POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Evaluating a side-channel simulation tool against real power traces of cryptographic software

Supervisor

Prof. Danilo BAZZANELLA

Co-Supervisor

Matteo BOCCHI

Candidate

Matteo CATTANEO

APRIL 2023

Summary

Side-channel attacks are a type of attack that involves the physical outputs of embedded devices while cryptographic operations are running on them to recover some secrets. In this work, we refer to power analysis due to the exploitation of power consumption. By studying it and using suitable techniques, it is possible to recover the secret key used for the encryption.

The thesis aims to the comparison of the traces generated by an emulator and by a real device, evaluating if the emulator can be a valid alternative to the generation of real traces. The emulator is Rainbow while the real target is an ARM Cortex-M4 microcontroller with the help of the NewAE Technology ChipWhisperer tool.

The comparison is based on the use of two AES implementations, one in C language and one in assembly, considering, for each of the analyses and implementations adopted, the difference between simulated and real. The benefits and limitations of the emulator will be discussed highlighting some improvements and, in the end, some possible future works to continue what is done here.

Acknowledgements

Un ringraziamento a tutto il gruppo di Agrate per avermi accolto, in particolare Matteo Bocchi per il supporto costante durante tutto il periodo di svolgimento di questo lavoro. Un altro ringraziamento al piccolo ufficio di Torino dove ho passato parte di questi mesi

Matteo

Table of Contents

List of Tables

List of Figures

Acronyms

SPA
Simple Power Analysis

DPA
Differential Power Analysis

CPA
Correlation Power Analysis

HW
Hamming Weight

HD
Hamming Distance

SD
Switching Distance

AES
Advanced Encryption Standard

Chapter 1

Introduction

Side-channel attacks are a real threat to embedded devices running cryptographic software. They are techniques in charge of measuring the physical outputs of a device running cryptographic operations and to find a relationship between them and those operations. The physical outputs can be, for example, power consumption, heat or electromagnetic radiation. Particularly, power analysis takes advantage of the power consumption information of a device in order to retrieve secret data involved in cryptographic computations, for example the secret key. There are mainly three kinds of power analysis techniques: simple power analysis (SPA) which directly interprets the power traces, differential power analysis (DPA) which is a step forward and it exploits statistical methods to recover the secret key, correlation power analysis (CPA) that exploits a hypothetical power model emulating the real power consumption and then computing the correlation between it and the real traces.

Some simulation tools have been developed to help with the analysis of those vulnerabilities, simplifying the process with respect to running on real embedded targets that require special and expensive tools to acquire the traces. The thesis aims to analyse how much the simulated traces produced by the simulation tool, in this case Rainbow, are comparable to real power consumption traces, and how much using these simulators can help in finding and removing side-channel vulnerabilities. The real traces are acquired on the ARM Cortex-M microcontroller, thanks to the NewAE Technology ChipWhisperer tool.

The following steps will be addressed:

1. Study the basic techniques of SPA, DPA and CPA
2. Understand how Rainbow and the Chipwhisperer work, trying to identify their benefits and limitations
3. Compute power analyses on simulated and real traces looking for differences

4. Develop a framework allowing better usability with ST embedded targets

The cryptographic protocol that will be attacked and analyzed is AES-128 in two implementations, one in C language and one in assembly. The C implementation is scholastic and not a very optimized version while the assembly implementation is an optimized version, with performance and security features aligned with the state-of-the-art.

1.1 Thesis outline

- **Chapter 2:** Overview about the power analysis techniques, with a detailed description and figures of the three different types of analysis. There is also an explanation of AES and its steps exploited by the attacks.
- **Chapter 3:** For the simulated traces the tool utilized is Rainbow while for the real acquisition it is the Chipwhisperer-Pro kit. This chapter explains all the software and hardware needed to replicate these analyses but also an explanation of how they work.
- **Chapter 4:** This is the main chapter because it includes all the results of the attacks. They are illustrated using a lot of charts and tables. The first section explains the context of the analyses and how to implement DPA and CPA providing the script in pseudo-code. The second and third section regards two different AES implementations where, for each one, the attacks tested make use of both simulated and real traces.
- **Chapter 5:** This is the chapter designated to the explanation of some improvement and some hints. These suggestions can be applied to any analysis and they can avoid waste of time.
- **Chapter 6:** The conclusions and some starting points for future works are explained here.
- **Appendix:** All the appendices are scripts that help the reader during the lecture of the work, but they can also be useful to the reader who wants to replicate the experiments on their own.

Chapter 2

Side Channel Analysis

Nowadays, cryptography has an essential role in everyday life, almost every electronic device uses it. Research has demonstrated that there are often relationships between the device's physical output like power consumption, heat and sound emanated, electromagnetic radiation and the encryption taking place on the device. This field of study, which has spread in recent years, is called Side-channel Analysis. This type of analysis is in charge of monitoring the external outputs of a device when some cryptographic operations are running on it. By analysing this data and using specific algorithms, is possible to recover, for example, the encryption key. The fundamental and starting hypothesis is that the physical outputs of a cryptographic device are correlated with the internal state of a device running some cryptographic operations.

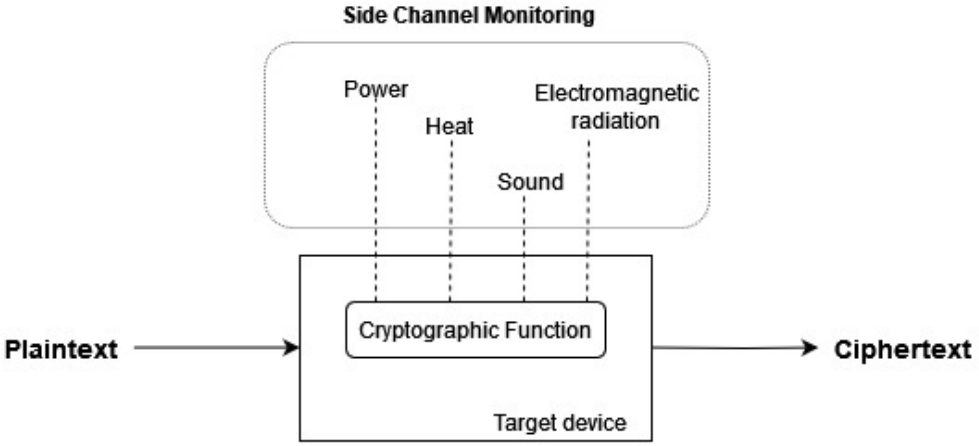

Figure 2.1: Side channel monitoring

2.1 Power Analysis

Power analysis is based on monitoring the power consumption on a cryptographic device [1]. Every electronic device manipulates data in terms of ones and zeros thanks to the transistors and the electric current. Adding or removing current from a transistor can change from one to zero or vice versa. The power consumption of a device reflects the data processed so it can reveal pieces of information about the processes running on it.

In this way, in the cryptographic field, when a cryptographic device is running some cryptographic operations, its data-dependent power consumption can expose secrets to attack. Common hardware used to monitor the device is an oscilloscope. Briefly, to measure the power consumption of a circuit, a known fixed stable resistor is introduced in series with the power input. Ohm's law $I = \frac{V}{R}$ says that the voltage is directly proportional to the current, so recovering the change in voltage in time is very easy. In the following sections, there will be an explanation of three different types of power analysis techniques.

2.1.1 Simple Power Analysis

Simple power analysis (SPA) is the most basic power analysis technique, as said by Kocher et al. «SPA is a technique that involves directly interpreting power consumption measurement collected during cryptographic operations» [2, p.2].

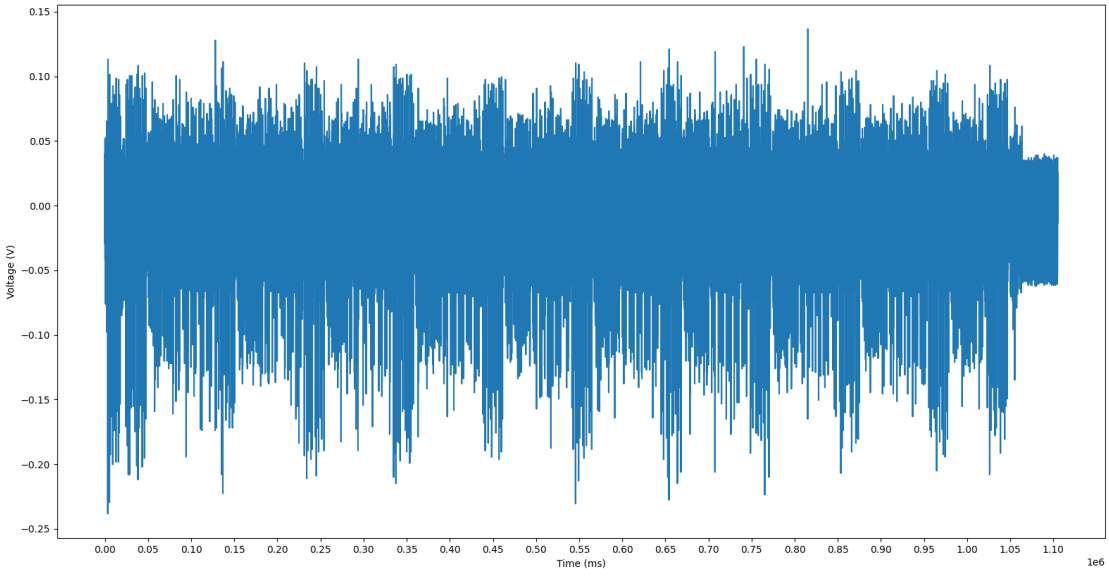

Figure 2.2: SPA trace showing AES-128 encryption operation

The basic element of the analysis is the *trace*, which represents the power

consumption measurement of a device performing a cryptographic operation. Figure 2.2 is an example: the AES-128 performs 10 rounds and they are clearly visible in the SPA trace. Different instructions involve variations in power consumption so, using SPA is difficult to recover the secret key but, only looking at the trace, can be very useful to understand the type of algorithm.

2.1.2 Differential Power Analysis

SPA is not easy to use to recover the secret key. There is often a lot of noise in the measurement and the analysis becomes more complicated. To reduce this fact there are some statistical methods, Differential power analysis (DPA) is based on these and permits to reach optimal results. DPA was first introduced by Kocher et al. in 1999 in an article entitled "Differential Power Analysis" [2].

The essential part to compute DPA is the selection function $D(C_i, K_n)$, it is necessary to group the collected traces in two sets related to the returned value of $D(C_i, K_n)$ (generally it is 0 or 1). The selection function has two parameters: C_i which is the known plaintext or the ciphertext relative to the ith trace, K_n which is the guessed key at the nth byte (for example, if the key is 16-byte length, n is between 0 and 15). The purpose of the DPA is to find when the hypothetical key K_n is correct, so there is some kind of relationship with the real power consumption.

The process of conducting a DPA attack should be resumed in the following steps:

- Observe m encryption operations and capture at the same time as many traces $T_{1..m}$, store also the plaintexts or the ciphertexts $C_{1..m}$.
- Choose a statistical method to compare the two groups (created using the selection function) and to decide, thanks to the statistic if they differ in some way. A classic statistical method is the mean: each group is reduced to only one trace that represents the average point-to-point.
- Once the statistical method has been applied, the two remaining traces are subtracted obtaining the final trace which is the one that will be analyzed looking for special patterns (peaks or nadirs).

$\Delta D[j]$ is the final trace at sample j. The equation 2.1 is the mathematical form of what was previously explained: the first fraction is the first set of traces while the second one is the other set.

$$\Delta D[j] = \frac{\sum_{i=1}^{m} D(C_i, K_n) T_i[j]}{\sum_{i=1}^{m} D(C_i, K_n)} - \frac{\sum_{i=1}^{m} (1 - D(C_i, K_n) T_i[j])}{\sum_{i=1}^{m} (1 - D(C_i, K_n))} \tag{2.1}$$

Now there are two possible ways:

- K_n is incorrect: the selection function has generated a subdivision of the traces that differs for about half from the correct target. Doing the mean and then the difference, the final result $\Delta D[j]$ should approach values around zero. There is no correlation between the subdivision of the selection function and the values processed by the device.

- K_n is correct: the selection function has generated a subdivision correct entirely, hence doing the mean and then the difference, the final result $\Delta D[j]$ should present spikes where there is a correlation with the values processed by the device.

Figure 2.3 shows the first 5000 samples of AES encryption and can be used as power reference while figure 2.4 shows three different results of DPA analysis. On top there is the trace of a correct key guess: a spike is visible approximately at sample 1500 and it is the proof of the correct guess. The lower two traces represent an incorrect key guess, a spike is again visible but it is significantly shorter.

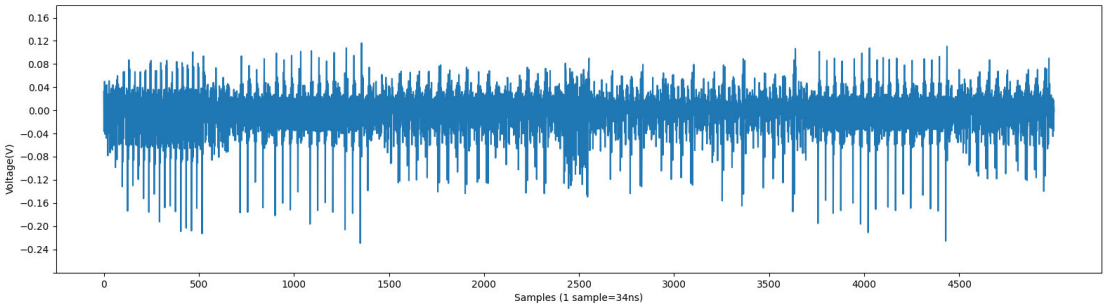

Figure 2.3: First 5000 samples of AES encryption

2.1.3 Correlation Power Analysis

Correlation Power Analysis was first introduced by Brier et al.[3] in 2004 and it is a further step forward the DPA. The main hypothesis for the CPA is that there is a correlation between the real measured power traces and the processed values by the device in time. This technique is based on a hypothetical power model, it must represent in the possible best way the power consumption of the cryptographic device under attack. As in DPA, analyzing the consumption, it should be possible to recover the secret key. An efficient and widespread way to compute the correlation between the measured traces and the power model is the Pearson correlation coefficient (eq. 2.2). This attack will make use of this function, looking for the highest correlation.

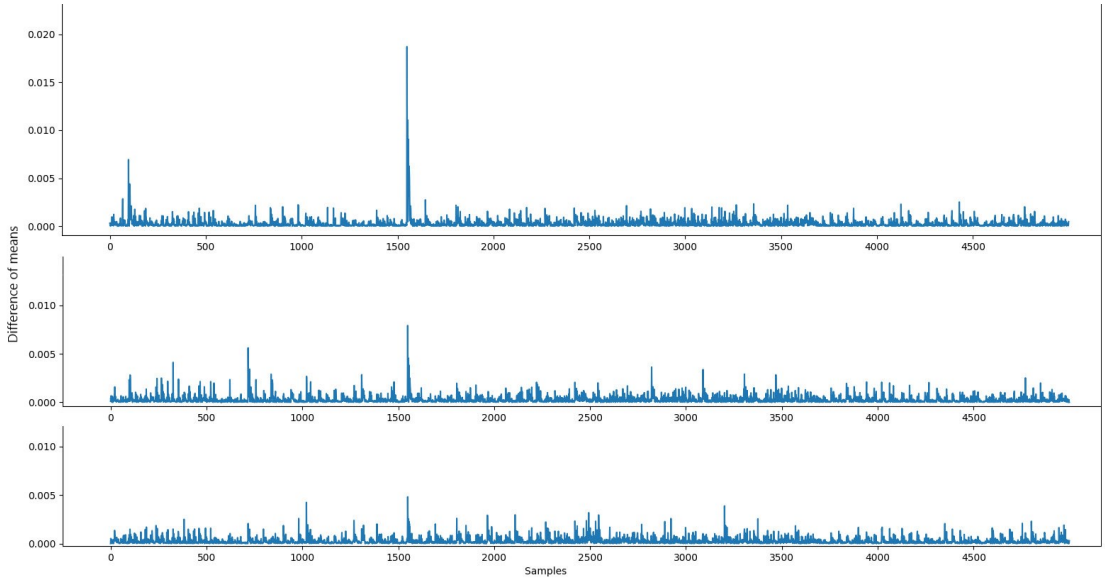

Figure 2.4: DPA traces, one correct and two incorrect

Given N known plaintexts or ciphertexts, P the predicted power calculated by the hypothetical power model and W the equivalent traces of real consumption, the Pearson correlation coefficient ρ is defined as:

$$\rho(W, P) = \frac{Cov(W, P)}{\sigma_W \sigma_P} \tag{2.2}$$

Where Cov is the covariance and σ is the standard deviation. The Pearson coefficient result always has a value between -1 and +1, when the absolute value of ρ close to 1 means there is a high correlation while close to 0 means there is no correlation.

The process of conducting a CPA attack is the following:

- Choose an intermediate point of the cryptographic algorithm that must depend on the known variable and the secret keys
- Measure the real power consumption of the device with an oscilloscope and store the traces
- Calculate the predicted power consumption using a hypothetical power consumption model
- Compute the correlation between the real power trace and the predicted one
- The value with the highest correlation coefficient will be, with a high probability, the correct key guess

2.1.4 CPA Power Consumption Model

The power consumption model has the task to predict the power consumption trying to be as much as possible close to the real. In this type of attack, the choice of a valuable power model has a direct impact on the performance of the attack. Scientific literature reports three power models: the Hamming Weight, the Hamming Distance and the Switching Distance. The Hamming Distance and the Switching Distance are quite similar, both are based on the relation between the power consumption and switching activity in CMOS devices, therefore the CMOS consumption is data-dependent [4, p.3]

Hamming Weight

The Hamming weight of a binary string is the number of bits equal to 1. For example in the string "00101101" the hamming weight is 4 and for "11110001" it is 5. If $D = \sum_{j=0}^{m-1} d_j 2^j$ is the m-bit binary data with d_j=0 or 1, the $HW(D) = \sum_{j=0}^{m-1} d_j$ will be the hamming weight.

The Hamming Weight model (HW) is the most basic consumption model, it is based on the fact that only a 1 involves a significant amount of power consumption while a 0 does not involve extra power consumption. In practice, using this model, the power consumption is proportional to the total number of bits set to 1 in the processed data.

Hamming Distance

The Hamming distance between two binary strings of equal length is the number of bits that change their value comparing the first string with the second one. For example, given the two strings "00101101" and "11110001", the hamming distance is 5 because there are five bits that change value. Given two m-bit binary string S_1 and S_2, the Hamming distance can be computed as $HD = HW(S_1 \oplus S_2)$.

The Hamming Distance model (HD) was proposed by Brier in [3], it is proportional to the number of transitions from 1 to 0 and vice versa. It is assumed that both transitions have the same amount of power consumption. Given D the data word and R its reference state, the power model is:

$$W = aHW(D \oplus R) + b \tag{2.3}$$

where a is a scaling factor between the Hamming distance and the power consumed (W), b is everything not related to the cryptographic operations. Note that if R is taken to be zero, this model collapses in the Hamming weight. Indeed, the Hamming distance model is a generalization of the Hamming weight

Switching Distance

The Switching distance model (SD) has been introduced in 2007 by Peeters et al. [5]. It is an evolution of the Hamming distance because it considers that the energy values required to flip the bits from one state to the other are different. The Switching distance of the transition 0→1 is assigned 1 but for the transition 1→0 is assigned Φ which is the Switching Distance factor.

Some studies have revealed that CPA attack with the Switching distance model performs better than other models and generally requires fewer traces to recover the entire secret key [4] [6].

The table 2.1 summarizes the behaviour of the three power models presented.

Transitions	HW	HD	SD
0→0	0	0	0
0→1	1	1	1
1→0	0	1	Φ
1→1	1	0	0

Table 2.1: Power Consumption Model of CMOS Transition

2.2 Advanced Encryption Standard

Advanced Encryption Standard "or simply AES" is an encryption standard. It is a symmetric block cipher that processes a data block of 128 bits using keys of 128, 192 or 256 bits length [7]. In general, the key used is specified in the name, for example with the key of 256 bits the algorithm's name will be AES-256. The blocks are called state and they are 4x4 column-major order matrix where each value is one byte. The AES is a round-based encryption algorithm, the number of rounds Nr can be 10, 12 or 14 when the key length is 128, 192 or 256 bits, respectively. Before the encryption, the cipher key needs to be expanded and it is called round key, the routine named Key Expansion is in charge of doing that. The encryption phase performs, in each round except the final one, four functions: AddRoundKey, SubBytes, ShiftRows and MixColumns, while the final round does not have the MixColumns transformation. The generic execution flow is shown in figure 2.5.

In the continuation of this work, AES-128 is used, but the theory may also be valid for other implementations (AES-192 and AES-256). AddRoundKey and SubBytes steps are the only relevant functions for this work and the following part describes how they work.

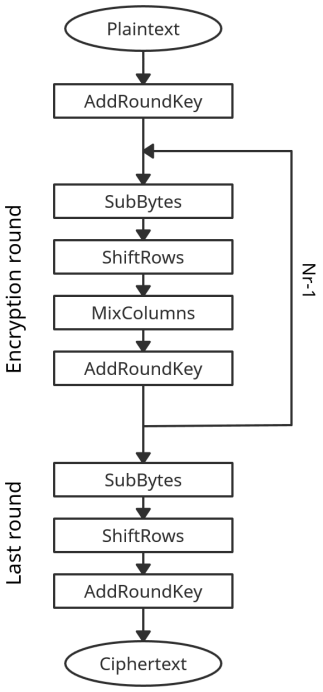

Figure 2.5: AES algorithm scheme

2.2.1 AddRoundKey and SubBytes steps

The first function executed is AddRoundKey, then the algorithm moves into the first round and performs SubBytes step (see figure 2.5). The algorithm takes two input parameters, the plaintext (the data to be encrypted) and the round key (the expanded cipher key).

During the first step, the AddRoundKey step, each plaintext value is XOR'd with a round key value at the same position in the block. The equation 2.4 shows, more simply, how it works, where P_i is the plaintext and K_i the round key at ith position.

$$[P_0 \oplus K_0, P_1 \oplus K_1, P_2 \oplus K_2 \dots P_{15} \oplus K_{15}] \tag{2.4}$$

$$[P_i \oplus K_i] \quad \text{i=0}\dots 15$$

Now, the SubBytes step takes the result of the previous step and performs a lookup for a value stored in the S-box. In other words, the input of this step is $P_i \oplus K_i$, and it is used for a lookup in the S-box that returns one byte value. The equation 2.5 represents the operation where S is the S-box lookup and O_i is the output.

$$O_i = S[P_i \oplus K_i] \quad \text{i=0...15} \tag{2.5}$$

The S-box is a 16x16 matrix of one byte values that remains constant for every AES implementation. Its purpose is to mix data and ensure the property of confusion, making the algorithm difficult to break. The S-box used in the AES algorithm is in figure 2.6.

Since $P_i \oplus K_i$ is one byte, the lookup is performed taking the first 4 bits as the column and the others as the row. For example if $P = C3$ and $K = 67$, $P_i \oplus K_i$ will be $A4$ then the S-box lookup will return 49. The following part provides a complete example of the execution of AddRoundKey and SubBytes steps using plaintext P=[01, 02, 03, 04, 08, 07, 06, 05, 09, 09, 0B, 0B, 06, 05, 06, 06] and cipher key K=[C0, C1, C2, C3, C4, C5, C6, C7, CA, CB, CD, CE CC, CC, CE, CF]. The figures 2.7 and 2.8 shown, respectively, the AddRoundKey and SubBytes function executed on an entire block.

The analysis that will be discussed exploits the monitoring of power consumption at the point the S-box lookup is completed, hence this point is at the end of the SubBytes function for each byte.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	1	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	4	C7	23	C3	18	96	5	9A	7	12	80	E2	EB	27	B2	75
4	9	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	0	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	2	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	6	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	8
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	3	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 2.6: S-box

01	08	09	06	⊕	C0	C4	CA	CC	=	C1	CC	C3	CA
02	07	09	05		C1	C5	CB	CC		C3	C2	C2	C9
03	06	0B	06		C2	C6	CD	CE		C1	C0	C6	C8
04	05	0B	04		C3	C7	CE	CF		C7	C2	C5	CB

Figure 2.7: AddRoundKey step

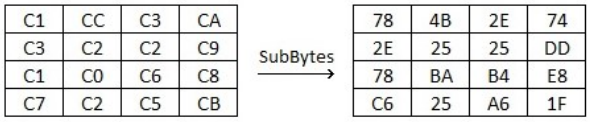

Figure 2.8: SubBytes step

Chapter 3

Trace acquisition setup

As mentioned in the introduction, the purpose of this work is to evaluate a side-channel simulation tool against real power traces. It is clear that for the same AES-128 implementation there will be two acquisitions of traces, one for the real and one for the simulated. Both traces will be analysed using the same procedures and algorithms. This section explains how the acquisition environment was set and how to replicate the experiments in other places, also the code will be provided, resulting in a better understanding. For some aspects, the Lo et al. article [8] does a similar work for the real traces acquisition but on a different board.

3.1 Hardware and software

The hardware used for the analyses is the following:

- **CW 1200 ChipWhisperer-Pro**: The capture board controlled by PC using python. Its role can be compared with a classic oscilloscope to gather power traces. It can perform power analysis synchronous to the target's clock helping the attacks succeed at much lower sampling frequencies than a conventional oscilloscope [9]
- **CW 308 UFO**: The base board is compatible with a lot of embedded target boards [10]
- **CW308T-STM32F4**: The target board, equipped with a Arm Cortex-M4 running the AES-128 algorithm [11]
- **Laptop**: Generic laptop to interface with the CW-Pro, save and elaborate traces

The software used for the analyses is the following:

- **ChipWhisperer API**: The python library needed to control the ChipWhisperer's boards from the laptop [12]

- **Rainbow**: The open-source side-channel simulation tool [13]. It is written in python and based on Unicorn Engine that is a lightweight multi-platform, multi-architecture CPU emulator framework [14]

- **Lascar**: Open source python library designed to help with side-channel analysis. It will be used to create an easy way to acquire and compare the two kinds of traces [15]

- **Text editor**: Generic text editor to write the scripts for the attacks

Figure 3.1: CW 1200 ChipWhisperer-Pro

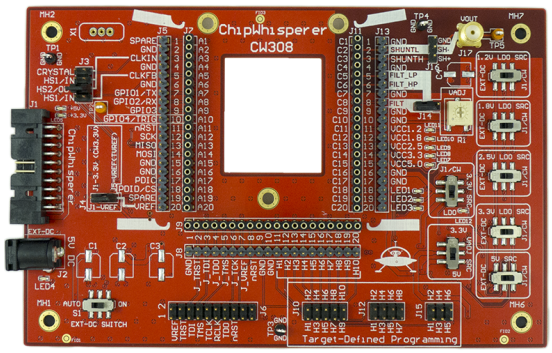

Figure 3.2: CW 308 UFO

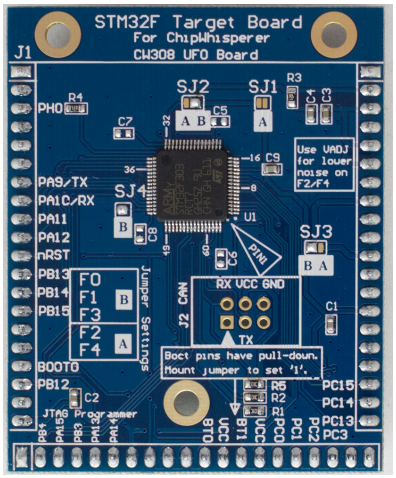

Figure 3.3: CW308T-STM32F

3.2 ChipWhisperer

ChipWhisperer is a collection of many tools useful for embedded security research, including side-channel power analysis and fault injection but this work will focus only on the first one. It includes all the tools and no external tools are needed.

There are mainly two types of hardware devices: the capture board and the target board. The capture board is the device in charge of being the channel of communication between the target board and the user, it performs the capture of traces and can be seen as a sort of oscilloscope. In our case it is the CW-Pro (fig. 3.1). The target board, instead, is the device under test (DUT) where specific algorithms run on it. In our case there are two boards: the CW 308 UFO (fig. 3.2) that stands as a baseboard and another board, the CW308T-STM32F4 (fig. 3.3), which will be connected to the baseboard.

3.2.1 Communication and firmware

The communication between the target and the capture board is done through the SimpleSerial protocol and it is always initiated by the capture board. This board is controlled using Python while the firmware of the target board is written in C and there are specific C functions to use. All the standard functions used in the communication are provided in the following list:

- *simpleserial_write(cmd, data)*: The function to start the communication, it is called from the capture side. *Cmd* is a char that says which command to use and *data* is the bytearray to send to the target board.

- *simpleserial_read(cmd, paylen)*: The function at the end of the communication to receive data sent from the target, it is called from the capture side. *Cmd* is the command (generally 'r') and *paylen* is the expected byte length of the received data
- *simpleserial_addcmd(cmd, len, func)*: C function declared in the firmware of the target. It adds a listener to the target for a specific command. *Cmd* is the letter for identifying the command, *len* is an integer and it is the amount of data bytes expected and *func* is the handler related to cmd
- *simpleserial_put(cmd, len, data)*: Writes data to the serial port which should send a packet from the target board to the capture board. *Cmd* is the command for the capture board, *len* is an integer and it is the size of the data buffer and *data* is the data buffer
- *simpleserial_get()*: It is used at the end of the code (in the target firmware) to keep checking if a known command is sent. When a packet from the capture board is found relevant, the appropriate callback function is called

Listing 3.1: Capture function example

```
#define number of traces
N = 100
for i in range(N):

    #send plaintext to the target
    target.simpleserial_write('p', plaintext)

    # ...snip (capture traces)

    #receive the result back
    response = target.simpleserial_read('r', 16)
```

Listing 3.2: Target firmware example

```
#include "simpleserial.h"

uint8_t encrypt_plaintext(uint8_t* plaintext, uint8_t data_len)
{
    trigger_high();
    // ...snip (do the encryption of plaintext).
    trigger_low();

    // Send the result back to the capture board.
    simpleserial_put('r', 16, result_buffer);

    return 0;
```

```
13 }
14
15 // Add a listener
16 simpleserial_addcmd('p', 16, encrypt_plaintext);
17
18 // Keep check if a command was send fitting one of the listeners.
19 while(1)
20     simpleserial_get();
```

The two codes above are a simple example to understand how the communication between the boards works. In practice, the target board encrypts the text sent from the capture board (the user) and returns the respective ciphertext. The code 3.2 is the target firmware.

Let's make an example using these scripts and trying to follow the communication flow. First, we want to encrypt some plaintext so, from the laptop, we send the plaintext associated with the command p to the target(line 6, 3.1). Then, thanks to *simpleserial_get* in the target firmware (line 20, 3.2), the command p is received and an associated function can handle the data. The *encrypt_plaintext* function will execute the encryption and returns the ciphertext to the user (line 10, 3.2). Now, the user with *simpleserial_read* (line 11, 3.1) can obtain the ciphertext relative to the data sent at the beginning.

3.2.2 Data capture workflow

As written in section 3.1, what is needed for capturing real traces is a laptop and the three boards included in the Chipwhisperer-Pro kit. This section provides a step-by-step description of the workflow:

1. The target is programmed to run AES-128 encryption when receiving some data. From the capture board 16 random bytes are sent through the serial communication and the target calculates the ciphertext. Two functions are used in the target firmware to delimit trace capture: *trigger_high()* and *trigger_low()*. The first function signals the point to start the acquisition that runs until a specific number of samples are collected (*scope.adc.samples*). *Trigger_low()* is used to set the trigger pin low so it can be ready for another capture

2. The capture board is set to start capturing at *trigger_high()* and, as default setup, it acquires 4 samples per clock cycle. What the board measure is the voltage at the ends of a shunt resistor so the unit of the points is Volt

3. The previous two steps are repeated as many times as there are traces to be collected. The traces and the random plaintext are saved into the disk for later analyses

4. After having gathered and saved the traces and the plaintext, some kind of offline analysis can then take place

Appendix A and B.1 provide real and working scripts as examples for a better understanding, moreover, there is a better explanation of the functions not explained in this section. For more details refer to the provided links or the chipwhisperer whitepaper [16].

3.3 Rainbow

Unicorn is a lightweight multi-platform, multi-architecture CPU emulator framework. It is implemented in pure C language with a lot of bindings for other languages and the performance is very good.

Rainbow is built on Unicorn and written in Python, it aims to provide an easy scripting interface to emulate embedded binaries and trace them to perform side-channel attacks. It has been created to have an easy and fast way to check the presence of side-channel vulnerabilities in the code and help the developers. Rainbow only produces an execution trace, without applying any processing on the values. This is left as some post-processing so that the user can apply its own leakage model and simulate various conditions from the same traces.

3.3.1 Basic usage

The basic usage is very intuitive, there are mainly some compulsory functions and others depending on the final purposes. For this work the following list contains the essential ones:

- *e=rainbow_arm(sca_mode=True, sca_HD=True):* Compulsory, defines the architecture to emulate, others architecture are available. *sca_mode=True* for HW power model, *sca_HD=True* for HD power model, there is an appropriate explanation of the Rainbow power model in section 3.3.2

- *e.load('firmware.elf', typ='elf'):* Compulsory, loading the binary to emulate, the loader supports elf, hex and bin extensions

- *e.start(start_address, stop_address, count=number_of_instructions):* Compulsory, starts the emulation from the first address and stops it at the second address or when the count is reached

- *e.start(functions['function_name'], stop_address, count=number_of_ _instructions):* Another way to start the emulation but this time specifying the function's name instead of the address

- *e[address] = value:* Write *value* in memory at *address*
- *e['reg_name'] = value:* Write *value* in the specific register
- *value = e[addr_start:addr_end]:* Read memory from *addr_start* to *addr_end* and put what read in *value*
- *e.sca_values_trace:* The array containing the points of the simulated trace. Each point corresponds to an instruction

Suppose we have the following little program (3.3) that encrypts the text and we want to emulate it. The first step is to compile the script with an appropriate compiler which depends on the architecture to simulate. We are using arm indeed the correct compiler is *arm-none-eabi-gcc* with the flags *-mthumb -mcpu=cortex-m4 -mfloat-abi=soft –specs=nosys.specs -Os*. Once we have generated the elf binary we are ready to set the Rainbow emulation. The binary contains only one function which is the one we want to emulate and requires one parameter, the plaintext.

Listing 3.3: Basic Rainbow firmware example

```
1 int main(){
2
3     //16 bytes of plaintext
4     uint8_t input[]={"aabbccddeeffgghh"};
5
6     encrypt_plaintext(input);
7
8     return 0;
9 }
```

The first two lines in the emulation script are the declaration of the architecture and the loading of the binary. Then, there is the setup of the function parameters. In arm architecture, the first four parameters are passed in the first four registers (r0, r1, r2, r3) therefore, in this case, the only parameter is passed in register $r0$. Lines 4, 5 and 6 do the following: an address is assigned to the variable *buf_in*, at the address pointed by *buf_in* is written the plaintext and finally, *buf_in* is assigned to $r0$. Now the emulation can start specifying either the function's name or the exact address and collecting the trace in *sca_values_trace*. The script 3.4 is the python form of what just described above.

Listing 3.4: Basic Rainbow emulation script

```
1 e = rainbow_arm(sca_mode=True)
2 e.load("firmware.elf")
3
4 buf_in = 0xCAFE1000
5 e[buf_in] = b"abcdefabcdef1234"
6 e["r0"] = buf_in
```

```
7
8 e.start(e.functions["encrypt_plaintext"], 0)
9
10 trace= e.sca_values_trace
```

3.3.2 Power models

Rainbow, by default, has two power models, one based on the hamming weight and one on the hamming distance. For each instruction, both models sum the HW or the HD of all written registers. For example, if we choose the HW power model, each point will be the sum of the HW of all written registers by that instruction. Let's see a practical example with some values.

Listing 3.5: Rainbow HW power model example

```
1 8000242:    c9f0         ldmia    r1!, {r4, r5, r6, r7}
2 8000244:    e8bc 000f    ldmia.w  ip!, {r0, r1, r2, r3}
3 8000248:    4044         eors     r4, r0
4 800024a:    404d         eors     r5, r1
5
6 0x8000244 [R12, R0, R1, R2, R3] HW_sum=74
7 r0=73736170 r1=64726f77 r2=73736170 r3=64726f77 r4=aa42f3f5 r5=7
    e1c9b60 r6=d4bc705c r7=b3e5e9d4 r12=90001010 r13=affffd4 r14
    =20000000 r15=08000248
8
9 0x8000248 [R4] HW_sum=14
10 r0=73736170 r1=64726f77 r2=73736170 r3=64726f77 r4=d9319285 r5=7
    e1c9b60 r6=d4bc705c r7=b3e5e9d4 r12=90001010 r13=affffd4 r14
    =20000000 r15=0800024a
```

The first four lines are instructions as example, line 6 contains the written registers by the instruction 0x8000244 and line 7 contains the values of the registers for the same instruction after it has been executed. Lines 9 and 10 are relative to the instruction 0x8000248. 0x8000244 writes five registers so the HW sum is 74 because it corresponds to the sum of the HW of each of the five registers. The next instruction writes only one register and the HW sum is 14 which corresponds to the HW of the value in r4.

The HD power model works in the same way but, instead of the HW, it calculates the HD. For this model there is an array that contains the last value of each register.

Listing 3.6: Rainbow HD power model example

```
1 8000242:    c9f0         ldmia    r1!, {r4, r5, r6, r7}
2 8000244:    e8bc 000f    ldmia.w  ip!, {r0, r1, r2, r3}
3 8000248:    4044         eors     r4, r0
4 800024a:    404d         eors     r5, r1
5
```

```
6  Last value: r12=90001000 r0=0 r1=90002010 r2=90003000 r3=900010a0
7  0x8000244 [R12, R0, R1, R2, R3] HD_sum=74
8  r0=73736170 r1=64726f77 r2=73736170 r3=64726f77 r4=8b4ad833 r5=6514
     e7f1 r6=2f65f7fd r7=c3d8d079 r12=90001010 r13=affffd4 r14
     =20000000 r15=08000248
9
10 Last value: r4=8b4ad833
11 0x8000248 [R4] HD_sum=16
12 r0=73736170 r1=64726f77 r2=73736170 r3=64726f77 r4=f839b943 r5=6514
     e7f1 r6=2f65f7fd r7=c3d8d079 r12=90001010 r13=affffd4 r14
     =20000000 r15=0800024a
```

Comparison between the power models

Since there are two power models, we need to decide which one to adopt for the analyses. This little section shows a light comparison between the two models and explains why we have chosen the HW power model. The implementation of AES used is the Cortexm-AES but it will be described in detail in the next chapter, for the moment we don't need further information about it.

The first idea to compare the models is to do the same analysis with the same number of traces but generated using the two different power models. For the traces generated with the HW power model, the results are the following:

- CPA
  - 25 traces ⇒ 95% key recovered
  - 50 traces ⇒ 100% key recovered
  - 100 traces ⇒ 100% key recovered
- DPA
  - 1000 traces ⇒ 95% key recovered
  - 2000 traces ⇒ 100% key recovered

For the HD power model, instead, the results are below:

- CPA
  - 25 traces ⇒ 65-70% key recovered
  - 50 traces ⇒ 95% key recovered
  - 100 traces ⇒ 100% key recovered
- DPA

- 2000 traces ⇒ 70-75% key recovered
- 3000 traces ⇒ 75% key recovered
- 5000 traces ⇒ 75% key recovered

If we have to base our choice on these data, it seems the HW model is better than the HD model. Using the same amount of traces the key recovering is easier for the key related to the traces generated with the HW model. This is the reason why we adopted it for the generation of all the simulated traces.

Chapter 4

Analysis results

This chapter wants to explain how the analyses have been done, their approaches, compare the results and provide scripts to replicate or improve this work.

4.1 Analysis context

As mentioned in section 2.2 the cryptographic algorithm used is AES-128 which works with 16 bytes blocks: for 16 bytes of plaintext it retrieves 16 bytes of ciphertext related to a 16 bytes key. There are a lot of AES implementations, written in several languages and with different performances. We have chosen to use two implementations: one written in C and one in assembly. Both implementations were performed on a real ARM microcontroller (STM32F4) and both were emulated using Rainbow. Every time, the traces were collected and saved on disk to make analyses at a later stage. Again, for both implementations, two types of analyses were done: DPA and CPA.

At the level of AES, the attacks exploit the S-Box output in the first round which corresponds to the SubBytes function and, for the success of the attack, the plaintext and the trace are mandatory to know (fig. 4.1).

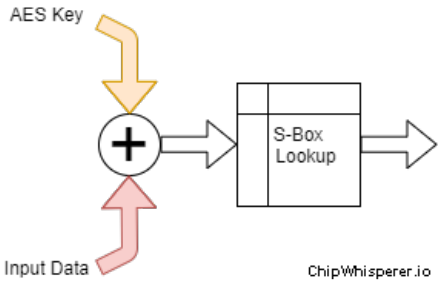

Figure 4.1: S-Box output: the point exploited in the attacks

At the level of the attacks carried out, the DPA model used is the difference of means and the selection function is based on the least significant bit (LSB). For the CPA the power consumption model used is the HW model.

The last important details are the sampling rate for the real traces which corresponds to 4 samples per clock cycle and for the simulated traces which is 1 sample per instruction.

4.1.1 DPA implementation

To implement the difference of means attack, we apply the fundamental hypothesis of the power analysis: there is a significant difference in power consumption if a bit of an output is 0 or 1. In particular, for this attack the bit considered is the LSB and the output is the output of the S-Box lookup. In order to predict the outputs, the script or software for the analysis replicates a model of the S-Box doing the same thing that the AddRoundKey and SubBytes functions do. Since we know the plaintext used by the AES algorithm, we can sort each S-Box output against a key guess into two sets: set 1 where LSB is 0 and set 2 where LSB is 1. This is done one byte at a time, testing all the possible values of the key (0-255) against the known plaintext. Then, the correct key should produce the highest significant difference.

A pseudocode is provided and can be adapted to any language.

```
for byteIndex 0 to 16:
    for keyGuess 0 to 255:
        for traceIndex 0 to N:
            sbox_output = sbox[keyGuess XOR plaintext[traceIndex][
    byteIndex]]

            if ( LSB(sbox_output) == 1 )
                add trace relative to traceIndex to set1
            else if ( LSB(sbox_output) == 0 )
                add trace relative to traceIndex to set0

    calculateAverage(set1)
    calculateAverage(set0)
    calculatePointToPointDifference(set1, set0)
```

The key to recover is 16 bytes long, `N` is the number of traces acquired, `sbox[]` is the array of 256 bytes for the lookup, `plaintext` is the array containing the plaintext associated to every trace, each entry is 16 bytes long but the total entries are N, `LSB()` is the function to get the LSB, `set0` and `set1` are the two sets where the traces are divided. The first for loop is to cycle over all the 16 bytes of the key, recovering one byte at a time, the second one iterates over all the possible values of

a byte and the third one iterates over all the acquired traces to make possible the classification in two sets. Appendix D.1 shows a working python implementation of the attack.

4.1.2 CPA implementation

For this attack, as in DPA, we want to predict the output of the SubBytes step of AES. Rather than splitting the traces into two groups, the goal of this technique is to use a power model to predict the S-Box output. Using the HW power model the thing to predict is the number of bits set to 1 in the SubBytes output. The correlation coefficient can then be calculated against the real power trace and the predicted power model against all the possible key values. Once the correlation is computed, the highest value should correspond to the correct key.

A pseudocode is provided and can be adapted to any language.

```
for byteIndex 0 to 16:
    for keyGuess 0 to 255:
        hw_array[] = 0
        pos = 0
        for traceIndex 0 to N:
            sbox_output = sbox[keyGuess XOR plaintext[traceIndex][
    byteIndex]]
            hw_array[pos] = HW(sbox_output)
            pos= pos + 1

        traces_array[] = getAllAcquiredTraces()
        calculatePearsonCoefficient(traces_array, hw_array)
```

The key to recover is 16 bytes long, `N` is the number of traces acquired, `sbox[]` is the 256-byte array for the lookup, `plaintext` is the array containing the plaintext associated with every trace, each entry is 16 bytes long but the total entries are N, `HW()` is the function to compute the HW and the result is put in the array `hw_array`. The first for loop is to cycle over all the 16 bytes of the key, recovering one byte at a time and the second one iterates over all the possible values of a byte. Appendix D.2 shows a working python implementation of the attack.

4.2 TinyAES - C implementation

The first implementation that is attacked is TinyAES [17], a small and portable AES implementation in C language. In the following analyses a slightly different version is used but the core is the same, it is the same used by the ChipWhisperer's tutorial.

The firmware programmed into the target board is the one presented in appendix B.1 where only the encryption function is captured. The firmware is compiled with the optimization flag *-Os*, for more details about the optimization options using GCC see [18]. The capture script in appendix A is good and can be adopted by changing only the number of traces to acquire. It is important to save into an appropriate format the traces to avoid a new capture each time you want to make an analysis, in the script it is not specified but an easy way is using the *NumPy* [19] Python library.

About the simulation with Rainbow, we must create a small firmware in C language and compile it with the following line *arm-none-eabi-gcc -mthumb -mcpu=cortex-m4 -mfloat-abi=soft –specs=nosys.specs -Os*. The C script must contain only the two functions needed by the AES to encrypt some text, one function is the one to expand the key and the other one is the proper encryption.

Listing 4.1: Rainbow firmware to emulate TinyAES

```
#include "aesTiny.h"

int main(){

    uint8_t input[]={"testcifratura123"};
    uint8_t key[16]={0x70, 0x61, 0x73, 0x73, 0x77, 0x6F, 0x72, 0x64,
    0x70, 0x61, 0x73, 0x73, 0x77, 0x6F, 0x72, 0x64};

    AES128_ECB_indp_setkey(key);

    AES128_ECB_indp_crypto(input);

    return 0;
}
```

The parameter must be set but then they will be overwritten by Rainbow. Both functions will be emulated but only the second one will be traced. Appendix C.1 shows the correct python script to set the emulation parameter and to capture N traces.

For the encryption, in the real and simulated case, the key is always the same and it is `[0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]`.

4.2.1 Simulated traces analysis

Figure 4.2 shows the entire execution trace of the encryption of 16 random bytes. The 10 rounds are clearly visible and the total length is 5000 samples, which corresponds to 5000 instructions. The results are provided in two manners: textual,

with a table containing the values computed with the scripts, and graphic, with a diagram.

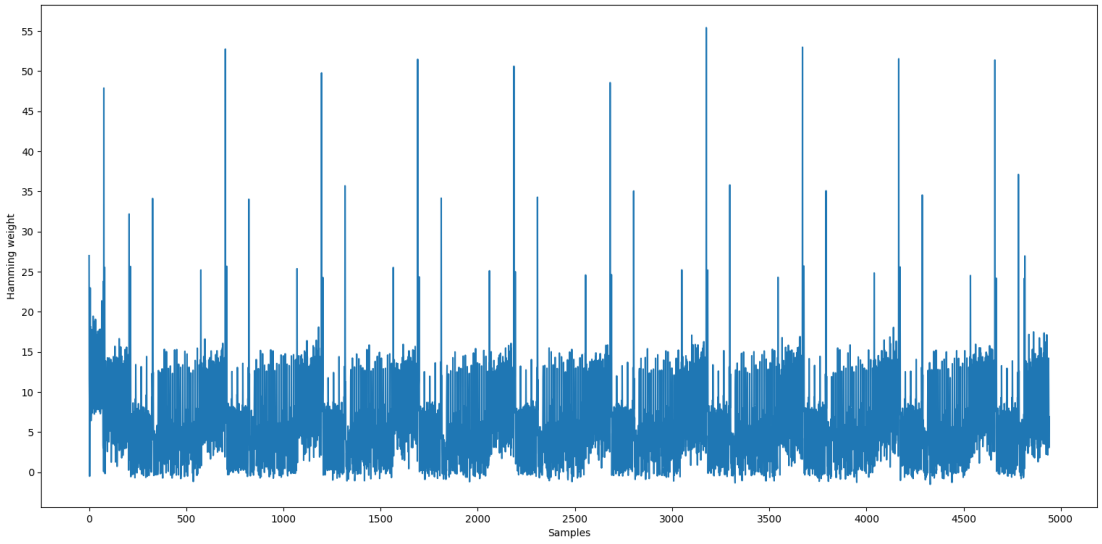

Figure 4.2: Simulated power trace of TinyAES implementation

DPA results

The first analysis done is the DPA, and the entire key is recovered with 600 traces. Table 4.1 is the textual view of the result, while figure 4.3 is the plotted data. Axis X is the number of samples and axis Y is the difference of means, each key byte corresponds to a colour and there is a legend to help in the consultation.

In the figure, it is possible to distinguish three groups of peaks related to different steps of AES. It means that DPA finds leakage in other points of the AES in addition to the SubBytes output that is the point we attack, as said at the beginning of the chapter. The first group, as expected, is the SubBytes step, where each spike corresponds to the load instruction from the S-Box for a total of 16 spikes. The second group is the ShiftRows step: there are only 12 peaks but they correspond again to a load instruction. The final group is related to the MixColumns: it is divided into 4 subgroups and each spike corresponds to a load instruction.

CPA results

The CPA finds the entire key with 50 traces. Table 4.2 is the textual view of the result while figure 4.4 is the plotted data. Axis X is the number of samples and axis Y is the correlation. The plotted data is similar to DPA, the points where there is

Byte	**Key**	**1° guess**		**2° guess**	
		Value	Avg diff	Value	Avg diff
0	2B	2B	0,894	B7	0,689
1	7E	7E	0,738	9C	0,720
2	15	15	0,865	38	0,698
3	16	16	1,205	E4	0,703
4	28	28	1,118	8F	0,742
5	AE	AE	0,946	94	0,743
6	D2	D2	0,992	57	0,735
7	A6	A6	1,073	2A	0,792
8	AB	AB	0,943	69	0,744
9	F7	F7	0,960	63	0,725
10	15	15	1,017	0F	0,794
11	88	88	1,092	11	0,774
12	09	09	1,079	E5	0,768
13	CF	CF	0,971	AE	0,705
14	4F	4F	0,982	12	0,739
15	3C	3C	0,933	E8	0,767

Table 4.1: DPA of 600 simulated traces - TinyAES

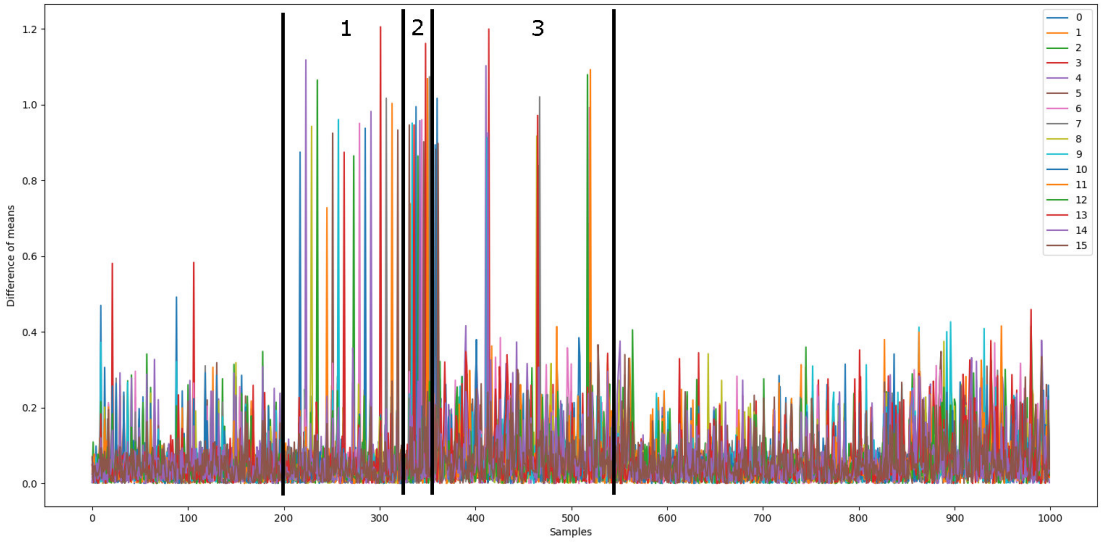

Figure 4.3: DPA of 600 simulated traces - TinyAES

leakage follow the same pattern: there are three groups of leakage corresponding

to three steps of AES and also the instructions that leak are the same.

Byte	**Key**	**1° guess**	
		Value	Correlation
0	2B	2B	0,8824
1	7E	7E	0,9005
2	15	15	0,8140
3	16	16	0,8423
4	28	28	0,7873
5	AE	AE	0,8267
6	D2	D2	0,8253
7	A6	A6	0,8306
8	AB	AB	0,8252
9	F7	F7	0,8501
10	15	15	0,8533
11	88	88	0,8388
12	09	09	0,8632
13	CF	CF	0,8241
14	4F	4F	0,8598
15	3C	3C	0,8741

Table 4.2: CPA of 50 simulated traces - TinyAES

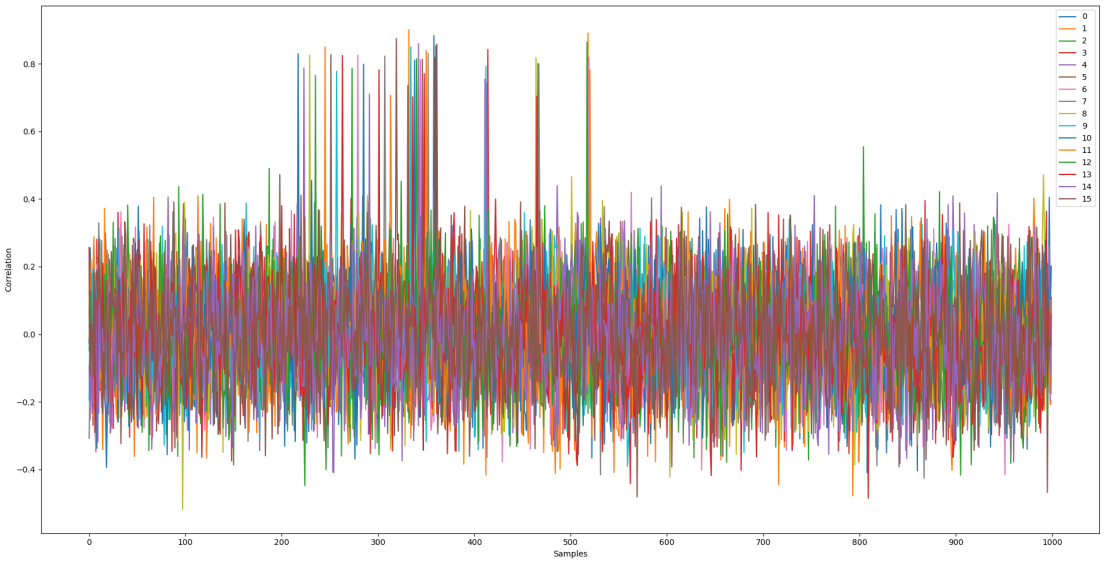

Figure 4.4: CPA of 50 simulated traces - TinyAES

Using different optimization flags

The analyses done before used a firmware compiled with the option *-Os* which produce a trace of about 5000 samples. What about using other flags? If we change the optimization we expect that the traces will change. With *-O0* we remove any optimization and we obtain a trace of 25000 samples, five times the traces obtained with *-Os*. If we want to optimize at the maximum level the correct flag is *-O3* and the resulting trace has 3500 samples. Since it is the highest optimize option, it is correct to obtain the shortest trace.

4.2.2 Real traces analysis

Figure 4.5 shows the entire trace of the execution of 16 random bytes encryption. The 10 rounds are clearly visible and the total length is 31000 samples that correspond to $31000/4 = 7750$ clock cycles due to the CW sampling rate.

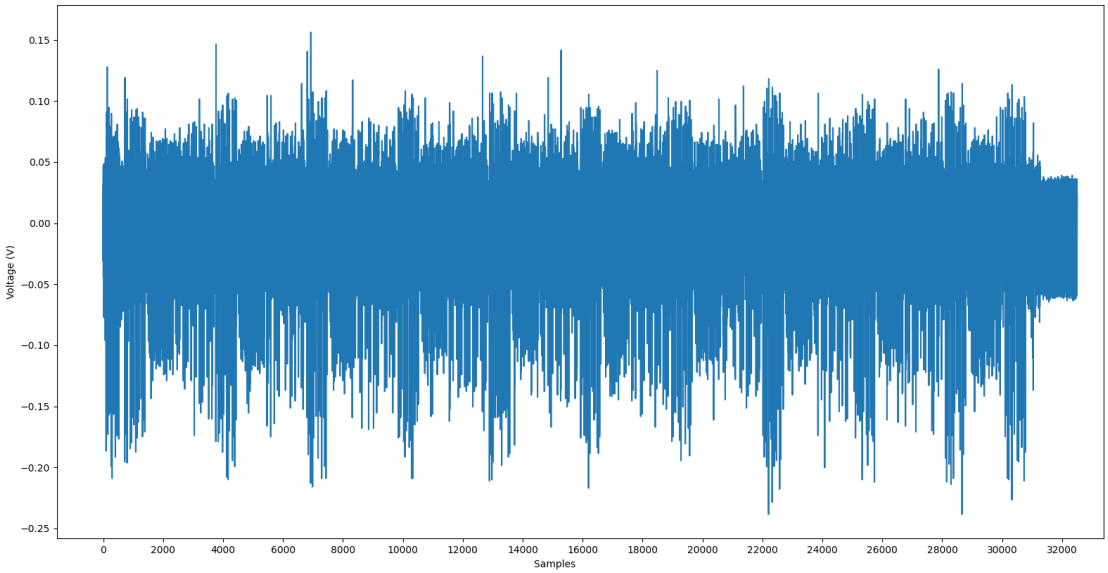

Figure 4.5: Real power trace of TinyAES implementation

DPA results

DPA recover the entire key with 300 traces, half the simulated case. Table 4.3 is the textual view of the result while figure 4.6 is the plotted data.

As we expected, the correct key byte produces the highest value and the 16 peaks are clearly visible (between sample 1500 and 2400), these peaks are related to the SubBytes function, while the others about 2500 samples are related to the

function ShiftRows. In the table we can notice the gap between the first guess (the correct byte) and the second guess (incorrect byte), for all the correct bytes their values are bigger, reflecting what the theory says.

For the simulated case we are able to know which instruction corresponds to each peak since there is a relationship sample-instruction, this fact is Rainbow dependent because it generates one point for each emulated instruction. For the real case, it is slightly different: we know that four samples represent 1 clock cycle but there is no direct relationship with the instructions. Moreover, some instructions take 1 clock cycle while others require 2 clock cycles. For example, if there are 40 samples they correspond to 10 clock cycles but we do not know the exact number of instructions executed, maybe ten or maybe less. This is why the manual counting of the samples to deduce the instruction is not feasible and we need another way to find where is the leakage. One possible method and the one we used is manually moving *trigger_high()* in the code in order to start the trace capture around the point we think there is the instruction that leaks. Capturing some traces and then computing DPA or CPA, we expect the first peak at the beginning of the trace. Counting the number of samples from the beginning to the first peak, we can estimate how many instructions correspond to that samples and if the leaking point is quite the same. Once we did some tests, the instructions seem to be the same as the simulated case with a gap of only a few clock cycles, hence all the peaks correspond to the load instruction.

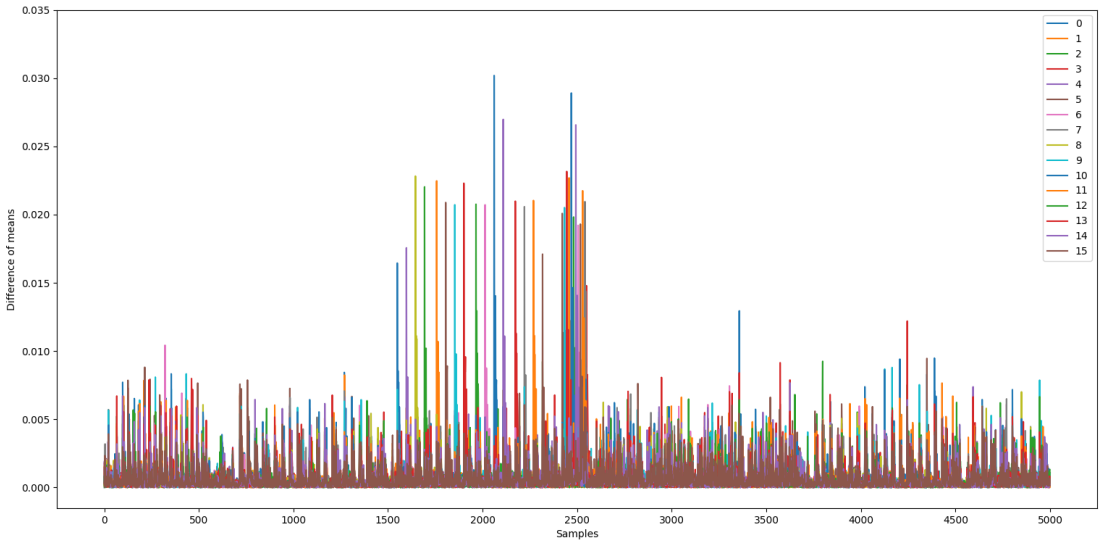

Figure 4.6: DPA of 300 real traces - TinyAES

Byte	Key	1° guess		2° guess	
		Value	Avg diff	Value	Avg diff
0	2B	2B	0,0164	5	0,0138
1	7E	7E	0,0226	15	0,0145
2	15	15	0,0207	0B	0,0160
3	16	16	0,0209	8F	0,0144
4	28	28	0,0175	4B	0,0161
5	AE	AE	0,0208	C5	0,0145
6	D2	D2	0,0207	B6	0,0137
7	A6	A6	0,0209	CE	0,0151
8	AB	AB	0,0228	16	0,0166
9	F7	F7	0,0207	58	0,0139
10	15	15	0,0301	83	0,0153
11	88	88	0,0217	1E	0,0163
12	09	09	0,0220	47	0,0128
13	CF	CF	0,0231	25	0,0148
14	4F	4F	0,0269	F2	0,0166
15	3C	3C	0,0193	77	0,0150

Table 4.3: DPA of 300 real traces - TinyAES

CPA results

For CPA analysis only 50 traces are necessary to recover the key and the correlation is almost perfect. Again, table 4.4 shows the textual results and figure 4.7 the plotted results that reflect what was found through the DPA with very little differences due to the different analysis.

Using different optimization flags

The firmware used in the real board was compiled with the flag *-Os*, the same flag utilized for the simulated case analyses. For that case, we have seen that different optimization flags impact in terms of the number of instructions executed by the microcontroller. With the option *-Os* the trace produced has about 30000 samples corresponding to 7750 clock cycles. The option *-O0* produces the longest traces, it has about 180000 samples. The option *-O3* produces, instead, the shortest trace with only 18000 samples. The results follow the same behaviour found in the simulated case, more optimization produces shorter traces and less optimization longer traces.

Byte	**Key**	**1° guess**	
		Value	Correlation
0	2B	2B	0,9949
1	7E	7E	0,9940
2	15	15	0,9945
3	16	16	0,9941
4	28	28	0,9957
5	AE	AE	0,9840
6	D2	D2	0,9940
7	A6	A6	0,9967
8	AB	AB	0,9937
9	F7	F7	0,9938
10	15	15	0,9919
11	88	88	0,9950
12	09	09	0,9934
13	CF	CF	0,9944
14	4F	4F	0,9947
15	3C	3C	0,9931

Table 4.4: CPA of 50 real traces - TinyAES

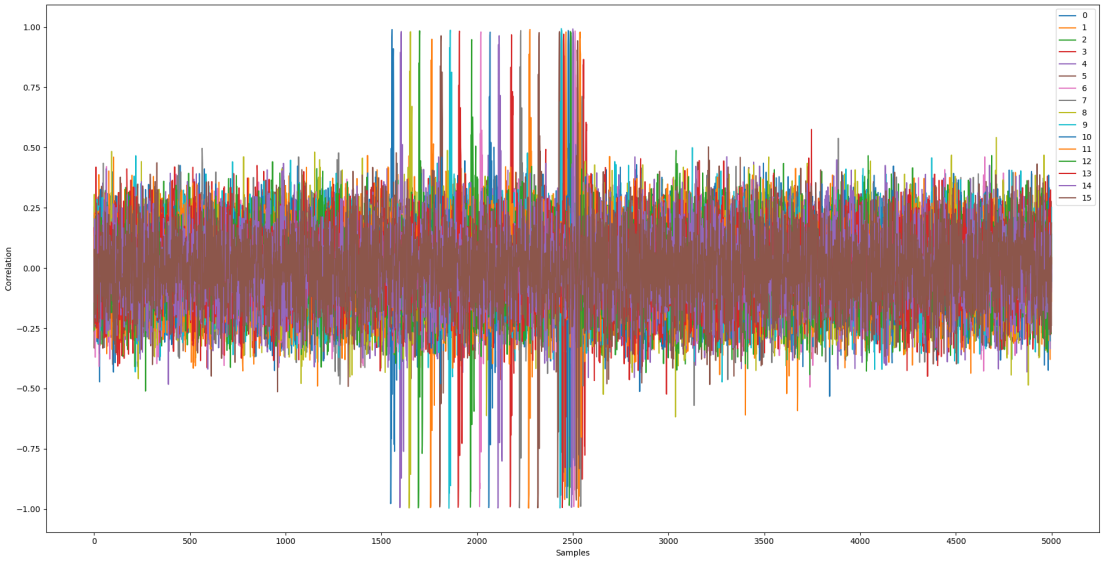

Figure 4.7: CPA of 50 real traces - TinyAES

4.3 Cortexm-AES - Assembly implementation

The second implementation that is attacked is Cortexm-AES [20], an implementation written in assembly and optimized for real world cortex-m microcontrollers. The repository includes implementations for several architectures, for our purposes we use *CM3_IT* that can be used for cortex-m4. In particular the files used are: *lookup_tables.c* which contains the S-Box, *CM3_1T_AES_128_keyschedule_enc.S* and *CM3_1T_AES_encrypt.S*.

The firmware programmed into the target board is the one presented in appendix B.2 where only the encryption function is captured. It is compiled with the optimize option *-Os*. The rules described and suggested for the other implementation are as valid for this one.

About the simulation with Rainbow, we must create again a small firmware in C language and compile it with the following line *arm-none-eabi-gcc -mthumb -mcpu=cortex-m4 -mfloat-abi=soft –specs=nosys.specs -Os*. An example of a C script is provided below.

Listing 4.2: Rainbow firmware to emulate Cortexm-AES

```
1  extern void CM3_1T_AES_128_keyschedule_enc(uint8_t *rk, const uint8_t
       *key);
2  extern void CM3_1T_AES_encrypt(uint8_t* rk, const uint8_t* in,
       uint8_t* out, size_t rounds);
3
4  int main(){
5
6      const uint8_t key[16]={0x70, 0x61, 0x73, 0x73, 0x77, 0x6F, 0x72,
       0x64, 0x70, 0x61, 0x73, 0x73, 0x77, 0x6F, 0x72, 0x64};
7      uint8_t rk[11*16];
8      const uint8_t in[16]={0,0,0,0,1,2,3,1,2,4,1,2,5,1,2,6};
9      uint8_t out[16];
10
11     CM3_1T_AES_128_keyschedule_enc(rk,key);
12
13     CM3_1T_AES_encrypt(rk,in,out,10);
14
15     return 0;
16 }
```

The parameter must be set but then they will be overwritten by Rainbow. Both functions will be emulated but only the second one will be traced. Appendix C.2 shows the correct python script to set the emulation parameter and to capture N traces. Comparing the two emulation scripts, this one requires more manual settings due to the more parameters of the functions.

For the encryption, in the real and simulated case, the key is always the same and it is `[0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,`

`0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]`.

4.3.1 Simulated traces analysis

The figure 4.8 shows the entire execution trace of the encryption of 16 random bytes. It is the average of 50 traces so in this way the 10 rounds are visible. The total length is about 550 samples that correspond to 550 instructions.

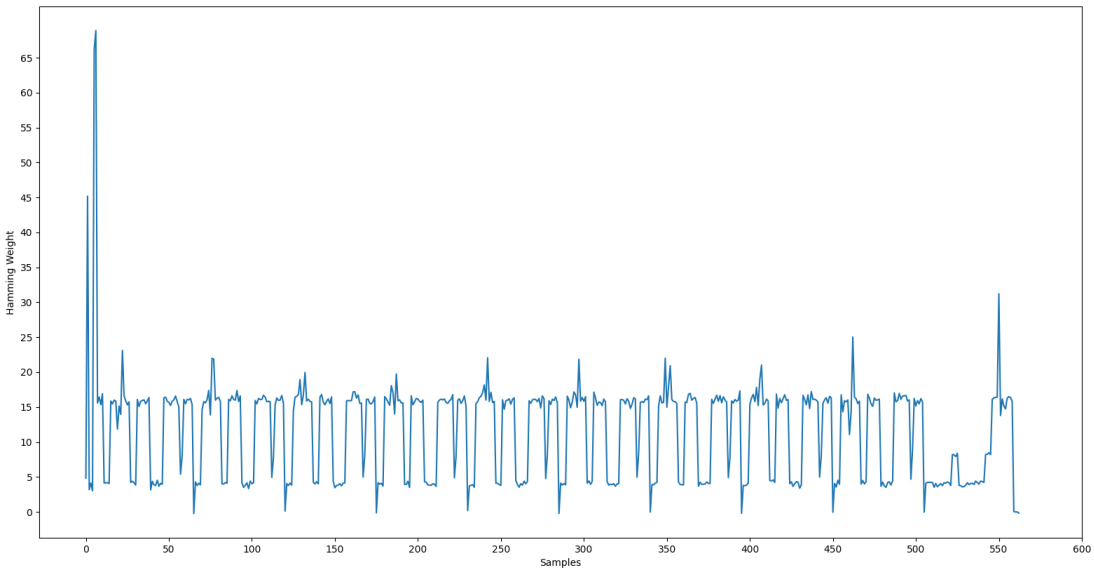

Figure 4.8: Average of simulated power traces of Cortexm-AES implementation

DPA results

The DPA finds the entire key with 1500 traces. Table 4.9 is the textual view of the results while figure 4.9 is the plotted data.

In the figure there are four groups of peaks but the first two are related to the same bytes (0, 4, 8 and 12). The point of AES we attack is the SubBytes output and we expect the peaks correspond to some instructions around that point. The code 4.3 shows which are the exact instructions and the following section does a deeper analysis, but now the question is: why there are two instructions that leak only for the bytes 0, 4, 8 and 12? Normally, as seen in the other analyses, it is always the load instruction that leaks. Doing some tests we found that these unusual spikes correspond to four `eor.w` instructions starting at address 0x8000278. Before that address, the registers r8, r9, r10 and r11 contain four bytes of the expanded key each one. The four `eor.w` do the xor between these values and the returned values from the S-Box lookup contained in the register r0, r1, r2, r3. The

next `eor.w` instructions no longer work with registers containing values of the expanded key. This is the reason why we can see leakage only in the first `eor.w` instructions and not in the others.

Byte	Key	1° guess		2° guess	
		Value	Avg diff	Value	Avg diff
0	2B	2B	2.145	B2	1.508
1	7E	7E	1.892	3C	1.760
2	15	15	2.067	87	1.699
3	16	16	2.215	8F	1.808
4	28	28	2.193	B1	1.851
5	AE	AE	1.841	37	1.534
6	D2	D2	1.956	90	1.826
7	A6	A6	1.566	34	1.393
8	AB	AB	2.177	32	1.755
9	F7	F7	2.344	3A	1.599
10	15	15	1.973	8C	1.590
11	88	88	2.130	D1	1.415
12	09	09	2.192	90	1.635
13	CF	CF	2.010	1F	1.886
14	4F	4F	1.882	BD	1.687
15	3C	3C	1.922	65	1.776

Table 4.5: DPA of 1500 simulated traces - Cortexm-AES

CPA results

The CPA finds the entire key with only 50 traces. Table 4.6 is the textual view of the results while figure 4.10 is the plotted data. Figure 4.11 represents in a different way the results of the analysis where each key byte has a single plot. If we zoom in around the peaks we can see at which sample they occur and consequently, thanks to Rainbow, we can retrieve the instruction corresponding to that sample. Finding the exact instruction means finding the exact point in the code where the leakage occurs. From Rainbow, during the emulation, we can know the state of the registers, the number of samples and also the instruction address corresponding to every sample. Thanks to these informations it is easy to identify all the leakage points. For example for byte 0, if we zoom in, the peak happens at sample 15 and we know that at this sample the corresponding instruction address is 0x8000258. Now, we have to inspect the code below looking for the instruction at address 0x8000258. At this point the instruction is `ldr.w r0, [lr, r0, lsl #2]`, it is a

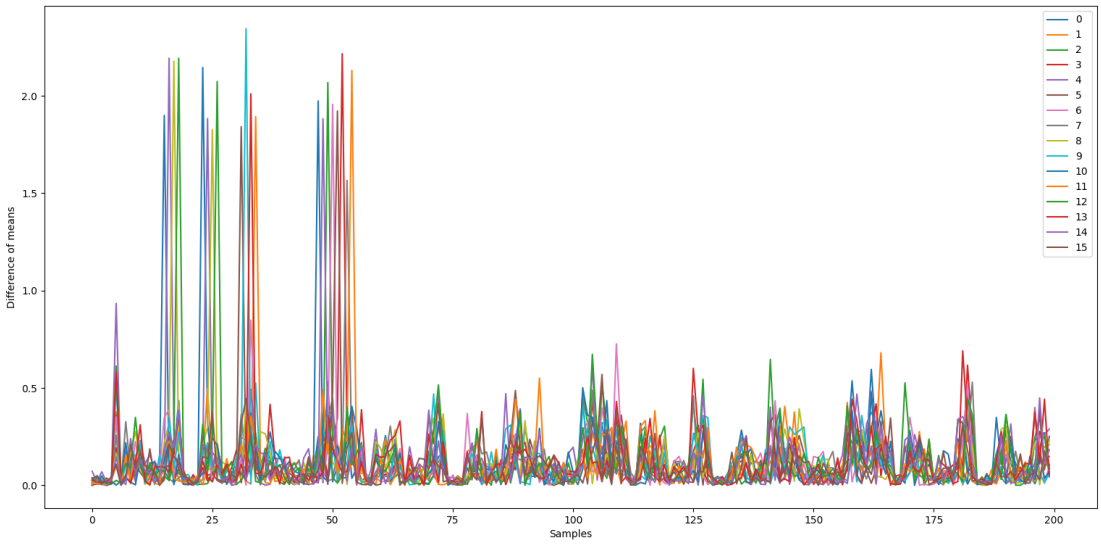

Figure 4.9: DPA of 1500 simulated traces - Cortexm-AES

load word from the S-Box table, the output of the SubBytes function, the exact point we exploit for the attack. For the other bytes we have to repeat this process and the results are shown in the code below where each byte is associated with the instruction that generates the leakage. Figure 4.10 has three groups of peaks, two of four and one of eight. This fact is confirmed by the assembly code where the load instruction are arranged in the same order.

Listing 4.3: Snippet of code of the encryption function - Cortexm-AES

```
08000230 <CM3_1T_AES_encrypt>:
 8000230:    eb00 1303    add.w    r3, r0, r3, lsl #4
 8000234:    e92d 4ffc    stmdb    sp!, {r2, r3, r4, r5, r6, r7, r8, r9,
     sl, fp, lr}
 8000238:    f240 0e00    movw     lr, #0
 800023c:    f2c2 0e00    movt     lr, #8192    ; 0x2000
 8000240:    4684         mov ip, r0
 8000242:    c9f0         ldmia    r1!, {r4, r5, r6, r7}
 8000244:    e8bc 000f    ldmia.w ip!, {r0, r1, r2, r3}
 8000248:    4044         eors     r4, r0
 800024a:    404d         eors     r5, r1
 800024c:    4056         eors     r6, r2
 800024e:    405f         eors     r7, r3
 8000250:    b2e0         uxtb     r0, r4
 8000252:    b2e9         uxtb     r1, r5
 8000254:    b2f2         uxtb     r2, r6
 8000256:    b2fb         uxtb     r3, r7
 8000258:    f85e 0020    ldr.w    r0, [lr, r0, lsl #2] //BYTE_0
 800025c:    f85e 1021    ldr.w    r1, [lr, r1, lsl #2] //BYTE_4
```

```
8000260:    f85e 2022    ldr.w    r2, [lr, r2, lsl #2] //BYTE_8
8000264:    f85e 3023    ldr.w    r3, [lr, r3, lsl #2] //BYTE_12
8000268:    f8dc 9004    ldr.w    r9, [ip, #4]
800026c:    f8dc a008    ldr.w    sl, [ip, #8]
8000270:    f8dc b00c    ldr.w    fp, [ip, #12]
8000274:    f85c 8b10    ldr.w    r8, [ip], #16
8000278:    ea88 4830    eor.w    r8, r8, r0, ror #16
800027c:    ea89 4931    eor.w    r9, r9, r1, ror #16
8000280:    ea8a 4a32    eor.w    sl, sl, r2, ror #16
8000284:    ea8b 4b33    eor.w    fp, fp, r3, ror #16
8000288:    fa5f f095    uxtb.w   r0, r5, ror #8
800028c:    fa5f f196    uxtb.w   r1, r6, ror #8
8000290:    fa5f f297    uxtb.w   r2, r7, ror #8
8000294:    fa5f f394    uxtb.w   r3, r4, ror #8
8000298:    f85e 0020    ldr.w    r0, [lr, r0, lsl #2] //BYTE_5
800029c:    f85e 1021    ldr.w    r1, [lr, r1, lsl #2] //BYTE_9
80002a0:    f85e 2022    ldr.w    r2, [lr, r2, lsl #2] //BYTE_13
80002a4:    f85e 3023    ldr.w    r3, [lr, r3, lsl #2] //BYTE_1
80002a8:    ea88 2830    eor.w    r8, r8, r0, ror #8
80002ac:    ea89 2931    eor.w    r9, r9, r1, ror #8
80002b0:    ea8a 2a32    eor.w    sl, sl, r2, ror #8
80002b4:    ea8b 2b33    eor.w    fp, fp, r3, ror #8
80002b8:    fa5f f0a6    uxtb.w   r0, r6, ror #16
80002bc:    fa5f f1a7    uxtb.w   r1, r7, ror #16
80002c0:    fa5f f2a4    uxtb.w   r2, r4, ror #16
80002c4:    fa5f f3a5    uxtb.w   r3, r5, ror #16
80002c8:    0e3f         lsrs     r7, r7, #24
80002ca:    0e24         lsrs     r4, r4, #24
80002cc:    0e2d         lsrs     r5, r5, #24
80002ce:    0e36         lsrs     r6, r6, #24
80002d0:    f85e 0020    ldr.w    r0, [lr, r0, lsl #2] //BYTE_10
80002d4:    f85e 1021    ldr.w    r1, [lr, r1, lsl #2] //BYTE_14
80002d8:    f85e 2022    ldr.w    r2, [lr, r2, lsl #2] //BYTE_2
80002dc:    f85e 3023    ldr.w    r3, [lr, r3, lsl #2] //BYTE_6
80002e0:    f85e 7027    ldr.w    r7, [lr, r7, lsl #2] //BYTE_15
80002e4:    f85e 4024    ldr.w    r4, [lr, r4, lsl #2] //BYTE_3
80002e8:    f85e 5025    ldr.w    r5, [lr, r5, lsl #2] //BYTE_7
80002ec:    f85e 6026    ldr.w    r6, [lr, r6, lsl #2] //BYTE_11
80002f0:    ea80 6037    eor.w    r0, r0, r7, ror #24
80002f4:    ea81 6134    eor.w    r1, r1, r4, ror #24
```

4.3.2 Real traces analysis

Figure 4.12 shows the entire trace of the execution of the encryption of 16 random bytes. It is the average of 10000 traces so in this way the 10 rounds are visible. The total length is about 3100 samples which corresponds to $3100/4 = 750$ clock cycles due to the CW sampling rate. Since it is an assembly implementation, it is

Byte	**Key**	**1° guess**	
		Value	Correlation
0	2B	2B	0.822
1	7E	7E	0.940
2	15	15	0.872
3	16	16	0.881
4	28	28	0.874
5	AE	AE	0.871
6	D2	D2	0.919
7	A6	A6	0.877
8	AB	AB	0.895
9	F7	F7	0.798
10	15	15	0.905
11	88	88	0.862
12	09	09	0.900
13	CF	CF	0.922
14	4F	4F	0.846
15	3C	3C	0.819

Table 4.6: CPA of 50 simulated traces - Cortexm-AES

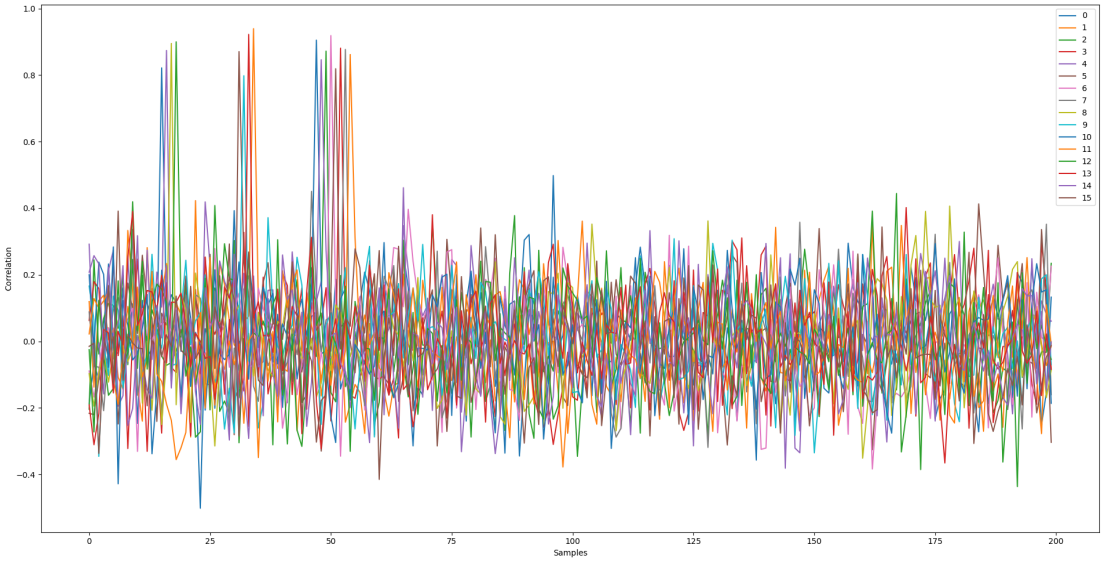

Figure 4.10: CPA of 50 simulated traces - Cortexm-AES

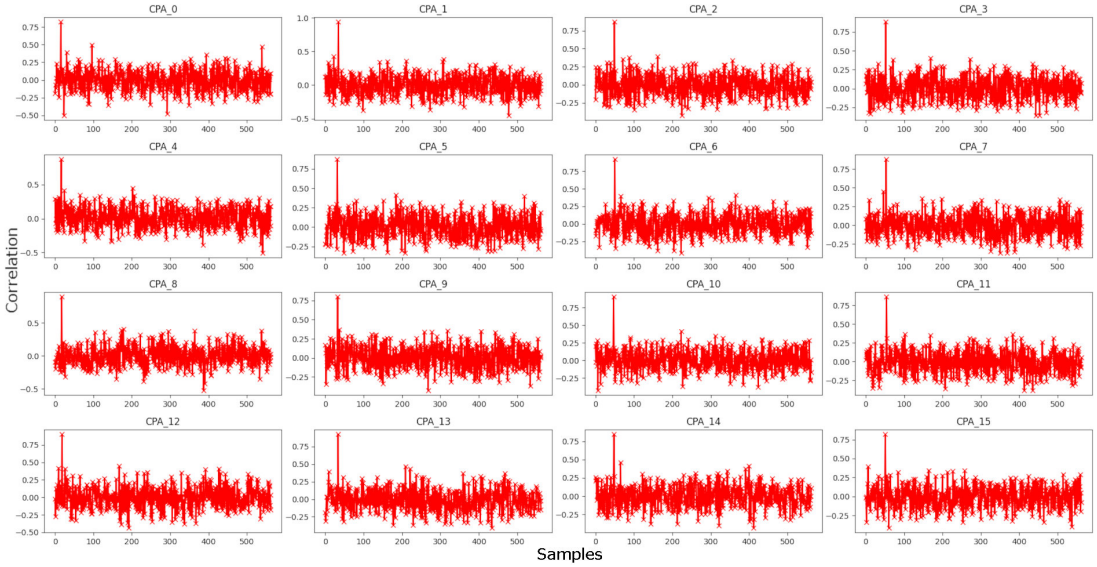

Figure 4.11: CPA of 50 simulated traces - Cortexm-AES

faster about 10 times than the C implementation and this fact is confirmed by the total samples. The analyses are performed in the same manner as before to have a direct comparison of the results, moreover, more diagrams are presented to have a better understanding of the analyses and their limitations.

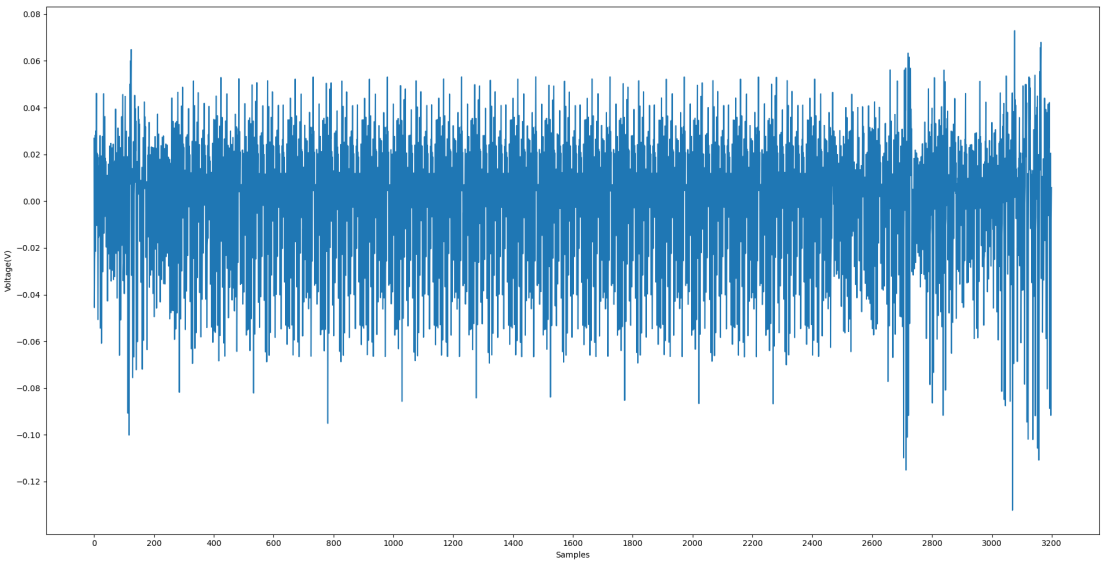

Figure 4.12: Average of real power traces of Cortexm-AES implementation

DPA results

DPA for this implementation was not very good. It requires a lot of traces and the bytes recovered are very few. For example with 20000 traces only three bytes were recovered and using 40000 traces only five were. This is the reason why the results are not provided and the focus is given only on CPA that finds more bytes. Despite the analyses have not been conducted, we tried to understand if the leakage pattern is similar to what was found for the simulated case. Moving *trigger_high()* in order to start the trace capture around the point we think there is the instruction that leaks, we found that the leakage points correspond only to load instructions and not also to some xor as we saw in the other case.

CPA results

In the previous implementation, CPA required only 50 traces but now there is a totally different behaviour. Using 50 traces zero bytes of the key are recovered and we need to acquire at least 2000 traces to recover about half of the key. Increasing the number of acquisitions reaching 40000 traces, the recovered key is about 75%. To better understand the analysis, in addition to the usual graphics, more will be presented and discussed.

The first results are presented in the table 4.7 and the recovered byte are highlighted. The correlation is generally higher for the correct byte but it is not the rule because for other bytes it is very low as well as incorrect ones. This behaviour can be seen in bytes 4, 6 and 9 where the correlation is comparable to that of the wrong bytes.

The following chart 4.13 shows the correlation progression in function of the traces acquired. The black line is the correlation of the correct byte, the coloured lines are the correlation of the other 255 bytes (the incorrect ones) and the four yellow highlighted are the not recovered key byte. In an ideal case the correlation of the correct byte should be the highest and remain stable, while for the wrong bytes, it decreases with the number of traces. For example, the first 4 bytes have the ideal behaviour because there is a significant gap between the black line and the coloured ones, the correlation even with few traces is the highest and remains constant until the end. For other bytes, for example 4, 6 and 9, at the beginning their correlation follows the trend but, when about 20000 traces are captured, the black line starts to separate from the other lines and it assumes the highest value, indeed these key bytes are recovered.

Another interesting view is the one below (fig. 4.14), it is the rank progression in function of the traces acquired. In other words, tell us how many ranks (1 to 256) away from the top (rank 1) the actual subkey is in our table of guesses. Rank 1 associated with a byte means this byte has the highest correlation. Again, some bytes are steady to rank 1 from the beginning while others reach the top with the

Byte	**Key**	**1° guess**	
		Value	Correlation
0	2B	2B	0.282
1	7E	7E	0.184
2	15	15	0.300
3	16	16	0.163
4	28	28	0.033
5	AE	AE	0.167
6	D2	D2	0.038
7	A6	A6	0.294
8	AB	B2	0.027
9	F7	F7	0.035
10	15	15	0.277
11	88	88	0.399
12	09	86	0.079
13	CF	4B	0.057
14	4F	F2	0.068
15	3C	3C	0.289

Table 4.7: CPA of 40000 real traces - Cortexm-AES

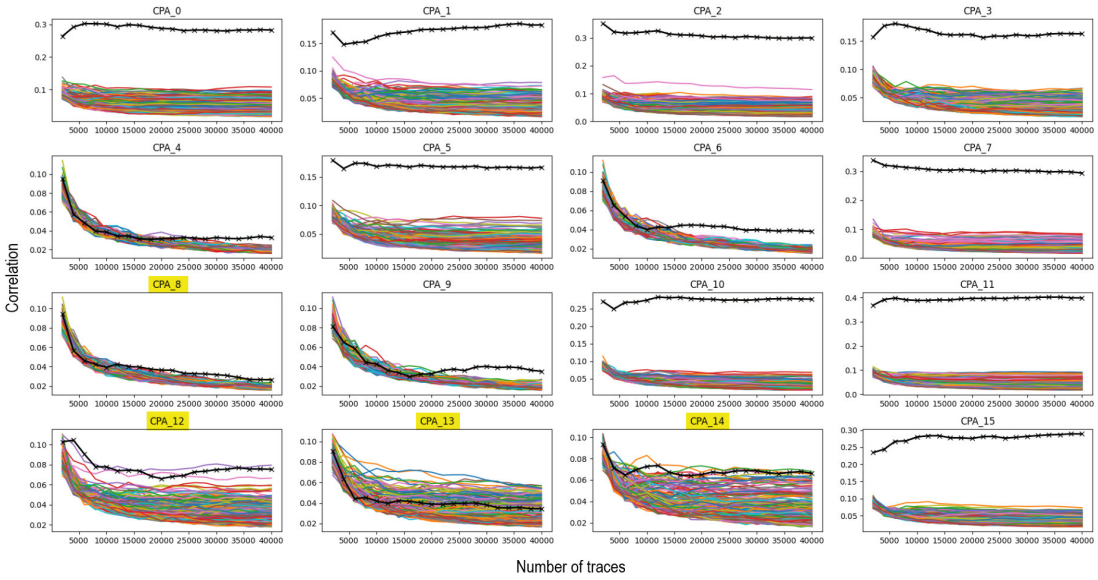

Figure 4.13: Correlation progression - Cortexm-AES

increasing number of traces acquired.

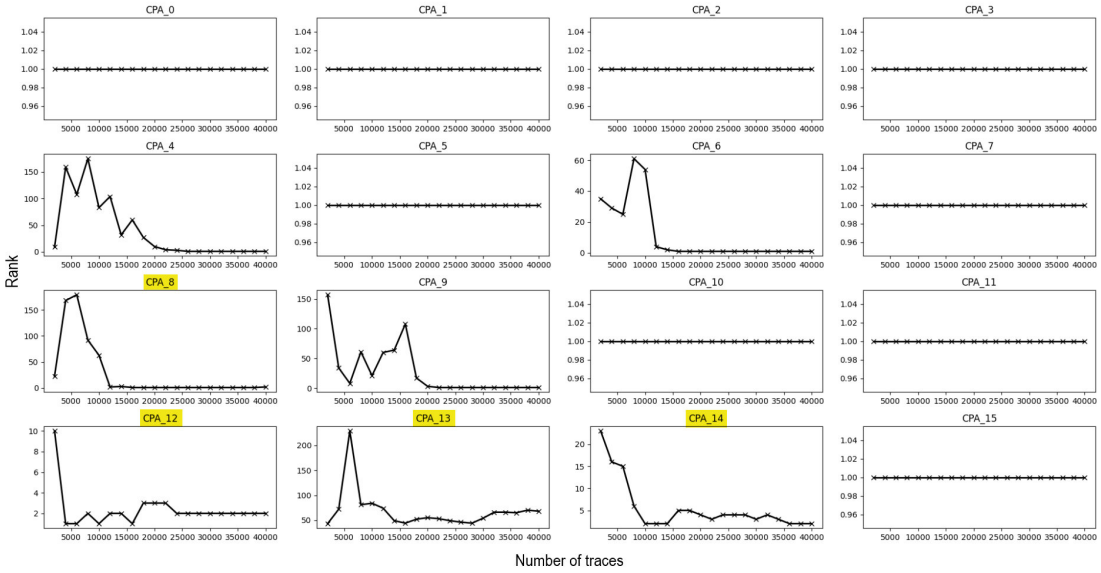

Figure 4.14: Rank progression - Cortexm-AES

The last diagram is the correlation plot, the figure 4.15 represents the correlation for each key byte. Where there is the peak, that point corresponds to the spot of the algorithm where there is leakage. It is easy to distinguish the recovered bytes because they have one peak isolated and the rest of the trace is flat with values close to zero, see for example byte 0. The others recovered as the number of traces collected increases (bytes 4, 6 and 9) have a bit more confusion, they have again the peaks but with a correlation close to zero. The not recovered bytes, instead, have more peaks and the chart is less clear than the previous one.

As done for TinyAES implementation, we want to know if the leakage in the real and simulated traces corresponds to the same instructions. We adopted the same technique, manually moving *trigger_high()* in order to start the trace capture around the point we think there is the instruction that leaks. In the simulated case the leakage was generated by the load word instructions, we expect the same for the real case. We expect also to find a similar pattern of the position of the peaks: two groups of four and one of eight (see fig. 4.10).

The first test done was putting the trigger before the instruction at line 9 (code 4.3). We computed CPA on 5000 traces and plotted the results: the peaks follow the expected pattern, two groups of four and one of eight and the first peak is not at the beginning because before the first load there are eight instructions which require at least 8 clock cycles (32 samples). The second test done was putting the trigger before the instruction at line 17, just before the first load, while the third

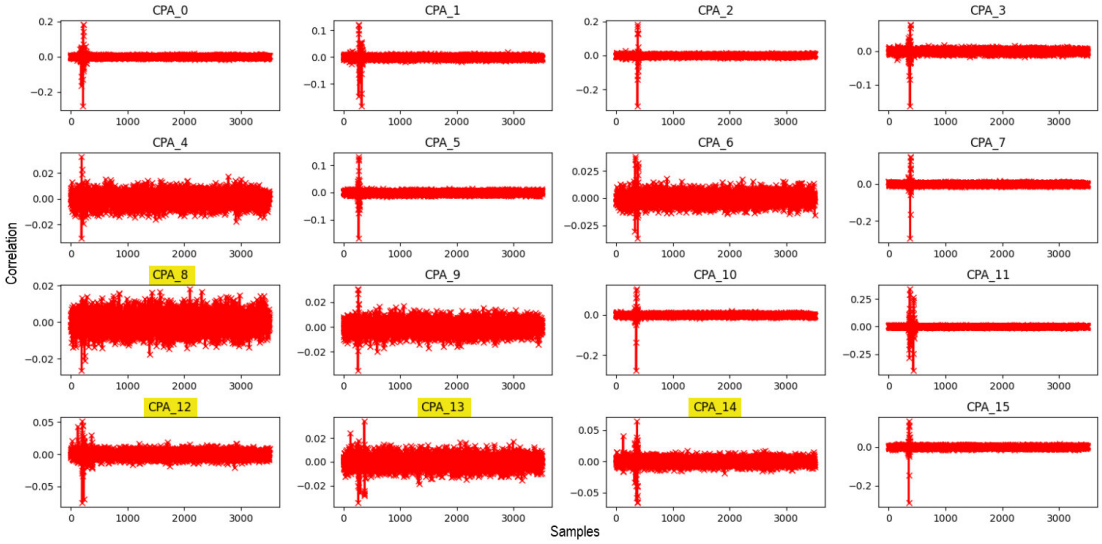

Figure 4.15: CPA of 40000 real traces - Cortexm-AES

test putting the trigger before line 21, just after the fourth load. The results of these two tests did not show the first group but confirmed what was discovered in the first test: the leakage seems to be generated by the load word instructions, the same as the simulated case.

Chapter 5

Improvements and hints

This fifth chapter has the role to provide the reader with hints and improvements that can be applied to the analyses presented in the previous chapter. Some of them are strictly related to the computation of the results, Lascar, for example. Other improvements are new ways of plotting data, for example VisPlot and Rainbow viewer.

5.1 VisPlot

The first hint is about VisPlot [21]. VisPlot is a useful tool created by the same team of Rainbow and provides a new way of plotting data. It is based on VisPy [22], a high-performance interactive 2D/3D data visualization library and it has been customized to be a side-channel trace visualizer. The most common python library to plot data is Matplotlib [23], all the diagrams in this work have been made using it. It has a lot of functionality and it is easy to use but on the contrary, it is slow when a lot of data need to be plotted. Citing the description of VisPlot on GitHub «Matplotlib is cumbersome to use during result analysis - when one needs to look around to find what's going on».

The main features are the fast display, responsive pan zoom and traces drawn with smooth colours until you select them. Selection is done with a click and holding CTRL for the multiple traces selection.

An example of how VisPlot makes the chart is the figure 5.1 where it is possible to see the multiple selections (in the example three lines are selected and are distinguishable).

At the python level, just two lines of code are requested: one to say the data to plot and one to open a new window with the diagram.

```
1 from visplot import plot
```

```
#.... compute data ...
v = plot(data_you_want_draw, dontrun=True)
v.run()
```

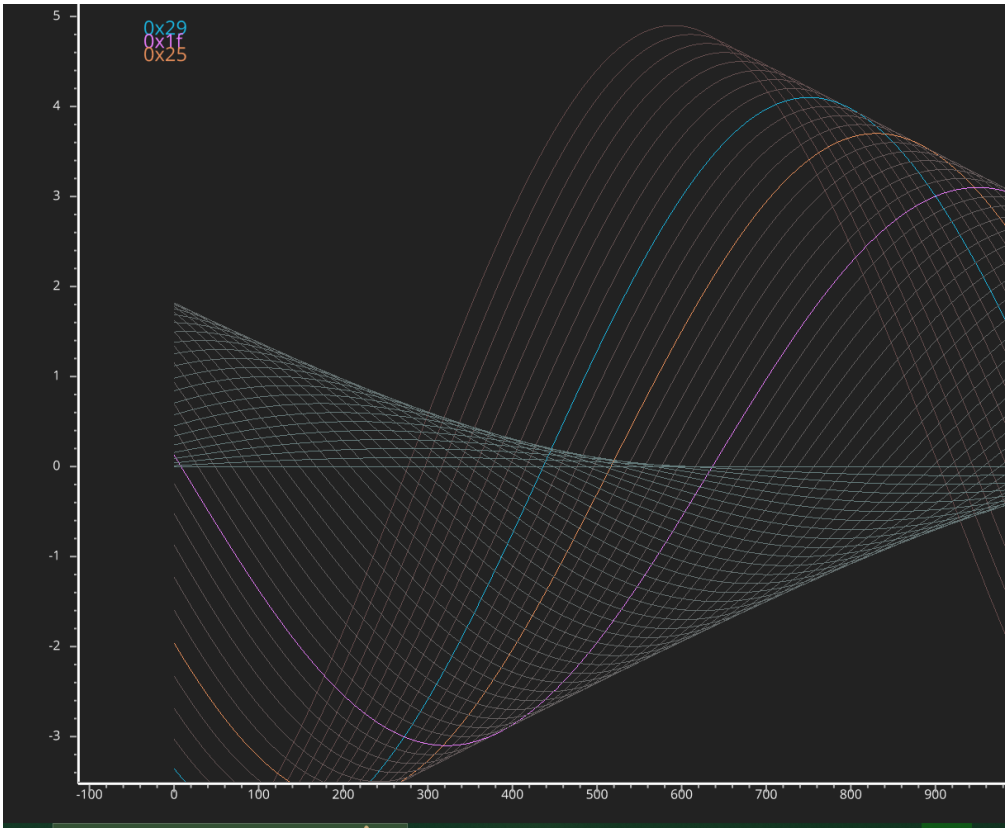

Figure 5.1: VisPlot example

5.2 LASCAR

Lascar (Ledger's Advanced Side Channel Analysis Repository) is a fast, versatile, and open-source python library designed to facilitate Side-Channel Analysis. It has already been mentioned in 3.1 as the software used during the analyses. There are many pros to the use of this tool:

- Openness: Lascar library is open source and can be customized for own purposes
- Simplicity: The script is easy and there are enough tutorials and examples to understand how it works

- Compatibility: It is written in python and relies on mainstream python libraries
- Flexibility: Implement your classes (for your already existing trace format, your specific attacks, and the way you want your output to be)

To understand better how to work with this tool, we need to define some basic objects that are fundamental for the correct use.

The first one is the `Container` object, it is role is to deliver side-channel data. Most of the time, side-channel data arises from several inputs for example a direct acquisition from a target board, an acquisition campaign already saved on the disk, simulated traces, an array of data and many others. In practise the Container class is the ad hoc class that collects the origin data in a standard way, you must implement the correct methods, but then it will represent your data who will be accessed by Lascar during the side-channel analysis.

The second fundamental object is `Session`, it is the class that will manage the reading of the Container traces by batch (to avoid loading all in RAM as much as possible), all the statistical computations that you would like to process on them (for example DPA or CPA) and finally the output that you seek from your analysis (simply graphic and textual format). All the traces are processed by the `Engine` class that does some computation as the mean (`MeanEngine`), the variance (`VarEngine`), the DPA (`DPAEngine`), the CPA (`CPAEngine`) and others. Once the entire computation is done we want to see the results in some way, `OutputMethod` is the class in charge of doing this. There are several methods: to store the result you can use `DictOutputMethod` or `Hdf5OutputMethod`, to see directly on the console `ConsoleOutputMethod`, to see graphically `MatPlotLibOutputMethod`, `ScoreProgressionOutputMethod` or `RankProgressionOutputMethod`. The last three methods were used in 4.3.2 to show the result in many ways instead of using a unique chart. Once the session class is set, it has the method `run()` which starts the whole process.

5.2.1 Acquisition and analysis from Chipwhisperer

This part describes how to perform the acquisition and the analysis of traces captured from the Chipwhisperer tool. In particular are shown the customized container that interacts with the CW and the setup for CPA analysis. The example is about CPA but with little modifications, the script is suitable for any other analysis or statistical computation.

The first five lines are variables needed by CW, the firmware must be in .hex format and `PLATFORM` says who is the target board. `AcquisitionSetupContainer` is the ad hoc class for our purposes that interact with CW. As a parameter there is the variable requested by CW, then it initialises the board and set the parameters

for the acquisition, finally defines a method for capturing traces. `CPAengine` is the class to make CPA analysis and must be declared for each byte indeed the array `CPAengines` contains all 16 engines. This class requires the selection function that for CPA is the HW power model of the S-Box output, `generate_guess_function()` is in charge of doing this. In case we wanted to perform DPA, we should use `DPAengine` and as a selection function take the LSB of the S-Box output. About the output method, the example shows three types: the first one prints the results directly on the console while the others make a chart. `Batch_size` defines how many traces to acquire and to analyse at a time. Appendix E.1 contains the entire code of the `AcquisitionSetupContainer`

Listing 5.1: Lascar and CW

```
KEY = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0
    x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]
PLATFORM = 'CW308_STM32F4'
SS_VER='SS_VER_1_1'
FIRMWARE_PATH="path/to/firmware.hex"
NUM_TRACES=10000

def generate_guess_function(byte):
     def guess_function(value, guess):
        return hamming_weight(sbox[value["plaintext"][byte]^guess])
    return guess_function

container = AcquisitionSetupContainer(NUM_TRACES, key=KEY, platform=
    PLATFORM, ss_ver=SS_VER, firmware_path=FIRMWARE_PATH) #acquire
    real traces from the target board

CPAengines = [
    CpaEngine("CPA_{}".format(byte), #the name
    generate_guess_function(byte),
    range(256), #the values for the guess
    solution=KEY[byte]) for byte in range(16)]

session = Session(
    container,
    engines=CPAengines,
    name="cpa on 16 bytes",
    output_method=[
            ConsoleOutputMethod(*CPAengines),
            MatPlotLibOutputMethod(*CPAengines, solution_only=True),
            ScoreProgressionOutputMethod(*CPAengines),
        ],
    #output_steps=1000, #if you want partial results or the plot
    output method
).run(batch_size=1000)
```

5.2.2 Acquisition and analysis from Rainbow

This part is very similar to the previous one, the only change is the container class `RainbowContainer` that simulates and traces the input firmware. At the beginning there are some declarations of variables for the simulation which will be passed as parameters in the container. The further code is the same as before because it depends on the analysis. Appendix E.2 contains the entire code of the `RainbowContainer`

Listing 5.2: Lascar and Rainbow

```
FILE="path/to/firmware.elf"
KEY = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0
    x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]
FUNCTION_KEY_EXP_NAME="CM3_1T_AES_128_keyschedule_enc"
FUNCTION_ENCRYP_NAME="CM3_1T_AES_encrypt"
NOISE=0.5
NUM_TRACES=1000

def generate_guess_function(byte):
    def guess_function(value, guess):
        return hamming_weight(sbox[value["plaintext"][byte]^guess])
    return guess_function

container = RainbowContainer(NUM_TRACES, noise=NOISE, key=KEY,
    binary_file=FILE, key_func=FUNCTION_KEY_EXP_NAME, encr_func=
    FUNCTION_ENCRYP_NAME)

CPAengines = [
    CpaEngine("CPA_{}".format(byte),
    generate_guess_function(byte),
    range(256),
    solution=KEY[byte]) for byte in range(16)]

session = Session(
    container,
    engines=CPAengines,
    name="cpa on 16 bytes",
    output_method=[
            ConsoleOutputMethod(*CPAengines),
            MatPlotLibOutputMethod(*CPAengines, solution_only=True),
            ScoreProgressionOutputMethod(*CPAengines),
        ],
    output_steps=100,
).run(batch_size=100)
```

How to emulate Chipwhisperer firmware

Until now, a special code for Rainbow has always been simulated, but it might be useful to emulate the same firmware running on the target board. For example, if we want to emulate the firmware using the cortexm-AES implementation, we can use the script provided for Rainbow in appendix C.2. If we use it without any changes we obtain a Rainbow error. This is caused because the S-Box table is not loaded in memory and when an instruction tries to read at its hypothetical address it reads an incorrect and unpredictable value. The simplest solution is to load manually the table and map the memory space with the script provided below. The memory addresses and the size of the table might be different but the procedure is always this. These few lines of code should be put immediately after Rainbow initialization and loading binary (example: put it at line 7 of appendix C.2).

```
1 self.device.map_space(0x20000000,0x20001000)     #map space manually
2 fwfile = open('simpleserial-cortexaes-CW308_STM32F4.bin',"rb") #open
    the .bin file
3 ROM_START = 0x8000000
4 AES_Te2_ADDR=ROM_START+0x12E0
5 fwfile.seek(0x12E0) #move the file object position to the beginning
    of the table
6 AES_Te_2=fwfile.read(1024)  #read the entire table
7 fwfile.close()
8 self.device.emu.mem_write(0x20000000,AES_Te_2)  #write table in
    memory
```

5.3 Rainbow viewer

By default, Rainbow generates traces with one point per instruction. A helpful function could be associating each point to the corresponding instruction so, in this way, plotting the results of an analysis it is possible to recover the peak and to what instruction it is associated. Manually it is a bit tricky but there is a tool inside Rainbow that is useful for our purpose. The tool is *Viewer* and is called like a function: `viewer(instructions, data_to_trace)` where the first parameter is the list of instructions computed by Rainbow and the second one is the data to trace. The instructions list is provided by Rainbow so no external tools are needed. Viewer builds a Qt application showing the instructions list next to VisPlot. Clicking an instruction, a vertical white line appears in the plot indicating at which point it corresponds. The figure below shows an example: instruction 827E corresponds to point 223 where there is a vertical white line.

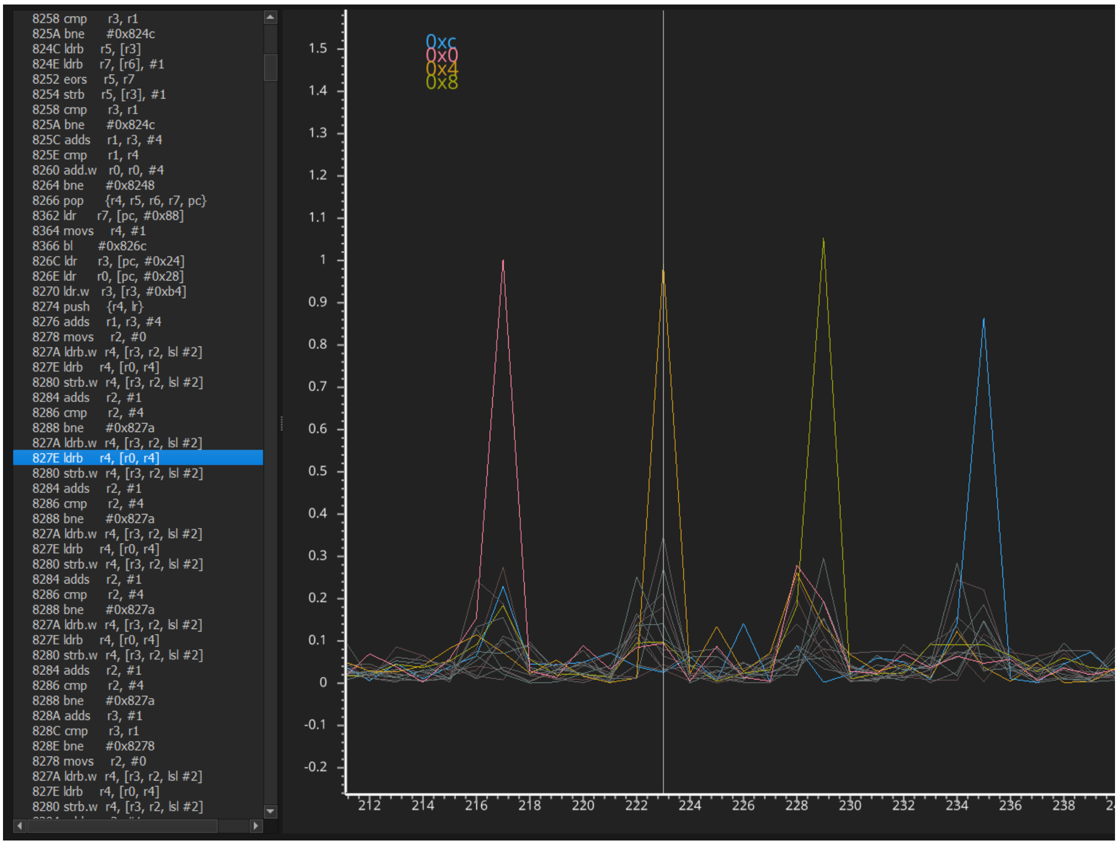

Figure 5.2: Viewer example

Chapter 6

Conclusion

The purpose of this thesis project is the evaluation of the traces produced by Rainbow, the embedded binaries emulator, against real power traces produced by a real board thanks to the NewAE Technology ChipWhisperer tool. For each implementation and type of analysis we defined four criteria for comparison:

1. Is the leakage generated by the same instruction?
2. Is the number of peaks the same?
3. Each peak corresponds to a key byte, are they in the same order?
4. Evaluation of the ratio between the distance from the first to the second peak and the total length of the 16 peaks (the distance from the first to the last peak)

The first criterion is satisfied, for both implementations the leakage between simulated and real corresponds, it is generated by load instructions. The second, instead, is not satisfied: in the C implementation, Rainbow generates more leakage because the analyses produce peaks related to the MixColumns step that there are not in the real scenario. About the assembly implementation is more difficult to do the same comparison but for example, DPA on simulated traces generates leakage associated with *eor* instructions for only one-quarter of the total bytes. The third criterion is again not satisfied, the comparison is only feasible for the TinyAES implementation because for the other one the entire key was not recovered. Only in the ShiftRows step the order of the spikes is different. The fourth and last criterion is satisfied, the ratios are not perfectly the same but they do not differ much: for TinyAES real it is $48/768 = 0{,}0625$ and for simulated $6/102 = 0{,}0588$, this shows that the pattern of the leakage is similar.

There are other things of Rainbow to consider that are not directly connected to the analyses, for example, the simplicity to set a script for the emulation of

binaries but also, since it is based on Unicorn Engine, the several architectures that it can simulate. On the contrary, trace generation is very dependent on the total instructions that it must emulate. Generally, the trace generation rate is always higher for Chipwhisperer with an average of 30 traces per second. For Rainbow, instead, for traces with 5000 points the generation rate is 10/15 traces per second while with 25000 points the rate drops to 2/3 traces per second. One possible improvement is to use directly Unicorn in Python and as a last chance Unicorn in C if the maximum performance is necessary.

Future work

To improve or extend this work some aspects can be evaluated. The first one is how Rainbow generates the traces: actually there are only two basic models described in section 3.3.2 but other better models can be developed. The second one is related to the analyses: the CPA power model was the HW power model but others should be better, HD or SD power model described in section 2.1.4, for example. About DPA, the selection function was always based on the LSB bit, but evaluating other bits or combinations of them can produce interesting results.

Appendix A

Basic capture script from CW

The following script is a working example on how to use Chipwhisperer to capture traces. There are some useful notes in the code.

```
import chipwhisperer as cw

SCOPETYPE = 'OPENADC'
PLATFORM = 'CW308_STM32F4'
CRYPTO_TARGET='TINYAES128C'
SS_VER='SS_VER_1_1'

#initialiaze the Chipwhisperer
scope = cw.scope()
target_type = cw.targets.SimpleSerial
target = cw.target(scope, target_type)
prog = cw.programmers.STM32FProgrammer

scope.default_setup()

#program the target with specific firmware
cw.program_target(scope, prog, "path/to/firmare.hex")

#set plaintext and key. The key used is a default key
#key=[0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0
    x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]
ktp = cw.ktp.Basic()
trace_array = []
textin_array = []
key, text = ktp.next()
target.set_key(key)
```

```
26
27 #number of traces
28 N = 100
29 for i in range(N):
30
31     #arm the Chipwhisperer
32     scope.arm()
33
34     #write plaintext to the target
35     target.simpleserial_write('p', text)
36
37     #capture the traces
38     ret = scope.capture()
39     if ret:
40         print("Target timed out!")
41         continue
42
43     #Read the ciphertext back from the target
44     response = target.simpleserial_read('r', 16)
45
46     #the actual trace is contained in "scope.get_last_trace()"
47     trace_array.append(scope.get_last_trace())
48     textin_array.append(text)
49
50     #create new plaintext
51     key, text = ktp.next()
52
53 #... save trace and plaintext ...
```

Appendix B

Target firmware

B.1 TinyAES

The following script is a working example of an AES firmware. Looking at this and appendix A, the user can have a full view about the workflow and how to customize for own purposes.

```
1
2  uint8_t get_key(uint8_t* k, uint8_t len){
3      AES128_ECB_indp_setkey(k);
4      return 0x00;
5  }
6
7  uint8_t get_pt(uint8_t* pt, uint8_t len){
8
9      //trigger to set the starting point of acquisition
10     trigger_high();
11
12     //encrypting the data block
13     AES128_ECB_indp_crypto(pt);
14
15     //trigger to reset the trigger pin
16     trigger_low();
17
18     simpleserial_put('r', 16, pt);
19     return 0x00;
20 }
21
22 int main(void){
23     uint8_t tmp[KEY_LENGTH] = {DEFAULT_KEY};
24
25     platform_init();
```

```
26      init_uart();
27      trigger_setup();
28
29      AES128_ECB_indp_setkey(tmp);
30
31      simpleserial_init();
32      simpleserial_addcmd('k', 16, get_key);
33      simpleserial_addcmd('p', 16,  get_pt);
34      while(1)
35          simpleserial_get();
36  }
```

B.2 Cortexm-AES

This is the working firmware of cortexm-AES implementation for the target board. It is like the TinyAES firmware with only the new functions.

```
1  #define ROUNDS 10
2  #define DEFAULT_KEY 0x2b,0x7e,0x15,0x16,0x28,0xae,
3            0xd2,0xa6,0xab,0xf7,0x15,0x88,0x09,0xcf,0x4f,0x3c
4  #define KEY_LENGTH 16
5
6  extern void CM3_1T_AES_128_keyschedule_enc(uint8_t *rk, const uint8_t
       *key);
7  extern void CM3_1T_AES_encrypt(uint8_t* rk, const uint8_t* in,
       uint8_t* out, size_t rounds);
8
9  uint8_t round_key[11*16];
10
11 uint8_t get_key(uint8_t* key, uint8_t len)
12 {
13      CM3_1T_AES_128_keyschedule_enc(round_key,key);
14      return 0x00;
15 }
16
17 uint8_t get_pt(uint8_t* pt, uint8_t len)
18 {
19      uint8_t output[16];
20
21      trigger_high();
22      CM3_1T_AES_encrypt(round_key, pt, output, ROUNDS);
23      trigger_low();
24
25      simpleserial_put('r', 16, output);
26      return 0x00;
27 }
```

```
28
29 int main(void)
30 {
31     uint8_t tmp_key[KEY_LENGTH] = {DEFAULT_KEY};
32
33     platform_init();
34     init_uart();
35     trigger_setup();
36
37     CM3_1T_AES_128_keyschedule_enc(round_key,tmp_key);
38
39     simpleserial_init();
40
41     simpleserial_addcmd('k', 16, get_key);
42     simpleserial_addcmd('p', 16,  get_pt);
43
44     while(1)
45         simpleserial_get();
46 }
```

Appendix C

Capture script from Rainbow

C.1 TinyAES

This is the working capture script for TinyAES implementation using Rainbow.

```
import numpy as np
from rainbow.generics import rainbow_arm

#define the architecture
e = rainbow_arm(sca_mode=True)
e.load("path/to/firmware.elf", typ=".elf")

def encrypt(key,plaintext):
    # Reset the emulator state
    e.reset()

    key_addr = 0xDEAD0000
    e[key_addr] = bytes(key)
    # AES128_ECB_indp_setkey(key);
    e["r0"] = key_addr
    e.start(e.functions["AES128_ECB_indp_setkey"] | 1, 0)

    buf_in = 0xDEAD1000
    e[buf_in] = plaintext
    # AES128_ECB_indp_crypto(input);
    e["r0"] = buf_in
    e["lr"] = 0
    #reset trace to collect only the ones related to the encryption
    function
```

```
24      e.trace_reset()
25      e.start(e.functions["AES128_ECB_indp_crypto"] | 1, 0)
26
27      #add some noise to make traces more realistic
28      trace= e.sca_values_trace + np.random.normal(0, 0.5, (len(e.
        sca_values_trace)))
29      return trace
30
31  def generate_trace(key):
32      plaintext = np.random.randint(0,256,(16,),np.uint8)
33      leakage = np.array(encrypt(key,plaintext.tobytes()))
34      return leakage, plaintext
35
36  N = 100 #number of traces
37  KEY = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0
        x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]
38  plaintext_array=[]
39  trace_array=[]
40
41  for i in trange(N):
42      trace_t,plain_t=generate_trace(KEY)
43      plaintext_array.append(plain_t)
44      trace_array.append(trace_t)
45
46  #... save trace and plaintext ...
```

C.2 Cortexm-AES

This is the working capture script for Cortexm-AES implementation using Rainbow.

```
1   import numpy as np
2   from rainbow.generics import rainbow_arm
3
4   #define the architecture
5   e = rainbow_arm(sca_mode=True)
6   e.load("path/to/firmware.elf", typ=".elf")
7
8   def aes_encrypt(key, plaintext):
9       # Reset the emulator state
10      e.reset()
11
12      key_addr = 0x90000000
13      e[key_addr] = key
14      rk_addr = 0x90001000
15      e[rk_addr] = key
16      # CM3_1T_AES_128_keyschedule_enc(rk, key)
```

```
17      e["r0"] = rk_addr
18      e["r1"] = key_addr
19      e.start(e.functions["CM3_1T_AES_128_keyschedule_enc"] | 1, 0)
20
21      buf_in = 0x90002000
22      buf_out = 0x90003000
23      e[buf_in] = plaintext
24      e[buf_out] = b"\x00" * 16  # Need to do this so this buffer is
        mapped into unicorn
25      # CM3_1T_AES_encrypt(rk, buf_in, buf_out,rounds)
26      e["r0"] = rk_addr
27      e["r1"] = buf_in
28      e["r2"] = buf_out
29      e["r3"] = 0xa   #number of rounds
30      e["lr"] = 0
31      #reset trace to collect only the ones related to the encryption
        function
32      e.trace_reset()
33      e.start(e.functions["CM3_1T_AES_encrypt"] | 1, 0)
34
35      #add some noise to make traces more realistic
36      trace= e.sca_values_trace + np.random.normal(0, 0.5, (len(e.
        sca_values_trace)))
37      return trace
38
39  def generate_trace(key):
40      plaintext = np.random.randint(0,256,(16,),np.uint8)
41      leakage = np.array(aes_encrypt(key, plaintext.tobytes()))
42      return leakage, plaintext
43
44  N = 100 #number of traces
45  KEY = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0
        x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]
46  plaintext_array=[]
47  trace_array=[]
48
49  for i in trange(N):
50      trace_t,plain_t=generate_trace(KEY)
51      plaintext_array.append(plain_t)
52      trace_array.append(trace_t)
53
54 #... save trace and plaintext ...
```

Appendix D

Python script for analysis

D.1 DPA

This is a basic script to compute DPA. It requires the traces and the plaintext to be previously saved in some way, for example numpy array.

```
import numpy as np

sbox = [
    #... fill with sbox values ...
]

def aes_internal(inputdata, key):
    return sbox[inputdata ^ key]

def calculate_diffs(guess, byteindex, bitnum):
    """Perform a simple DPA on two traces, uses global 'textin_array'
    and 'traces_array' """

    one_list = []
    zero_list = []

    for trace_index in range(numtraces):
        hypothetical_leakage = aes_internal(guess, textin_array[
   trace_index][byteindex])

        #Mask off the requested bit
        if hypothetical_leakage & (1<<bitnum):
            one_list.append(traces_array[trace_index])
        else:
            zero_list.append(traces_array[trace_index])

```

```
25      one_avg = np.asarray(one_list).mean(axis=0)
26      zero_avg = np.asarray(zero_list).mean(axis=0)
27      return abs(one_avg - zero_avg)
28
29  #array containing real traces previously saved
30  traces_array=np.load("path/to/traces.npy")
31  #array containing plaintexts previously saved
32  textin_array=np.load("path/to/plaintext.npy")
33
34  numtraces = np.shape(traces_array)[0] #total number of traces
35  numpoints = np.shape(traces_array)[1] #samples per trace
36
37  #Store your key_guess here
38  key_guess = []
39
40  #Which bit to target
41  bitnum = 0
42
43  full_diffs_list = []
44
45  for subkey in range(0, 16):
46
47      max_diffs = [0]*256
48      full_diffs = [0]*256
49
50      for guess in range(0, 256):
51          full_diff_trace = calculate_diffs(guess, subkey, bitnum)
52          max_diffs[guess] = np.max(full_diff_trace)
53          full_diffs[guess] = full_diff_trace
54
55      #Make copy of the list
56      full_diffs_list.append(full_diffs[:])
57
58      #Get argument sort, as each index is the actual key guess.
59      sorted_args = np.argsort(max_diffs)[::-1]
60
61      #Keep most likely
62      key_guess.append(sorted_args[0])
63
64      #Print results
65      print("Subkey %2d - most likely %02X"%(subkey, key_guess[subkey])
       , end="")
```

D.2 CPA

This is a basic script to compute CPA. It requires the traces and the plaintext to be previously saved in some way, for example numpy array.

```
import numpy as np

sbox = [
    #... fill with sbox values ...
]

def aes_internal(inputdata, key):
    return sbox[inputdata ^ key]

def mean(X):
    return np.mean(X, axis=0)

def std_dev(X, X_bar):
    sum=np.sum(np.square(X-X_bar),axis=0)
    return np.sqrt(sum)

def cov(X, X_bar, Y, Y_bar):
    return np.sum(((X-X_bar)*(Y-Y_bar)),axis=0)

HW = [bin(n).count("1") for n in range(0, 256)]

#array containing real traces previously saved
traces_array=np.load("path/to/traces.npy")
#array containing plaintexts previously saved
textin_array=np.load("path/to/plaintext.npy")

numtraces = np.shape(traces_array)[0] #total number of traces
numpoints = np.shape(traces_array)[1] #samples per trace

t_bar = np.sum(traces_array, axis=0)/len(traces_array)
o_t = np.sqrt(np.sum((traces_array - t_bar)**2, axis=0))

cparefs = [0] * 16 #put your key byte guess correlations here
bestguess = [0] * 16 #put your key byte guesses here

for bnum in range(0, 16):
    maxcpa = [0] * 256
    for kguess in range(0, 256):
        hws = np.array([[HW[aes_internal(textin[bnum],kguess)] for
    textin in textin_array]]).transpose()

        hws_bar = mean(hws)
        o_hws = std_dev(hws,hws_bar)
        cov_xy = cov(traces_array,t_bar,hws,hws_bar)
        cpaoutput = cov_xy/(o_t*o_hws)
        cpaoutput[np.isnan(cpaoutput)] = 0
        maxcpa[kguess] = max(abs(cpaoutput))

```

```
48      bestguess[bnum]=np.argmax(maxcpa)
49      cparefs[bnum]=max(maxcpa)
50
51  print("Best Key Guess: ", end="")
52  for b in bestguess: print("%02x " % b, end="")
53  print("\n", cparefs)
```

Appendix E

Lascar integration

E.1 with Chipwhisperer

This is the working script to use Lascar to capture traces from CW and analyse them.

```
import numpy as np
import chipwhisperer as cw
from lascar import (
    AbstractContainer,
    Trace,
    CpaEngine,
    Session,
    MatPlotLibOutputMethod,
    ConsoleOutputMethod,
)
from lascar.tools.leakage_model import hamming_weight
from lascar.tools.aes import sbox

# the Oscilloscope:
class ChipwhispererSetup:
    def __init__(self,key=[40]*16, platform=None, ss_ver=None, path=
    None):

        self.key=bytes(key)

        #initialize the chipwhisperer
        try:
            if not scope.connectStatus:
                scope.con()
        except NameError:
            scope = cw.scope()
```

```
26          try:
27              if ss_ver == "SS_VER_2_1":
28                  target_type = cw.targets.SimpleSerial2
29              elif ss_ver == "SS_VER_2_0":
30                  raise OSError("SS_VER_2_0 is deprecated. Use
    SS_VER_2_1")
31              else:
32                  target_type = cw.targets.SimpleSerial
33          except:
34              ss_ver="SS_VER_1_1"
35              target_type = cw.targets.SimpleSerial
36          try:
37              target = cw.target(scope, target_type)
38          except:
39              print("INFO: Caught exception on reconnecting to target -
     attempting to reconnect to scope first.")
40              print("INFO: This is a work-around when USB has died
    without Python knowing. Ignore errors above this line.")
41              scope = cw.scope()
42              target = cw.target(scope, target_type)
43
44          print("INFO: Found ChipWhisperer")
45
46          if "STM" in platform or platform == "CWLITEARM" or platform
    == "CWNANO":
47              prog = cw.programmers.STM32FProgrammer
48          elif platform == "CW303" or platform == "CWLITEXMEGA":
49              prog = cw.programmers.XMEGAProgrammer
50          elif "neorv32" in platform.lower():
51              prog = cw.programmers.NEORV32Programmer
52          else:
53              prog = None
54
55          self.scope=scope
56          self.target=target
57          self.prog=prog
58          self.platform=platform
59          self.path=path
60
61      def default_setup(self):
62          try:
63              self.scope.default_setup()
64              #set here your configuration if you don't want the
    default
65              self.scope.adc.samples=3200
66          except:
67              print("ERROR: default setup")
68
69      def flash_device(self):
```

```
70          #program the device with the appropriate firmware
71          try:
72              cw.program_target(self.scope, self.prog, self.path)
73          except:
74              print("ERROR: programming target")
75
76      def get_trace(self):
77          # return the side-channel leakage
78          ktp = cw.ktp.Basic()
79          text = ktp.next_text()
80          trace = cw.capture_trace(self.scope, self.target, text, self.
    key)
81          return trace #This trace is namedtuple of four pieces of data
     (wave, textin, textout, key)
82
83 # Then the AbstractContainer:
84 class AcquisitionSetupContainer(AbstractContainer):
85      def __init__(self, number_of_traces, key=[40]*16, platform=None,
    ss_ver=None, firmware_path=None):
86          """
87          :param number_of_traces: how many traces for the lascar
    container
88          :param key: key to be used by the subbytes function. Default
    is [40]*16
89          :param platform: needed by the CW. It identifies the base
    board and the target device (es. CW308_STM32F4)
90          :param ss_ver: needed by the CW. It identifies the version of
     the SimpleSerial protocol
91          :path firm_path: path of the firmware to load into the target
     board
92          """
93          self.key=key
94          self.platform=platform
95          self.ssver=ss_ver
96          self.path=firmware_path
97
98          self.oscilloscope = ChipwhispererSetup(self.key, self.
    platform, self.ssver, self.path)
99          self.oscilloscope.default_setup()
100
101         #flash only the first time then comment the line
102         self.oscilloscope.flash_device()
103
104         # This part is lascar dependent:
105         # We define what will be the type of the value generated at
    each trace
106         # (here, 16 bytes of plaintext + 16 bytes of key)
107         value_dtype = np.dtype(
```

```
108             [("plaintext", np.uint8, (16,)), ("key", np.uint8, (16,))
    ,]
109         )
110         self.value = np.zeros((), value_dtype)
111         self.value["key"] = key
112         AbstractContainer.__init__(self, number_of_traces)
113
114     def generate_trace(self, index):
115         """
116         generate_trace is the only method needed by
    AcquisitionSetupContainer to work properly
117         :param index: index of trace, not used here.
118         :return:  Trace with leakage from oscilloscope, and value
    from the dut plaintext/ciphertext
119         """
120
121         self.logger.debug(
122             "Generate trace %d", index
123         )  # The container logger can be used!
124
125         CW_trace = self.oscilloscope.get_trace()
126
127         leakage=CW_trace.wave    #on CW Trace.wave return the array of
     power trace captured
128         self.value["plaintext"]=CW_trace.textin
129         return Trace(leakage, self.value)
130
131 if __name__ == "__main__":
132
133     #fill here with the code presented in the Lascar and CW section
    and the script will work
```

E.2 with Rainbow

This is the working script to use Lascar to capture traces from Rainbow and analyse them.

```
1 import numpy as np
2 from rainbow.generics import rainbow_arm
3 from lascar import (
4     AbstractContainer,
5     Trace,
6     CpaEngine,
7     Session,
8     MatPlotLibOutputMethod,
9     ConsoleOutputMethod,
```

```
)
from lascar.tools.leakage_model import hamming_weight
from lascar.tools.aes import sbox

class RainbowContainer(AbstractContainer):
    """
    RainbowSubBytesContainer is a class that will define a lascar container using a rainbow device.
    """

    def __init__(
        self, number_of_traces, key=[40] * 16, noise=2, binary_file=None, key_func=None, encr_func=None, **kwargs
    ):
        """
        :param number_of_traces: how many traces for the lascar container
        :param key: key to be used by the subbytes function. Default is [40]*16
        :param noise: Noise added to the simulated leakage
        :param binary_file: the file used by rainbow. Must be an arm compiled .elf file containing a "subbytes" function, MUST BE SET
        :param key_func: the key schedule function name inside the binary file, MUST BE SET
        :param encr_func: the encrypt function name inside the binary file, MUST BE SET
        """

        # Initialize rainbow device:
        self.device = rainbow_arm(sca_mode=True)
        self.device.load(binary_file)
        # This part is lascar dependent:
        # We define what will be the type of the value generated at each trace
        # (here, 16 bytes of plaintext + 16 bytes of key)
        value_dtype = np.dtype(
            [("plaintext", np.uint8, (16,)), ("key", np.uint8, (16,))
,]
        )
        self.value = np.zeros((), value_dtype)
        self.value["key"] = key
        #the name of the two function to simulate
        self.func_1=key_func
        self.func_2=encr_func
        self.noise = noise
        AbstractContainer.__init__(self, number_of_traces, **kwargs)

    def generate_trace(self, idx):
        # We prepare the value for this trace:
```

```
50          self.value["plaintext"] = np.random.randint(0, 256, (16,), np
    .uint8)
51          self.device.reset()  # device must be reset/parametred at
    each function execution
52
53          key_addr = 0xDEAD0000
54          self.device[key_addr] = bytes(self.value["key"])
55          rk_addr = 0xDEAD1000
56          self.device[rk_addr] = bytes(self.value["key"])
57
58          # CM3_1T_AES_128_keyschedule_enc(rk, key)
59          self.device["r0"] = rk_addr
60          self.device["r1"] = key_addr
61          self.device["lr"] = 0xdeac
62          self.device.start(self.device.functions[self.func_1] | 1, 0
    xdeac)
63
64          self.device.trace_reset()   #reset the array "
    sca_values_trace"
65          buf_in = 0xDEAD2000
66          buf_out = 0xDEAD3000
67          self.device[buf_in] = bytes(self.value["plaintext"])
68          self.device[buf_out] = b"\x00" * 16  # Need to do this so
    this buffer is mapped into unicorn
69
70          # CM3_1T_AES_encrypt(rk, buf_in, buf_out, rounds)
71          self.device["r0"] = rk_addr
72          self.device["r1"] = buf_in
73          self.device["r2"] = buf_out
74          self.device["r3"] = 0xa #number of rounds
75          self.device["lr"] = 0xdeac
76          self.device.start(self.device.functions[self.func_2] | 1, 0
    xdeac)
77
78          #generate trace values and add noise to make them similar to
    real power traces
79          leakage= self.device.sca_values_trace + np.random.normal(0,
    self.noise, (len(self.device.sca_values_trace)))
80          return Trace(leakage, self.value)
81
82  if __name__ == "__main__":
83
84      #fill here with the code presented in the Lascar and Rainbow
    section and the script will work
```

Bibliography

[1] Mark Randolph and William Diehl. «Power side-channel attack analysis: A review of 20 years of study for the layman». In: *Cryptography* 4.2 (2020), p. 15 (cit. on p. 4).

[2] Paul Kocher, Joshua Jaffe, and Benjamin Jun. «Differential power analysis». In: *Annual international cryptology conference*. Springer. 1999, pp. 388–397 (cit. on pp. 4, 5).

[3] Eric Brier, Christophe Clavier, and Francis Olivier. «Correlation power analysis with a leakage model». In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2004, pp. 16–29 (cit. on pp. 6, 8).

[4] Hassen Mestiri, Noura Benhadjyoussef, Mohsen Machhout, and Rached Tourki. «A comparative study of power consumption models for cpa attack». In: *International Journal of Computer Network and Information Security* 5.3 (2013), p. 25 (cit. on pp. 8, 9).

[5] Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. «Power and electromagnetic analysis: Improved model, consequences and comparisons». In: *Integration* 40.1 (2007), pp. 52–60 (cit. on p. 9).

[6] Hongying Liu, Guoyu Qian, Satoshi Goto, and Yukiyasu Tsunoo. «AES key recovery based on Switching Distance model». In: *2010 Third International Symposium on Electronic Commerce and Security*. IEEE. 2010, pp. 218–222 (cit. on p. 9).

[7] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. *Advanced Encryption Standard (AES)*. en. Nov. 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197 (cit. on p. 9).

[8] Owen Lo, William J Buchanan, and Douglas Carson. «Power analysis attacks on the AES-128 S-box using differential power analysis (DPA) and correlation power analysis (CPA)». In: *Journal of Cyber Security Technology* 1.2 (2017), pp. 88–107 (cit. on p. 13).

[9] NewAE Technology Inc. *CW1200 ChipWhisperer-Pro*. [Online: 30-December-2022]. URL: https://rtfm.newae.com/Capture/ChipWhisperer-Pro/ (cit. on p. 13).

[10] NewAE Technology Inc. *CW308 UFO*. [Online: 30-December-2022]. URL: https://rtfm.newae.com/Targets/CW308%20UFO/ (cit. on p. 13).

[11] NewAE Technology Inc. *CW308T-STM32F*. [Online: 30-December-2022]. URL: https://rtfm.newae.com/Targets/UFO%20Targets/CW308T-STM32F/ (cit. on p. 13).

[12] NewAE Technology Inc. *ChipWhisperer API Documentation*. [Online: 30-December-2022]. URL: https://chipwhisperer.readthedocs.io/en/latest/index.html#api (cit. on p. 14).

[13] Ledger Donjon. *Rainbow*. [Online: 30-December-2022]. URL: https://github.com/Ledger-Donjon/rainbow (cit. on p. 14).

[14] Unicorn engine. *Unicorn-Engine*. [Online: 30-December-2022]. URL: https://www.unicorn-engine.org/ (cit. on p. 14).

[15] Ledger Donjon. *LASCAR*. [Online: 30-December-2022]. URL: https://github.com/Ledger-Donjon/lascar (cit. on p. 14).

[16] Alex Dewar, Jean-Pierre Thibault, and Colin O'Flynn. *NAEAN0010: Power Analysis on FPGA Implementation of AES Using CW305 & ChipWhisperer*. 2020 (cit. on p. 18).

[17] kokke. *Tiny-AES*. [Online: 20-January-2023]. URL: https://github.com/kokke/tiny-AES-c (cit. on p. 25).

[18] Free Software Foundation. *GCC Optimization Options*. [Online: 23-February-2023]. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options (cit. on p. 26).

[19] NumPy. *NumPy Python library*. [Online: 20-January-2023]. URL: https://numpy.org/ (cit. on p. 26).

[20] jnk0le. *Cortexm-AES*. [Online: 24-January-2023]. URL: https://github.com/jnk0le/cortexm-AES (cit. on p. 34).

[21] Ledger Donjon. *VisPlot*. [Online: 26-January-2023]. URL: https://github.com/Ledger-Donjon/visplot (cit. on p. 45).

[22] VisPy developers. *VisPy*. [Online: 26-January-2023]. URL: https://vispy.org/ (cit. on p. 45).

[23] The Matplotlib development team. *Matplotlib*. [Online: 26-January-2023]. URL: https://matplotlib.org/ (cit. on p. 45).