SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

CE/CZ4055 Project Report

Side-Channel Attack Using Correlation Power Analysis

Team Members:

NG XIN YI (U1821432J)

LIM SHANG MEI (U1822757J)

OOI KOK YIH (U1921262E)

TAN ZHI YONG (U1922445A)

Academic Year 2021/2022

1.0 Introduction

Side-channel attack is a security exploit that uses indirect effects of the system or its hardware to acquire information from the program execution of a system, rather than directly targeting the program or its code. These attacks typically aimed to exfiltrate sensitive information such as the cryptographic key. There are several types of side-channel attacks namely, cache attack, timing attack, electromagnetic attack and power-monitoring attack. This project will be focusing on performing Correlation Power Analysis (CPA).

CPA attack allows an attacker to obtain the secret encryption key that is stored in the victim's device. Generally, CPA attack consists of 4 steps:

1. Modeling the victim's power consumption
2. Victims encrypt several different plaintext and record the victim's power consumption traces during each encryption
3. Attack the key one byte at a time
4. Put together all the subkey to get the full secret key

2.0 Implementation

This project's implementation was separated into two parts:

1. Correlation Power Analysis
2. Data visualization

Our team has decided to build the Correlation Power Analysis in Java since it is statically typed and compiled, as opposed to Python, which is dynamic and interpreted. To speed up the analysis, the implementation will be multithreaded.

However, because we will be using libraries for plotting and interpreting graphs, the data visualization will be done in Python.

2.1. Correlation Power Analysis

2.1.1 Trace Acquisition

The executable file of the SCA328p ctrl software will be used to create the traces from the Efflux SCA Evaluation Board. The Tektronix TDS2012C scope is linked to the Efflux SCA board for the hardware setup. The scope's Channel 1 probe is connected to Pin 1 of the Efflux SCA Evaluation Board's header J2. The pin is in charge of collecting the 8-bit MCU's power leakage output. On the Efflux SCA Evaluation Board, channel 2 of the probe is connected to pin PB1 of header J1. This pin serves as the scope's trigger.

On the SCA328p_ctrl software, the key used to generate the 100 traces is:

4C494D5348414E474D45493132333435

Figure xx shows the generated ciphertext using the key provided.

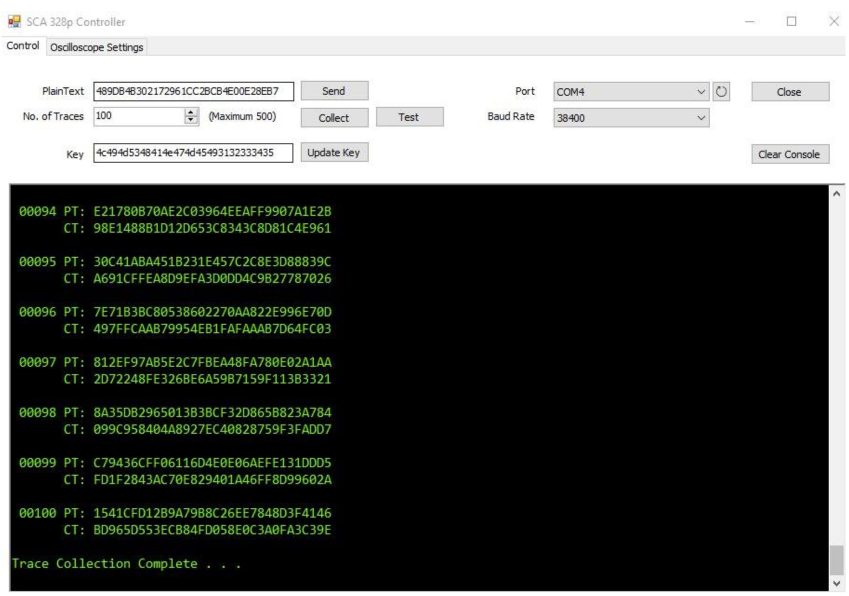


Figure xx

During the operation, we will be able to see a file being generated “waveform.csv” shown in figure xx. that will generate all the power consumption information. The first column will be the plain text while the second column will be the ciphertext and the rest of the values on the right of will be the actual trace values of the trace themselves

CF8CFBDB8CEA69C7FBFA55951CB0D612	B900368455086F7920006339274681A8	0.5080	0.4860	0.4540	0.4820	0.4400	0.4080	0.4480	0.5000	0.4580	0.4920	0.4460	0.3980	0.4480	0.4560	0.4140	0.4100	0.4180	0.3780	0.4000	0.5220	0.4760	0.4420
90C8D9BD7327D5BF526BCF8E010957B6	9FFF9AB9E6427EA443A67BFBA5749D2B	0.4920	0.5180	0.4800	0.4920	0.4440	0.4100	0.4560	0.5000	0.4960	0.5000	0.4840	0.4380	0.4400	0.4900	0.4440	0.4320	0.4540	0.4200	0.4020	0.5380	0.5120	0.4680
99CFBC6C244649FF3755AE7BAB2A5D90	DFECCB564235773181FB7741F4C81A52	0.5120	0.4780	0.4740	0.4580	0.4160	0.4220	0.4740	0.4980	0.4660	0.4740	0.4340	0.4000	0.4820	0.4620	0.4140	0.4600	0.4120	0.3840	0.4640	0.5080	0.4480	0.4820
5B7903D649ABA06EF89E6CBA8464289C	F26411DFF9433DA496209A3AB7E24A9C	0.5100	0.4960	0.4500	0.4900	0.4320	0.4000	0.4540	0.4960	0.4720	0.4940	0.4540	0.4060	0.4620	0.4600	0.4340	0.4400	0.4220	0.3860	0.4160	0.5300	0.4820	0.4600
6380E585F9CA15AEDD894BA72E852E76	E435BD7329A586F930CBC8C6A45E7655	0.4600	0.5100	0.4700	0.4960	0.4400	0.4100	0.4420	0.4860	0.4920	0.4800	0.4700	0.4260	0.4200	0.4720	0.4440	0.4160	0.4580	0.4100	0.3820	0.4960	0.5060	0.4580
F8F88354FDE481DA4D912C2F9F5A171D	BEBB851F32C0D941B8250E8498BD982F	0.5140	0.4780	0.4760	0.4580	0.4120	0.4140	0.4720	0.5020	0.4640	0.4860	0.4320	0.4120	0.4780	0.4660	0.4260	0.4580	0.4220	0.3860	0.4900	0.5140	0.4660	0.4880
01FF6603AD03F51A327C0A1B497B1DF7	3DEA091BE2F2995796E84912834F7D77	0.5080	0.4760	0.4760	0.4460	0.4040	0.4180	0.4680	0.4840	0.4440	0.4560	0.4280	0.3940	0.4780	0.4440	0.4080	0.4540	0.4240	0.3820	0.4640	0.5100	0.4540	0.4900
090648B25D22695A1766E908F39C23D1	B2E2DCEA378A18CEBD8B102CC77A9176	0.4860	0.4520	0.4320	0.4400	0.3900	0.3960	0.4400	0.4540	0.4220	0.4540	0.4100	0.3760	0.4520	0.4280	0.4100	0.4220	0.3920	0.3760	0.4520	0.5060	0.4500	0.4700
CA422695455ED4526ED76301D8F5A475	881A9007141325D45F7D6BE15D433A5F	0.4760	0.4920	0.4340	0.4840	0.4100	0.3960	0.4380	0.4540	0.4340	0.4480	0.4280	0.3720	0.4160	0.4480	0.3960	0.3800	0.4300	0.3740	0.3680	0.4920	0.4780	0.4440
18EFBF97809737375065DF96850309E6	2F5D3A7C36E40EE6F896C0967BFE6967	0.5260	0.5040	0.4680	0.4860	0.4520	0.4080	0.4660	0.5040	0.4820	0.5140	0.4600	0.4220	0.4640	0.4820	0.4300	0.4300	0.4340	0.3900	0.4140	0.5280	0.4880	0.4500
20F6A14630B6AC76354FBE822F240EC0	910A8AEC2022D04A33AF14F1D012E6E8	0.4660	0.5080	0.4580	0.4880	0.4400	0.4060	0.4380	0.4760	0.4680	0.4560	0.4540	0.4220	0.4240	0.4720	0.4360	0.4120	0.4480	0.4100	0.3820	0.5000	0.5060	0.4500
24B3E7036545EC6FF88D10D85390C85C	D188667B2F09EE4B39F0775E74FDD052	0.4860	0.4420	0.4460	0.4360	0.4020	0.3980	0.4560	0.4680	0.4420	0.4580	0.4260	0.3920	0.4580	0.4380	0.3940	0.4480	0.4020	0.3760	0.4660	0.5160	0.4520	0.4860
E5EEC5E64C8157674FFD8AD138E94900	C31700E0E028BE49B93939F2AE1266A1	0.4680	0.5120	0.4500	0.4960	0.4460	0.4080	0.4420	0.4820	0.4620	0.4540	0.4540	0.4060	0.4200	0.4620	0.4320	0.4020	0.4440	0.4060	0.3740	0.4940	0.4880	0.4520
339B5EE987BABA4C308B0766E5F7AE71	0C5E2716098C480875F0BC0BE3DB7EF6	0.4720	0.5200	0.4780	0.5060	0.4500	0.4180	0.4520	0.4820	0.5080	0.4920	0.5000	0.4360	0.4280	0.4920	0.4580	0.4280	0.4620	0.4260	0.3920	0.5140	0.5180	0.4740
3BA2419838D92F8C1576E5528F18B44B	A3429361EBA7DD0407705215B87D2ABF	0.5060	0.4960	0.4460	0.4820	0.4440	0.4000	0.4520	0.4840	0.4660	0.4900	0.4440	0.4000	0.4420	0.4580	0.4240	0.4160	0.4140	0.3960	0.3960	0.5300	0.4860	0.4540
D01BDF673CF39AB8857EC6DA00EE9DF2	3C10A15DD868C5C3C119AA4221B4AB78	0.5180	0.4980	0.4660	0.4840	0.4380	0.4040	0.4640	0.5080	0.4800	0.5100	0.4520	0.4220	0.4700	0.4640	0.4220	0.4480	0.4360	0.3940	0.4140	0.5360	0.4960	0.4680
4A8BB77D5A4EFD684E74DCE0207F9960	E75B14B5820C68B0107387BD7B03B219	0.5000	0.4680	0.4700	0.4540	0.4060	0.4100	0.4700	0.4880	0.4520	0.4580	0.4260	0.3940	0.4600	0.4380	0.4020	0.4520	0.4080	0.3780	0.4700	0.5100	0.4560	0.4880
9738517F9587604D30035875CD8DFED1	34B8F40CA9E1B6A103B32DEAF93ED10C	0.4920	0.4520	0.4680	0.4380	0.3920	0.4240	0.4600	0.4800	0.4500	0.4640	0.4160	0.4020	0.4780	0.4280	0.4020	0.4380	0.4160	0.3720	0.4780	0.4940	0.4500	0.4800
A03F332E46A5D48C15ED376177AE04AB	C400B0420A2907AF7F6BE78DA4343E0E	0.4960	0.4700	0.4680	0.4360	0.4080	0.4200	0.4620	0.4740	0.4460	0.4640	0.4280	0.3900	0.4680	0.4380	0.4140	0.4520	0.4160	0.3840	0.4640	0.4980	0.4560	0.4860
6A1049F52402A712B1B8CB8767721DC9	E836D67EC464D599DDB2E2C604C3020A	0.4960	0.5180	0.4600	0.4900	0.4400	0.4120	0.4540	0.5000	0.4860	0.4940	0.4520	0.4200	0.4360	0.4660	0.4380	0.4180	0.4360	0.3920	0.3960	0.5200	0.5000	0.4520
2B4B27D70C3E130A082945804CCB9F6D	13AD97C8F6D35D87141066D8C75661CC	0.5060	0.4780	0.4500	0.4580	0.4180	0.3980	0.4540	0.4580	0.4320	0.4620	0.4240	0.3760	0.4620	0.4500	0.4080	0.4620	0.4040	0.3800	0.4500	0.5140	0.4580	0.4880
3F56ABF1F1A6DB4E9B4781E0F436872A	1DB96172156A46536D3DED3A0FF872B9	0.4920	0.4640	0.4360	0.4520	0.4100	0.3880	0.4440	0.4840	0.4560	0.4820	0.4400	0.3960	0.4660	0.4500	0.4140	0.4520	0.4060	0.3720	0.4420	0.5160	0.4540	0.4780
8C0345F42CDE3E337DD5FE75A144EB9B	341F2E3909CF3CF9D5E6D10F12A51A6D	0.4660	0.5080	0.4600	0.4960	0.4560	0.4120	0.4400	0.4860	0.4840	0.4700	0.4620	0.4200	0.4140	0.4780	0.4460	0.4000	0.4420	0.4080	0.3780	0.4980	0.4920	0.4500
06731D094B39A0E346CB147BC1D6E809	8C44F67B76D957F0B73A609BC7EE3EB0	0.5280	0.4940	0.4620	0.4700	0.4300	0.3880	0.4760	0.4980	0.4500	0.5000	0.4460	0.4100	0.4580	0.4660	0.4180	0.4480	0.4300	0.3900	0.4340	0.5120	0.4760	0.4540
5320B60C867203C72859900F6DE44C7A	87848268E932623955F7DDFD9D4D3973	0.4600	0.4880	0.4500	0.4760	0.4380	0.4020	0.4440	0.4820	0.4840	0.4660	0.4560	0.4160	0.4240	0.4680	0.4240	0.4180	0.4360	0.4000	0.3860	0.5100	0.5040	0.4580
145C94EF6DAE6FBF7FCA0B08523DCD1E	7036486080648C89775597FE2E49ED61	0.5060	0.4720	0.4560	0.4500	0.4060	0.4120	0.4540	0.4580	0.4220	0.4540	0.4200	0.3860	0.4760	0.4360	0.4100	0.4500	0.4180	0.3760	0.4580	0.5200	0.4660	0.4900
62092DF1A8E7D2A46158879DFF4B328F	DFADFDE9F7E30A3599FCEECA6FCBDF90	0.5160	0.4740	0.4960	0.4560	0.4140	0.4360	0.4940	0.4860	0.4560	0.4660	0.4200	0.4140	0.4800	0.4520	0.4000	0.4580	0.4160	0.3860	0.4840	0.4960	0.4580	0.4880
F781CBC0AC023ED0D160682570211B36	1B0C53DE4231DA6A0550697812EBB0DE	0.4880	0.4640	0.4520	0.4460	0.3960	0.4040	0.4500	0.4720	0.4360	0.4620	0.4240	0.3920	0.4560	0.4380	0.4000	0.4460	0.4020	0.3780	0.4520	0.4960	0.4560	0.4840
40E4C8D16CAB6D6E90415723977A346A	E725A7506BC32F00A3D6E2A22CB366F0	0.4940	0.5100	0.4580	0.4840	0.4360	0.4120	0.4500	0.4820	0.4620	0.4740	0.4440	0.4000	0.4320	0.4480	0.4260	0.4140	0.4240	0.3900	0.3980	0.5080	0.4760	0.4400
0120A6B453E7D866E7B1D21C7CD3B50E	87B829B322C58C879932141E0E299A14	0.5060	0.4780	0.4380	0.4700	0.4200	0.4020	0.4440	0.4800	0.4500	0.4800	0.4340	0.3880	0.4500	0.4460	0.4120	0.4160	0.4140	0.3640	0.4140	0.5160	0.4680	0.4360
4ECD3FB68F203C4AC93F4EB129E11A7F	FE07E1F54D62074936F0448FD2BB752E	0.5240	0.4920	0.4620	0.4840	0.4360	0.4080	0.4620	0.4720	0.4440	0.4780	0.4300	0.3860	0.4600	0.4580	0.4220	0.4420	0.4200	0.3860	0.4280	0.5060	0.4580	0.4440
C83D18CCAD7B9EFA923664B6497216ED	B17EA9FD27864EF2E35B65B8CF7F778A	0.4700	0.5120	0.4740	0.4980	0.4340	0.4160	0.4360	0.4880	0.4900	0.4840	0.4720	0.4340	0.4380	0.4720	0.4460	0.4080	0.4640	0.4100	0.3920	0.4980	0.5060	0.4680
D044FA7B5D99123A772043A3F3931CC7	43B21BFA9366C254BC4FC8EB0CEEAEC0	0.5220	0.4760	0.4540	0.4620	0.4240	0.4080	0.4620	0.4900	0.4600	0.4920	0.4400	0.4060	0.4700	0.4640	0.4160	0.4580	0.4140	0.3820	0.4560	0.5180	0.4600	0.4640
66BC984A61B47E66E728232B6469056E	2EDC8729E525965288A2EBCAE8CD1C25	0.4600	0.4940	0.4580	0.4780	0.4440	0.4080	0.4340	0.4780	0.4640	0.4580	0.4660	0.4360	0.4040	0.4580	0.4420	0.4080	0.4460	0.4060	0.3860	0.4880	0.5040	0.4460
6979DE089643BF5FAA66758189D5BF0A	9682EF7C9FF7511C401FF67D8AB53EAA	0.4820	0.4840	0.4460	0.4800	0.4320	0.3860	0.4300	0.4620	0.4480	0.4580	0.4300	0.3880	0.4200	0.4520	0.4140	0.3860	0.4180	0.3760	0.3780	0.4860	0.4720	0.4220
2AB5BCEA7D7F2A5701D6F07A6E2E40AE	DEB3288F5E7DD7FE22AEC58D352DA0A5	0.4680	0.4740	0.4360	0.4600	0.4200	0.3940	0.4440	0.5000	0.4660	0.4880	0.4500	0.4160	0.4380	0.4400	0.4200	0.4120	0.4380	0.3800	0.3780	0.5140	0.4960	0.4440

Figure xx

2.1.2 Model Trace Matrix Initialization

A 2D Array with the size of [a][b] was used to create the model trace matrix.

a: the total number of traces in a sample

b: 2^8 = 256 is the number of potential values for one byte of the key.

```
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 xx:

2.1.3 Data Preparation

```
//variables
int marg = 60;
public float [][]powerMatrix;
public String[] plainText;
public float[][] hammingPowerMatrix;
public float[][] correlationMatrix;
private  int numPowerTracePoint;
private  int numPowerTrace;
public String fileName = "waveform.csv";
public String originalkey = "Key.txt";
private String[] key = new String[16];
public String[][] plot1 = new String[16][256];
public String[][] plot2 = new String[160][256];
public String[] keyByteList = new String[160];
```

Figure X shows the variables we created to store the data generated. We created arrays for power matrix, hamming power matrix, correlation matrix and also plot 1 and 2 to store the data for plotting of graph.

2.1.4 Compute Hamming weight

```
public void createHammingPowerMatrix(int num) {
    int sBoxValue;
    hammingPowerMatrix = new float[numPowerTrace][256];

    for(int i=0; i<numPowerTrace; i++)
    {
        int plainTextOneByte =  Integer.parseInt(plainText[i].substring(2*(num-1),2*num),16);

        for (int count = 0; count<256; count++)
        {
            sBoxValue = sBox[plainTextOneByte^count];
            hammingPowerMatrix[i][count]= Integer.bitCount(sBoxValue); //to get hamming weight
        }
    }
}
```

Figure X shows the code snippet to compute the hamming weight. We loop through the number of plain text (100 plain text in this case) and work 1 byte of the plain text at a time. The hex value of will be converted into integer. To compute the hamming weight of each key bye, we find the corresponding sbox value by xor-ing 1 byte of plaintext with every possible key byte from 0x00 to 0xFF (or 0-255 in decimal). After finding out the sbox value, we find the hamming weight by counting the number of ‘binary 1’ in the sbox value by using the bitCount() function.

2.1.5 Pearson's Correlation Coefficient

To find the correlation between the model trace matrix and the actual trace matrix, Pearson's Correlation Coefficient equation was utilized. The correlation ranges from -1 to 1, where 1 indicates a strong positive correlation while -1 indicates a strong negative correlation and 0 being no correlation at all.

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

Where,

- r = Pearson Coefficient
- n = size of array
- $\sum x$ = summation of elements in array X
- $\sum y$ = summation of elements in array Y
- $\sum x^2$ = summation of elements squared in array X
- $\sum y^2$= summation of elements squared in array Y
- $(\sum x)^2$ = summation of elements in array X squared
- $(\sum y)^2$ = summation of elements in array Y squared

Figure X shows the formula for Pearson's Correlation Coefficient.

```
public float pearsonCoeff(float[] x, float[] y, int n) {

    float sumX = 0, sumY = 0, sumXY = 0;
    float sqSumX = 0, sqSumY = 0;


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

    float correlation = (float) (n*sumXY - sumX*sumY)/ (float)(Math.sqrt((n*sqSumX - sumX*sumX) * (n*sqSumY - sumY*sumY)));
    return correlation;
}
```

Figure X shows the code for calculating the correlation using the Pearson's Correlation Coefficient equation.

```java
public void createCorrelationMatrix() {
    float powerMatrixColumn[] = new float[numPowerTrace];
    float hammingPowerMatrixColumn[] = new float[numPowerTrace];

    for (int i=0; i<correlationMatrix.length; i++)
    {
        //get hamming version of power matrix for each column
        //too many loop so this one do in function lol
        hammingPowerMatrixColumn = hammingPowerMatrixAtColumn(i);


        for(int k=0; k<numPowerTracePoint; k++)
        {
            //get actual power matrix for each column then calculate coefficient with hamming version column by column
            powerMatrixColumn = powerMatrixAtColumn(k);
            correlationMatrix[i][k] = pearsonCoeff(powerMatrixColumn,hammingPowerMatrixColumn, numPowerTrace);
        }
    }
}
```

Figure X is the code snippet to calculate the correlation matrix. We compute the correlation matrix by using the Pearson's correlation function above by passing in the actual power matrix for each column, and the hamming weight of the power matrix for each column and number of traces.

2.1.6 Finding correct key

```java
public String guessKey(int row) {
    float max = -1;
    int key = 0;
    //here we check one by one and get the highest correlation
    for (int i=0; i<256; i++) {
        float maxRowCor = -1;

        for (int j=0; j<numPowerTracePoint; j++)
        {
            if(correlationMatrix[i][j]>maxRowCor)
            {
                maxRowCor = correlationMatrix[i][j];
            }
            if(correlationMatrix[i][j]> max)
            {
                max = correlationMatrix[i][j];
                key = i;
            }
        }

        if(numPowerTrace == 100) {

            plot1[(row+1)/10-1][i] = maxRowCor+ "";
        }

        plot2[row][i] = maxRowCor + "";
    }

    keyByteList[row] = max+ "";

    return Integer.toHexString(key);
}
```

In this function, we find the highest correlation for all the traces and store the data in plot1 and plot2. The highest value from the correlation matrix are also stored inside the key byte list.

2.1.7 Number of bytes recovered

```java
//Calculate the number of bytes recovered as number of traces increase
int numberOfBytesRecovered[] = new int[16];;
StringBuilder b4 = new StringBuilder();
for(int i=0;i<10;i++) {
    for(int j=1;j<17;j++) {
        if(key[j-1].equalsIgnoreCase(Allkey[j][i])) {
            numberOfBytesRecovered[i] = numberOfBytesRecovered[i] + 1;
        }

    }
    b4.append(numberOfBytesRecovered[i]);
    b4.append("\n");
}
BufferedWriter w4=null;
try {
    w4 = new BufferedWriter(new FileWriter("numberbytesrecovered.csv"));
    w4.write(b4.toString());
    w4.close();
}
catch (IOException error) {
    error.printStackTrace();
    System.exit(0);
}
```

Figure X depicts the code to get number of correct key bytes recovered as the number of traces increase. We compare the correct key array with all the generated keys array, if there's a match with the key, the number of bytes count increase by 1. Next, we write the array of number of correct bytes recovered into a csv file for plotting later.

2.1.8 Running the Java file

```
Console
<terminated> CPAProject [Java Application] C:\Program Files\AdoptOpenJDK\jdk-11.0.8.10-openj9\bin\javaw.exe (Apr 15, 2022, 6:18:40 PM – 6:19:35 PM)
No of Power Traces: 100 No of Points in a Trace: 2500
The original key is: 4C 49 4D 53 48 41 4E 47 4D 45 49 31 32 33 34 35
The key recovered is: 4C 49 4D 53 48 41 4E 47 4D 45 49 31 32 33 34 35
```

Figure xx: Recovering secret key using Java file

By running the written java programme, we will be able to see from figure xx that the secret key originally generated is able to be recovered.

3.0 Results

3.1 Plot 1

The code snippet in Figure xx will be used to generate the correlation plot for plot 1.

```
for i in range(0, 16, 1): #byte 0 to 15
    plot.figure()
    plot.plot(plot1[i])  # plot1 correlation values for key from 0x00 to 0xFF for current byte
    plot.xlabel("Possible Key Bytes")
    plot.ylabel("Correlation Values")
    #plot.savefig(f'correlation_plot1_for_byte_{index//2}')

```

Figure xx: Code snippet for generating plot 1

Assuming you get a paste in order of the figure generated, each graph represents 1 byte of the key. The correlation plots for all 16 key bytes are shown in Figures XX reading vertically down. The key is 4C494D5348414E474D45493132333435,

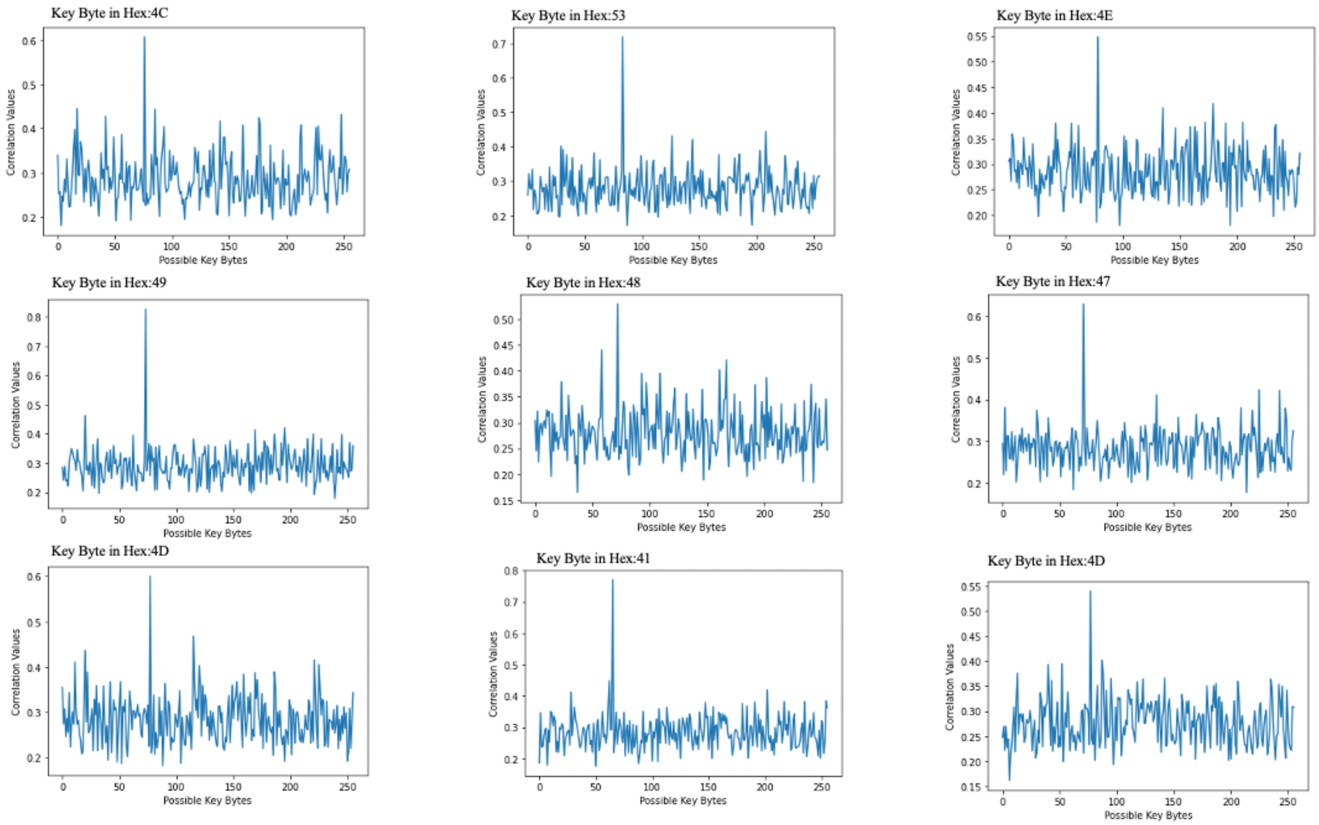


Figure xx: First 9 Key Bytes

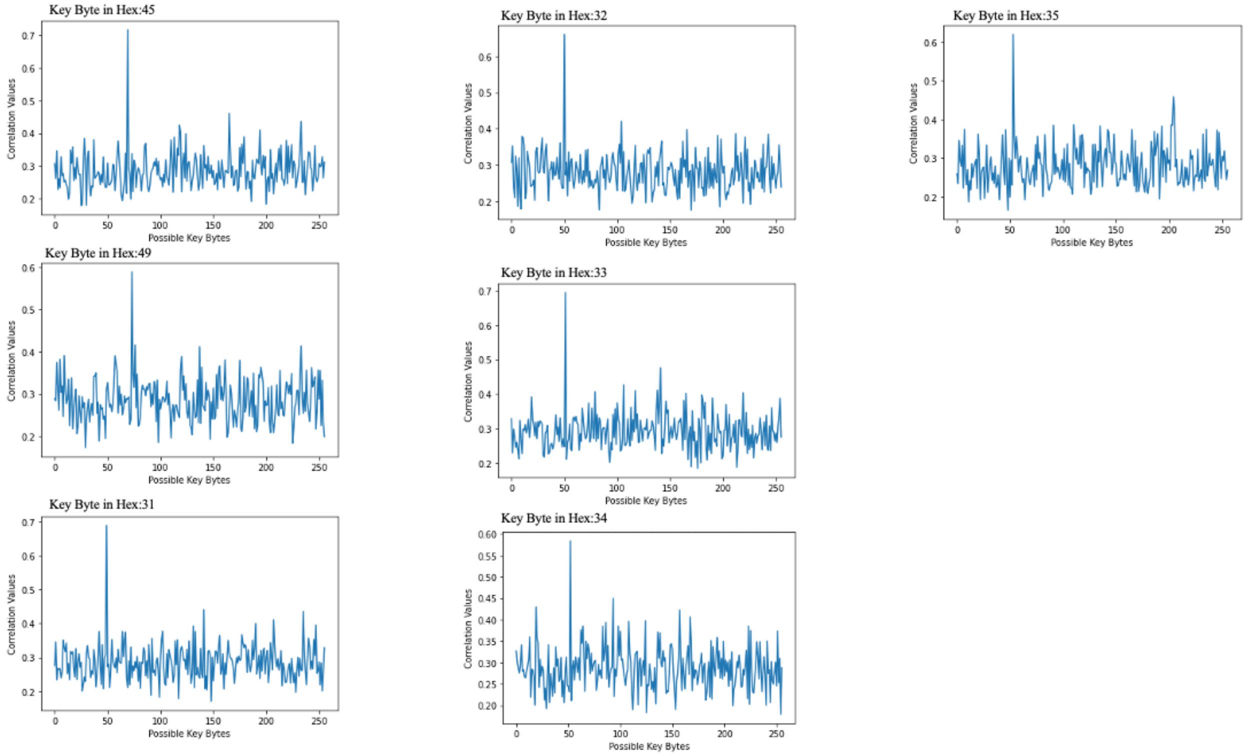


Figure xx: Last 7 Key Bytes

Figure XX and XX shows the correlation coefficient vs hypothesis for 100 traces.

3.2 Plot 2

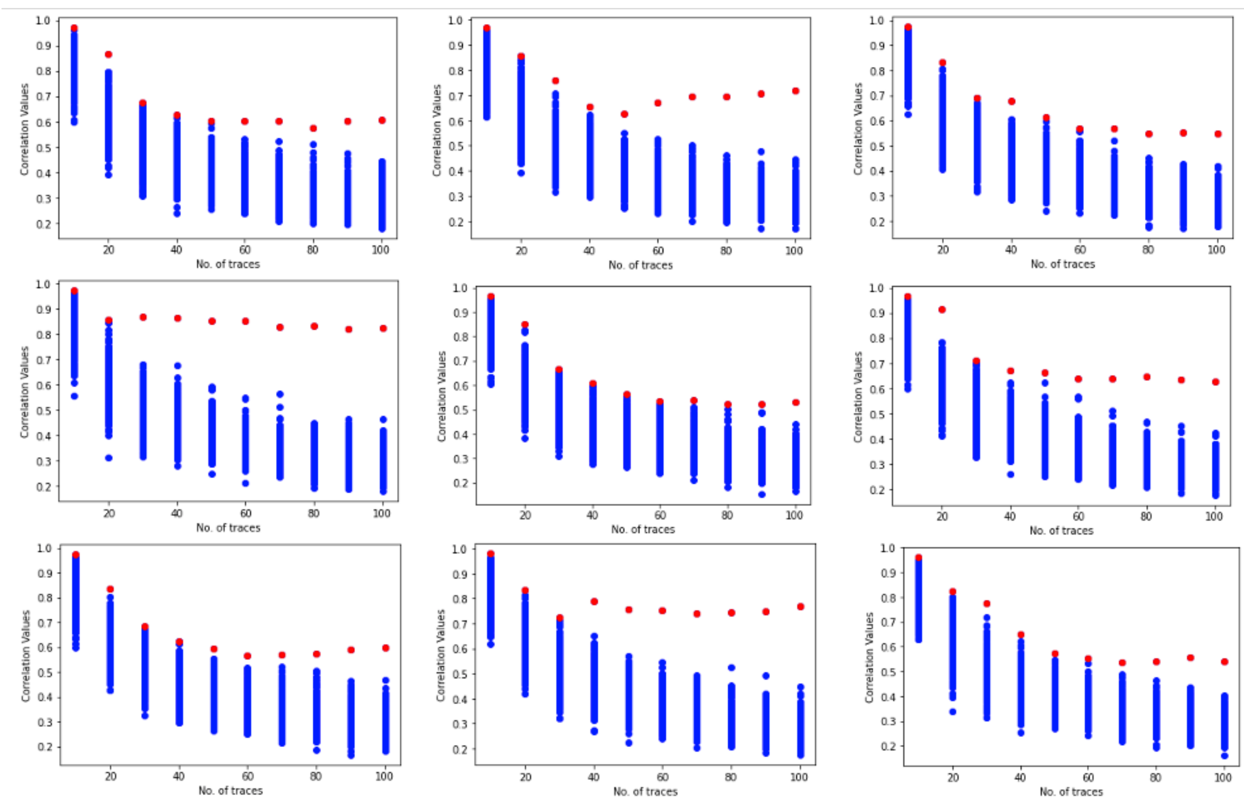
The steps for making correlation plot 2 are similar to those for making plot 1.

```
step_list = [i for i in range(10, 110, 10)]

l = 0
for j in range(0, 160, 10):
    plot2list = []
    for k in range(0,10,1):
        plot2list.append(plot2[j+k])
        # print(plot2list)

    plot.figure()
    plot.plot(step_list, plot2list, 'o', color='blue')  # plo
    plot.plot(step_list, keyByteList[l], 'o', color='red')  #
    plot.xlabel("No. of traces")
    plot.ylabel("Correlation Values")
    l+=1
```

Figure xx: Code snippet for generating plot 2



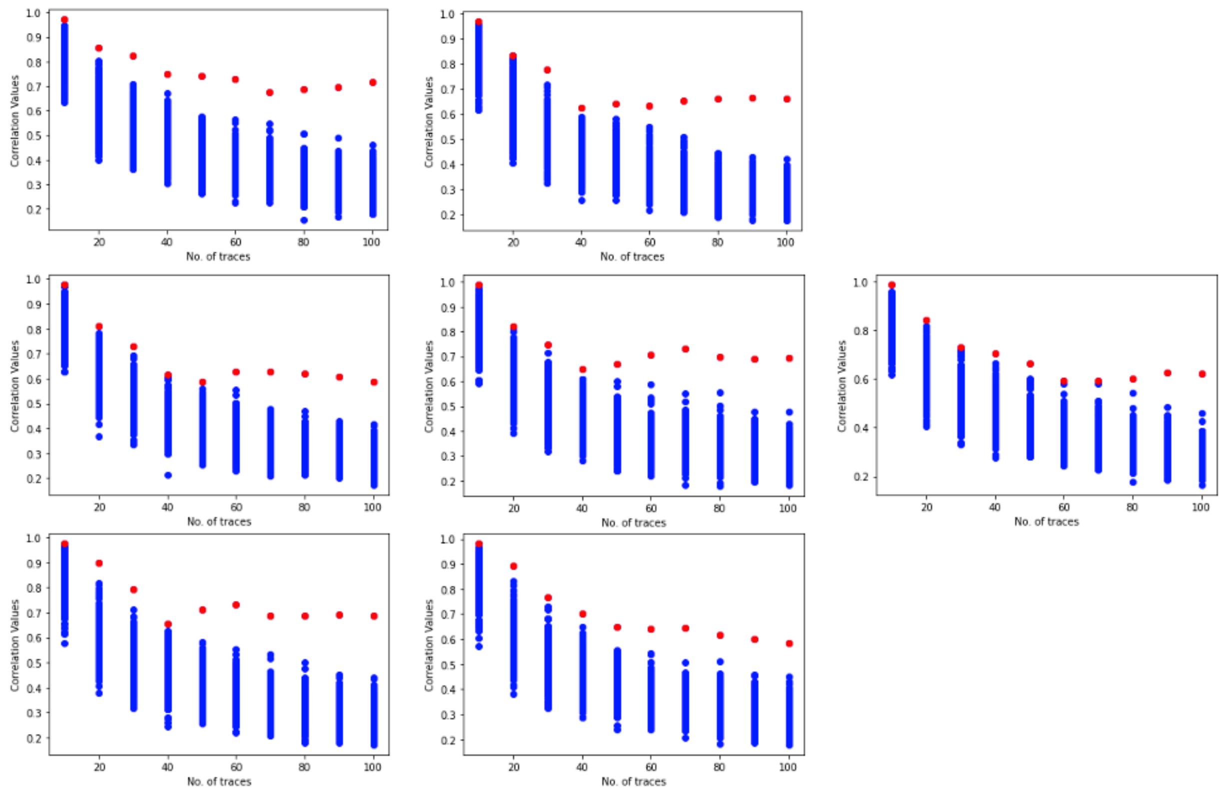


Figure xx: Correlation Plot with Variable Trace

Figure XX shows the correlation of correct key byte vs number of traces for all of the individual byte of the key. From the observation of the plots, the correlation coefficient eventually converge and the correct key byte starts to emerge as the number of traces increase.

3.3 Plot for Correct bytes recovered

```
with open("numberbytesrecovered.csv") as file_name:
    recoveredlist = np.loadtxt(file_name)
```

```
#Recovered bytes
step = [i for i in range(10, 110, 10)]
plot.figure()
plot.plot(step, recoveredlist)
plot.xlabel("Number of traces")
plot.ylabel("Number of correct bytes")
```

Figure X shows the code to plot the graph for the number of correct bytes against the number of traces. Firstly, we open the csv file generated earlier by running the java program. Next we plot the number of traces on the x-axis and the list of number of recovered correct bytes on the y-axis.

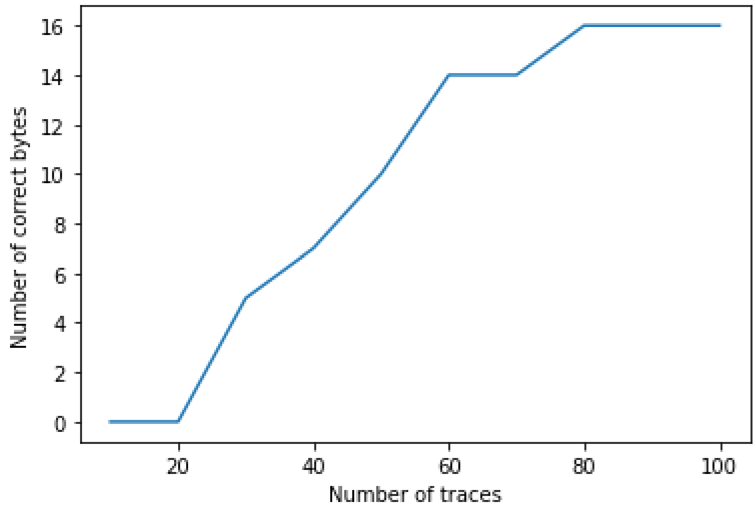


Figure X shows the plot for number of correct key bytes recovered against the number of traces. From the plot, we can see that as the number of traces increase, the number of correct bytes increase as well.

4.0 Countermeasures against Power Analysis Attack

The countermeasures against power analysis attacks are a series of strategies designed to make the power consumption of cryptographic devices independent of the data they process. Both hardware and software can be modified or improved to counter power analysis attacks. Hardware provides more variety of solutions to avoid leakage of information as compared to software countermeasures.

The following countermeasures can be split into 2 parts:

1. Hiding
2. Masking

Hiding Scheme

The hiding schemes randomly change the execution times of the operation to be attacked or the vertical height of the side-channel signals of the operation to be attacked. The goal of this scheme is to make the power consumption of cryptographic devices independent of the performed operations and the processed values.

Some techniques that the hiding scheme uses are random insertion of dummy operations and shuffling of instructions to ensure that in different executions the same operation does not happen at the same moment in time.

Masking Scheme

The masking schemes modify the intermediate value randomly by adding random values as masks so that the attacker cannot guess intermediate values. Masking prevents power analysis attacks because the randomly masked intermediate values cause a power consumption that is not predictable by the attacker. The goal of this scheme is to make the intermediate values that are processed by the device independent of the intermediate values of the algorithm.

5.0 Conclusion

Power analysis attacks are frequently undetectable by the affected device. Because the attack is non-invasive and the power consumption monitoring is passive. As a result, cryptosystem engineers are continually improving countermeasures to ensure that power analysis attacks can be prevented by making an attack difficult enough that the reward for breaking the system is less than the cost of doing so.