lo, W. J. Buchanan 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,* vol. 1, no. 2, pp. 88-107, 2016.