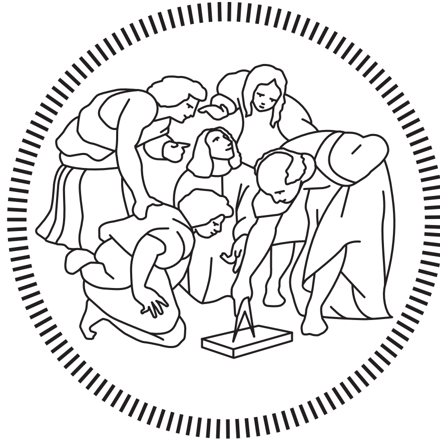
Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria



Achieving low-cost side channel attacks via quantitative information leakage analysis

Relatore: Prof. Alessandro Barenghi

Tesi di Laurea di:
Alain Federico Carlucci Matr. 862746

Anno Accademico 2017–2018

To my grandparents,
Ai miei nonni.

Ringraziamenti

Il ringraziamento principale va al mio relatore, *prof. Alessandro Barenghi*, per avermi offerto la possibilità di lavorare a questo progetto, per l'immensa conoscenza trasmessami e per avermi aiutato - grazie alle sue intuizioni e ad una pazienza smisurata - a concretizzare un'idea che altrimenti sarebbe rimasta un sogno irrealizzato.

Ringrazio poi i miei genitori, nonni, fratelli e tutti i familiari che mi sono stati vicino e mi hanno sostenuto in questi anni di studio, dandomi fiducia e coraggio a non arrendermi mai.

Inoltre, non posso evitare di nominare le associazioni studentesche *POuL*, *Tower of Hanoi*, *Skyward Experimental Rocketry* e la stanza virtuale `#polimi@irc.azzurra.org`. Tutte poli d'attrazione per persone favolosamente nerd che hanno costantemente stimolato la mia creatività e il mio intuito.

Segue un elenco alfabetico di persone che meritano di essere nominate individualmente per avermi supportato (e sopportato) di loro spontanea volontà in tutti questi anni: Alessandro DF, Alex P, Antonio S, Armando B, Daniele I, Davide B, Davide O, Edoardo C, Federico T, Filippo C, Gabriele F, Giulio DP, Luca C, Mario P, Matteo F, Matteo P, Michele A, Nicola P, Noemi F, Pietro DN, Riccardo B, Silvano S, Stefano M.

Grazie.

Sommario

La correttezza matematica degli algoritmi crittografici non è sufficiente ad assicurarne le proprietà garantite. Infatti, la loro implementazione può essere vulnerabile ad attacchi *side-channel*, ossia metodi noti in letteratura per ottenere informazioni riguardanti la chiave di cifratura sfruttando informazioni esfiltrate durante l'esecuzione dell'algoritmo. I primi attacchi - effettuati più di venti anni fa - hanno permesso il recupero di chiavi private situate in smart card e microcontrollori. Nel tempo sono state sviluppate tecniche più sofisticate, arrivando a permettere il recupero di chiavi private GPG effettuando una singola firma digitale su piattaforma x86. Mentre le contromisure sviluppate per evitare questo tipo di attacchi sono state efficaci, una variante chiamata Differential Power Analysis (DPA) ha iniziato - e continua tutt'oggi - a mietere vittime. Quest'ultima tipologia permette di attaccare sistemi di cifratura richiedendo solamente che nel consumo di corrente della CPU vi sia una componente correlata ai valori in elaborazione al momento.

La prima parte di questo lavoro mostra un metodo innovativo per identificare quali componenti spettrali permettano attacchi DPA. Dimostriamo che, con l'ausilio di questi dati, è possibile effettuare il recupero di una chiave di cifratura AES misurando le emissioni elettromagnetiche della CPU con una Software Defined Radio (SDR).

Implementare sistemi di crittografia sicuri da attacchi side-channel, ad oggi, risulta arduo e l'implementazione ottenuta è strettamente dipendente dall'architettura del processore su cui verrà eseguito l'algoritmo. È necessario ottenere un modello di esecuzione delle istruzioni nella CPU, quando le specifiche pubbliche di questa non sono disponibili. Nella seconda parte di questo lavoro mostriamo come è possibile modificare una tecnica per effettuare la ricostruzione dell'architettura della CPU in modo che sia completamente automatizzata ed eseguibile utilizzando unicamente una linea seriale di comunicazione invece di oscilloscopi da laboratorio. Mostriamo inoltre come affrontare i problemi generati dalla latenza del protocollo, la quale è elevata e molto variabile.

Abstract

The mathematical correctness of a cryptosystem is not sufficient to achieve confidentiality. This is because an implementation could be vulnerable to side-channel attacks: a set of techniques - already known in literature - to obtain information regarding the encryption key by exploiting information leaks during the algorithm execution. First attacks were made more than twenty years ago and allowed to recover keys stored in smartcards and microcontrollers. More advanced techniques were developed during these years, which allowed, for example, to obtain a GPG private key by analyzing the electro-magnetic emissions of a x86 CPU while performing a digital signature.

The countermeasures developed to avoid these attacks were effective, but at the same time a new kind of side-channel attack called Differential Power Analysis (DPA) started claiming victims. Modern DPA techniques allow key-recovery attacks to cryptosystems by observing the power consumption of a CPU as long as it has a correlation with one of the values that are currently processed.

The first part of this work shows an innovative method to identify which spectral components allow DPA attacks. Using this information, we demonstrate the feasibility of a DPA attack using exclusively a Software Defined Radio (SDR).

Writing side-channel aware code is difficult and target-dependent. To do that, an execution model of the code as seen from an Instruction Set Architecture (ISA) point of view is needed and this task recently gained traction in the community.

The second part of this work shows how to automate an existing technique to perform reverse-engineering of the architecture of a CPU without using an oscilloscope. In this context, we have developed a technique to minimize UART jittering to the point that it becomes negligible and it does not interfere with the analysis.

Contents

1 Introduction **1**

1.1 Motivation 3

1.2 Results 5

2 Background **7**

2.1 Fundamentals of Signal Processing 7

2.1.1 Digital Signal Processing 15

2.1.2 Amplitude Modulation 20

2.1.3 In-Phase/Quadrature Demodulation 23

2.1.4 Signal reconstruction 25

2.2 Side-Channel Attacks 28

2.2.1 Simple Power Analysis 29

2.2.2 Correlation Power Analysis 30

2.3 Trace warping 32

2.3.1 Dynamic Time Warping 32

2.3.2 Fast Dynamic Time Warping 35

3 A DSP Approach to DPA-oriented signal processing **37**

3.1 Characterization of leakage in frequency domain 37

3.1.1 Frequency-oriented leakage model 37

3.1.2 Validation 39

3.1.3 AM-like leakage detection 48

3.2 DSP Pipeline for Side-Channel Attacks 49

3.2.1 Elastic-Alignment Pipeline 50

3.3 Attacking with a SDR 51

3.3.1 SDR Acquisition pipeline 52

3.3.2 Signal reconstruction on a different carrier 53

4 Architectural Reverse Engineering 57

4.1 Inferring architecture using CPI-index 58

4.2 From the oscilloscope to UART 59

4.3 UART Jitter Correction 61

4.4 Implementation 62

5 Experimental results 65

5.1 Experimental Workbench 65

5.1.1 Device 65

5.1.2 Workbench 66

5.1.3 SDR 67

5.1.4 Filtering software 69

5.2 A DSP Approach to DPA-oriented signal processing 69

5.2.1 Leakage Model 70

5.2.2 Robustness against clock frequency shift 73

5.3 Parameter characterization 78

5.3.1 Leakage map at 32 MHz 78

5.3.2 Leakage map at higher frequencies 81

5.3.3 Results 87

5.4 Attacks using SDR 88

5.4.1 Basic attack 88

5.4.2 Attacks at higher frequencies 90

5.5 Architectural Reverse Engineering 94

5.5.1 Public information 94

5.5.2 CPI Analysis 95

5.5.3 Results 100

6 Conclusions 103

Bibliography 105

List of Figures

2.1 A visual representation of the Fourier Transform 8
2.2 $rect(t)$ and its Fourier Transform $sinc(\omega)$ 10
2.3 $rect(t)$ and its Discrete Fourier Transform $sinc(\omega)$ 14
2.4 Rectangular window function and its spectral leakage. First sidelobes at $-13dB$ 16
2.5 Tukey window with $\alpha = 0.5$ 17
2.6 Magnitude and leakage of a Chebyshev window Type I, order 30 18
2.7 Magnitude of a Chebyshev window Type II, order 30 19
2.8 A signal is modulated using the Amplitude Modulation technique 20
2.9 AM demodulation using a phase-coherent demodulator 22
2.10 AM demodulation using the envelope detector 23
2.11 IQ Demodulation pipeline 24
2.12 Power variations observed while running an unprotected square and multiply algorithm. 29
2.13 CPA Pipeline 30

3.1 The DSP pipeline used to filter traces 49
3.2 The Elastic-Alignment Pipeline 50
3.3 Expected magnitude of a raw trace captured using the SDR 52
3.4 SDR Trace acquisition pipeline 53
3.5 Magnitude and Phase of the signal recostructed on a different carrier . . . 55

4.1 The pipeline used to automatically obtain CPI from a target board 60
4.2 UART RTT Jittering in μs, 460800 bps 62

5.1 The custom loop probe 67
5.2 Amplitude of a trace captured using the SDR, board clocked at 64 MHz. . 68

5.3 Screenshot of the DSP tool 69
5.4 Window function found by the linear-scan tool, Cortex-M7 clocked at 32 MHz 70
5.5 Window function found by the dichotomic-scan tool, Cortex-M7 clocked at 32 MHz 71
5.6 Window function found by the genetic tool, Cortex-M7 clocked at 32 MHz 72
5.7 Window function found by applying a slice-wise AND, Cortex-M7 clocked at 32 MHz 72
5.8 Window function found by the linear-scan tool, Cortex-M7 clocked at 64 MHz 73
5.9 Window function found by the dichotomic-scan tool, Cortex-M7 clocked at 64 MHz 74
5.10 Window function found by the genetic tool, Cortex-M7 clocked at 64 MHz 75
5.11 Window function found by applying a slice-wise AND, Cortex-M7 clocked at 64 MHz 75
5.12 Window function found by the linear-scan tool, Cortex-M7 clocked at 128 MHz 76
5.13 Window function found by the dichotomic-scan tool, Cortex-M7 clocked at 128 MHz 76
5.14 Window function found by the genetic tool, Cortex-M7 clocked at 128 MHz 77
5.15 Window function found by applying a slice-wise AND, Cortex-M7 clocked at 128 MHz 77
5.16 3D-plot and heatmap, $f_{clk} = 32$ MHz, $\Delta f \in [-10, 10]$ MHz 79
5.17 3D-plot and heatmap, $f_{clk} = 32$ MHz, $\Delta f \in [-1, 1]$ MHz 79
5.18 3D-plot and heatmap, $f_{clk} = 32$ MHz, $\Delta f = [0, 1]$ MHz, double-sideband . 80
5.19 3D-plot and heatmap, $f_{clk} = 64$ MHz, $\Delta f \in [-5, 5]$ MHz, restricted to the leaking area 82
5.20 3D-plot and heatmap, double-sideband, $f_{clk} = 64.1$ MHz, $\Delta f \in [0, 3]$ MHz 83
5.21 3D-plot and heatmap, $f_{clk} = 64$ MHz, $f \in [0, 250]$ MHz; plot restricted to zones with leakage. 84
5.22 3D-plot and heatmap, $f_{clk} = 128$ MHz, $f \in [0, 250]$ MHz restricted to $f_{clk} \pm 75$ MHz 84
5.23 3D-plot and heatmap, $f_{clk} = 128$ MHz, $\Delta f \in [-50, 50]$ MHz 86
5.24 3D-plot and heatmap, double-sideband, $f_{clk} = 128$ MHz, $\Delta f \in [0, 10]$ MHz 86

5.25 Results table which maps clock frequency to frequency and bandwidth of the leakage 87
5.26 Expected leakage zone with respect to CPU clock frequency 87
5.27 GQRX tuned to the clock frequency, when $f_{clk} = 31.80$ MHz 88
5.28 CPA attack using SDR without filtering ($f_{clk} = 32$ MHz) 89
5.29 CPA attack using SDR with filters applied ($f_{clk} = 32$ MHz) 89
5.30 GQRX tuned to the clock frequency, when $f_{clk} = 63.9$ MHz 90
5.31 CPA attack using SDR without filtering ($f_{clk} = 64$ MHz) 91
5.32 CPA attack using SDR with filtering ($f_{clk} = 64$ MHz) 91
5.33 GQRX tuned to the clock frequency, when $f_{clk} = 127.5$ MHz 92
5.34 CPA attack using SDR without filtering ($f_{clk} = 128$ MHz) 93
5.35 CPA attack using SDR with filtering ($f_{clk} = 128$ MHz) 93
5.36 Reverse-engineered ARM Cortex-M7 internal structure 102

List of Tables

3.1 Scoring functions 40

5.1 Comparison between CPI without(a) and with(b) RAW hazards 96
5.2 Various CPI w/ and w/o RAW hazards 97
5.3 Load/Store CPI 98
5.4 Cortex-M7 Unit Latencies reported in the GCC Back-end 98
5.5 Reverse-engineered ARM Cortex-M7 pipeline schematics 99
5.6 Cortex-M7 dual issue table 100

List of Algorithms

2.3.1 Dynamic Time Warping 34
2.3.2 Build Path from Cost Matrix 35
2.3.3 Time-Series Warping Algorithm 35
2.3.4 FastDTW 36

3.1.1 MERGE function used in the bisection algorithm 43
3.1.2 BISECTIONFINDFILTER, search the best filter using bisection 44
3.1.3 Genetic FindFilter 47
3.3.1 Modified UPMIX algorithm. It performs $IQ \rightarrow RF$ modulation on a different carrier 54

4.4.1 Benchmark (Host) 63
4.4.2 Benchmark (Board) 63

Chapter 1

Introduction

Cryptography is the study of techniques and methods for sending and receiving secrets, so that unauthorized people or systems cannot access them. It began thousands of years ago and nowadays can be split in classic cryptography and modern cryptography. The latter is based on the Kerckhoffs' principles[15], stating that the method used to encipher data is known to the opponent, and that security must lie in the choice of key; any currently used modern cryptographic algorithm is designed around computational hardness assumptions. Modern cryptography is the intersection of mathematics, computer science, electrical engineering, communication science and physics; applications of that include electronic commerce, digital currencies, computer password and military communication. Cryptosystems can be divided in two categories: symmetric and asymmetric ciphers; both have been designed and analyzed by considering the primitives as ideal mathematical objects; this means that they are immune to all known theoretical attacks (e.g. symmetric cryptosystems are analyzed with linear and differential cryptanalysis), but not to the attacks which exploit the hardware implementation. Modern cryptosystems are often implemented on embedded systems, which are systems designed for special-purpose applications usually running on an ad-hoc hardware platform (including mechanical parts) and often with real-time computing constraints. Embedded systems control many devices in common use today. Ninety-eight percent of all microprocessors are manufactured components of embedded systems. Since the first iPhone (2007), the market of Smartphones started growing and the sales almost topped 1 billion units in Q3 2012 [3]. The main operating systems running on nowadays smartphones are iOS (by Apple) and Android (by Google). As of 2017, over than 100 billion ARM processors have been produced. Almost all smartphones and a wide range of embedded

devices (75% in 2005 [1]) run on ARM CPUs. ARM is a RISC Instruction Set Architecture (ISA) and the CPUs implementing it are praised for their power-efficiency. Modern ARM CPUs have the Thumb-2 instruction-set, which is variable-length and consists in a set of 16-bit instructions used to achieve high code-density, with additional 32-bit instructions allowing performance comparable to the original ARM instruction-set. The security of personal and business information now stored on smartphones has become increasingly important nowadays. Due to their usage, smartphones collect an increasing amount of sensitive and private information regarding their users' work and private life. The access to these information must be controlled to protect the privacy. On the other hand, even embedded devices carry out sensitive information, such as hard-coded encryption keys or, because of the typical "security-by-obscurity" design, the leakage of any information regarding the internal structure might compromise the entire security model.

Side-Channel Attacks A new class of attacks has been developed in recent years[17], which aims to break cryptosystems or leak information from embedded devices by exploiting the so-called Side-Channel Attacks (SCA): information leakages coming from physical parameters of the actual hardware implementation. Nowadays SCA is the most effective way to break a theoretically-secure cryptosystem. They usually exploit differences in the time required to perform some operations (e.g. in cryptography, the primitives), the power consumption of the device or the electromagnetic (EM) emissions, which are roughly proportional to the power consumption. The easiest - and the first known - side-channel attack is known as Simple Power Analysis (SPA)[18], which works if there is a dependence between which instruction being executed and the data that is going to be processed (or sometimes the bits of the encryption key). For example, the core decryption/sign operation of the asymmetric cryptosystem "RSA" is a modular exponentiation where the exponent is the private key. This operation is implemented in software or hardware by the use of an algorithm called "Square & Multiply", which loops for each bit of the exponent and executes a squaring of an accumulator if this bit is zero, or a squaring and a multiplication if this bit is one. An unprotected, naive implementation of this algorithm makes the multiplication happen only if the corresponding bit of the key is one. A protected implementation, instead, makes it happen every time, even if in this way the execution time of the algorithm is slower. The problem with the unprotected implementation is that, the power consumption of the multiplication is different than the one of the squaring, hence if the power consumption of the whole

device is drawn on a chart, the encryption key appears as a binary wave, where a low-power slice corresponds to a zero and a low-power/high-power transition corresponds to a one. There is the Differential Power Analysis (DPA) [18], which consists the collecting a batch of power traces from the device while it is encrypting and is using the key, partition them in two subsets depending of the hypothesis we are interested in and then use it to validate a guess on the value of a portion of the secret key. The Correlation Power Analysis (CPA) is an improvement of the DPA and is the one we used in this work. The latter is a ciphertext-only attack which uses statistical correlation while processing all the traces to validate (as in DPA) a guess about a portion of the key. The last class of attacks are the so-called "Template Attacks" which proceeds in two phases: a *Training Phase* which needs a clone of the attacked system completely in control of the attacker so that they can profile the power dissipation for each subkey; an *Attack Phase* where the attacker can proceed to measure the power consumption during encryption with the secret key and match the measurements with one of the models.

1.1 Motivation

During these years, SCAs have improved their capabilities: they started more than twenty years ago by allowing to crack smart-cards and small microcontrollers, and allowed in these years to break encryption running on small single-core and dual-core CPUs. Recently, even x86 CPUs were attacked. Thanks to SPA it was possible to recover the GPG private key by analyzing the EM emission of a CPU while encrypting or signing a single message [13]. SPA protections developed in the past years have been solid and solved completely the problem; DPA and CPA instead continued claiming victims[22][2]. Encryption algorithms are analyzed by considering them as ideal mathematical constructs and Advanced Encryption Standard (AES), for example, is strongly believed to be mathematically sound and completely secure. SCA instead exploit the physical implementation of a given cryptosystem.

Side-Channel countermeasures have been developed since the first attack. The goal is to make the power consumption of the cryptographic device independent of the intermediate values of the executed cryptographic algorithm[20]. The countermeasures published so far can categorized into two groups: masking and hiding. The former consists in randomizing the intermediate values that are processed by the device; the latter, instead, consists in processing the same intermediate results as an unprotected implementation but the resistance is achieved by altering the power consumption characteristics

of the cryptography device. The advantage of masking is that the power consumption characteristics of the device do not need to be changed. Writing side-channel aware code after considering all leakages in a given Central Processing Unit (CPU) is difficult and target-dependent. Not taking into account the CPU architecture, instead, may invalidate the effectiveness of the countermeasures. This is because every CPU model has a different internal architecture, which means different internal buffers[2]. These buffers, if not known before creating the countermeasure, may leak even when a generic countermeasure is applied. Building a side-channel aware CPU execution-model is a task that recently gained interest in the community: past works shown how to reverse the CPU architecture by analyzing the Cycles Per Instruction (CPI) indicator of some instructions. This technique works but it is quite expensive in terms of time spent and human interaction while analyzing the target. There is the need to develop a set of tools to automatize the process of reversing the CPU architecture in a way that is feasible and not expensive to do it in scale. These tools could help to build countermeasures even if CPU details and information are not be released due to intellectual property (e.g. Intel, ARM).

Nowadays a typical Side-Channel setup is made using an expensive oscilloscope who has a large bandwidth and a set of expensive calibrated antennas. An alternative setup could be made using Software Defined Radio (SDR) receiver, which is a radio communication system where the components usually made in hardware are implemented in software. Nowadays there are cheap SDR receivers made using an USB dongle which performs in hardware signal amplification, up-mixing, digital-to-analog conversion and low-pass filtering. There are SDR hardware dongles of every kind, we are using the `rad1o badge`, a HackRF clone whose bandwidth is 20 MHz. It could be found on eBay for less than 200 euros. To reduce the costs of a SCA setup, the oscilloscope could be replaced with an SDR, even if this brings up a new set of challenges. Signal locking, limited bandwidth, poor signal-to-noise ratio and frequency drifting are some examples of that. At first we must understand where is the leakage information placed, if it is highly packed and is feasible to use an SDR or if it is too widespread and the SDR bandwidth doesn't allow any attack. There is also no way to start/stop the capture aligned to the target device, hence we should also find a way to realign all the traces. This brings up the problem to recognise the leakage waveform through the noise.

1.2 Results

With this work we will develop a set of tools which will allow us to discover how the leakages are spread around the spectrum, which part of it leaks information and how much information is leaked for each slice of it; we found a correlation between CPU clock frequency and the location of the leakage. All the measurements will be done using at first the oscilloscope and then the SDR, allowing us to perform full attacks with just that. With each attack we will recover the encryption key, having the board clocked from 32 MHz up to 128 MHz. We will try multiple Digital Signal Processing (DSP) pipelines to convert the In-Phase/Quadrature (IQ) samples to a convenient format, which will be used to improve the attacks by filtering out the noise and correcting clock drifting. The SDR does not allow automatic realignment of the collected traces; to solve this problem, we will use a new algorithm to realign traces collected by exploiting a custom execution pattern. We will show the improvements that happen while performing full attacks (which will recover the encryption key) using the SDR both with and without filtering. To reduce the size of the traces and speed-up the attacks, we will show a leakage-invariant transform which reconstructs the original signal on a different carrier; with the latter, the size of the traces will stop depending on the clock frequency.

We also will provide an innovative way to perform architecture reversing without any human intervention. It consists in a software which drives a device, measures the CPI indicator for any combination of a chosen set of instructions and builds reports with all the results. The target device should just be connected using the Universal Asynchronous Receiver-Transmitter (UART) serial interface to the host machine, which obtains each measure in less than 10 seconds and a full analysis requires few hours. The host machine runs hundreds of tests with each combination of these instructions by building - for each set of instructions - a custom firmware. As a last step, this software fills a table with all the results. We developed a way to remove the UART jitter and the host jitter so that the results are stable enough to allow the reverse-engineering.

Chapter 2

Background

This chapter aims to give the background knowledge needed to understand the contributions of this work. It is divided in three sections. The first, *Fundamentals of Signal Processing* introduces some basic notions about the Fourier Transform, Digital Signal Processing and the various modulation and demodulation techniques used during this work. This section aims to give a brief recap, we refer the interested reader to Prati [24], course-book for the Digital Signal Processing (DSP) class in *Politecnico di Milano*, which contains a better explanation, all the theorems, properties and proof of each concept named here. The second section, *Side-Channel Attacks*, recaps the state of the art of Side-Channel Attacks and Countermeasures. It is focused on Simple Power Analysis (SPA) and Correlation Power Analysis (CPA), which had been used extensively during this work. The third section shows some algorithms useful to perform trace alignment.

2.1 Fundamentals of Signal Processing

Signal Processing is a field of mathematics that concerns analysis, synthesis and modifications of signals. A generic *signal* could be defined as an observable change in a quantifiable entity, which may or may not contain any information. It is usually accompanied by the term *noise*, which represents any indesirable, more or less random disturbance. An important distinction in this field is between *discrete-valued* and *continuous-valued* signals. The former, which is often referred to as *time-series* in other fields, represent the signal as a finite series of numbers which can be seen like "points in time": the corresponding signal jumps from one value to the other as time moves on. The latter, instead, views variables as having a particular value for an infinitesimally

short amount of time: between any two points in time there are an infinite number of other points. In this case, time is also viewed as a continuous variable.

One of the most powerful tools in signal processing is the *Fourier Transform*, which allows to change the representation of a signal from points in time to sum of sines and cosines. This allows us to see and analyze a signal from a completely different point of view: the *Frequency Domain*. Some analysis, filters and modifications can be done there and, thanks to the *Inverse Fourier Transform*, the signal could be moved back again in the time representation.

Fourier Transform The Fourier Transform (FT) is a mathematical transform which decomposes a function of time into a sum of sines and cosines that make it up. The transform is applied to a complex function of time and the output is a complex function of frequencies: $f : \mathbb{C} \to \mathbb{C}$. It is defined only when both the time and the signal are continuous. In this work the FT is applied to real signals, hence the corresponding function of time has $Im(f(x)) = 0$. The Fourier Transform of a signal $f(\xi)$ is denoted as $\mathfrak{F}\{f(x)\}$ and the corresponding function in the frequency-domain is denoted as $\hat{f}(\xi)$.

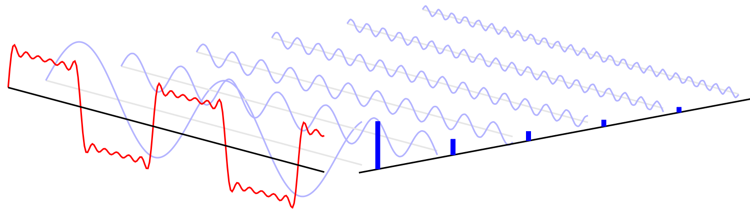


Figure 2.1: A visual representation of the Fourier Transform. The time-domain representation of the signal is drawn in red; the frequency-domain samples are in blue, whose corresponding sine-wave is drawn in lilac.

The Fourier Transform is defined by the following formula:

$$\hat{f}(\xi) = \mathfrak{F}\{f(x)\} = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} \, dx, \tag{2.1}$$

for any real number ξ. The variable x represents time, the variable ξ represents the frequency. $\hat{f}(\xi)$ is called *spectrum* of the signal. Using Euler's Formula we can rewrite the complex exponential as:

$$e^{-2\pi i x \xi} = \cos(-2\pi x \xi) + i \cdot \sin(-2\pi x \xi).$$

Sine and cosine are an orthogonal basis for the function space in $\mathbb{C}$. The Fourier Transform is an invertible transformation (under certain conditions, available for the interested reader at Folland [10]), hence given the spectrum of a signal we can reconstruct the original waveform in the time domain. The Inverse Fourier Transform (IFT) allows the reconstruction, and it is defined by the following formula:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i x \xi} \, dx, \tag{2.2}$$

for any complex number x. Figure 2.1 depicts both the time-domain representation - the approximate red-colored square-wave - and the frequency-domain representation - the blue histogram - of the same signal. The time-domain signal can be decomposed using the Fourier Transform to a set of sinusoidal *components*, which, if summed again using the Inverse Fourier Transform give back the original time-represented signal.

Before showing a schoolbook example of the Fourier Transform, we define two functions which will be used: $rect(x)$ and $sinc(x)$. The *Rectangular Function* depicted in Figure 2.2a is a real function also known as $rect(x)$, whose value is 0 outside some bounds (± 0.5) and 1 inside. It is defined as:

$$rect(x) = \begin{cases} 0 & \text{if } |t| > \frac{1}{2} \\ \frac{1}{2} & \text{if } |t| = \frac{1}{2} \\ 1 & \text{if } |t| < \frac{1}{2} \end{cases}$$

The *Cardinal Sine*, written as $sinc(x)$ and depicted in Figure 2.2b is a function defined as:

$$sinc(x) = \begin{cases} \frac{\sin(x)}{x} & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases}$$

The value at $x = 0$ can be found by computing the limit at 0 as:

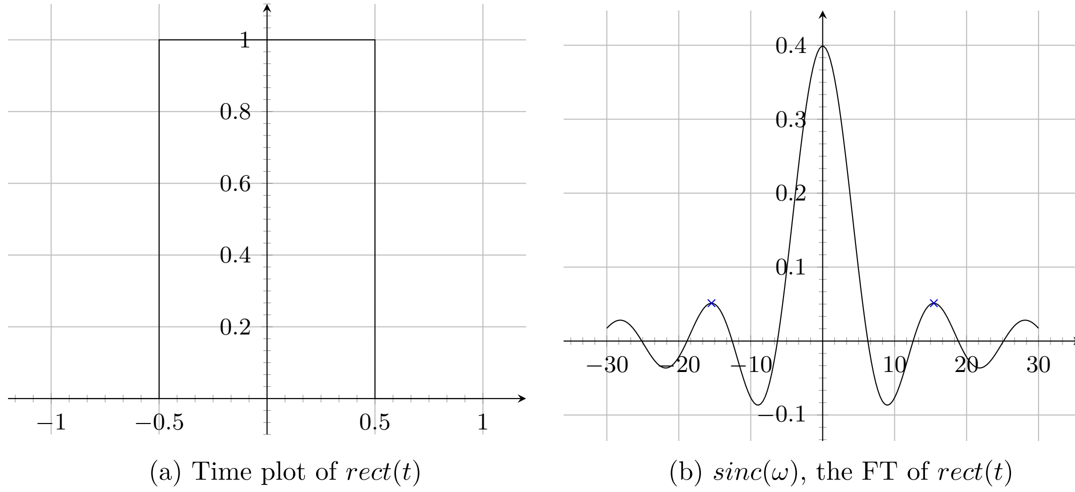
$$sinc(0) = \lim_{x \to 0} \frac{\sin(ax)}{ax} = 1 \quad \text{for all real } a \neq 0.$$

The Fourier Transform of $rect(t)$ is a $sinc(\xi)$ using ordinary frequencies or $\frac{1}{\sqrt{2\pi}} sinc(\frac{\omega}{2})$ using w as angular frequency: $w = 2\pi\xi$.

As example, we show the full derivation of it:

$$
\begin{aligned}
\mathfrak{F}\{rect(t)\} &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} rect(t) e^{-i\omega t} dt = \frac{1}{\sqrt{2\pi}} \int_{\frac{1}{2}}^{\frac{1}{2}} e^{-i\omega t} dt = -\frac{1}{\sqrt{2\pi}} \frac{1}{i\omega} \left[e^{-i\omega t} \right]_{-\frac{1}{2}}^{+\frac{1}{2}} \\
&= -\frac{1}{\sqrt{2\pi}} \frac{1}{i\omega} \left[e^{-i\frac{\omega}{2}} - e^{i\frac{\omega}{2}} \right] = \frac{1}{\sqrt{2\pi}} \frac{2}{\omega} \left[\frac{e^{i\frac{\omega}{2}} - e^{-i\frac{\omega}{2}}}{2i} \right] \\
&= \frac{1}{\sqrt{2\pi}} \frac{\sin(\frac{\omega}{2})}{\frac{\omega}{2}} = \frac{1}{\sqrt{2\pi}} sinc(\frac{\omega}{2})
\end{aligned}
\tag{2.3}
$$

Figure 2.2a depicts a plot of both the function $rect(t)$, whose amplitude is 1, and it is non-zero in the range from -0.5 to 0.5. The corresponding function in the frequency domain is shown in Figure 2.2b, where the zeros in the frequency domain are every $2\pi\xi$ with $\xi \neq 0, \xi \in \mathbb{Z}$.



(a) Time plot of $rect(t)$

(b) $sinc(\omega)$, the FT of $rect(t)$

Figure 2.2: $rect(t)$ and its Fourier Transform $sinc(\omega)$

Due to the properties of complex numbers, each component in the frequency domain has its own amplitude, which is obtained by calculating the magnitude of a complex number:

$$|\hat{f}(\xi)| = \sqrt{Re\{\hat{f}(\xi)\}^2 + Im\{\hat{f}(\xi)\}^2}\ ,$$

and a relative phase, which is calculated as the argument of the complex number:

$$\angle\{\hat{f}(\xi)\} = tan^{-1}\left(\frac{Im\{\hat{f}(\xi)\}}{Re\{\hat{f}(\xi)\}}\right).$$

Magnitude and *Phase* are the two main indicators used to analyze components of a signal

in frequency domain. The magnitude shows how strong a single component is, the phase shows what is the initial point of the sine-wave generated by that component.

Due to the properties of complex exponentials, the representation of a real signal in the frequency-domain has an interesting property:

$$|\hat{f}(-\xi)| = |\hat{f}(\xi)|$$

$$\angle\{\hat{f}(-\xi)\} = \angle\{\hat{f}(\xi)\} + \frac{\pi}{2}$$

The magnitude is symmetric and each component with a negative frequency has the angle shifted by 90 degrees of the corresponding positive component.

Sampling theorem The journey through Digital Filters and Digital Signal Processing techniques begins with the Nyquist-Shannon theorem, which states:

Theorem 1. *If a function $x(t)$ contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced $\frac{1}{2B}$ seconds apart.*

The *bandwidth* of a signal could be defined as the greatest component in the frequency-domain whose value is not zero. Sampling a signal whose bandwidth is greater than B Hertz using a sample rate lower than $\frac{1}{2B}$ introduces artifacts in the sampled signal. This phenomenon is called *aliasing*.

Discrete-Time Fourier Transform The Discrete-Time Fourier Transform (DTFT) is a transform that maps a discrete-time signal into a description in the frequency domain. The procedure to obtain a discrete-time signal given a continuous one is called *sampling* and it is implemented in the Analog-to-Digital Converters (ADCs): devices connected to a continuous analog signal which are able to sample it every some fixed instants of time (T) and convert it to a digital information, a number.

The mathematical formulation of the DTFT is the following:

$$X(\xi) = \sum_{t=-\infty}^{\infty} x[t]e^{-i\omega t} \tag{2.4}$$

The DTFT has an inverse transform which maps a function in the frequency domain back to the time domain. This transform is called Inverse Discrete-Time Fourier Transform

(IDTFT) and is defined as:

$$x[t] = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\xi) e^{i\omega t} \, d\omega \tag{2.5}$$

Given a continuous signal $x(t)$, the discretization is obtained by multiplying $x(t)$ by the repetition (with period T) of the ideal impulse.

$$x[t] = \sum_{n=-\infty}^{\infty} x(nT)\delta(t - nT) = x(t) \sum_{n=-\infty}^{\infty} \delta(t - nT) \tag{2.6}$$

Due to the Nyquist-Shannon theorem, the Discrete-Time Fourier Transform of a sampled signal is the Fourier Transform of part of the continuous signal, repeated with period of length $\frac{1}{T}$. If the continuous signal is bandwidth-limited, with maximum frequency f_{max}:

$$\hat{f}(f_{max} + \epsilon) = 0 \; \forall \epsilon > 0$$

and the bandwidth B is less than the sampling frequency $f_s = \frac{1}{T}$, then the DTFT shows the full frequency spectrum of the signal, because the spectral replicas do not overlap. In case of real signals, the Fourier Transform is symmetric. Hence, the spectral replicas do not overlap if the maximum frequency of the signal (f_{max}) is less than *half* the sampling frequency. This frequency limit f_{Ny} is called *Nyquist Frequency*, and will be explained extensively in the next subsection. For now it is useful only to know that given a real signal:

$$f_{max} < \frac{1}{2T} = \frac{f_s}{2} = f_{Ny}.$$

If this condition holds, the Fourier Transform of a sampled signal is obtained by calculating the DTFT and then removing all the replicas (except for the one in the baseband) by using a low-pass filter, which will be explained later in this chapter.

Discrete Fourier Transform The Discrete Fourier Transform (DFT) is a transform that maps a finite sequence of equally-spaced samples of a function into a sequence of coefficients of a linear-combination of complex sine-waves.

$$X[\xi] = \sum_{t=0}^{N-1} x[t] e^{-i\xi \frac{2\pi}{N} t} \qquad \xi = 0, 1, \cdots, N-1, \tag{2.7}$$

where i is the imaginary unit and $e^{\frac{2\pi i}{N}}$ is a primitive N^{th} root of unity, a complex number that gives 1 when raised to some positive integer power.

The opposite transform, called Inverse Discrete Fourier Transform (IDFT) is described by the following formula:

$$x[t] = \frac{1}{N} \sum_{\xi=0}^{N-1} X[\xi] e^{\frac{2\pi i}{N} \xi t} \qquad t = 0, 1, \cdots, N-1 \tag{2.8}$$

Unlike the FT, the DFT views both the time domain and the frequency domain as periodic. This is referred to as being circular, and is identical to viewing the signal as being periodic. The periodicity can be shown directly from the formula:

$$X_{\xi+N} = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}(\xi+N)n} = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}\xi n} e^{-2\pi i n} = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}\xi n} = X_\xi. \tag{2.9}$$

Figure 2.3 depicts an example of transformation. It is made of three plots. The first plot depicted in Figure 2.3a, which shows a discrete $y[n] = rect(x_n)$ in time domain; Figure 2.3b depicts a small part of the spectrum of that signal in the frequency domain: this has been obtained by applying the DFT to the discrete time signal. The last plot shown in Figure 2.3c, depicts a wide view of the frequency spectrum where some replicas in the frequency domain are visible.

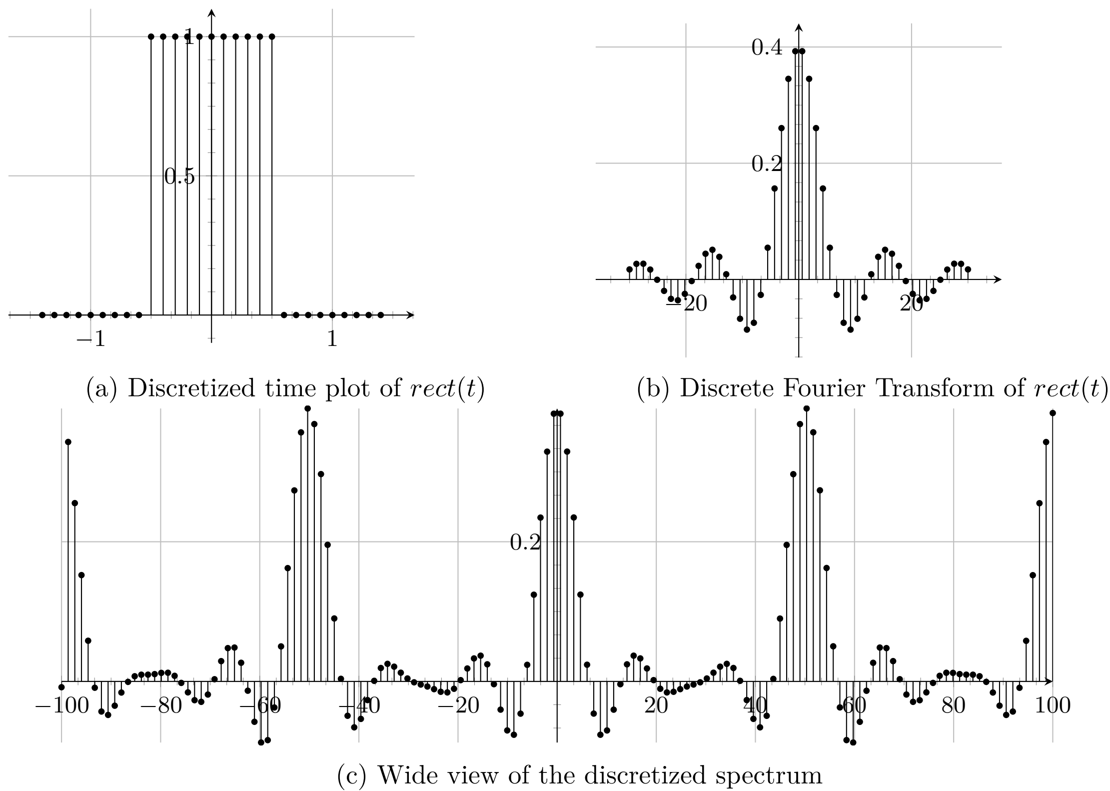
The DFT can be implemented in a computer program by applying its definition reported in Equation 2.7 in a naive way and its asymptotic complexity is $O(n^2)$. There is a set of fast algorithms categorized with the term Fast Fourier Transform (FFT) which aim to reduce the asymptotic complexity from $O(n^2)$ to the order of $O(n \cdot log(n))$. Each algorithm usually works only when some conditions meet. For example, if the number of samples N is a power of two, the Cooley–Tukey Algorithm [5] computes the DFT in $O(n \cdot log(n))$ using a divide-and-conquer approach.

Convolution The convolution is a mathematical formula that given two functions f and g, shows the amount of overlap of one function as it is shifted over the other. The convolution of $f(t)$ and $g(t)$ is written as $f * g$ ($*$ is called *Convolution Operator*) and is defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\, d\tau \tag{2.10}$$

The convolution theorem states that the Fourier Transform of the convolution of two signals is the point-wise product of their Fourier Transform. In formulas:

$$\mathfrak{F}\{f * g\} = \mathfrak{F}\{f\} \cdot \mathfrak{F}\{g\}.$$



(a) Discretized time plot of $rect(t)$

(b) Discrete Fourier Transform of $rect(t)$

(c) Wide view of the discretized spectrum

Figure 2.3: $rect(t)$ and its Discrete Fourier Transform $sinc(\omega)$

It also works on the other way round:

$$\mathfrak{F}\{f \cdot g\} = \mathfrak{F}\{f\} * \mathfrak{F}\{g\}.$$

By applying the Inverse Fourier Transform, we can write:

$$(f * g)(t) = \mathfrak{F}^{-1}\{\mathfrak{F}\{f\} \cdot \mathfrak{F}\{g\}\} \tag{2.11}$$

The implementation of Equation 2.10 is an algorithm whose asymptotic complexity is $O(n^2)$. The implementation of Equation 2.11, instead, takes the advantage of the FFT by speeding-up the computation from $O(n^2)$ operations to $O(n \cdot log(n))$ assuming that the number of samples of the discrete signal is a power of two.

Cross-Correlation The cross-correlation is a mathematical operation that shows how much two functions (e.g. $f(t)$ and $g(t)$) are similar. It is commonly used for searching if a shorter signal is inside a longer signal. The mathematical description is similar to the

one of the convolution:

$$(f \star g)(t) = \int_{-\infty}^{\infty} f^*(t)g(t+\tau)\, dt,$$

where f^* is the *complex conjugate* of f and τ is the displacement. The cross-correlation also follows the convolution theorem:

$$\mathfrak{F}\{f \star g\} = \mathfrak{F}\{f\}^* \star \mathfrak{F}\{g\}.$$

2.1.1 Digital Signal Processing

DSP is the use of digital tools, such as by computers or more specialized embedded devices, to perform a wide variety of operations on signals, usually mathematical transformations or analysis. A *digital filter* is a processing step that modifies a signal by applying a mathematical transform on that. A *digital filter* is a mathematical function defined on the frequency domain $g : \mathbb{C} \rightarrow \mathbb{C}$ which is applied to a signal by performing a pointwise multiplication with the spectrum of it. For example, given a signal $f(t)$ and a filter $\hat{g}(\xi)$, the filtered signal $f'(t)$ is obtained by:

$$f'(t) = \mathfrak{F}^{-1}\{\hat{f}'(\xi)\} = \mathfrak{F}^{-1}\{\hat{f}(\xi) \cdot \hat{g}(\xi)\} = \mathfrak{F}^{-1}\{\mathfrak{F}\{f(t)\} \cdot \hat{g}(\xi)\} \tag{2.12}$$

As an example on the behavior of each frequency component, let us consider a real digital filter. The multiplication of it by the spectrum of the signal causes the amplification or deamplification of each component. In particular, any value of the filter greater than 1 causes the amplification of the corresponding spectral component; any value from 0 to 1, instead, causes a deamplification of it. A digital filter can also be seen in time domain, which becomes a window that needs to be convolved with the signal in time domain. Filters are classified in two main categories: Infinite Input Response (IIR) filters and Finite Input Response (FIR) filters. IIR is a property of some types of digital filters who are distinguished by having an impulse response which continues to modify the original signal indefinitely. FIR filters, instead, are a typology of digital filters whose response decreases in time until it becomes zero after a certain moment, they are usually used for real-time processing (embedded devices) because they do not need the whole signal to work with, but only a fixed-size window. In this work, we use discretized IIR filters because the unwanted effects introduced with the discretization are negligible for our purpose. Also, the IIR filters we are working with have a contri-

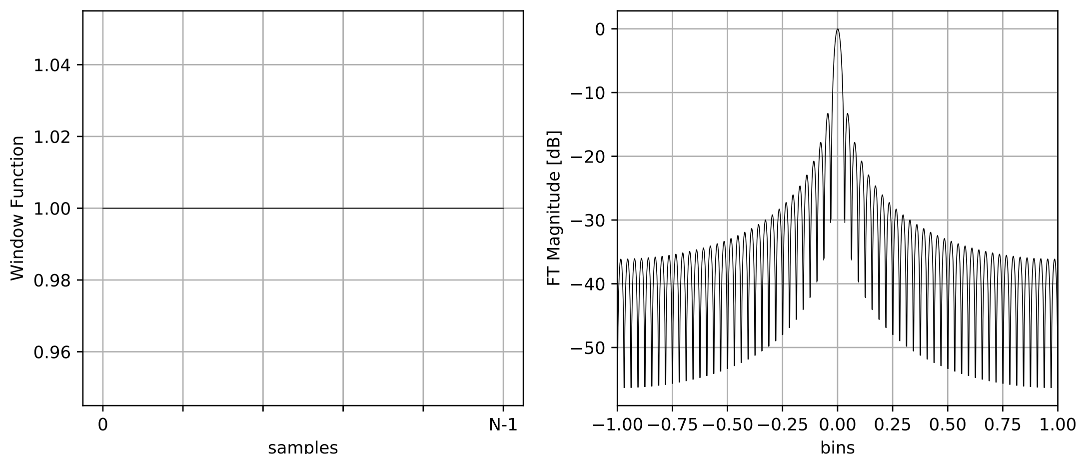


Figure 2.4: Rectangular window function and its spectral leakage. First sidelobes at $-13dB$

bution in time-domain which drops close to zero after few samples, working in practice like FIR filters. *Linear filters* are the filters who affect linearly the phase of the signal, depending on a value α. There are the so-called *zero-phase* filters, whose application does not affect the phase of the signal. They can be seen as a linear filter with $\alpha = 0$. The *window function* is a particular type of filter, also defined as $f : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$, whose representation in the frequency domain is always between 0 and 1, and it is 0 outside some chosen interval, hence it has finite energy. There are two main regions in a window function: *pass-band* and *block-band*, the former is the region of the spectrum of the signal which will be (almost) untouched after applying the window function; the latter, instead, is the region that will be removed. The simplest example of window function is called *rectangular*, which is based on the $rect(t)$ function described above: it multiplies the band-pass region by 1 and the block-band region by 0; the simplicity and the steepness of the borders are paid with extremely high artifacts, called *ripples*, which are present in the block-band. To see and analyze these ripples, the Fourier Transform is applied to the window function, showing the so-called *spectral leakage* plot, which is obtained by applying the FT to the window function. An example of that can be seen in Figure 2.4, where the rectangular window function is analyzed and its spectral leakage is drawn.

There are both real window functions and complex window functions. Each real window function has a compromise between steepness of the filter and magnitude of *ripples* in the block-band, the worst one (as depicted in Figure 2.4) is the rectangular

window function, which has the first sidelobes at $-13dB$. The complex window functions may also change the phase of a signal, hence they might have both steeper response and low ripples at the cost of adding delays to some of the spectral components of the signal. In this work we have mainly used the Tukey Window and the Chebyshev window, which are described below.

Tukey Window This window is a zero-phase window function made of a cosine convolved with a rectangular window and it is tunable using the parameter $0 \leq \alpha \leq 1$. If $\alpha = 0$, the Tukey window becomes a rectangular window; otherwise if $\alpha = 1$, it becomes a Hann window:

$$w(n) = \sin^2\left(\frac{\pi n}{N-1}\right)$$

Figure 2.5 shows an example of Tukey window ($\alpha = 0.5$) and its DFT, which shows the nearest sidelobes at $-15dB$, $-23dB$ and $-34dB$ respectively.

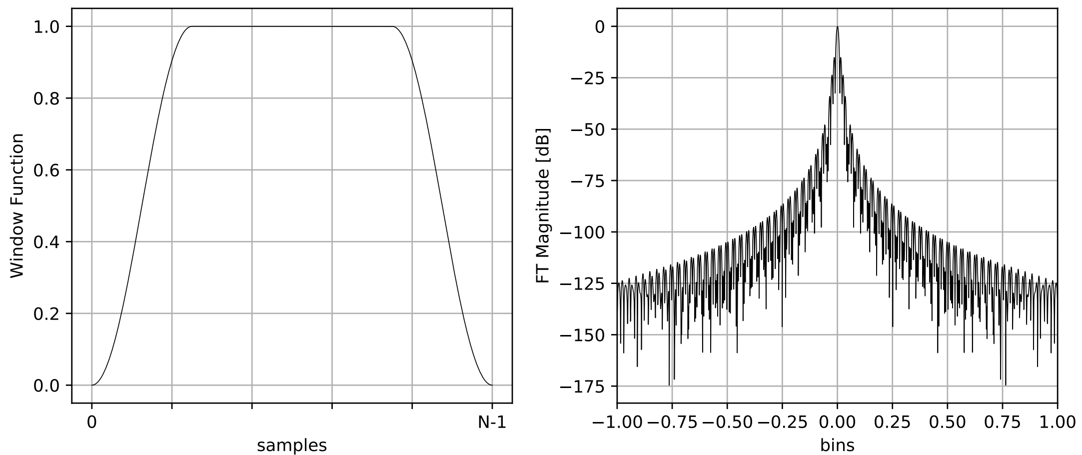


Figure 2.5: Tukey window with $\alpha = 0.5$

A Tukey window is constructed piecewise using the following formula:

$$w(n) = \begin{cases} \frac{1}{2}[1 + \cos(\pi(\frac{2n}{\alpha(N-1)} - 1))] & 0 \leq n < \frac{\alpha(N-1)}{2} \\ 1 & \frac{\alpha(N-1)}{2} \leq n < (N-1)(1 - \frac{\alpha}{2}) \\ \frac{1}{2}[1 + \cos(\pi(\frac{2n}{\alpha(N-1)} - \frac{2}{\alpha} + 1))] & (N-1)(1 - \frac{\alpha}{2} \leq n < \frac{\alpha(N-1)}{2} \end{cases}$$

where N is the size of the window and α is the tuning parameter.

Chebyshev Window This window function is based on Chebyshev polynomials. It could be configured to have a steeper transition between pass-band and block-band regions than any other filter and the magnitude of all sidelobes could go down to $-100dB$. On the other hand, as said before, this is not a zero-phase filter, and a complex multiplication with the original signal is needed to apply it.
The Nth-order Chebyshev polynomial can be computed as:

$$w_N(t) = \begin{cases} \cos(N \cos^{-1}(t)) & \text{if } |t| \leq 1 \\ \cosh(N \cosh^{-1}(t)) & \text{if } |t| > 1 \end{cases}$$

Using $w_0(t) = 1$ and $w_1(t) = t$, each other polynomial can be generated recursively using:

$$w_{N+1}(t) = 2\ t\ w_N(t) - w_{N-1}(t) \qquad \text{if } N \geq 1$$

There are two types of Chebyshev filters and both of them are based on Chebyshev polynomials. The *Type I* Chebyshev filter has an equiripple pass-band and exhibits a monotonic behavior in the block-band, as depicted in Figure 2.6. It has the following magnitude response ($|H_N^1(j\xi)|^2$):

$$|H_N^1(j\xi)|^2 = \frac{1}{1 + \epsilon^2 w_N^2(\frac{\xi}{\xi_p})}$$

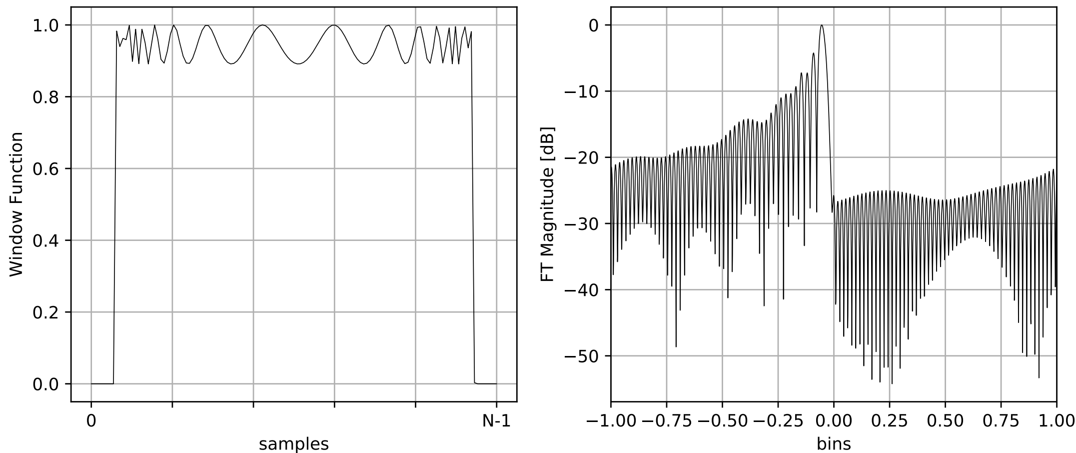


Figure 2.6: Magnitude and leakage of a Chebyshev window Type I, order 30

where ϵ is a parameter that controls the amount of pass-band ripple, ξ_p is the upper pass band edge and w_N is the Chebyshev polynomial of order N.

In the interval $0 < \xi < \xi_p$, all polynomials oscillates between 0 and 1 and this causes $H_N^1(j\xi)$ to oscillate between 1 and $\frac{1}{1+\epsilon^2}$. For $\xi > \xi_p$, the magnitude response decreases monotonically. The *Type II* Chebyshev filter (depicted in Figure 2.7) is indeed the opposite of Type I: it exhibits an equiripple response (ripples have equal height) in the block-band and a monotonically decreasing pass-band. It is also called "inverse Chebyshev filter". The magnitude response ($|H_N^2(j\xi)|^2$) is:

$$|H_N^2(j\xi)|^2 = \frac{1}{1 + \frac{1}{\epsilon^2 w_N^2(\frac{\xi}{\xi_p})}}$$

By allowing some ripple in pass-band or block-band magnitude response, the roll-off of a Chebyshev filter is dramatically faster than a same order Butterworth or Tukey filter.

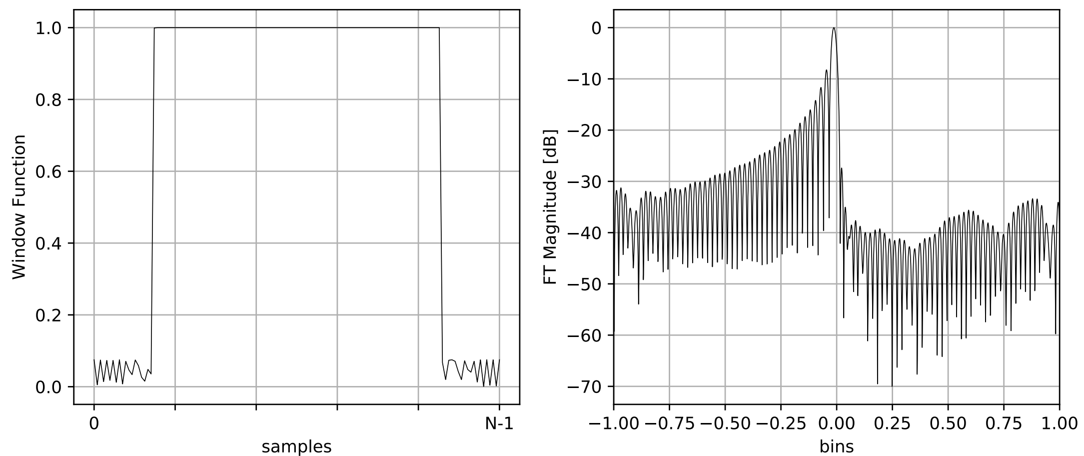


Figure 2.7: Magnitude of a Chebyshev window Type II, order 30

2.1.2 Amplitude Modulation

In telecommunications, the *Modulation* is a process which consists in varying some properties of a periodic signal, called *Carrier* with respect to a *modulating signal* that contains information to be transmitted. The device which performs modulation is called *Modulator*; the device which, instead, perform the inverse process is called *Demodulator*. To successfully transmit some information using the modulation, both Modulator and Demodulator are needed. The Amplitude Modulation (AM) is a modulation technique which consists in varying the amplitude of the carrier signal proportionally to the value of the signal to be transmitted. An example of a sample carrier, modulating signal and modulated signal is depicted in Figure 2.8. The Amplitude Modulation can be easily analyzed mathematically. At first we define the carrier-wave as:

$$c(t) = A_c \cdot \sin(2\pi f_c t),$$

where A_c is the amplitude, f_c is the frequency. We also define the modulating signal $m(t)$ as:

$$m(t) = M \cdot \cos(2\pi f_m t + \phi) = A_c \cdot m \cdot \cos(2\pi f_m t + \phi),$$

where f_m is the frequency of this signal, ϕ is the phase and M is the amplitude.

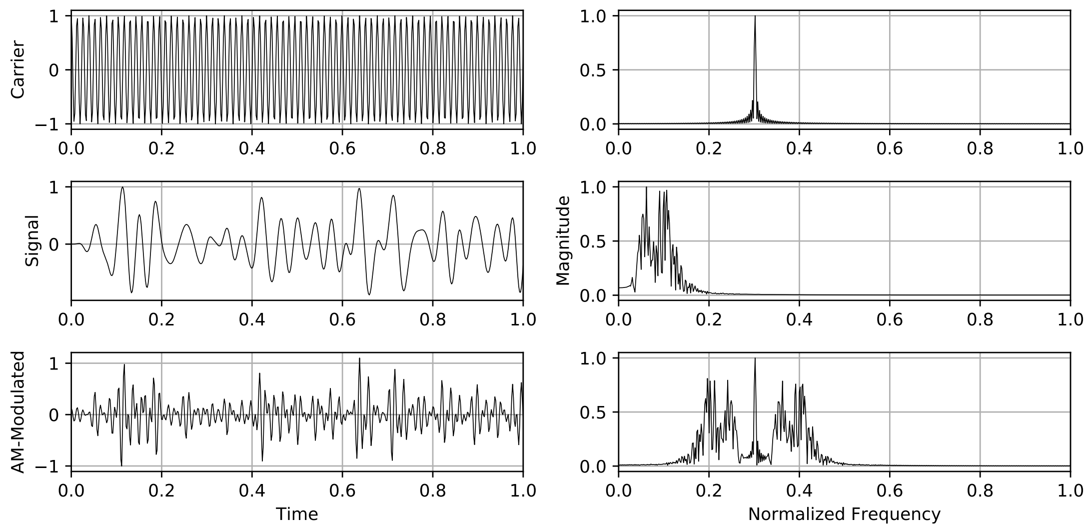


Figure 2.8: A signal is modulated using the Amplitude Modulation technique

We use a cosine-wave for the purpose of the analysis; in practice any signal could be the modulating signal because it could be seen as a sum of sine-waves, where each one is modulated on the carrier signal. We can rewrite $M = A_c m$ with m indicating the *amplitude sensitivity*: how much the amplitude of the carrier signal should be modified with respect to the modulating signal. As side note, $0 \leq m \leq 1$ allow a perfect reconstruction of the signal, $m > 1$ causes over-modulation: the amplitude of the carrier stops being exactly correlated to the modulating signal when the amplitude of the modulating signal causes the result signal to have (a theoretical) amplitude less than zero. Given the carrier $c(t)$ and the signal $m(t)$, the resulting AM-modulated signal is represented by:

$$y(t) = \left(1 + \frac{m(t)}{A_c}\right) c(t) = (1 + m \cdot \cos(2\pi f_m t + \phi)) \cdot A_c \cdot \sin(2\pi f_c t).$$

Using prosthaphaeresis identities, $y(t)$ can be shown to be the sum of three sine waves:

$$y(t) = A_c \cdot \sin(2\pi f_c t) + \frac{1}{2} A_c \cdot m[\sin(2\pi(f_c + f_m)t + \phi) + \sin(2\pi(f_c - f_m)t - \phi)].$$

A modulated signal can be seen as the sum of three components: the carrier wave $c(t)$ is unchanged and two pure sine waves (known as *sidelobes*) are placed symmetrically with respect to the carrier, with frequencies slightly above and below the carrier f_c: $f_c + f_m$ and $f_c - f_m$. The collection of the former frequencies above the carrier frequency is known as the *upper sideband*, and those below constitute the *lower sideband*.

AM Demodulation There are two methods to retrieve the modulating signal given a modulated signal: the *envelope detector* and the *product detector*. The product detector takes the product of the modulated signal and a local oscillator tuned to the same frequency with the same phase of the modulated signal. If $m(t)$ is the original modulating signal and $\cos(\omega t)$ is the carrier, the modulated signal is:

$$y(t) = (1 + \frac{m(t)}{A_c}) \cos(\omega t).$$

After the multiplication with the local oscillator we have:

$$y(t) = (1 + \frac{m(t)}{A_c}) \cos(\omega t) \cos(\omega t)$$

which can be rewritten as:

$$y(t) = (1 + \frac{m(t)}{A_c})(\frac{1}{2} + \frac{1}{2}\cos(2\omega t)).$$

The original signal $m(t)$ can be reconstructed after filtering out both the DC component and the high-frequency component around $\cos(2\omega t)$ by the use of a low-pass filter.

$$y(t) = \frac{1}{2} + \frac{1}{2}\cos(2\omega t)(1 + \frac{m(t)}{A_c}) + \frac{1}{2}\frac{m(t)}{A_c}.$$

The envelope detector, instead, is a method that does not require phase locking. It consists in three steps. Given a modulating signal having bandwidth B, at first a band-pass filter (whose pass-band zone is in range $[f_c - B, f_c + B]$) is applied to the modulated signal, so that only the modulated signal is processed and the noise or other signals are ignored. Then, the signal is fed into a signal rectifier followed by a low-pass filter with bandwidth B. In electronics, the signal rectifier is made by using a diode rectifier followed by a passive low-pass filter. Both methods can be applied digitally, but the signal should be sampled at a sample rate that allows the perfect reconstruction of the upper side-band. The envelope detector also requires the simulation of the rectifier diode, which is usually approximated with taking the square-root of the squared signal.

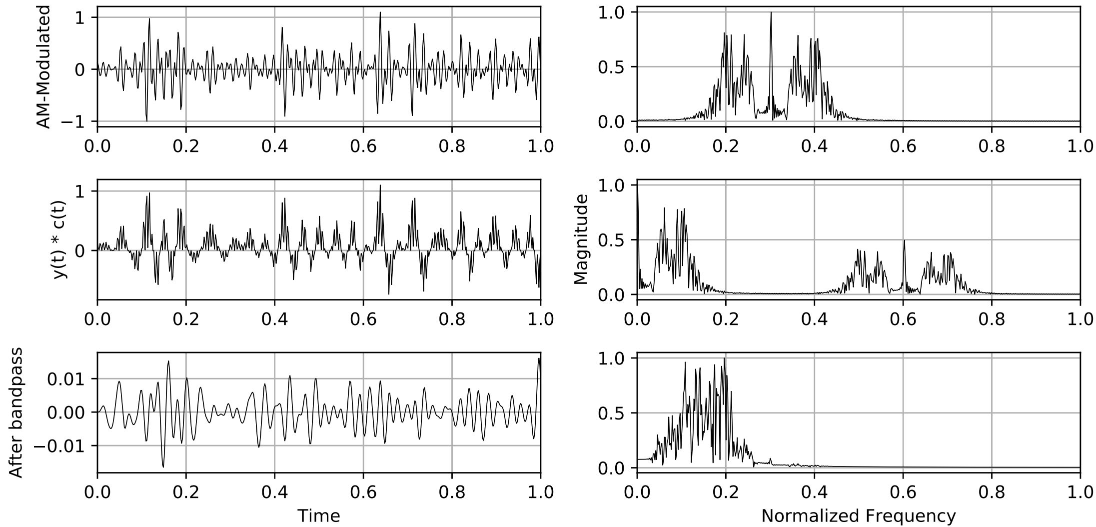


Figure 2.9: AM demodulation using a phase-coherent demodulator

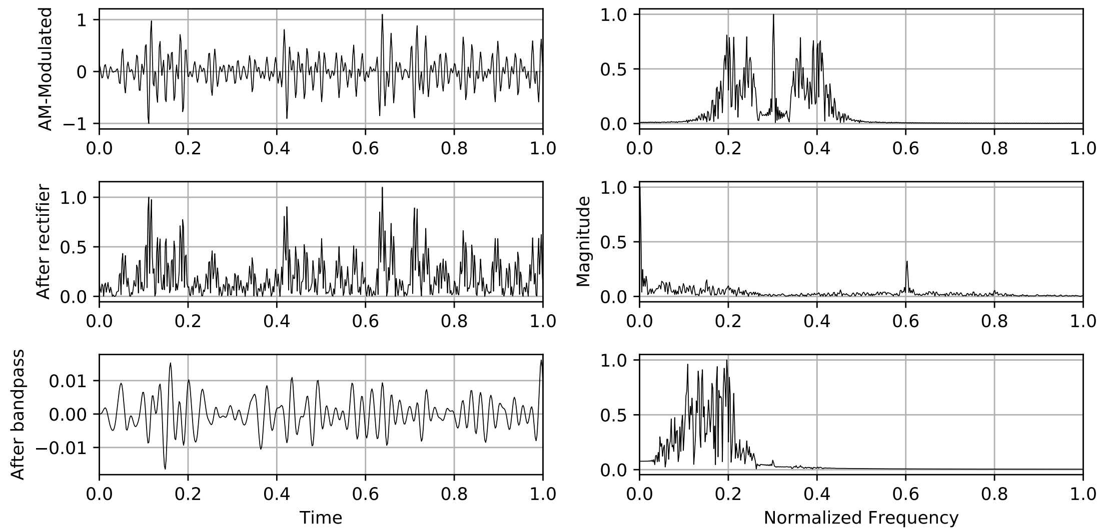


Figure 2.10: AM demodulation using the envelope detector

The software implementation of an envelope detector is the standard method use to perform AM-demodulation if it is not possible to phase-lock to the signal. The low-pass filter can be applied between the two operations, in formulas:

$$y(t) = \sqrt{\text{Low-Pass}(x(t)^2)}$$

2.1.3 In-Phase/Quadrature Demodulation

In-Phase/Quadrature (IQ) Demodulation is a technique used to build devices able to receive any modulated signal [16] having a carrier frequency f_c and a bandwidth B. This technique splits a signal into two components, called In-Phase and Quadrature component. It has two main points on his behalf: it brings any desired signal to the baseband and it allows to use two Analog to Digital Converter (ADC) where each one has bandwidth $\frac{B}{2}$ instead of a single ADC with bandwidth B. The two components (I and Q) are 90-degrees off, the product of them has zero mean:

$$\sin(2\pi f_c t)\cos(2\pi f_c t) = \frac{1}{2}\sin(4\pi f_c t)$$

and for this reason, sine and cosine are a valid *orthogonal* basis for the signal.

The IQ-demodulation can be seen as a basis change of the modulated signal, used to split it in two orthogonal components, so that performing a sample of them at the same time allows us to have the original signal sampled twice as much as each ADC sample rate. The electronic devices needed to perform IQ-demodulation are: an oscillator able to oscillate at f_c, a 90-degree phase-shifter and two ADCs, where each one has bandwidth $\frac{B}{2}$. This demodulation is performed in three main steps: The first is called *Down-mixing* and consists in frequency shifting the whole spectrum, which allows a device to tune to any desired frequency f_c, moving that to the baseband. The second is the *low-pass filtering*, which removes everything outside the chosen pass-band area, so that aliasing during sampling is not introduced. The third part is the sampling, which converts the analog signal (having bandwidth B) to digital.

IQ-demodulation can be used to receive and decode any AM-modulated signal. On the other side, a device able to perform IQ-modulation can be used to perform AM-modulation. Figure 2.11 depicts the full IQ-Demodulation pipeline, which converts the *RF Signal* to the *IQ Signal*.

Figure 2.11: IQ Demodulation pipeline

Down-mixing The down-mixing step is the one which performs frequency shifting. Theoretically, it consists in the multiplication of the original signal in time domain by a complex exponential having the $\omega = -f_c$, where f_c is the center frequency. After this step, the center frequency is shifted to the base-band.

$$x_{IQ}(t) = x_{RF}(t)e^{-j\omega t} \tag{2.13}$$

We notice that after splitting the complex exponential using Euler's Formula, the down-mixing becomes a multiplication by a cosine and a sine, which can be easily implemented in hardware using a splitter followed by two mixers:

$$e^{-j\omega t} = \cos(-\omega t) + i \cdot \sin(-\omega t) = \cos(\omega t) - i \cdot \sin(\omega t)$$

As we can see, a part is multiplied by a cosine generated by the local oscillator and the other part by the sine, which is built by shifting the cosine wave by 90 degrees. The output of the down-mixer are two signals called "In-Phase Component" and "Quadrature

Component", which can be seen as the real and the imaginary component of the same signal.

Low-pass filtering The second part of the IQ demodulation consists in applying a low-pass filter to remove both the noise outside the desired bandwidth and the negative frequency spectrum. The low-pass filter is applied separately to each part (In-Phase and Quadrature) and the cut-off frequency is chosen so that the slow sampling of the two ADCs will not introduce aliasing ($f_{\text{cutoff}} = f_{\text{aliasing}}$).

Sampling In this step, the signal is converted from analog to digital. Thanks to the fact the we are working separately with the two components, we can sample each component at half of the Nyquist frequency of the original signal, hence we can use two ADCs with half of the desired bandwidth $\frac{B}{2}$ each instead of one having a full bandwidth B, which costs more.

The opposite technique is called IQ Modulation and it can be used both to build a transmitter or to reconstruct the original signal after is has been IQ-demodulated.

2.1.4 Signal reconstruction

To reconstruct the original Radio Frequency (RF) signal from IQ data, we must reverse the IQ demodulation. This is done in three steps. The first step is called *zero-interleaving*, which is a technique used to increase the sample rate by interleaving the original signal with N_I zeros between each sample in time domain. N_I is called Interpolation Factor and it is obtained by calculating how many times the bandwidth should be increased, so that the highest frequency could be represented without aliasing. In formulas:

$$N_I = ceil\left(\frac{f_{RF}}{f_{MF}}\right) - 1 \tag{2.14}$$

where f_{MF} is the maximum representable frequency in the IQ domain and $f_{RF} = f_c + \frac{f_{MF}}{2}$ is the maximum representable frequency in the RF domain, calculated as the sum of the carrier frequency and half of the bandwidth of the signal.

After zero-interleaving, the spectrum of the signal is made of the original and a set of N_I replicas, this is due to the fact that the original signal is sampled and we see a small slice of the DFT, which is made of infinite replicas of it. In order to remove the

unwanted replicas of the original signal, a *low-pass* filter with $f_{cutoff} = f_{MF}$ should be used. The last part is called *Up-Mixing*, which is the one responsible for shifting the center frequency from the base-band to its original position. As shown in Equation 2.13, this is achieved by multiplying the upsampled (and low-passed) signal with another complex exponential similar to the one used in the down-mixing phase.

Theorem 2. *Given* $\omega = 2\pi f_c \frac{n}{f_s}$, $t = \frac{k}{2n}$ *where* k *is the current sample,* n *is the total number of samples,* f_c *is the shift frequency and* f_s *is the maximum representable frequency,*

the original signal $x_{rf}(t)$ *can be reconstructed using:*

$$x_{rf}(t) = Re\{x_{iq}(t)e^{j\omega t}\} = x_i(t)\cos(\omega t) - x_q(t)\sin(\omega t) \tag{2.15}$$

Proof. Given the following definitions:

$$x_i(t) = x_{RF}\cos(\omega t) \tag{2.16a}$$

$$x_q(t) = x_{RF}(-\sin(\omega t)) \tag{2.16b}$$

it is easy to prove the reconstruction by substituting (2.16a) and (2.16b) into (2.15):

$$x_i(t)\cos(\omega t) - x_q(t)\sin(\omega t) = x_{RF}cos^2(\omega t) + x_{RF}sin^2(\omega t)$$

which can be rewritten as:

$$x_{RF}(1 - sin^2(\omega t)) + x_{RF}sin^2(\omega t) = x_{RF}$$

□

Optimal Signal Reconstruction The Whittaker-Shannon Interpolation Formula is a way to reconstruct a bandwidth-limited continuous-time signals, given its discrete representation. The formula states that, given a sequence of numbers $x[n]$, the continuous function $x(t)$ is given by:

$$x(t) = \sum_{n=-\infty}^{\infty} x[n] \cdot sinc\left(\frac{t - nT}{T}\right) \tag{2.17}$$

where $T = \frac{1}{f_s}$ is the distance in time between two sample, f_s is the sampling frequency and $sinc(x)$ is the normalized *sinc* function. This formula reconstructs correctly the signal (i.e. without aliasing) if the bandwidth of the spectrum $\hat{x}(\xi)$ of the signal is limited:

$$\hat{x}(\xi) = 0 \qquad \text{for all } |\xi| \leq \frac{1}{2T}.$$

This formula is energy-preserving, hence we do not need the Optimal Signal Reconstruction to perform attacks because it provides the exact same result as performing the attacks on the discrete signal.

2.2 Side-Channel Attacks

Side-Channel Attacks (SCA) are attacks based on information leaked from the implementation of a system, instead of the algorithm itself. Some side-channel attacks require knowledge of the inner workings of the system; others are effective even as black-box attacks. They can be classified as *Active* side-channel attacks, where the attacker can tamper the analyzed system (for example by injecting faults) and *Passive* side-channel attacks, where the attacker can only observe without performing any interaction. In all cases, the underlying principle is that physical effects caused by the computation of an operation of a cryptosystem can provide useful extra information about secrets in the system. Examples of side-channels are memory load time due to caches, execution time of an algorithm, sound emission, power consumption, the electromagnetic (EM) field or behavior in case of faults. These channels could leak some information useful to break an encryption algorithm running on the target machine.

The countermeasures for side-channel attacks usually fall in two main categories: either eliminate (or reduce) the leak of information or eliminate the connection between the released information and the secret data, by making the leaked information uncorrelated with the secret data, for example using some form of randomization. In this work we focus on analyzing power consumption and EM emissions of a Central Processing Unit (CPU) while it is performing encryption of a set of messages. We are performing *Power-Analysis Attacks*, which are roughly categorized into SPA and CPA, which is an improvement of Differential Power Analysis (DPA).

Each attack works by recording one or more traces and performing analysis on them. A *trace* refers to a set of equally time-spaced power consumption measurements taken during a cryptographic operation. For example, a 1 millisecond operation sampled at 20 MHz yields a trace containing 20000 points. A *trace-set* is a collection of traces, it is mainly used to perform CPA and each trace contained in the set has a corresponding plaintext or ciphertext.

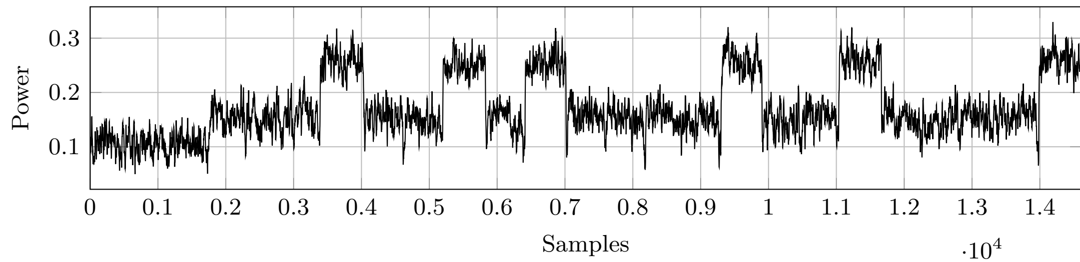


Figure 2.12: Power variations observed while running an unprotected square and multiply algorithm.

2.2.1 Simple Power Analysis

The form of side-channel attacks that involves directly interpreting the power traces collected during cryptographic operations is called *Simple Power Analysis* (SPA), where, sometimes, the interpretations becomes a visual inspection of the traces. It has been introduced the first time by Kocher et al. [18], even if the same principle was already known and used in `TEMPEST` attacks [19] to spy far computer monitors by reconstructing the image from the leaked EM emanation.

The goal of SPA in cryptography is to reveal the secret key when given only a small number of power traces. The main distinction in this field is between single-shot SPA attacks and multiple-shot SPA attacks. The former happens when only a single power trace can be recorded and the key extraction may happen with just that trace [13]; the latter happens, instead, when the attacks is performed after recording multiple power traces [12]. The core principle of the Simple Power Analysis Attack is that, it manages to extract the secret key successfully if there are key-dependent differences within a single trace, if there is some redundancy in the algorithm and if the control-flow is key-dependent. Using SPA it is possible to recover secret-key bits directly from a single trace. With modern CPUs, this means that it works if there is a dependency between part of the key and *which* instructions are executed. Genkin et al. [12] managed to extract the private key of the RSA implementation in GnuPG via acoustic cryptanalysis. The attack was able to extract full 4096-bit RSA decryption keys from laptop computers (of various models), within an hour, using the sound generated by the computer during the decryption of some chosen ciphertexts. Multiple traces are required because the bandwidth of a single trace is made of few KHz, hence it contains very little information. Then, they showed[13] that it was possible to successfully attack GnuPG before 1.4.19 and libgcrypt 1.6.3 (used by GnuPG 2.x) running on a consumer-grade x86 CPU using

SPA, causing the exfiltration of the private key in a single-shot and in a non-intrusive way: by measuring the EM emanations for a few seconds from a distance of 50 cm. This attack was possible in three different flavors: by recording the EM emanations using a Software Defined Radio (SDR) with a shielded loop antenna (made with a coaxial cable) and using a computer for the processing part; by building a small portable device SDR-based which records the emanations and then running the processing offline or by using a small consumer radio connected to the microphone of a smartphone, feasible because the leakage bandwidth was circa 1.7 MHz, located in the range of the commercial AM radio frequency band.

2.2.2 Correlation Power Analysis

The most popular type of side-channel Attacks nowadays is called Correlation Power Analysis (CPA), which is an improvement (and a mathematically correct version) of the original Differential Power Analysis (DPA). This category of attacks works also without a detailed knowledge of the attacked device, but, differently from the SPA, this attack requires the collection of a batch of traces (a *trace-set*), where each trace contains the recording of the power variations (or EM emission) during each encryption, which should be also associated to a known plaintext or ciphertext.

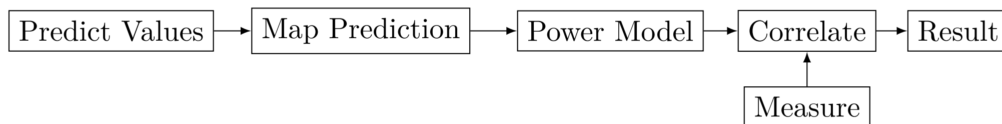


Figure 2.13: CPA Pipeline

Hence, CPA is a known plaintext (or known ciphertext) attack, which makes predictions based on the hypothetical value of a portion of the key. Figure 2.13 depicts the skeleton of CPA. To perform this attack, at first we measure the power consumption of a running device; we predict the intermediate value for each run by knowing the plaintext and the hypothetical portion of the key and then the prediction of the intermediate value is mapped to the power consumption by using a power model (e.g. Hamming distance). The guessing of the key is done piecewise, by trying to recover a small portion of that (e.g. 8 bits) during each analysis. The correct guess of the key will be revealed by comparing the power traces with the predicted power model using a statistical correlation method (e.g. Pearson) for each sample (instant of time t) of the traces: the power consumption prediction relying on the correct key will show a significant correlation in

the time instant when the modeled operation is performed.

The Pearson correlation coefficient is a number $-1 \leq \rho_{X,Y} \leq 1$ which express how much two random variables X and Y are linearly correlated. A value close to 0 means that X and Y are not linearly correlated; a value close to 1 means that they are completely linearly correlated and a value close to -1 means that they are inversely correlated. The Pearson correlation coefficient is defined as:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y} \tag{2.18}$$

We can use a matrix representation for a trace-set, where each row is a trace and each column is a sample of the trace. Given the traces $\boldsymbol{t}_j = [t_{1,j}, t_{2,j}, \cdots, t_{i,j}, \cdots, t_{n,j}]$ and the power prediction matrix for a given key-byte $\boldsymbol{p}_j = [p_{1,l}, p_{2,l}, \cdots, p_{i,l}, \cdots, p_{n,l}]$, the CPA attack is performed by computing the correlation vector $\boldsymbol{r}_{j,l}$ by computing:

$$r_{j,l} = \frac{\sum_i (t_{i,j} - \bar{t}_j)(p_{i,l} - \bar{p}_l)}{\sqrt{\sum_i (t_{i,j} - \bar{t}_j)^2 \sum_i (p_{i,l} - \bar{p}_l)^2}} \tag{2.19}$$

As result, given a key k,

$$r_k = \max(r_{\boldsymbol{t}_j, \boldsymbol{p}_l}) \tag{2.20}$$

is taken as index for the whole correlation attack.

Correlation Power Attacks nowadays have a number of open challenges. The work by Barenghi and Pelosi [2] showed that the leakages in a modern, complex CPU are due to all internal buffers. This means that having an execution model of the CPU allows spotting leakages usually proportional to the Hamming Distance of some intermediate values. Gilbert Goodwill et al. [14] showed how to assess whether a cryptographic module can provide resistance to side-channel attacks commensurate with the desired security level without having a power model of the target device. This methodology could be combined with Veyrat-Charvillon and Standaert [29], to perform model-free attacks by splitting a trace-set in two subset depending on a key-dependent intermediate value. If the device is leaking, then statistical test with the correct guess will spot a difference in the behavior of the two subsets.

The poor Signal-to-Noise Ratio (SNR) of the leakages with respect to the noise generated by the other parts of the circuit provides another set of challenges: theoretically, removing the unwanted components should result in an improvement of the attack quality, i.e. a CPA should be able to recover portion of the key with less power traces.

The last set of challenges is given by the fact that, to perform correctly a CPA, the traces should be aligned with the maximum tolerable delay being equal to a clock cycle of the computing device. This is because each estimation is done pointwise in time, hence even the smallest misalignment causes a misinterpretation of the power consumption as belonging to a different clock cycle, thus reducing the effective SNR. Temporal hiding is a technique used on the modern smartcards [20] based on this idea: a random number k is generated (where $0 \leq k \leq N$) before each encryption and then, k "dummy" (the result is discarded) operations are added before the encryption and $N - k$ dummy operations are added after the encryption, so that there is no known way to align correctly all the traces. In this way $\sqrt{N}$-times the number of standard traces should be required to perform a CPA, as shown in Mangard et al. [20]: thanks to the *integration method* the number of required traces can be reduced from N to $\sqrt{N}$ by summing and averaging near samples. An relatively-low upper limit on the number of the transactions is then added, so that after it is crossed, the smart-card stops working preventing the successful key-recovery.

2.3 Trace warping

During trace recording two types of misalignment can occur. The first is what we call *delay*-misalignment, which happens when, given two traces representing the same phenomena, the second one has an initial delay which prevents them to be perfectly overlappable. A rigid realignment (e.g. by the use of cross-correlation) may solve this problem. The second type of misalignment is called the *clock-jitter* misalignment, which happens when the first sample of the phenomena represented in the two traces is aligned but, due to clock drifting, after few samples the two traces starts being misaligned. In this section we explore two signal-warping algorithms which are used to correct the clock-jitter misalignment in the recorded traces by warping non-linearly (stretching and compressing) parts of one trace to fit the other.

2.3.1 Dynamic Time Warping

Given two time series, the optimal alignment between them happens when a desired distance (Euclidean or Manhattan) between them is minimized. Dynamic Time Warping (DTW) is an algorithm used to find the optimal alignment between two time-series if one may be "warped" non-linearly by stretching or shrinking it along its time axis. This

warping can also be used to find corresponding regions between the two time-series or to determine the similarity between them.

Given two time-series $x(n)$ and $y(n)$, of length $|x|$ and $|y|$,

$$x(n) = x_1, x_2, ..., x_i, ..., x_{|x|}$$

$$y(n) = y_1, y_2, ..., y_i, ..., y_{|y|}$$

construct a warp-path W

$$W = w_1, w_2, ..., w_K \qquad max(|x(n)|, |y(n)|) \leq K < |x| + |y|$$

where K is the length of the warp-path and the kth element of the warp-path is $w_k = (i, j)$ where i is an index from time-series $x(n)$, and j is an index from time-series $y(n)$.
The warp-path must ensure the following property:

$$w_k = (i, j),\ w_{k+1} = (i', j') \qquad i \leq i' \leq i + 1,\ j \leq j' \leq j + 1$$

The optimal warp-path is defined as the minimum-distance warp-path, where the distance of a warp-path W is calculated as

$$Dist(W) = \sum_{k=1}^{k=K} Dist(w_{ki}, w_{kj})$$

where $Dist(W)$ is the distance (Euclidean, in this work) of a warp-path W and $Dist(w_{ki}, w_{kj})$ is the distance between the two data point indexes (one from $x(n)$ and one from $y(n)$) in the kth element of the warp-path.

Algorithm Algorithm 2.3.1 illustrates the implementation of the Dynamic Time Warping algorithm when the two sequences s and t are strings of discrete symbols. $d(x, y)$ is a function that returns the distance between the symbols x and y.

Each cell in the Cost Matrix ($|x| \cdot |y|$) is filled exactly once, and each cell is filled in constant time, hence time and space complexity of the DTW algorithm is $O(|x| \cdot |y|) \simeq O(N^2)$.

Algorithm 2.3.1: Dynamic Time Warping

```
   Input: x ∈ ℝ^n : Signal, y ∈ ℝ^m : Reference
   Output: (Dist(s,t) ∈ ℝ, costMatrix ∈ ℝ^n × ℝ^m)
1  c_m ← {0}^{n×m}
2  c_m[0,0] ← 0
3  for i ← 1 to n do  c_m[i,0] ← ∞
4  for j ← 1 to m do  c_m[0,j] ← ∞
5  for i ← 1 to n do
6  |   for j ← 1 to m do
7  |   |   cost ← d(x[i], y[j])
8  |   |   c_m[i,j] ← cost + minimum( c_m[i−1,j], c_m[i,j−1], c_m[i−1,j−1] )
9  maxDist ← c_m[n,m]
10 return (maxDist, c_m)
```

Building the path Given two time-series $x(n)$ and $y(n)$, if the Cost Matrix between $x(n)$ and $y(n)$ is known, the algorithm that builds the warp-path minimizing the distance between $x(n)$ and $y(n)$ by stretching $y(n)$ is easily buildable by moving backward on the Cost Matrix, starting from cell $(|x|-1, |y|-1)$ and going backward at each step in the nearest cell having:

$$C_{i_{x+1}\ j_{x+1}} = min(C_{i_x\ j_{x-1}}, C_{i_{x-1}\ j_{x-1}}, C_{i_{x-1}\ j_x})$$

and decrementing the corresponding counter(i, j, or both of them). Algorithm 2.3.2 implements these rules.

The algorithm starts visiting the last cell ($C_{|x|-1\ |y|-1}$) and then goes backward. At each iteration either j or i are decremented. Hence the worst case happens when at first j is decremented until it reaches 0 and then i is decremented until it reaches 0: this means that the time complexity is $O(|x|+|y|)$. The space complexity is also $O(|x|+|y|)$ if the cost of storing the cost matrix (c_m) is not counted, because in the worst case the wrap-path contains every point of the first time-series and then every point of the second one.

Warping the time-series We are using a zero-order holder to warp a time-series $x(n)$ given a warp-path w_p. The warping algorithm is described in [25] and reported in Algorithm 2.3.3. It warps the time-series so that if w_p contains the cell D_{ij} in the c_m, the ith point of time series $x(n)$ will be moved to the jth point of the time series $y(n)$.

Algorithm 2.3.3 loops one time through *path*. If *path* is built using Algorithm 2.3.2

Algorithm 2.3.2: Build Path from Cost Matrix

```
    Input: c_m ∈ ℝ^n × ℝ^m : Cost Matrix
    Output: w_p ∈ ℝ^(n · m) : Warp path
 1  i ← n − 1
 2  j ← m − 1
 3  while i ≥ 0 ∨ j ≥ 0 do
 4  |   APPEND(w_p, (i, j))
 5  |   if i = 0 ∨ j = 0 then  break
 6  |   if i = 0 ∧ j > 0 then
 7  |   |   i ← i − 1
 8  |   else if j > 0 ∧ i = 0 then
 9  |   |   j ← j − 1
10  |   else
11  |   |   v_1 ← c_m[i − 1, j]
12  |   |   v_2 ← c_m[i, j − 1]
13  |   |   v_3 ← c_m[i − 1, j − 1]
14  |   |   m ← min(v_1, v_2, v_3)
15  |   |   if m = v_1 ∨ m = v_3 then  i ← i − 1
16  |   |   if m = v_2 ∨ m = v_3 then  j ← j − 1
17  return w_p
```

Algorithm 2.3.3: Time-Series Warping Algorithm

```
    Input: x ∈ ℝ^m : Signal, w_p ∈ ℝ^(n · m) : Warp path
    Output: o ∈ mathbbR^k : Warped signal
 1  foreach e in w_p do
 2  |   o[e_x] ← x[e_y]
 3  return o
```

then the size of *path* is $O(|x| + |y|)$. This means that Algorithm 2.3.3 has a linear (w.r.t. the input time-series) space and time complexity. The asymptotic complexity is then $O(|x| + |y|)$.

2.3.2 Fast Dynamic Time Warping

The asymptotic complexity of the original DTW algorithm has both space and complexity $O(n^2)$, which is practically unusable with the signals we are using. Fortunately for us, there is a suboptimal but faster version of the standard DTW algorithm called FastDTW [25], which is linear both in time and space. It uses a multilevel approach with three key operations:

1. *Coarsening*: Halve the length of a time-series by averaging adjacent pairs of points.
2. *Projection*: Find a warp-path at lower resolution and use it as first guess for a higher resolution's warp-path.
3. *Refinement*: Find the optimal warp-path in the neighborhood of the projected path, where the size of the neighborhood is controlled by the *radius* parameter.

The pseudo-code for FastDTW is reported in Algorithm 2.3.4. It switches between two modes: if the search radius is less or (almost) equal to the size of one time-series, the standard DTW (Algorithm 2.3.1) is used. Otherwise, it shrinks the size of the time-series, it gets a low-resolution warp-path, creates a list of which cells should be visited in the high-resolution map and then uses it as search-window for the last `DTW` call (line 9).

Algorithm 2.3.4: FastDTW

Input: $x \in \mathbb{R}^n$: TimeSeries, $y \in \mathbb{R}^m$: TimeSeries, $r \in \mathbb{N}$: distance to search outside of the projected warp-path

Output: $(d \in \mathbb{R},\ w_p \in \mathbb{R}^(n \cdot m))$: Minimum distance, WarpPath

```
1 minTSize ← radius + 2
2 if n ≤ minTSize ∨ m ≤ minTSize then
3     return DTW(x,y)
4 else
5     shrunkX ← reduceByHalf(x)
6     shrunkY ← reduceByHalf(y)
7     lowResPath ← FastDTW(shrunkX, shrunkY, radius)
8     window ← SearchWindow(lowResPath, x, y, radius)
9     return DTW(x, y, window)
```

Time and space complexity Given n as the length of the input signals, the total number of filled cells is $2n\ (4r+3)$, the time to create all resolution is $4n$, so the FastDTW time complexity is proven to be $n(8r+14)$. At last, the space complexity of FastDTW is the sum of the following components: $2n$ is the space used to shrink time-series, $n(4r+3)$ is the space used by the cost matrix and $2n$ is the space used to store the warp-path. Hence, the total space complexity of the FastDTW is then $O(5n(4r+7)) = O(n)$.

Chapter 3

A DSP Approach to DPA-oriented signal processing

In this chapter we show the improvements that a side-channel attack can have after filtering it using Digital Signal Processing (DSP) techniques.

This chapter is divided in three sections, the first is about methods and tools to perform automatic characterization of side-channel leakages in the frequency domain. The second section describes the DSP pipeline we are using, problems and solutions we found, and ways to perform track alignment, divided in coarse alignment and fine alignment. The third section describes the steps to use a Software Defined Radio (SDR) to perform attacks, instead of an oscilloscope.

3.1 Characterization of leakage in frequency domain

The aim of this section is to characterize which spectral components of the electromagnetic (EM) emissions of a Central Processing Unit (CPU) contain enough information to allow a full key-recovery by recording them and performing a Correlation Power Analysis (CPA) attack. This information will be used to understand what is the bandwidth of the leakage, which part of the spectrum can be discarded and what equipment is needed to measure it without (or with minimum) loss of information.

3.1.1 Frequency-oriented leakage model

The power consumption of the target device is the side-channel we measure by recording the EM emissions of the CPU. This gives us the information useful to perform Differ-

ential Power Analysis (DPA) attacks. The target device has a fixed clock signal, which imposes a periodicity to the power consumption. This periodicity is strictly linked to the clock frequency that drives it.

A model for this behavior has been made by analyzing the power consumption of the CPU during a single clock-cycle and by replicating it at regular intervals. This model corresponds to the convolution of the power consumption of a single clock-cycle with a series of stems equally spaced in time. When the D flip-flops become transparent, they start to expose the new value at their output, usually at the rising edge of the clock signal. All the logic gates at the base of the combinational logic circuit start switching together (we expect them to be many). As the computed signal gradually spreads in the combinational circuit, the number of switching logic gates starts decreasing. A model to represent this consists in a high initial consumption peak followed by an area of low power consumption that lasts proportionally to the length of the combinational circuit. The switching activity of the logic circuit should end before the raising edge of the next clock signal, where the latch stores the result. Otherwise a *setup-time violation* is generated, which may cause incorrect data to be stored. Increasing the working frequency causes the clock generator to reduce the distance between the spikes and this may cause to the combinational logic one of these two behaviors: either the logic is already switching at a reasonable speed (so that it is not needed to increase its speed) or the power supply unit should increase the voltage so that the logic starts switching enough fast to avoid failures.

The model for power consumption we are going to use is a simplification of the real model we described in the previous paragraph. Our model is obtained by the use of two discrete *rect* functions: the first tall and short in time, the second lasts long but has less energy than the former. Given a series of stems where the distance between them is t_{clk}, the frequency spectrum shows a set of stems whose distance is f_{clk}. The effects of the two *rect* functions modeling the power consumption of the logic are spread near each stem in the frequency spectrum. By looking at a single stem in the frequency spectrum, this happened to be placed at f_{clk} and the components representing the power consumption of a single clock-cycle are placed at both sides of each stem. Hence, it is possible to consider the leakage as it is AM-modulated on the clock frequency and measure it consequently.

3.1.2 Validation

To perform validation of the frequency-domain leakage model described before, we need:

1. A synthetic index that represents how informative is a portion of the spectrum from the point of view of the information contained in the side-channel.
2. A methodology to search which spectral components are those with a high informative content, useful to side-channel attacks.

The synthetic index, called *score*, has been designed by using the CPA as information detection technique because it is widely used and optimal when the leakage is proportional to the considered intermediate value. In particular, a CPA attack is considered successful when the confidence interval of the sample Pearson correlation coefficient r for the correct key guess is disjoint with respect to the any other and whose absolute value is the greatest. An alternative method is to find the leakage by the use of non value-specific tests instead of the CPA. This alternative can easily replace the CPA test without varying the overall structure. As we have seen, the evaluation metric of a side-channel attack via CPA is binary: either an attack is successful or it is failed. We need, instead, a metric of *how much* is an attack effective.

Scores Each algorithm used to find leakage sections needs a monotonic metric, expressed as real number, to quantify the effectiveness of an attack. This metric should be used to understand both the quality of an attack and, if the attack failed, how far are we from making it work. We called this metric the *Score* of an attack. Usually the maximum correlation value is used as index for the attack, as described in Equation 2.20. For each CPA attack, we can obtain other information useful to understand the status of the attack by measuring the correlation values in more instants of time. The correlation with the correct hypothesis may happen in more instants of time and, measuring only the maximum distance between confidence intervals given a time t does not capture the whole information of "successfulness" of a CPA attack. Not every instant of time which correlates could be known, for example some of them could have low Signal-to-Noise Ratio (SNR) and they cannot be seen without filtering.

We tested the various algorithms with a common set of scoring functions designed to extract more information from each CPA attack and useful to drive the search algorithms. All these scoring function are described in Table 3.1, where t is a point in time, $\delta_k(t)$ is

the gap between the confidence interval of the correct key and confidence interval of the wrong key having highest correlation. N_T is the minimum number of traces so that the two confidence intervals are no longer overlapping ($\delta_k(t) > 0$).

Name	Score Formula
`MinTracks`	$s_{mt} = \frac{1}{N_T} \cdot M_x$
`MaxGap`	$s_{mg} = \delta_k(t)$
`Area`	$s_a = \sum_{t=0}^{\infty} \delta_k(t) \;\text{ iif } \delta_k(t) > 0$
`Mix_Gap_Tracks`	$s_{mgt} = \sqrt{s_a^2 + s_{mg}^2} - e^{1-s_{mg}}$
`Mix_Area_Tracks`	$s_{mat} = \sqrt{s_a^2 + s_{mt}^2}$
`Mix_All`	$s_{all} = \sqrt{s_a^2 + s_{mt}^2 + s_{mg}^2} - e^{1-s_{mg}}$

Table 3.1: Scoring functions

The first score, and the one we used most, is `MinTracks`, which is based on the minimum number of traces that allow to recover the encryption key with a confidence greater than a chosen value ($\alpha = 0.1$ most of the times). $M_x = 1'000'000$ is a constant multiplier used to display and work with integer numbers, this is because $0\frac{1}{N_T} \leq 1$. If the attack fails, $s_{mt} = 0$ so that the function . Alternatively a number depending on the trace-set could have also been used (e.g. M_x is the number of traces in the trace-set) but this choice does not allow comparison between two scores.

`MaxGap`, instead, performs a full attack and uses as score the distance between the confidence interval of the correct key and the confidence interval of the best (highest) wrong key.

`Area` uses as score the area between the interval of the correct key and the one of the best wrong key, when this is greater than zero.

The other three functions are a combination of the first three, using a function which maps $f : \mathbb{R}^N \rightarrow \mathbb{R}$ because the score should be a scalar number and not a vector. We used mainly the Euclidean distance with a correction, as shown in the table.

$w_p = -e^{1-s_{mg}}$ is a correction factor used to rapidly decrease the score if $s_{mg} < 0$. Any other monotonic function (which drops to zero if $s_{mg} > 0$) could be used. This correction factor is used in the score functions that mix more values using the Euclidean distance, where a $s_m g < 0$ would increase the resulting score. A piecewise definition of the scoring functions could have also been used.

Search algorithms The easiest way to perform the search is to consider each component without amplification: either is included or excluded. Given a slice of the spectrum around the f_{clk}; $s = f_{clk} \pm B$ and given a fixed width for each component w, the number of available components is $n = \frac{2B}{n}$. There are 2^n possible combination of components and an exhaustive search becomes not feasible when n is big enough (e.g. 64, 128, 256).

We have developed and compared three strategies to workaround this problem and scan the whole spectrum: *linear scanning*, *dichotomic search*, and *genetic algorithm search*.

- The *linear scan* of the spectrum consists in removing a single window at time with a band-stop filter and checking if the score increases or decreases. At the end, all the slices who contribute to the attack are kept and the other are discarded. The advantage of this technique is that highly informative components are easily identifiable. The main disadvantage, instead, is that this technique is slow for high-precision scans.

- The *dichotomic search* consists in searching which components contribute to the attack by dividing at each step the spectrum in half and performing the attack after keeping only a part of the spectrum and removing the other. The selected part is discarded if the attack fails, otherwise it is split again and searched in the next iteration. A maximum number of recursions is set and the depth search stops when it is crossed.

  This technique has the advantage that it is extremely fast with respect to the other and it is able to isolate small components who contribute significantly. On the other side, the optimality is not guaranteed and sometimes small components surrounded by noise or by other components who correlate inversely are discarded.

- The *genetic algorithm* performs a search which is close to exhaustive search. It starts by trying random windows and during the iterations, it tries to maximize the score by performing a genetic crossover and genetic mutation between them. This is the only algorithm able to find filters even after scrambling the spectrum (e.g. using the *abs* function); it finds small group of slices that, if kept in pass-band together, improve the attacks. The disadvantage, is the speed: this algorithm is extremely slow and there is no known terminate condition, hence by its own does not terminate.

All of these search algorithms internally describe a window function as a binary vector: $\{0,1\}^n$, where n is the length of the array. Each bit represents a Hann window having bandwidth B and having center frequency positioned as described in Equation 3.1. In this way, each window is partially overlapping (exactly by $\frac{B}{2}$) the next one and the previous one so that the whole spectrum is covered.

$$pos_i = \text{offset} + i \cdot \frac{B}{2} \tag{3.1}$$

Linear-scan The first, and easiest, algorithm used to find which parts of the spectrum leak has been called *Linear-Scan Find-Filter*. It consists in dividing the spectrum in a finite number N of slices and then check for each slice what is its contribution to the score function. This can be done in two ways: either a band-pass filter is applied so that everything is masked out except for that slice, or a band-stop filter is applied to that slice, removing it. If the first way is chosen, the score of each slice can be directly obtained from the score of the attack. If the second way is chosen, at first the score without any filter should be calculated, then for each slice check if the score function increases or decreases after applying a band-stop filter that masks out that slice. If the score function increases after masking out, and each slice contributes to the result independently to the others, then that slice is marked as `BAD`: it introduces noise and should be removed from the final result. We have chosen the second way because the other one drops slices which contribute to the attack but they are not enough to perform an attack if taken alone. Each slice is removed using a Hann window (Tukey window with $\alpha = 1$) and the slices are partially overlapped. In particular, if a slice has bandwidth B, at each step the band-stop filter is moved by $\frac{B}{2}$ MHz, in this way the whole spectrum is covered exactly once.

Dichotomic search The second algorithm we implemented is called *Bisection find-filter* and is the implementation of the dichotomic search, which helped us to check if there are cases where the SNR is so low that some components are discarded even if they contain information. It can be also used as proof for the linear independence of the slices of spectrum: if the slices are independent, this algorithm should give the same result as the Linear-Scan in less steps.

At each iteration, the spectrum is divided in half and the algorithm performs two attacks: one keeping the first half and masking out the second half and the other one by keeping only the half removed before. For each of the two attacks, if they are successful,

the algorithm is applied again after reducing the search range to only that slice. The recursion stops when the maximum depth is reached and the best shape for each of the two halves is returned. Algorithm 3.1.2 describes in pseudo-code this technique. The MERGE function used in both Algorithm 3.1.2 is described in Algorithm 3.1.1. Given two non-overlapping bit-array description of window functions in input and given a middle point, the MERGE function generates a new (description of) window function by merging the left-most bits of the first window function and the right-most bits of the second one. More precisely, the $[l, m]$ bits of the first one will be copied to the $[l, m]$ bits of the output array and the $[m+1, r]$ bits of the second one to the range $[m+1, r]$ of the output.

The functions WINDOWOF and SCOREOF reported in Algorithm 3.1.1 and Algorithm 3.1.2 extract respectively the window and the score from the tuple passed as argument. In our implementation, the tuple is a `struct` and both WINDOWOF and SCOREOF return the corresponding field of it.

Algorithm 3.1.1: MERGE function used in the bisection algorithm

Input:
- **in** $\in \{0,1\}^n$ — Original array
- `tplLeft` $\in \{0,1\}^n \times \mathbb{R}$ — Tuple containing Left window & score
- `tplRight` $\in \{0,1\}^n \times \mathbb{R}$ — Tuple containing Right window & score
- $l \in \mathbb{N}_0$ — Left pointer
- $m \in \mathbb{N}_0$ — Middle pointer
- $r \in \mathbb{N}_0$ — Right pointer

Output: **out** $\in \{0,1\}^n$ — Resulting filter,

```
1 leftWin ← WindowOf(tplLeft)
2 rightWin ← WindowOf(tplRight)
3 for i ← l to m do
4 |  out[i] ← leftWin[i]
5 for i ← m + 1 to r do
6 |  out[i] ← rightWin[i]
7 return out
```

Algorithm 3.1.2: BisectionFindFilter, search the best filter using bisection

```
Input:  ComputeScore : {0,1}^n → ℝ          Score evaluation function
        in ∈ {0,1}^n                         Filter shape
        l ∈ ℕ0                               Left pointer
        r ∈ ℕ0                               Right pointer
        remDepth ∈ ℕ0                        Remaining recursion depth
Output: arr ∈ {0,1}^n                        Resulting filter
        score ∈ ℝ                            Obtained score

   /* Termination condition                                          */
1  if l = r ∨ remDepth = 0 then
2  |   newFilt ← Write(in, l, r, 1)
3  |   newScore ← ComputeScore(newFilt)
4  |   if newScore > 0 then
5  |   |   return (newFilt, newScore)
6  |   return (in, 0)
7  m ← (l + r)/2
8  scoreLeft ← ComputeScore(Write(f, in, l, m, 1))
9  scoreRight ← ComputeScore(Write(f, in, m + 1, r, 1))
   /* Recursion                                                      */
10 if scoreLeft > 0 then
11 |   resLeft ← BisectionFindFilter(f, in, l, m, remDepth − 1)
12 if scoreRight > 0 then
13 |   resRight ← BisectionFindFilter(f, in, m + 1, r, remDepth − 1)
   /* Evaluation                                                     */
14 if ScoreOf(resLeft) = 0 ∧ ScoreOf(resRight) > 0 then
15 |   return resRight
16 else if ScoreOf(resRight) = 0 ∧ ScoreOf(resLeft) > 0 then
17 |   return resLeft
   /* Merge                                                          */
18 finalFilter ← Merge(in, resLeft, resRight, l, m, r)
19 finalScore ← ComputeScore(finalFilter)
20 return (finalFilter, finalScore)
```

Genetic We used a genetic algorithm to perform exhaustive validation, because it was impossible, as reported before, to perform by using a simple brute-force set of attacks. More precisely, we worked performing a subdivision of the spectrum from 60 to 120 slices, hence a standard brute-force attack would had required from 2^{60} to 2^{120} attacks. The pseudo-code of the genetic algorithm we used is reported in Algorithm 3.1.3, which is based on the following skeleton:

1. Initialise the population
2. Evaluate fitness
3. Check exit conditions and save best filter
4. Selection (where the probability to be chosen is proportional to the fitness)
5. Crossover
6. Mutation
7. Replacement of the population
8. Go to (2)

The genetic algorithm we have implemented left us some choices about which function to use for each step. It needs a fitness function and a set of algorithm for the selection, crossover and mutation parts. We mainly used two algorithms for the selection part: the first one, reported also in Algorithm 3.1.3, picks two random solutions using a Normal distribution and uses them. The other one, instead, picks one solution using a Random Number Generator (RNG) based on Normal distribution and the other is picked by choosing the best solution of the current iteration.

PickRandomNormal($\mathbf{x}$) : $\{\Sigma\}^n \rightarrow \Sigma$ is a function which chooses a random item from the array passed as argument using a Normal distribution, where $\mu = 0$ and $\sigma =$ SizeOf($\mathbf{x}$). The Normal distribution is used because a solution with higher score should have a higher chance to be chosen. If the chosen element does not exists, (for example because the RNG chooses a negative number or a number outside the bounds), another number is generated until a valid item is chosen.

UniformRandomBits(n) returns a vector of random bits ($\{0, 1\}$) with uniform distribution, whose length is n.

$\textsc{IndexOf}(\mathbf{x}, \mathbf{y}) : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{N}_0$ is a function which, given an array of elements $\mathbf{x}$ and an element y, finds and returns the position of the element y in the array $\mathbf{x}$. If there are multiple elements y, the index of the first is returned.

$\textsc{SortByScore}(\mathbf{x}) : (\{0,1\}^n \times \mathbb{R})^m \rightarrow (\{0,1\}^n \times \mathbb{R})^m$ is a function which, given an array of m tuples made of solutions and corresponding score, performs a reverse-sort of the array, so that the first element of $\mathbf{x}$ is the one with the highest score.

$\textsc{ComputeScore}(\mathbf{x}) : \{0,1\}^n \rightarrow \mathbb{R}$ is the fitness evaluation function, which is passed as argument and works in exact the same way of the linear-scan or bisection-scan algorithms. We tried all the scoring functions described above, focusing on `MinTracks` and `MaxGap`, which in practice performed better than the other functions.

$\textsc{CrossOver}(\mathbf{x}, y, z) : \{0,1\}^n \times \{0,1\}^n \times \mathbb{N}_0^2 \rightarrow \{0,1\}^n$ performs the genetic crossover between two solutions. Any function which mixes two solutions $f : \{0,1\}^n \times \{0,1\}^n \times \mathbb{N}_0^2 \rightarrow \{0,1\}^n$ is a valid crossover function. Our implementation consisted in choosing two random points in the first solution and replace the subsequence between them with the corresponding part taken from the second solution.

$\textsc{Mutate}(\mathbf{x}) : \{0,1\}^n \rightarrow \{0,1\}^n$ flips some bits in the solution. Our current implementation chooses the number of flipped bits using a uniform random generator: $\alpha = rand([0, 0.1]) \cdot l_s$, where l_s is the length of the binary vector representing the solution. This means that the mutation rate is in range $[0, 10\%]$. When the number α of bits to be flipped is chosen, the function picks α random bits (using a uniform distribution) in the solution and flips them. A flipped bit is not removed from the choices of the random generator, which can choose it again and it could be flipped twice.

Algorithm 3.1.3: Genetic FindFilter

```
Input:  populationSize ∈ N_0                Size of the population
        ComputeScore : {0,1}^n → R          Score evaluation function
        maxIterations ∈ N_0                 Number of maximum iterations
Output: filter ∈ {0,1}^n                    Bit-array description of the optimal filter
/* Initialization                                                    */
1  bestFilter ← {0}^n
2  curIteration ← 0
3  population ← {0^n}^populationSize
4  for i ∈ population do
5  |  i ← UniformRandomBits(n)
/* Main Loop                                                         */
6  while true do
7  |  curIteration ← curIteration + 1
8  |  scores ← {0}^populationSize
   |  /* Evaluate fitness                                            */
9  |  for i ∈ population do
10 |  |  scores[i] ← ComputeScore(i)
   |  /* Save best filter                                            */
11 |  curBestFilter ← IndexOf(scores, max(scores))
12 |  if scores[curBestFilter] > ScoreOf(bestFilter) then
13 |  |  bestFilter ← population[curBestFilter]
14 |  if curIteration > maxIterations then
15 |  |  return bestFilter
   |  /* Selection and crossover                                      */
16 |  newPopulation ← {0^n}^populationSize
17 |  population ← SortByScore((population, scores))
18 |  for i ∈ newPopulation do
19 |  |  a ← −1, b ← −1
20 |  |  while a = b do
21 |  |  |  a ← PickRandomNormal(population)
22 |  |  |  b ← PickRandomNormal(population)
23 |  |  i ← CrossOver(newPopulation, a, b)
   |  /* Apply mutation                                              */
24 |  for i ∈ newPopulation do
25 |  |  Mutate(i)
   |  /* Replace population                                          */
26 |  population ← newPopulation
```

3.1.3 AM-like leakage detection

Assuming that the leakage is compatible with the model described in the previous section, we can now proceed to analyze exhaustively all these leakages, by considering the information as it is AM-modulated. To do that, we have three free variables: The carrier frequency f_c, the width of the sidebands B, the distance d_{bw} between the carrier and the center of both sidebands.

The first free variable f_c, is clearly the clock frequency or any multiple of it, $f_c = n \cdot f_{clk}$, with $n \in \mathbb{N}$. The other two variables are searchable exaustively, by building a tool which tries all the window functions in a given range and plots all of them in a tridimensional plot (or bidimensional image, using the color as third axis).

The algorithm we developed works on two axis: f_c is fixed to f_{clk} $(n = 1)$, the first axis represents the distance d_{bw} and it is searched by sliding a Hann window (Tukey window with $\alpha = 1$) in the frequency domain. The second axis, instead, represents the bandwidth of the sidebands (B) and is searched simply by changing the bandwidth of the chosen window. In this way we try all the windows having a center frequency $f_c = f_{clk} \pm d_{bw}$, with $d_{bw} \in \mathbb{R}$ and $f_{min} \leq f_c \leq f_{max}$ and a bandwidth $B_{min} \leq B \leq B_{max}$ with $B \in \mathbb{R}$. For each window tried, a score is obtained using a ScoreFun. The results are drawn with a tridimensional plot, which shows on the X axis the distance from the clock; on Y axis the window width and on the Z axis the score of the attacks.

This plot can be used to discover immediately which zones of the spectrum leak and how much. If the hypothesis written in Paragraph 3.1.1 is correct, then we should see that the leakage is placed symmetric with respect to the f_{clk}. We do not expect a perfect symmetry in the amplitude because of the noise. We can modify this algorithm so that two window function are used at the same time, both with the same bandwidth. The former is placed around the upper-sideband $f_{clk} + f$ and the latter at the lower-sideband $f_{clk} - f$. With this modification, we consider the leakage as its AM-modulated on the clock carrier and, assuming that the background noise is uniform, the score of each attack should improve.

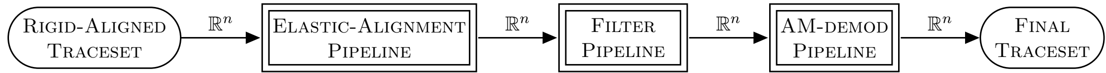


Figure 3.1: The DSP pipeline used to filter traces

3.2 DSP Pipeline for Side-Channel Attacks

In Chapter 2 there are described the most common DSP filtering techniques, the Amplitude Modulation (AM) and trace-alignment algorithms: rigid using the Cross-Correlation and elastic using the Dynamic Time Warping (DTW). Before going on, we remind the definitions of rigid-alignment and elastic-alignment.

Rigid-Alignment: Given two power traces $\mathbf{t}_1$ and $\mathbf{t}_2$ representing two instances of the same phenomena (e.g. the power consumption of a CPU while performing the same actions), the rigid-alignment consist in shifting $\mathbf{t}_2$ so that the point-wise distance between the two traces is minimized.

Elastic-Alignment: Given two power traces $\mathbf{t}_1$ and $\mathbf{t}_2$ as before, the elastic-alignment consists in warping $\mathbf{t}_2$ by stretching or compressing non-linearly so that the distance between them is minimized. The DTW is an algorithm to find a warp-path which minimizes a score, e.g. euclidean distance between the traces.

In this section we show a DSP pipeline built on these concepts, that, if applied to each trace of a trace-set, improves the SNR and the score of a CPA attack using them, which means that a key-recovery attacks becomes feasible using less power traces. If the trace-set is acquired using the setup based on the oscilloscope, the trace-set is already rigid-aligned: the power consumption related to the first instructions of the Advanced Encryption Standard (AES) encryption of each trace start at the same sample. Otherwise, if we are using a setup based on the SDR, after the acquisition we have a set of unprocessed IQ-modulated traces and, before applying this DSP pipeline we must convert them to a stream of real samples (by performing a signal reconstruction), cut them and perform at least a rigid-realignment; this preprocessing is described in detail in the next section. Once we have a trace-set made of real rigid-aligned traces, we can use the DSP pipeline depicted in Figure 3.1 to apply elastic alignment, filtering and finally AM-demodulation. The second block, called *Elastic-Alignment Pipeline*, is described in detail in the next paragraph. Basically, it warps every trace of the set to correct the clock drifting and other phenomena that may cause a misalignment of the last samples even if the first samples are perfectly aligned. After this step, all traces in the traces

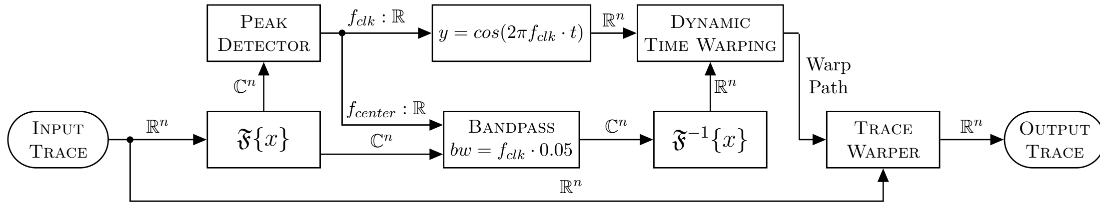


Figure 3.2: The Elastic-Alignment Pipeline

are aligned up to the clock-cycle from the beginning to the end. The Filter Pipeline consists in removing the frequency components not useful to the attack by applying a filter obtained as described in Section 3.1, performing a point-wise multiplication in the frequency domain. The resulting trace is then AM-demodulated by the last block (AM-demod Pipeline) by considering as carrier frequency the clock frequency of the target device.

3.2.1 Elastic-Alignment Pipeline

After we had performed a rigid-alignment of the trace-set by the use of the oscilloscope or the pipeline described in Section 3.3.1, given two traces $\mathbf{t}_1$ and $\mathbf{t}_2$ taken from the same trace-set, the sample representing the power consumption of the first instruction (it can be seen as the first sample) of both $\mathbf{t}_1$ and $\mathbf{t}_2$ should be now perfectly aligned, this is a precondition to the pipeline we are going to describe. Due to the fact that the clock generator of any electronic device is sensitive to temperature, the frequency of the clock signal generated changes up to 5%. This change happens fast enough that, even if the first sample of $\mathbf{t}_1$ is aligned with the first sample of $\mathbf{t}_2$, the Nth sample of $\mathbf{t}_1$ is usually not aligned with the Nth sample of $\mathbf{t}_2$. Also, two traces never have the same length. We must stretch t_2 so that the length of both traces becomes equal and all the clock stems become aligned from the beginning to the end. To do this, we use the FastDTW, a fast Dynamic Type Warping algorithm, introduced in Chapter 2.

A filtering pipeline which corrects the clock drifting is depicted in Figure 3.2. This pipeline is based on the DTW algorithm already described in Section 2.3. The easiest way to build a pipeline based on the DTW consists in warping the current trace so that the distance between that and the first trace of the set is minimized. We tried this approach but, in practice, the noise made the DTW find a warp path which - even if minimizes the distance between the two traces - does not align the traces as we wanted, voiding in this way the usefulness of this approach. This is because the SNR is too low

and the DTW aligns the noise instead of the signal which carries information. Hence, we built this pipeline which avoids the issues generated by the noise by applying the DTW after the application of a bandpass filter that keeps only the clock frequency and the closest surroundings. In this way, the DTW tries to build a path which warps the real clock signal to become similar to a generated cosine wave. At first, the Discrete Fourier Transform (DFT) is applied to the input trace, so that a representation in the frequency-domain is used. After that, the *Peak Detector* detects the clock frequency by searching the highest peak in the spectrum of the trace. In practice, knowing the clock frequency allows to skip this step and force the generation of a cosine wave using the precomputed frequency. A zero-phased cosine-wave having $f = f_{clk}$ is then generated and, at this point, the DTW is used to build a path which is able to warp the band-passed clock signal ($f_c lk \pm 5\%$) so that it resembles the generated one. The amplitude of the spectral component generated by the clock signal is dominant in the pass-band area of the filter, making the DTW work as expected. Once that the warp path is built, the original trace is elastically warped, correcting in this way all the issues derived from the jitter of the hardware clock generator.

3.3 Attacking with a SDR

If the hypothesis described in Section 3.1.1 is correct, then we should be able to perform a full attack using a SDR. The process of obtaining traces using a SDR is difficult because of two main problems: alignment and size of the traces. The alignment problem is caused by the unavailability of an accurate trigger mechanism as the one available with the oscilloscope. In Section 3.3.1 we describe how we managed to record and align all the traces. Due the fact that there is no trigger mechanism, the traces obtained using the SDR capture longer time spans than the ones obtained using the oscilloscope; after the signal reconstruction is performed, the size of each trace becomes proportional both to the length of the trace and to the clock frequency. The size of each trace becomes a problem when we are working with CPUs clocked to high frequencies. Section 3.3.2 shows an alternative way to perform signal reconstruction so that the size of the trace is independent from the clock frequency.

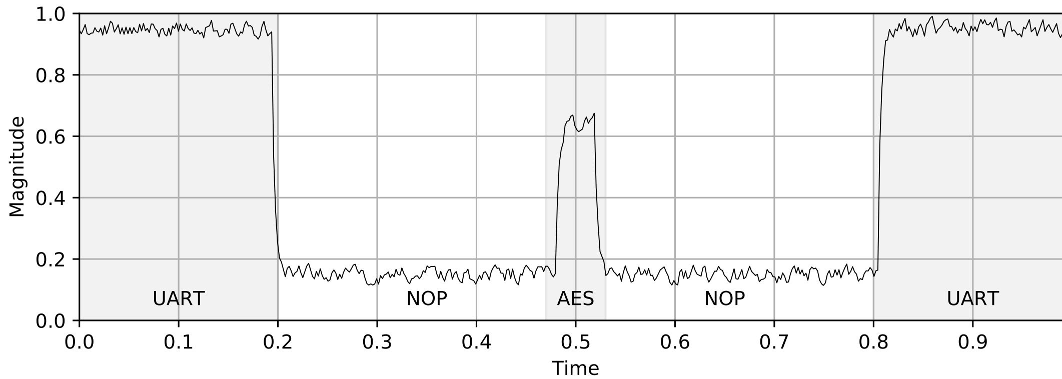


Figure 3.3: Expected magnitude of a raw trace captured using the SDR

3.3.1 SDR Acquisition pipeline

To record traces and perform the rigid alignment, we used the pipeline shown in Figure 3.4. We wrote a software (running on the target board) which communicates with a host machine using the Universal Asynchronous Receiver-Transmitter (UART) interface. The host machine starts the recording using the SDR and sends a plaintext to the target board, which performs the AES encryption and sends the ciphertext back via the UART interface. The AES encryption routine running in the target board is surrounded with `NOP`s (before and after), so that the power consumption of the encryption is easily machine-recognizable, even in noisy conditions. We expect each trace recorded using the SDR to have a magnitude similar to the one depicted in Figure 3.3, which can be divided in five parts. The first part (extremely similar to the last) consists in a high energy emission due to the UART transmission; a second part (similar to the fourth) with low energy, where the board runs through the `NOP`-sled and a third part with middle-to-high energy, almost centered with respect to the recorded trace, where the board is computing the AES encryption. Regarding the pipeline depicted in Figure 3.4, the SDR Source and the Plaintext Generator are the sources of our traceset. At first the IQ-demodulated signal coming from the SDR is modulated using the $IQ \rightarrow RF$ block, which performs the *Reconstruction on a different carrier* ($f_{carrier} = \frac{BW}{2}$) described in detail in Section 3.3.2. After that block, the resulting signal is made only of real samples and it is proportional to the power consumption of the CPU. After that, a pattern-matching algorithm implemented in the Coarse Cutter detects the five areas depicted in Figure 3.3, finds the third area, corresponding to the EM emissions of the AES encryption (surrounded by the `NOP`s) and cuts the trace from slightly before the

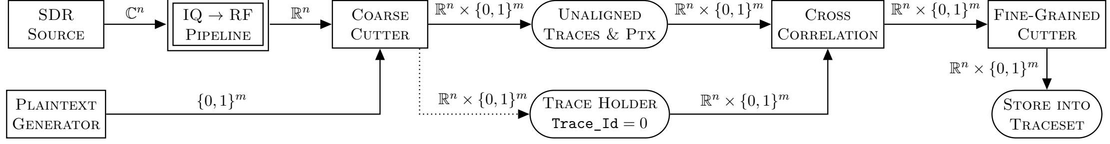


Figure 3.4: SDR Trace acquisition pipeline

AES encryption to slightly after, e.g. it keeps the range $[0.45, 0.55]$ in Figure 3.3, so that no useful information is lost and there is almost no memory wasted by storing the non-informative EM emissions while the CPU is running a `NOP`-sled. At this point, given the current trace and the first trace of the traceset (modeled in Figure 3.4 using a Trace Holder), the Cross-Correlation block finds a time delay that, if applied to the current trace, minimizes the distance between that and the first trace of the set. This time delay will be used by the *Fine Cutter* to rigid-align this trace with all the other traces of the set. The Coarse Cutter block is necessary, otherwise the Cross-Correlation matches the AES peak of the current trace with the high-energy noise generated by the UART transmission, resulting in a wrong time-delay applied to the current trace.

3.3.2 Signal reconstruction on a different carrier

The standard signal reconstruction technique described in Section 2.1.4 consists in performing the inverse transform of the In-Phase/Quadrature (IQ)-demodulation, which moves back the center frequency of the acquired signal to its original position. The SDR has limited bandwidth B, the interpolation factor N_I used in the up-sampling step is dependent from the target center frequency, hence an IQ-demodulated trace whose size was only linearly-dependent from the bandwidth B of the SDR after the reconstruction becomes also proportional to the target frequency. The standard signal reconstruction is, then, not useful to our purposes because due to the fact that the effective bandwidth is B, the upsampling corresponds to performing an extremely dense interpolation without any improvement in the contents. The need for real signals to perform CPA attacks makes a frequency shift and a $\mathbb{C} \to \mathbb{R}$ transform necessary.

In this section, we present an alternative technique to perform a signal reconstruction suitable for our purposes, which allows to perform CPA attacks on the reconstructed power traces and whose size of the traceset is not dependent from the clock frequency. The strict precondition for technique to work is that the input signal should be complex with a limited bandwidth B, the IQ-demodulated signal of any SDR matches this pre-

condition. Instead of up-mixing the signal back to its original frequency range, we can reconstruct a similar signal by shifting the target center frequency to a fixed number ω_2. In this way, we create a set of traces which is suitable to perform side-channel attacks (because the waveform is the same) but whose size does not depend on the clock frequency (f_{clk}) of the target device. Every step of this signal reconstruction technique is the same of the standard signal reconstruction reported in Section 2.1.4, except for the up-mix equation, which is changed to:

$$x_{IQ}(t)e^{j\omega_2 t} = x_{RF}(t)e^{j(\omega-\omega_2)t} \tag{3.2}$$

Where ω_2 is the new center frequency. To avoid lost of information and produce traces having the minimum size as possible, ω_2 should be chosen carefully (and should be the minimum of the set of valid values): an ω_2 too low causes loss of information; a too high ω_2 causes, instead, waste of space. The minimum ω_2 that allows the full reconstruction corresponds to half of the bandwidth of the SDR. In this way the reconstructed signal has the DC moved from $f = 0$ to $f = \frac{B}{2}$ and has the sampling rate of the SDR kept constant also during the attacks (the bandwidth is equal to the SDR bandwidth). Algorithm 9 performs the up-mix of an upsampled IQ signal as described in the paragraph above.

Algorithm 3.3.1: Modified UpMix algorithm. It performs $IQ \rightarrow RF$ modulation on a different carrier

Input:	**in** $\in \mathbb{C}^n$	Input IQ signal
	$bw \in \mathbb{R}$	Bandwidth of the input signal
	$f_C \in \mathbb{R}$	Carrier/Shift frequency
Output:	**out** $\in \mathbb{C}^m$	Output real signal

1 $f_S \leftarrow f_C + \frac{bw}{2}$ `/* Maximum representable frequency after upscaling */`
2 $nTimes \leftarrow \left\lceil \frac{f_S}{bw/2} \right\rceil$
3 **upSampled** $\leftarrow$ Upsample$(in, nTimes)$
4 $m \leftarrow length(upSampled)$
5 $f \leftarrow \pi \frac{f_C}{f_S}$ `/* Obtained after simplification of:` $f \leftarrow 2\pi f_C \frac{n}{f_S} \frac{1}{2n}$ `*/`
6 **for** $k \leftarrow 1$ **to** m **do**
7 | **out**$[k] \leftarrow$ **upSampled**$[k] \cdot cos(f \cdot k) -$ **upSampled**$[k] \cdot sin(f \cdot k)$
8 **return out**
9 .

Figure 3.5 depicts a sample signal reconstructed using this transform. We can see both magnitude and phase representation of the spectrum of the signal, when the correct ω_2 is used. We can see that if the SDR is centered to the clock frequency, this one is shifted to half of the bandwidth. In the example in Figure 3.5 the bandwidth is 20 MHz and the clock peak can be found at 10 MHz.

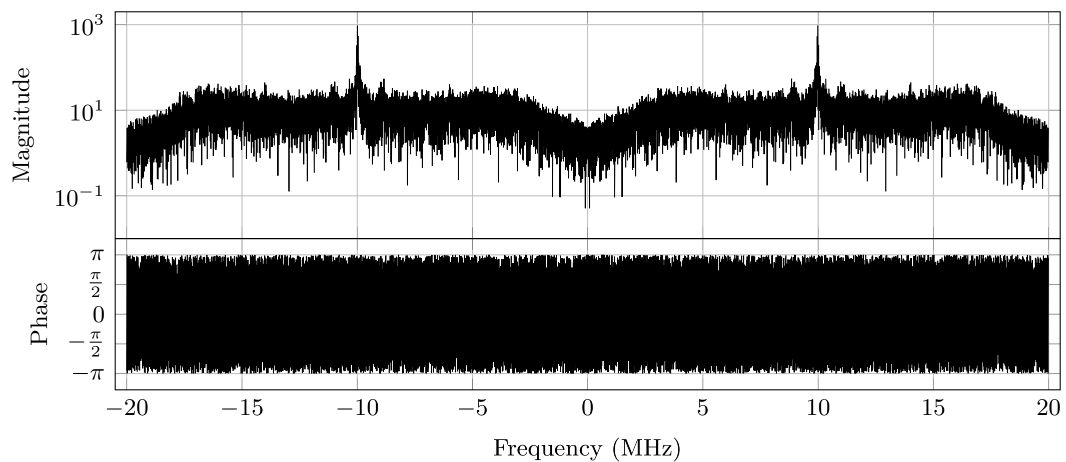


Figure 3.5: Magnitude and Phase of the signal recostructed on a different carrier

Chapter 4

Architectural Reverse Engineering

Countermeasures to side-channel attacks have been applied keeping into account only an execution model of the code as seen from an Instruction Set Architecture (ISA) level [4, 8]. However Seuschek et al. [27] and Seuschek and Rass [26] have shown a leakage behavior explainable only with a detailed instruction execution model: micro-architectural features may invalidate side-channel protections because of leakages due to internal Central Processing Unit (CPU) buffers. So, a knowledge of the microarchitectural execution model is needed to understand the internal architecture of a CPU to build better (and effective) protections against Side-Channel Attacks (SCA). Unfortunately, most of the times a Hardware Description Language (HDL) specification of the CPU is not publicly available. In such case, Barenghi and Pelosi [2] shown an innovative architecture characterization technique via Cycles Per Instruction (CPI) index: they have reverse engineered the internal architecture through the use of an oscilloscope to measure timings and by writing custom benchmarks aimed to spot the CPI index of some chosen instructions. This led to understand which instructions are able to run in dual-issue and which one not. The set of instructions that should be able to be dual-issued but in some circumstances they are not helped to reverse engineer the architecture. The characterization technique via oscilloscope works perfectly but requires:

- An oscilloscope (which quite expensive) used just for measure timing information which constrain the procedure to one device at time
- The availability of high-speed General Purpose Input/Output (GPIO) pins on the

target device that needs to be reverse-engineered

- A procedure to update the bare-metal firmware before each test-case, which requires human intervention.

In this section we present a similar technique to perform reverse-engineering of the architecture, but, which uses only the Universal Asynchronous Receiver-Transmitter (UART) interface of the target device and it is completely automatic.

4.1 Inferring architecture using CPI-index

Theoretically, a CPU is designed to run at its maximum capabilities. Therefore, it is reasonable to expect that, whenever no structural or data conflicts take place, the CPU will compute all the instructions, multiple-issuing them in the best, i.e., lowest, number of clock CPI. A direct consequence of this fact is that the CPI is a source of information on the inner architecture of a CPU. From a reverse engineering standpoint, it is interesting to note that causing data dependency conflicts can be controlled in an orthogonal fashion with respect to the structural constraints, simply by changing the operands of a given set of instructions.

For example, if the two instructions are completely independent one to the other and they use different execution units, they usually can be issued together. Instead, if a Read After Write (RAW) hazard happens, the depending instruction should be stalled. For example, given a generic modern ARM CPU, the following instructions cannot be dual-issued because they contain a RAW hazard:

```
DIV r0, r1, r2
ADD r3, r0, r0
```

Using a Cortex-M3 as a sample architecture for the execution latencies, the first instruction takes from 2 to 12 clock cycles [6] to be executed, the second instruction will wait the writeback of the results of the first before its execution happens, unless forwarding paths are present.

Given a chosen set of instruction, it is possible to determine if theoretically (using an available CPU model) they could be dual-issued or not. Considering the previous example, and assuming that the instructions are employing an orthogonal set of registers, they can only be dual-issued if enough functional units are available. In the same way, given the CPI index of a set of instructions, it is possible to infer if the available CPU

model is correct by checking if every set of instructions that theoretically should or should not run in dual issue is being executed as predicted. Also, given a set of instructions, the corresponding hazards and the CPI of a set of combination of them, it is easy to understand how many buses, registers, execution units and read/write ports are being used. This is the key principle behind that work, where they measure CPI of a set of chosen instructions and they do architectural reverse engineering using them.

4.2 From the oscilloscope to UART

In Barenghi and Pelosi [2] to obtain the CPI-index of the chosen instructions, they used an oscilloscope whose input channel was connected to a digital output pin of the target board used as trigger. In this way the oscilloscope recorded with a high precision the execution time of the microbenchmark: the digital pin was asserted before running the instructions and deasserted right after. To avoid nondeterministic execution due to caches, while retaining single cycle latencies in instruction fetches, they wrote at first a simple firmware which ran a sequence of instructions constituting the microbenchmark, enclosed between two runs of 100 `NOP`s between the trigger state-change and they measured the time to perform them. After that, they started benchmarking the instructions using a firmware which was running `100 NOPs`, `microbenchmark`, `100 NOPs`. The execution time of the `microbenchmark` was then obtained by subtracting the timing obtained by the first measurement from the timing of the second measurement.

We want to fully automate this process, so that it becomes possible to run multiple tests at the same time, aiming to obtain CPI-indices. For each test, the software should build a custom firmware which runs the chosen packet of instructions and communicates with the host machine using the UART interface of the target board; the host machine should use that channel to measure time instead of an oscilloscope. This approach has the advantage of simplifying and automating the tests, but it has two main issues that need to be solved. The first regards the UART communication: the oscilloscope they are using has a precision up to $\pm 2ns$[2], the UART serial port, instead, has a jittering which varies and it is up to few milliseconds (at 115200 bps). The second issue regards the UART interface of the host machine, which is driven by a non-realtime OS (Linux) which sometimes may introduce random latency and introduce artifacts in the results.

With this work we propose a framework that builds template-based firmwares on-the-fly, runs the tests and fills a table with all the CPI indicators. We also provide a way to reduce the UART jittering and remove the host latency, so that the measures are

Figure 4.1: The pipeline used to automatically obtain CPI from a target board

accurate enough to perform architectural reverse engineering.

Obtaining the CPI The first tool, called `GetCPI`, manages to run a single CPI-test by building a firmware, flashing it and running the tests. The pipeline depicted in Figure 4.1 shows the full procedure ran by this tool. The first block, Build The Firmare creates a custom firmware based on a template and on the specified instruction packet. After that, during Reboot the Board In Programming Mode, the board is rebooted in programming mode using `OpenOCD`. The third block Flash the new Firmware uses the UART connection to flash and verify the firmware. Reboot the target board runs `OpenOCD` and forces another reset of the target board, which now will run the new firmware and Measure the CPI indicator will run the software to perform the measurement. Our software then waits for the standard boot of the target board and starts the routine to measure CPI. All these steps are performed using a simple command-line interface:

```
./getCPI <instruction1> [instruction2 [instruction3] [...]]
```

which builds a firmware running multiple times the sequence of instructions specified via command line, flashes it into the target device and obtains the corresponding CPI index. The instruction packet written in the custom firmware consists in a long sequence of interleaved instructions passed as parameters. For example, a call to `getCPI` passing as parameters `INS1 r0, r1`; `INS2 r2, r3`; `INS3 r4, r5`; will build a firmware where at its core there is a sequence of:

```
...
INS1 r0, r1
INS2 r2, r3
INS3 r4, r5
INS1 r0, r1
INS2 r2, r3
INS3 r4, r5
...
```

```
INS1 r0, r1
INS2 r2, r3
INS3 r4, r5
      ...
```

which will run for few milliseconds. This sequence is then repeated multiple times, as described in the next paragraphs. The host machine measures the time that the target board takes to execute hundreds of thousands of instructions. This time is generally in the order of a few seconds. The host machine knows the number of instructions that the target device is executing, its working clock frequency and can easily obtain the CPI index of each instruction sequence.

Building the result-table Another tool of the framework is based on the first tool and uses it multiple times to fill up a table of CPI indices, made of the cartesian product of all the chosen instructions. Each run of `GetCPI` is done multiple times (ten times, in the current implementation), and the final CPI is obtained by averaging the CPI of the lower quarter of the test, or by taking the lowest (i.e. fastest) result. In this way any possible latency of the host machine is removed.

4.3 UART Jitter Correction

To neutralize the UART jitter, we used a combination of two techniques: the first aims at measuring the average Round Trip Time (RTT) of the UART channel; the latter, given the former, reduces the noise of this measurement so that it becomes negligible. Before each test, we compute the average RTT of the UART channel by running $N = 30$ measurements and averaging them, obtaining:

$$a_u = \frac{1}{N}\sum_{i=1}^{N} t_i \qquad \sigma_u = \sqrt{\frac{\sum_{i=1}^{N}(t_i - a_u)^2}{N-1}} \ .$$

where a_u is the average RTT and σ_u is the standard deviation of the UART RTT. During each test, we run N times the same micro-benchmark, so that the total running time makes the estimator of the running time more reliable. For each test, we collect the running time T, which can be written as:

$$T = N \cdot t_s + r_s \ ,$$

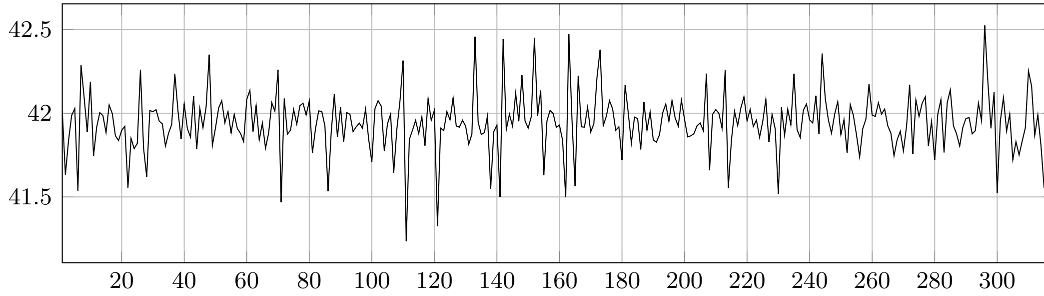


Figure 4.2: UART RTT Jittering in μs, 460800 bps

where t_s is the running time of a single micro-benchmark and r_s is the random jitter of the UART interface. T can be then written as:

$$T = N \cdot t_s + (a_u + t_\eta) \ ,$$

where t_η is the standard deviation of the jitter, a random variable. If T is large enough with respect to t_η, we can compute the time of a single micro-benchmark as:

$$t_i = \frac{T - a_u}{N} = \frac{N \cdot t_s + t_\eta}{N} = t_s + \frac{t_\eta}{N},$$

which reduces the jitter to $\frac{t_\eta}{N}$. For example, if the UART RTT is 10 $ms \pm 2ms$ and a single micro-benchmark lasts 1 us, we run 10 million micro-benchmarks, so that the total running time is $T = 10\ s$. We subtract from T the UART RTT (10 ms) and we obtain a total time of 9.99 $s \pm 0.002\ s$, which divided by the number of benchmarks is 0.999 $us \pm 0.0002\ us$.

4.4 Implementation

The firmware that our software will build and run is made using a template source code with placeholders. This template set-ups the target board, turns on the UART communication and writes a message to the host machine when it is ready to run tests.

Algorithm 4.4.1: Benchmark (Host)

```
1  Function HostBench () is
       /* UART Benchmark                    */
2      t0 ← TimePoint()
3      UARTSend (StartCMD)
4      UARTWaitFor (DoneCMD)
5      t1 ← TimePoint()
       /* Main Benchmark                    */
6      UARTSend (StartCMD)
7      UARTWaitFor (DoneCMD)
8      t2 ← TimePoint()
9      dt ← (t2 − t1) − (t1 − t0)
10     return dt
```

Algorithm 4.4.2: Benchmark (Board)

```
1  Function BRDBench () is
       /* UART Benchmark                    */
2      UARTWaitFor (StartCMD)
3      Nops ()
4      Nops ()
5      UARTSend (DoneCMD)
       /* Main Benchmark                    */
6      UARTWaitFor (StartCMD)
7      Nops ()
8      Benchmark ()
9      Nops ()
10     UARTSend (DoneCMD)
```

Each test is made of two steps: the first step measures the UART RTT and is useful to remove unwanted delays from the second step, which measures the CPI of a chosen sequence of instructions. Algorithm 4.4.1 and Algorithm 4.4.2 were used to perform our benchmarks. For each pair of instructions to test, the corresponding firmware containing the `Benchmark` function reported in Algorithm 4.4.2 has been implemented as a long sequence of these instructions interleaved.

Micro-Benchmarks A single micro-benchmark is a run of interleaved assembly instructions which aims to check if two instructions can be issued at the same time. The category of instructions we tested are the following:

- No-Operation: `NOP`
- Move operation: `MOV rA, rB`
- Unsigned ADD: `ADD rA, rB, rC`
- Signed ADD `ADDS rA, rB, rC`
- Addition with Shifting: `ADD rA, rB, rC, LSL rD`
- Multiplication: `MUL rA, rB, rC`
- Logical Shift w/, w/o immediates
- Load/Store, `LDR`, `STR`

The CPI indices of the cartesian product of the instructions reported above helped us to find: flash latencies, number of Register File (RF) read/write ports, forwarding paths, number of execution units and Arithmetic-Logic Unit(s) (ALU)/Load-Store Unit (LSU) timing. The choice of the list of instructions to be executed has been driven by the number and the type of functional units inside the target device, so that we can discover potential hazards in the functional units of the target CPU.

Chapter 5

Experimental results

This chapter describes the setup we have used and the results obtained after we had performed the experiments described in Chapter 3 and Chapter 4. The first section describes the experimental setup: the target device, the oscilloscope, the workbench and the Software Defined Radio (SDR) dongle we have used. We move then to validate the spectrum of the leakage using the oscilloscope and to characterize the parameters used in the device. Once we have validated the model, we present the results obtained by performing the Correlation Power Analysis (CPA) attacks using the SDR. The last section shows the results of the new technique we developed to perform architectural characterization without the use of an oscilloscope.

5.1 Experimental Workbench

In this section we describe the hardware and software used to perform experiments. These include a description of the target device, the measurement workbench and the SDR.

5.1.1 Device

Our target device is the board NUCLEO-F746ZG, a commercial prototyping board made by STMicroelectronics, equipped with a STM32-F746ZG Central Processing Unit (CPU) in a LQFP144 package, which is based on the ARM Cortex-M7 architecture and it can be clocked from 16 MHz to 216 MHz. This board features two Universal Serial Bus (USB) ports (one usable by the CPU and one usable both as Universal Asynchronous Receiver-Transmitter (UART) port and as programming port), an Ethernet port that

we have been keeping off for the whole time, an extensive set of General Purpose Input/Output (GPIO) and UART support also via GPIO.

The ARM Cortex-M7 architecture is a dual-issue ARMv7E-M CPU which features a 4-stage integer pipeline and an optional 5-stage Floating Point Unit (FPU) pipeline with branch prediction. It supports the Thumb-2 instruction set with Digital Signal Processing (DSP) extension (single-cycle 16/32-bit Multiply And Accumulate (MAC), single-cycle dual 16-bit MAC and 8/16-bit Single Instruction Multiple Data (SIMD) arithmetic). The target board has been powered via the USB port and we have chosen `PB0` as the GPIO trigger pin.

5.1.2 Workbench

The oscilloscope we used is a Picoscope 5203, which is able to sample up to 1 Gsamples/s using a single-channel configuration. We used it in the dual-channel mode, which consists in sampling the input of two channels at the same time. The first channel, called `Channel A` is connected to the antenna and is used to measure the electromagnetic (EM) emissions of the target device; the second channel, `Channel B`, is connected to the trigger of the target device. This configuration works with a maximum data-rate of 500 Msamples/s per channel, yielding a Nyquist limit for the non-aliased sampling of 250 MHz. Capturing at 250 MHz implies a maximum uncertainty of 4 ns (±2 ns), enough for this work because the target board is clocked at most at 128 MHz. This oscilloscope has an internal 32 MiB memory, used as store buffer for rapid acquisition.

To record the traces, we used a tool based on the Picoscope library (`libps5000`), configured with the following settings:

- The sampling sensitivity (vertical resolution) was set to 200 mV.
- The trigger threshold was set to 700 mV.
- The number of pre-trigger samples to be recorded was 50.

`Channel B` is used as trigger: it is connected to a GPIO port of the target device, which outputs 1 (3.3 V) when the board is encrypting and 0 (0.0 V) otherwise. To measure EM-emissions we used a custom loop probe made out of a 50 Ω coax cable, exposing 2 mm of the coax core, depicted in Figure 5.1. This probe is connected to the oscilloscope via two Agilent INA-10386 amplifiers and is placed on top of the SoC, centered with respect to the package borders.

5.1.3 SDR

The SDR we used for this work is the *rad1o-badge*, a HackRF clone which features a half-duplex transceiver, operating in a theoretical frequency range of [50 MHz, 4000 MHz], even if, in practice, worked perfectly down to 12 MHz. The rad1o-badge is software-compatible with the HackRF. It is based on a Maxim MAX2837 [21] Wimax Transceiver, which is designed specifically for 2.3GHz to 2.7GHz wireless broadband systems and it has an external mixer for up and down converting. The I/Q samples are sent to a NXP LPC4330FET180 [23], an ARM Cortex M4 CPU which is able to run some preprocessing, but, for our purposes, forwards the data to the host machine through the USB port.

It features an on-board antenna, which has been disabled, and it has the support for an external 50 Ω/75 Ω SMA antennas, such as the one we used. We have connected the same 50 Ω probe described in Section 5.1.2 to the input SMA connector of this device. Each In-Phase/Quadrature (IQ) sample sent by the rad1o-badge SDR to the host machine is made of two bytes, one byte (±127) is the In-Phase component and one byte (±127) is the Quadrature component. Given this information we obtain the resolution of the two ADCs: 8-bit per sample. The measured maximum data-rate via USB is 40 MB/s. Given the ADC resolution and the USB data-rate, we obtain the maximum sample-rate, 20 Msamples/s, and the maximum bandwidth of the SDR: ±10 MHz.

The rad1o-badge exposes to the HackRF driver three tunable signal amplifiers. For this work we used the following configuration:

- The antenna-port power, called `amp` in libHackRF (`RF` in OsmoSDR), has been turned on.
- The LNA gain, tunable in range $[0, 40]$ dB, has been set to 32 dB.
- The VGA gain, having range $[0, 62]$ dB, has been set to 20 dB.

Figure 5.1: The custom loop probe

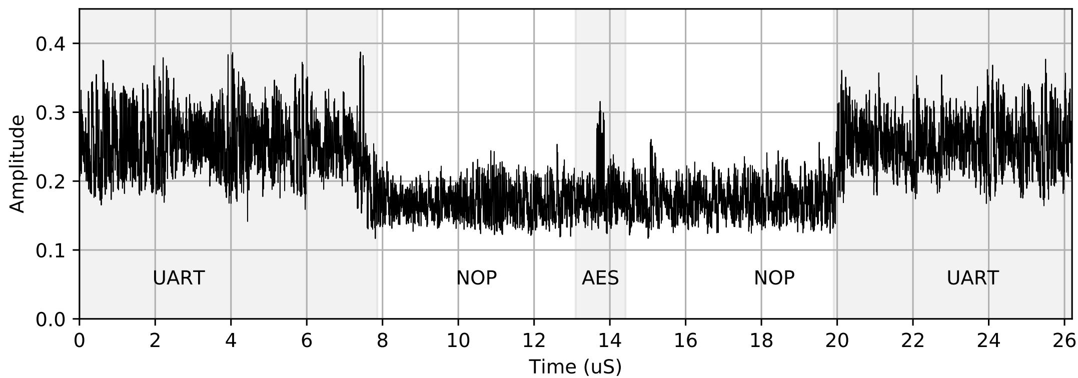


Figure 5.2: Amplitude of a trace captured using the SDR, target board clocked at 64 MHz. UART noise in $[0, 8]$ ms and $[20, 26]$ ms. AES peak at 13.8 ms surrounded by a `NOP`-sled.

Both LNA and VGA gain have been tuned so that during the capture, the signal uses $\frac{3}{4}$ of the vertical ADC range.

Software To capture a trace-set using the SDR, the host machine ran a software to collect traces using the SDR, called `sdr_dump`. This software uses `libHackRF` to read data from the rad1o SDR. It starts recording just before sending the `encrypt` command to the target device and stops the recording right after the `done` command is received. Due to the slowness of the rad1o device in starting and stopping the sampling, we keep it always running and we store only the samples that are received while the `recording` flag is raised. This software aims to collect about 1.2x the number of traces required for the trace-set because some of them (usually $\frac{1}{1000}$) are too noisy or damaged due to external factors, e.g. environmental transmissions near the clock frequency. Each trace is saved as stream of raw complex data obtained from the SDR (IQ-demodulated samples) associated with the corresponding plaintext.

Figure 5.2 shows the amplitude of IQ-demodulated samples of a collected trace using the SDR, where the target board was clocked at 64 MHz. The first 8 ms and the last 6 ms correspond to the UART transmission, the low-energy part between 8 ms and 20 ms happens while the board is running a `NOP`-sled and the peak near 13.8 ms is the EM emission happened while the target device was performing the Advanced Encryption Standard (AES) encryption of the given plaintext.

5.1.4 Filtering software

Our DSP filtering software is based on the publicly available DSP filter collection by Falco [9], which includes the implementation of Chebyshev, Elliptic and Butterworth filters and does not require any external dependency (except for the C++ Standard Library). To have a Fast Fourier Transform (FFT) in our C/C++ framework, we have used the FFTW library written by Frigo and Johnson [11]. We are also using a modified version of the OpenSCA framework [28] released by Bristol University to prototype attacks. Based on the software collection reported above, we made a new DSP side-channel toolkit called `dsp_sca_pipeline`. Figure 5.3 shows a screenshot of this tool while running some analysis in time-domain and frequency-domain.

Figure 5.3: Screenshot of the DSP tool

5.2 A DSP Approach to DPA-oriented signal processing

The aim of this experimental campaign is to validate the leakage model proposed in Section 3.1 by the use of a sampling instrumentation whose bandwidth is a proper subset of the leakage bandwidth. To perform them, we use the oscilloscope described at the beginning of this chapter, which is able to sample at 500 Msamples/s, the maximum sampling-rate available using the dual-channel configuration. In this way we can reconstruct the EM emission without aliasing up to 250 MHz. To perform the validation of the leakage model we use the methodology described in Section 3.1.2 to find which frequency components leak information. We perform the search using the algorithms described (linear-scan, dichotomic-scan and genetic) and, after a comparison of the results, we merge them using a slice-wise `AND` to find the common components, if there are any.

5.2.1 Leakage Model

The first experiment is made by clocking the target device at 32 MHz and collecting, using the oscilloscope, `15000` unaveraged traces having `110367` samples each. The antenna is placed on top of the SoC, the sensitivity of the oscilloscope is tuned to 200 mV, so that almost all the samples cover at most $\frac{3}{4}$ of the Analog to Digital Converter (ADC) sampling resolution. The collected traces are aligned at first by performing the automatic alignment available using the trigger mechanism of the oscilloscope, then by the use of the cross-correlation, which corrected sub-clock-cycle misalignment. At first an attack without filtering has been performed, so that the score obtained can be used as baseline for the search algorithms: if a search algorithm finds a window whose score, if applied, is lower than the baseline, the search is considered failed.

The attack without filters obtained a score (using the `MinTracks` function) $s = 74.07$, which means that an attack ($\alpha = 0.1$) is feasible using 13500 traces.

Linear-scan The first search is done by running the linear-scan tool, with the following configuration:

- Spectrum division: 30 slices.
- Score function: `MinTracks`.
- Pearson p-value: $\alpha = 0.1$.

The scan tool found the window function depicted in Figure 5.4, which improved the score to $s = 111.11$: an attack with $\alpha = 0.1$ is feasible using 9000 traces.

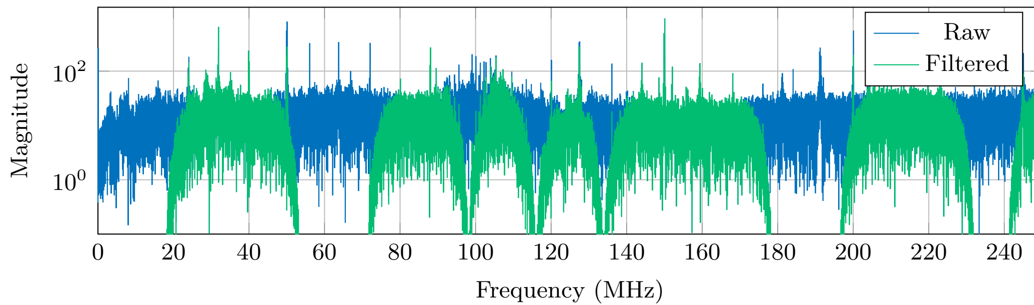


Figure 5.4: Window function found by the linear-scan tool, Cortex-M7 clocked at 32 MHz

The linear-scan algorithm designed to remove a slice per iteration (instead of keeping only once at time) keeps all the slices of the spectrum around $k \cdot f_{clk}$, with $k \geq 1$.

dichotomic-scan After the linear-scan tool, we used the dichotomic-scan to perform another search. We remind that the bisection tool tries the attack by band-passing only the selected slice for each iteration.

We configured the dichotomic-scan tool with the following configuration:

- Number of recursions: $n = 4$.
- Score function: `MinTracks`.
- Pearson p-value: $\alpha = 0.1$.
- Filter roll-off: 120 slices, hence $f_r = f_{max}/120 = 250$ MHz$/120 \cong 2$ MHz.

This algorithm found the window function depicted in Figure 5.5, which obtained a score $s = 210.53$, hence an attack after filtering the traces with this window function is feasible using 4750 unaveraged traces.

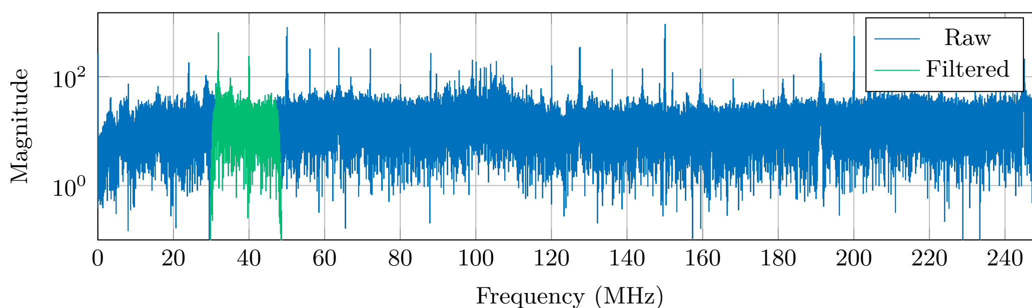


Figure 5.5: Window function found by the dichotomic-scan tool, Cortex-M7 clocked at 32 MHz

Exhaustive search After we performed the search of window functions using the linear-scan algorithm and the dichotomic-scan, we performed the pseudo-exhaustive search using the genetic algorithm described in Section 3.1.2. We configured the tool with the following parameters:

- Spectrum division: 60 slices.
- Population: 60 solutions per iteration.
- Selection pick algorithm: two random solutions using normal distribution.
- Score function: `MinTracks`.

- Pearson p-value: $\alpha = 0.1$.

The genetic algorithm found the window function depicted in Figure 5.6, which has a score $s = 333.33$. Using this window function an attack is feasible using only 3000 unaveraged traces.

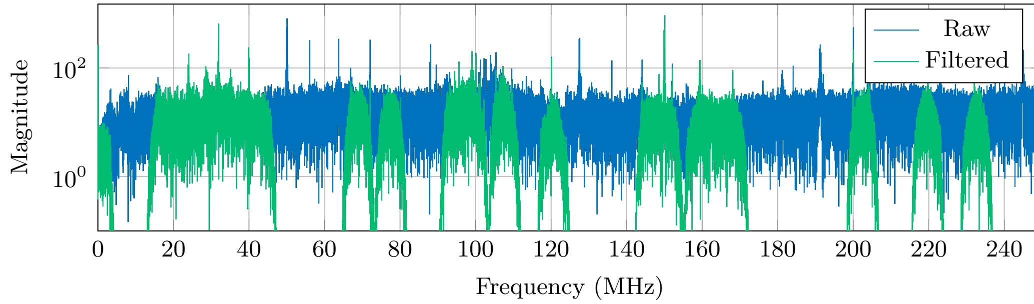


Figure 5.6: Window function found by the genetic tool, Cortex-M7 clocked at 32 MHz

Filter mix Once we got all the window functions, we found the common components between them by applying the bitwise-`AND` operation to the bit-arrays found by each tool.

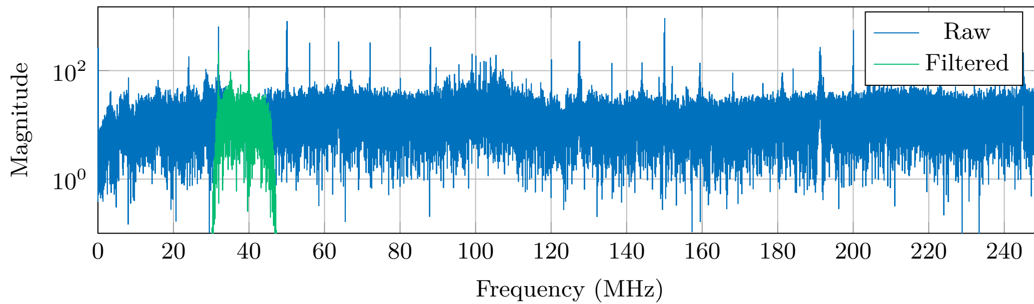


Figure 5.7: Window function found by applying a slice-wise `AND`, Cortex-M7 clocked at 32 MHz

Figure 5.7 shows the resulting window function, which is extremely similar to the one found by the bisection tool due to the fact that the window found by it is the strictest one. The linear-scan tool and the genetic tool found some components also near $f_p = k \cdot f_{clk}, k > 1$. Instead, the bisection tool found only the components near $f_{clk}, k = 1$. The resulting window function shows that the common components between all the results above are around f_{clk}, with a bandwidth of (at most) ± 10 MHz.

5.2.2 Robustness against clock frequency shift

In the previous subsection we validated the model by clocking the target device at $f_{clk} = 32$ MHz. We aim now to discover which correlation there is between a change of the clock frequency of the target device f_{clk} and the position and size of the leakage. To do that, we applied the same methodology, but this time the target device has been clocked to $f_{clk} = 64$ MHz and $f_{clk} = 128$ MHz.

64 MHz As shown before, at first we perform an attack without applying any filter. Using the raw trace-set, an attack becomes feasible with $\alpha = 0.1$ using 36000 traces. Using this number as baseline for this trace-set, we start the filter search. The acquisition setup and parameters were the same of the ones presented in the subsection above, where the f_{clk} was 32 MHz. At first we ran the linear-scan tool with the following configuration:

- Spectrum division: 60 slices.
- Score function: `MinTracks`.
- Pearson p-value: $\alpha = 0.1$.

The tool found the window function depicted in Figure 5.8, which got the score $s = 54.05$. Filtering the signal with this window function allows attacks with $\alpha = 0.1$ using 18500 unaveraged traces.

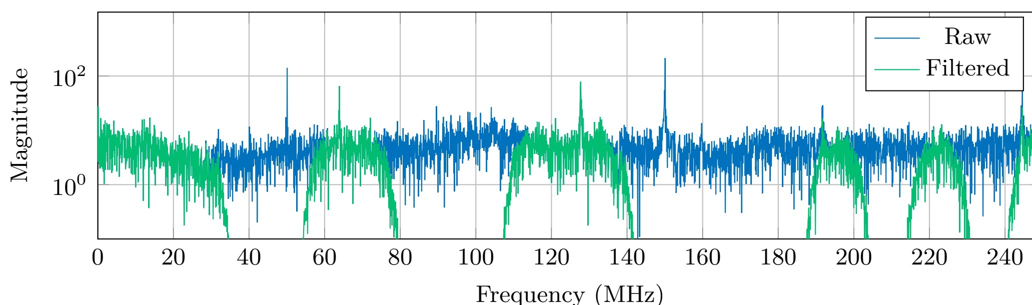


Figure 5.8: Window function found by the linear-scan tool, Cortex-M7 clocked at 64 MHz

We ran the dichotomic-scan tool on the 64 MHz trace-set, configured with the following parameters:

- Number of recursions: $n = 4$.
- Score function: `MinTracks`.

- Pearson p-value: $\alpha = 0.1$.

- Filter roll-off: 60 slices, hence $f_r = f_{max}/60 = 250$ MHz$/60 \cong 4$ MHz.

The bisection search found the window function depicted in Figure 5.9, which got a score $s = 285.71$. If this window function is applied to a signal, an attack becomes feasible using 3500 unaveraged traces.

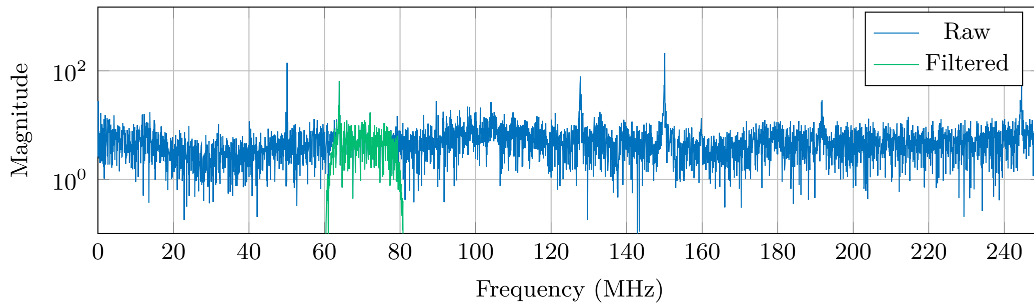


Figure 5.9: Window function found by the dichotomic-scan tool, Cortex-M7 clocked at 64 MHz

For the pseudo-exhaustive search we ran the genetic tool with the following parameters:

- Spectrum division: 60 slices.

- Population: 80 solutions per iteration.

- Selection pick algorithm: one random solution, cross-over with the best solution of the current iteration.

- Score function: `MinTracks`.

- Pearson p-value: $\alpha = 0.1$.

The genetic tool found the window depicted in Figure 5.10, which scored $s = 294.118$. Filtering a signal using this window function makes an attack feasible using only 3400 unaveraged traces.

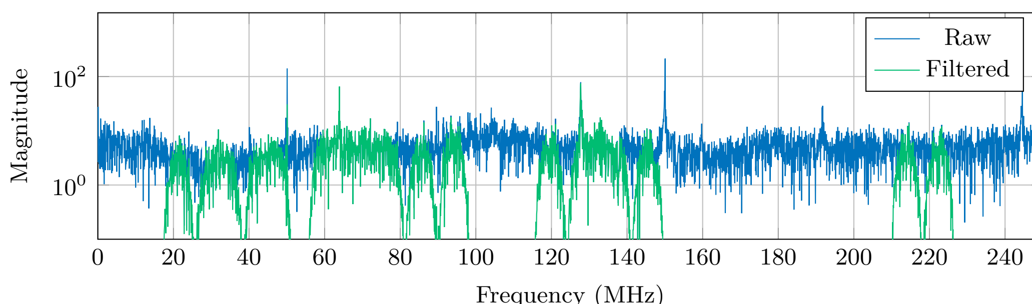


Figure 5.10: Window function found by the genetic tool, Cortex-M7 clocked at 64 MHz

Figure 5.11 depicts the window function obtained by applying a slice-wise AND between the three window function reported above. Due to the fact that the bisection tool already found the smallest window, this result is exactly the same as the one found by the bisection tool.

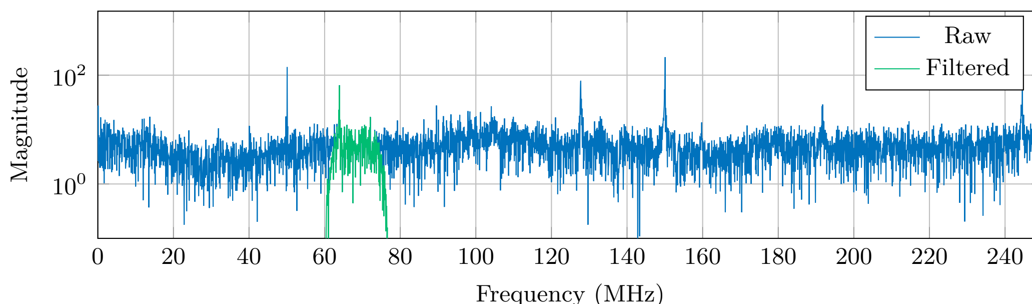


Figure 5.11: Window function found by applying a slice-wise `AND`, Cortex-M7 clocked at 64 MHz

128 MHz We applied the same methodology used for f_{clk} = 32 MHz and f_{clk} = 64 MHz. The trace-set collected with f_{clk} = 128 MHz makes an attack feasible without filtering, with $\alpha = 0.1$ using 18200 traces.

At first we ran the linear-scan tool with the following configuration:

- Spectrum division: 60 slices.
- Score function: `MinTracks`.
- Pearson p-value: $\alpha = 0.1$.

which found the window function depicted in Figure 5.12. This window function has a score $s = 161.29$, which makes an attack possible with $\alpha = 0.1$ using 6200 unaveraged traces.

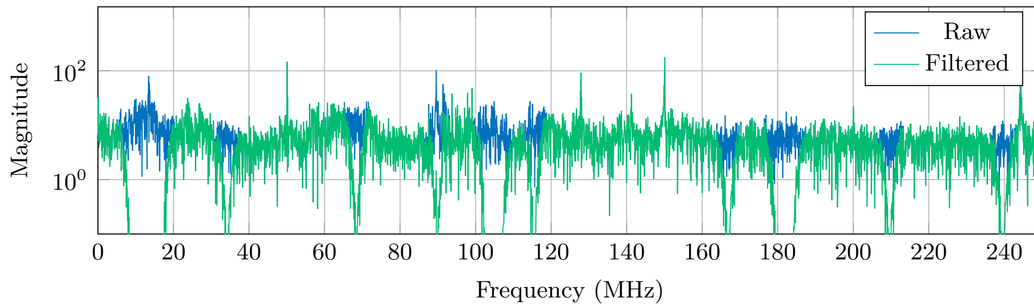


Figure 5.12: Window function found by the linear-scan tool, Cortex-M7 clocked at 128 MHz

We moved then to search the spectrum using the dichotomic-scan tool, with this configuration:

- Number of recursions: $n = 4$.
- Score function: `MinTracks`.
- Pearson p-value: $\alpha = 0.1$.
- Filter roll-off: 60 slices, hence $f_r = f_{max}/60 = 250$ MHz$/60 \cong 4$ MHz.

The dichotomic-scan tool performed extremely well in this condition, by finding a narrow window function depicted in Figure 5.13 which has a score $s = 555.56$. With this window function, an attack becomes feasible using 1800 unaveraged traces.

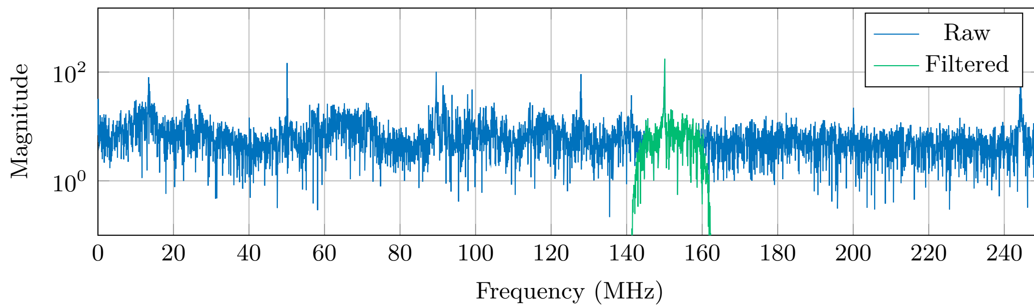


Figure 5.13: Window function found by the dichotomic-scan tool, Cortex-M7 clocked at 128 MHz

At last we ran the pseudo-exhaustive search using the genetic algorithm using the following parameters:

- Spectrum division: 60 slices.

- Population: 60 solutions per iteration.
- Selection pick algorithm: two random solutions using normal distribution.
- Score function: `MinTracks`.
- Pearson p-value: $\alpha = 0.1$.

Figure 5.14 depicts the window function found by the genetic algorithm, which performed worst than the dichotomic-scan tool yielding a score $s = 416.67$. This means that an attack (with $\alpha = 0.1$) needs at least 2400 unaveraged traces to be performed successfully.

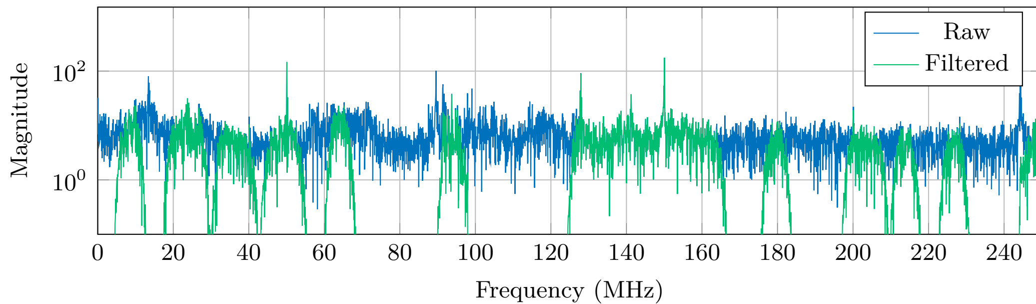


Figure 5.14: Window function found by the genetic tool, Cortex-M7 clocked at 128 MHz

We did, as before, the slice-wise **AND** between the window functions, obtaining in this way the filter depicted in Figure 5.15, which is the same as the one found by the dichotomic-scan tool.

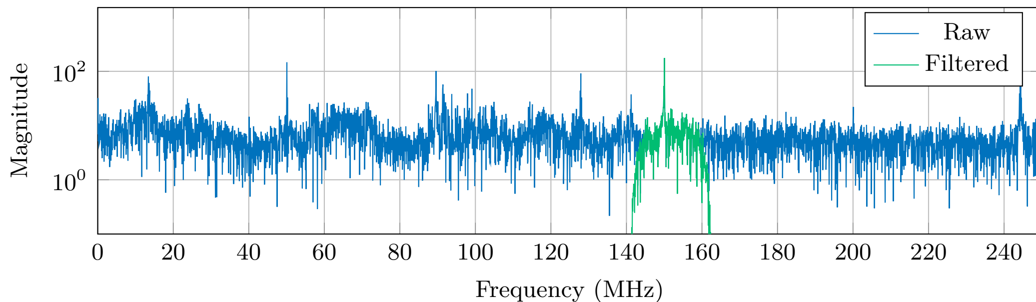


Figure 5.15: Window function found by applying a slice-wise **AND**, Cortex-M7 clocked at 128 MHz

5.3 Parameter characterization

After we validated the leakage model using the oscilloscope, we proceeded to obtain the leakage parameters for the target device, which can be uniquely identified by two parameters: the width of both upper and lower sidelobes and the distance from the clock frequency f_{clk}. The aim of this section is to determine which performance specification should a SDR have to be able to perform successfully a CPA attack.

To characterize the spectrum and identify how the leaking frequencies are spread around the spectrum, we created another tool which plots a map of leakage sections by performing a set of attacks as described in Section 3.1.3. For each attack, a band-pass filter is placed near the clock frequency and the tuple $\langle Position, Bandwidth, Score \rangle$ is stored, where the *Score* (displayed as Z-axis in the figures below) is obtained using a score function described in Table 3.1. The collection of these tuples is drawn both as 3d-plot and as heatmap, darker color means higher score. Every filter is made using one (single-sideband) or two (double-sideban) Hann window functions, which can be seen as a Tukey window having $\alpha = 1$.

5.3.1 Leakage map at 32 MHz

To perform spectrum characterization when $f_{clk} = 32$ MHz, we used the same trace-set as in Section 5.2.1, which contains 100000 traces.

To draw Figure 5.16, the leakage mapping tool was configured with the following parameters:

- Pearson p-value: $\alpha = 0.1$.
- Score function: `MinTracks`.
- Search step: $\Delta f = 0.1$ MHz.
- Search range: $f \in f_{clk} + [-10, +10]$ MHz.
- Bandwidth range: $BW \in [0.5, 5]$ MHz.
- Window type: Single window, Hann window in $[f - BW/2, f + BW/2]$.

Two separate leaking zones were found, as depicted in Figure 5.16. The former has center in $f = f_{clk} - 8$ MHz, the latter is centered in $f = f_{clk}$ and from the picture we see it extends in range $f_{clk} \pm 1.25$ MHz. As the reader can see, Figure 5.16 depicts only the areas with leakage: the range $[5, 10]$ MHz has been removed because it is empty.

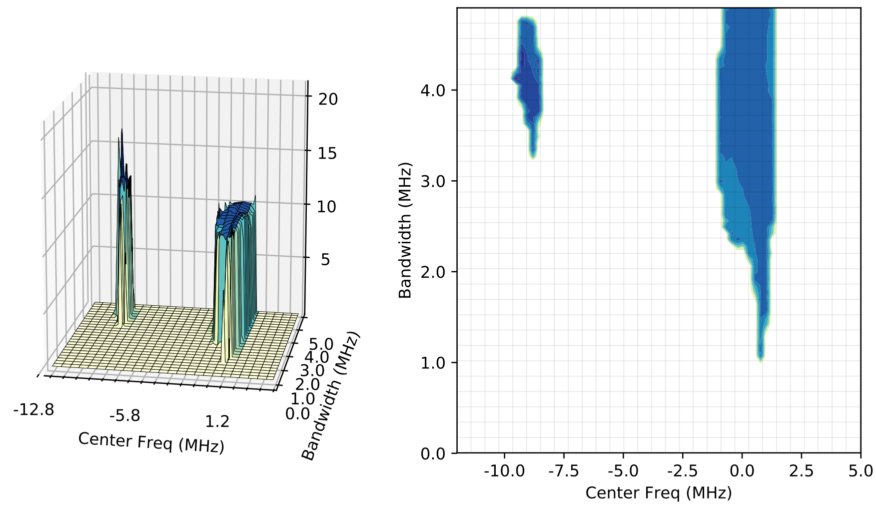


Figure 5.16: 3D-plot and heatmap, $f_{clk} = 32$ MHz, $\Delta f \in [-10, 10]$ MHz

To visualize the leakage area more in detail, we ran again the leakage map tool, configuring it so that the search range is restricted to the interesting area and the figure shows a higher resolution map of the leakage around the clock. We obtained in this way Figure 5.17, which shows a symmetric leakage with respect to the clock frequency (except for some noise), the highest peaks are at ± 0.5 MHz and the optimal

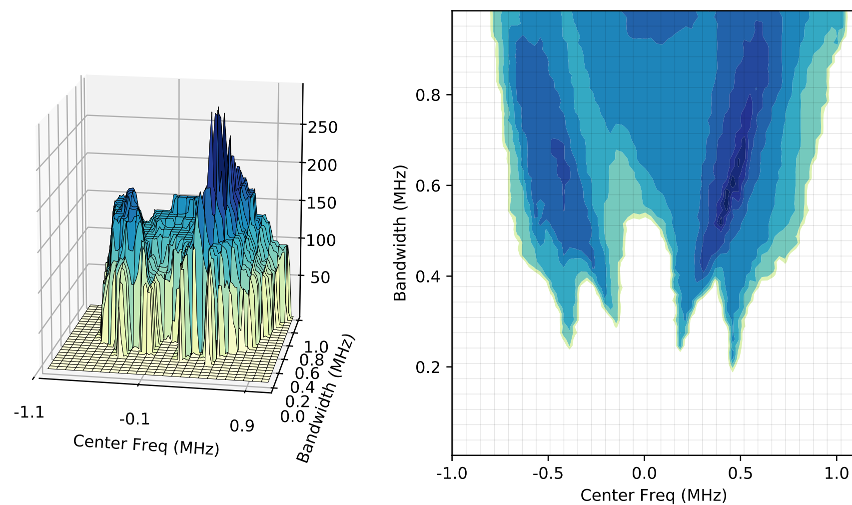


Figure 5.17: 3D-plot and heatmap, $f_{clk} = 32$ MHz, $\Delta f \in [-1, 1]$ MHz

bandwidth is 0.6 MHz, hence the optimal sidelobes for AM demodulation are placed at $f_{clk} \pm [0.2 \text{ MHz}, 0.8 \text{ MHz}]$.

To perform the scan depicted in Figure 5.17, the tool was configured with the following parameters:

- Pearson p-value: $\alpha = 0.1$.
- Score function: `MinTracks`.
- Search step: $\Delta f = 0.02$ MHz.
- Search range: $f \in f_{clk} + [-1, +1]$ MHz.
- Bandwidth range: $BW \in [0.01, 1]$ MHz.
- Window type: Single window, Hann window in $[f - BW/2, f + BW/2]$.

Double window After we validated that the leakage is located around the clock frequency and that is symmetric with respect of it, we performed a new analysis by modifying the leakage map tool so that it uses two symmetric windows having the same bandwidth and placed at the same distance with respect to f_{clk}.

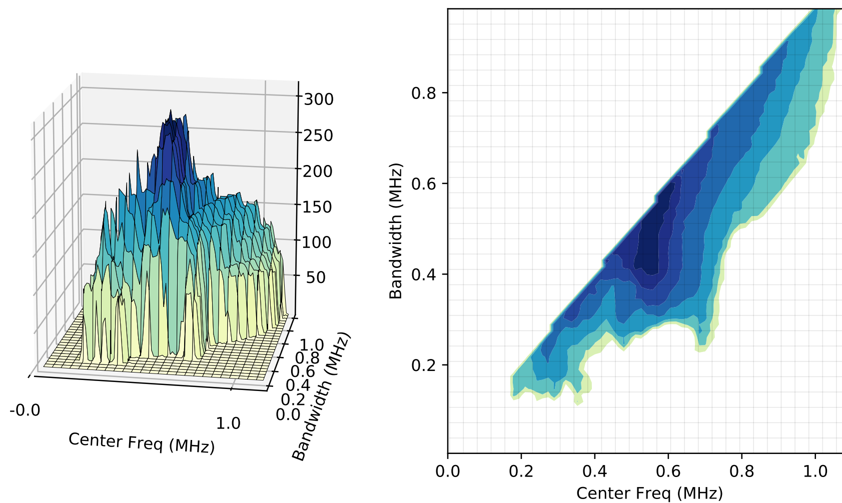


Figure 5.18: 3D-plot and heatmap, $f_{clk} = 32$ MHz, $\Delta f = [0, 1]$ MHz, double-sideband

To draw Figure 5.18, the tools was configured using the following parameters:

- Pearson p-value: $\alpha = 0.1$.
- Score function: `MinTracks`.
- Search step: $\Delta f = 0.02$ MHz.
- Search range: $f \in [0, 1]$ MHz.
- Bandwidth range: $BW \in [0.01, 1]$ MHz.
- Window type: Double symmetric window, Hann window in $f_{clk} \pm [f - BW/2, f + BW/2]$.

The window obtained using this configuration is depicted in Figure 5.18, shows - as the one depicted in Figure 5.17 - that the highest peak is located in range $f \in [0.55, 0.6]$ MHz, with bandwidth 0.4 MHz. We remind that the plot is skewed because in the upper triangle (where $Bandwidth > CenterFreq$) the two sidebands are overlapping. With this plot we can clearly see the best spots to place the double-sideband filter and perform Amplitude Modulation (AM)-demodulation.

5.3.2 Leakage map at higher frequencies

We applied the same methodology described in Section to characterize the spectrum when the clock of the target board is set to $f_{clk} = 64$ MHz and $f_{clk} = 128$ MHz. In this way we can check if there is a correlation between CPU clock frequency and leakage zones.

64 MHz The first test is done using the trace-set with 100000 traces collected before, where the target board was clocked to $f_{clk} = 64$ MHz. The tool was configured with the following parameters:

- Pearson p-value: $\alpha = 0.1$.
- Score function: `MinTracks`.
- Search step: $\Delta f = 0.1$ MHz.
- Search range: $f \in f_{clk} + [-5, +5]$ MHz.
- Bandwidth range: $BW \in [0.1, 1]$ MHz.

- Window type: Single window, Hann window in $[f - BW/2, f + BW/2]$.

Figure 5.19 depicts the results of this test, where a leakage in range $f \in [-2, 2]$ MHz is found. We see that even in this case, there are two leakage areas, symmetric with respect to the center frequency ($f_{clk} = 64$ MHz).

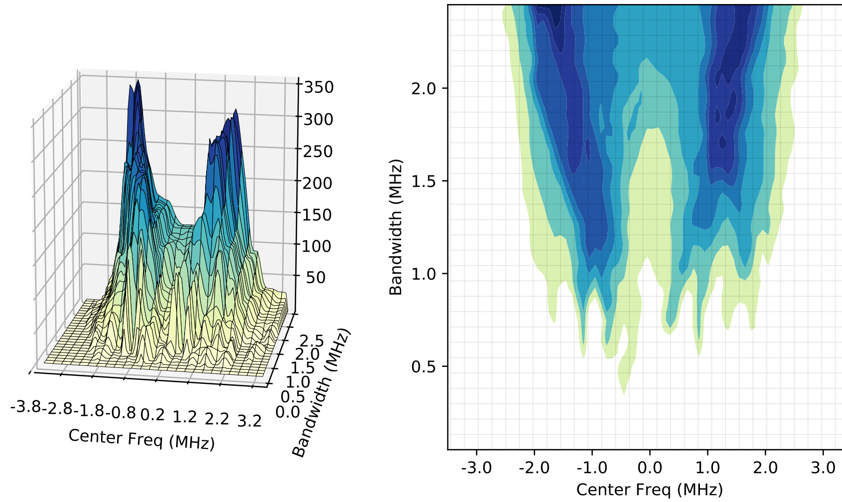


Figure 5.19: 3D-plot and heatmap, $f_{clk} = 64$ MHz, $\Delta f \in [-5, 5]$ MHz, restricted to the leaking area

We performed, then, the search using a double-sideband window, to spot components which improve leakage if taken together (AM-modulated components). To do that, we configured the tool with the following parameters:

- Pearson p-value: $\alpha = 0.1$.
- Score function: `MinTracks`.
- Search step: $\Delta f = 0.02$ MHz.
- Search range: $f \in [0, +3]$ MHz.
- Bandwidth range: $BW \in [0.01, 3]$ MHz.
- Window type: Double symmetric window, Hann window in $f_{clk} \pm [f - BW/2, f + BW/2]$.

With this configuration we obtained the window depicted in Figure 5.20, which shows two peaks when the center frequency is near $f_{clk} \pm 1.0$ MHz and the bandwidth is respectively 1.0 MHz and 1.2 MHz.

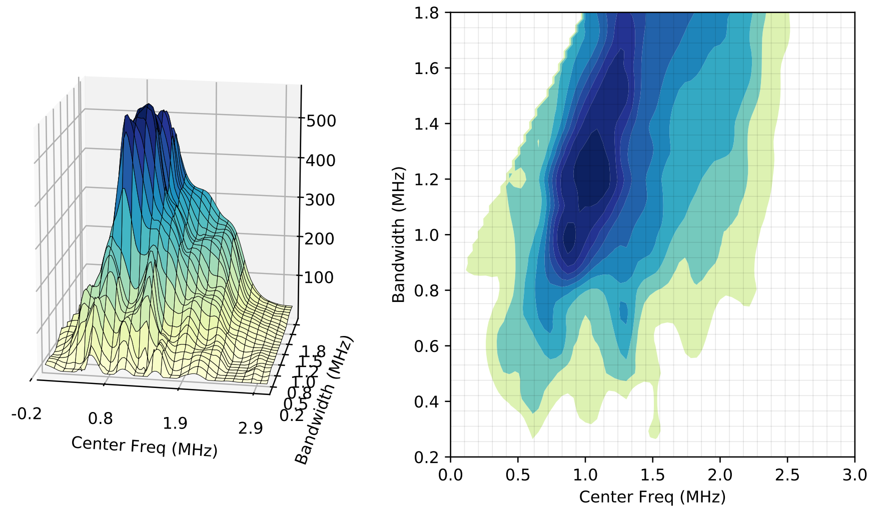


Figure 5.20: 3D-plot and heatmap, double-sideband, $f_{clk} = 64.1$ MHz, $\Delta f \in [0, 3]$ MHz

To have a global overview, we ran the tool to perform a coarse-grained scan of the whole spectrum, configuring it with the following parameters:

- Pearson p-value: $\alpha = 0.1$.
- Score function: `MinTracks`.
- Search step: $\Delta f = 0.5$ MHz.
- Search range: $f \in [0, 250]$ MHz.
- Bandwidth range: $BW \in [0.05, 0.5]$ MHz.
- Window type: Single window, Hann window in $[f - BW/2, f + BW/2]$.

Figure 5.21 depicts a plot of the whole spectrum, where the slice around the clock is the one analyzed in detail above. There are three main leaking zones: one before 20 MHz, one around the clock (64 MHz) and the last one at $2 \cdot f_{clk}$.

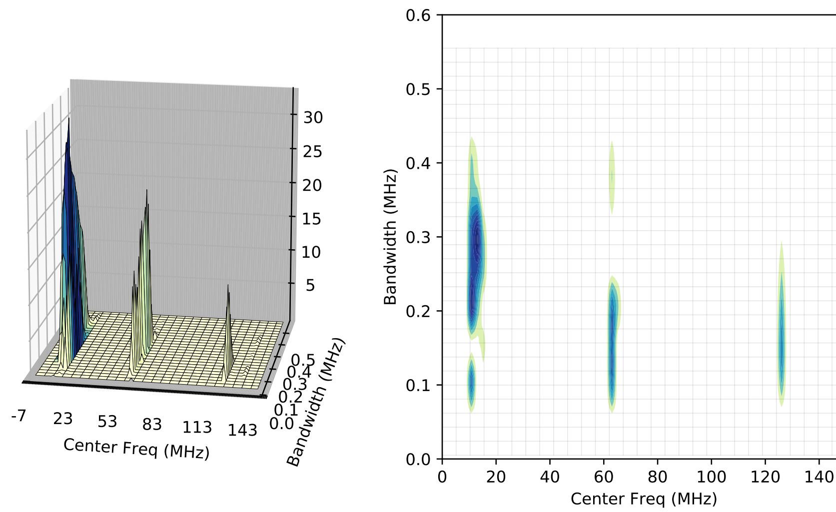


Figure 5.21: 3D-plot and heatmap, $f_{clk} = 64$ MHz, $f \in [0, 250]$ MHz; plot restricted to zones with leakage.

128 MHz The last test was performed by clocking the target board to $f_{clk} = 128$ MHz. In this case we expect only a single leakage due to the fact that the bandwidth of the oscilloscope is 250 MHz. Figure 5.22 depicts the leakage map of the whole spectrum,

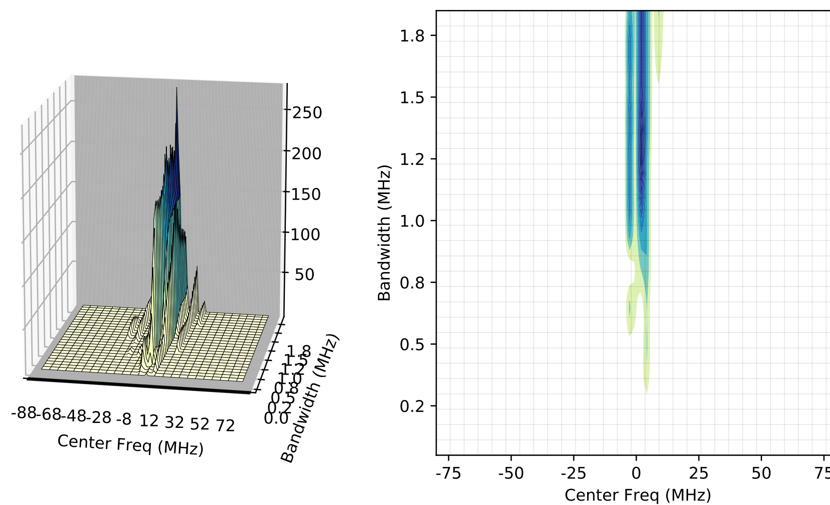


Figure 5.22: 3D-plot and heatmap, $f_{clk} = 128$ MHz, $f \in [0, 250]$ MHz restricted to $f_{clk} \pm 75$ MHz

with the figure restricted to ±75 MHz. The result of this scan shows, as expected, a single leaking zone near the clock frequency. The map reported in Figure 5.22 was obtained by configuring the tool with the following parameters:

- Pearson p-value: $\alpha = 0.1$.
- Score function: `MinTracks`.
- Search step: $\Delta f = 0.03$ MHz.
- Search range: $f \in [0, 250]$ MHz.
- Bandwidth range: $BW \in [0.1, 4]$ MHz.
- Window type: Single window, Hann window in $f_{clk} \pm [f - BW/2, f + BW/2]$.

Figure 5.23 depicts a zoom of the leakage map in range $f \in [-20, 20]$ MHz, which is created using the following configuration:

- Pearson p-value: $\alpha = 0.1$.
- Score function: `MinTracks`.
- Search step: $\Delta f = 0.03$ MHz.
- Search range: $f \in f_{clk} + [-20, 20]$ MHz.
- Bandwidth range: $BW \in [0.1, 4]$ MHz.
- Window type: Single window, Hann window in $f_{clk} \pm [f - BW/2, f + BW/2]$.

We have proven that even in this case the leakage is symmetric with respect to the clock frequency, even if the left sidelobes, due to environmental noise, have less informative content. We proceed to configure the tool using a double-sideband window. The result if depicted in Figure 5.24. We notice that the highest peak is located at $f = \pm 2.0$ MHz with bandwidth 1.2 MHz.

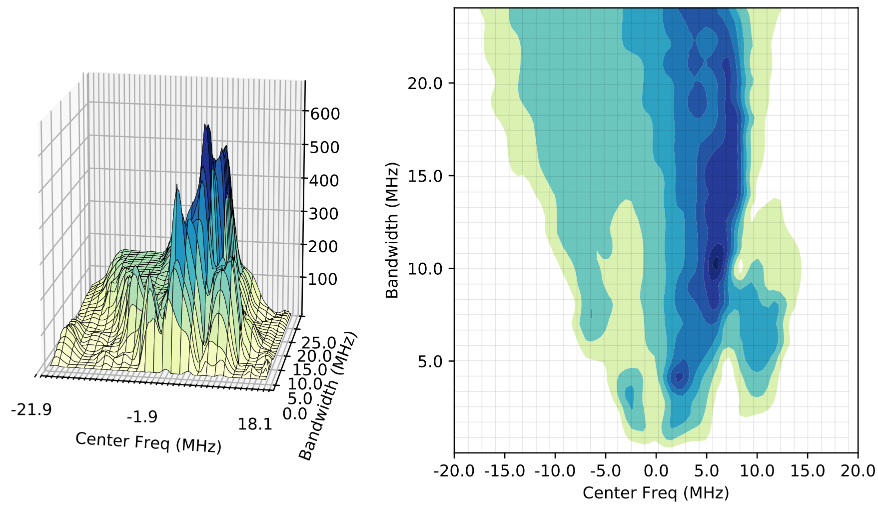


Figure 5.23: 3D-plot and heatmap, $f_{clk} = 128$ MHz, $\Delta f \in [-50, 50]$ MHz

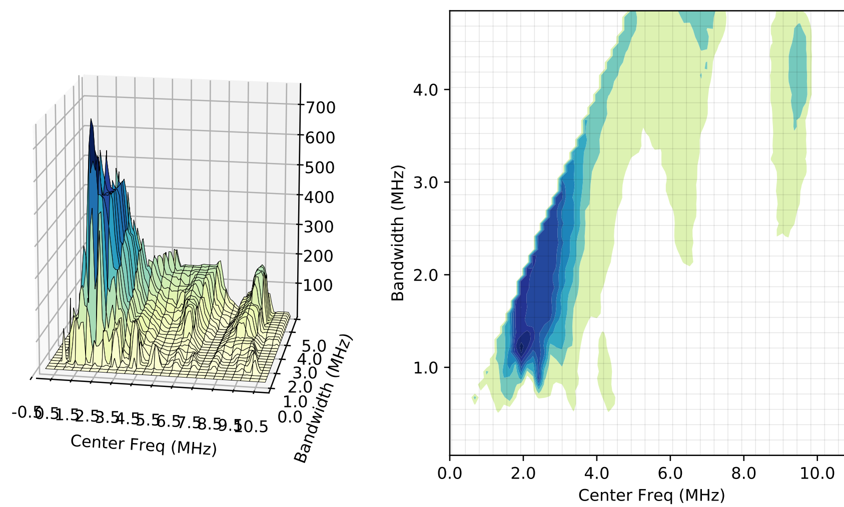


Figure 5.24: 3D-plot and heatmap, double-sideband, $f_{clk} = 128$ MHz, $\Delta f \in [0, 10]$ MHz

5.3.3 Results

Table 5.25 reports the results of this analysis, correlating the target clock frequency with the center frequency of the two sidelobes of the leakage and the bandwidth of each sidelobe. With this configuration the attack score is maximized, which, in our case means that a side-channel CPA attack becomes feasible with less number of traces.

Clock	Center Frequency	Bandwidth
32 MHz	$f_{clk} \pm 0.55$ MHz	0.40 MHz
64 MHz	$f_{clk} \pm 1.00$ MHz	1.00 MHz
128 MHz	$f_{clk} \pm 2.00$ MHz	1.20 MHz

Figure 5.25: Results table which maps clock frequency to frequency and bandwidth of the leakage

Figure 5.26 depicts the leakage area with respect to the clock frequency, the points are interpolated using a spline function. The two dashed lines represent the upper and lower limit of the leakage and the solid line at the center depicts the center frequency and the bandwidth. We see that all the three lines are linear with respect to the target clock frequency.

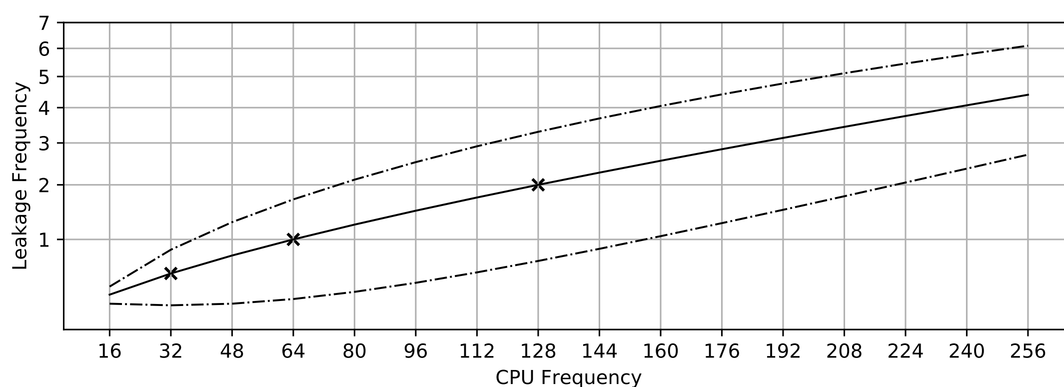


Figure 5.26: Expected leakage zone with respect to CPU clock frequency

5.4 Attacks using SDR

A frequency model for our target device is now available and, given the results in Table 5.25, it is safe to say that the bandwidth of the leakage when $f_{clk} \in [32, 64, 128]$ MHz is less than the bandwidth available using our SDR (rad1o-badge): 20 MHz. We can now safely perform CPA attacks using the SDR to capture EM emissions, instead of the oscilloscope.

To be sure that the center frequency of the SDR is set correctly, we developed a firmware that runs an alternate pattern of AES and `NOP`s, so that the power consumption adopts a visible pattern. This firmware is used to generate the wave patterns depicted in Figure 5.27, Figure 5.30 and Figure 5.33.

5.4.1 Basic attack

The first attack is done by clocking the target board to $f_{clk} = 32$ MHz, we tuned the SDR to that frequency and discovered the clock peak is exactly at $f = 31.80$ MHz. Figure 5.27 depicts *GQRX*, a popular interactive SDR receiver software showing the Discrete Fourier Transform (DFT) and the waterfall diagram of the spectrum while the antenna is places near the CPU and the firmware on the target board is running the alternated pattern described before.

We recorded a trace-set with 100000 traces and we performed the CPA attack after we had performed the $IQ \rightarrow RF$ reconstruction. The CPA attack is performed with p-value $\alpha = 0.1$. Figure 5.28 contains the results of the CPA attack, where Figure 5.28a depicts

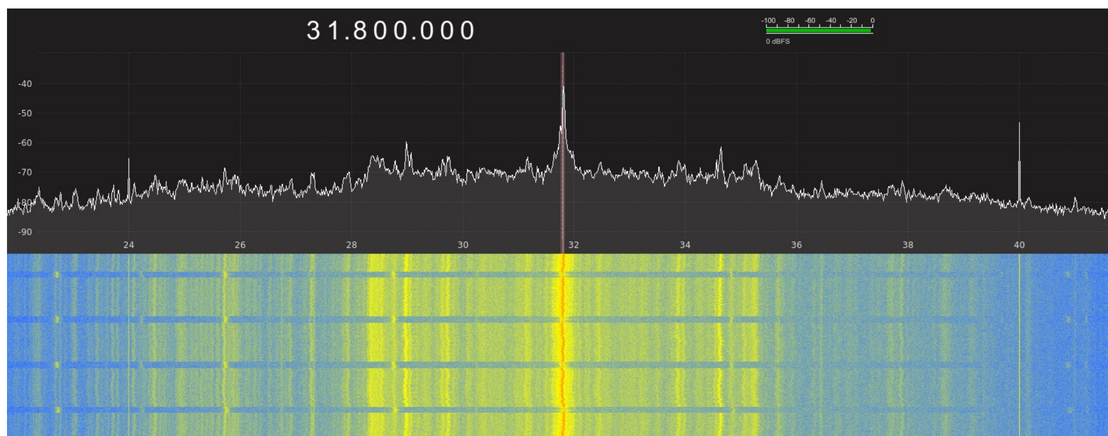
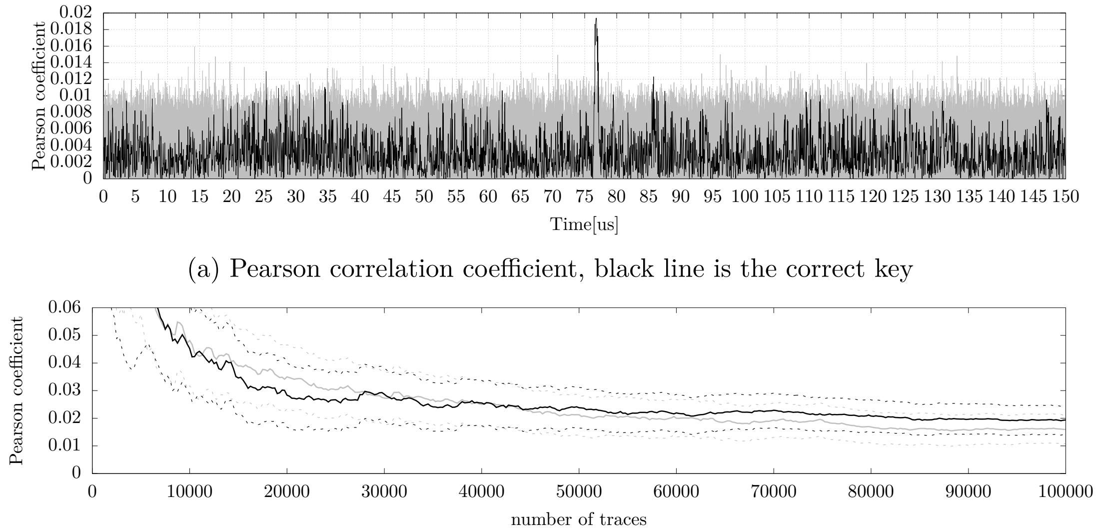


Figure 5.27: GQRX tuned to the clock frequency, when $f_{clk} = 31.80$ MHz



(a) Pearson correlation coefficient, black line is the correct key

(b) Confidence interval of the correct key (black) and wrong key with highest correlation coefficient (grey)

Figure 5.28: CPA attack using SDR without filtering ($f_{clk} = 32$ MHz)

the correlation coefficient and Figure 5.28b depicts two confidence interval: correct key in black and wrong key with highest correlation in grey. The confidence interval of the correct key and the interval of a wrong key are overlapping for the whole trace-set, hence an attack is not feasible using $\alpha = 0.1$.

We filtered each trace of the trace-set by performing a pointwise multiplication with a

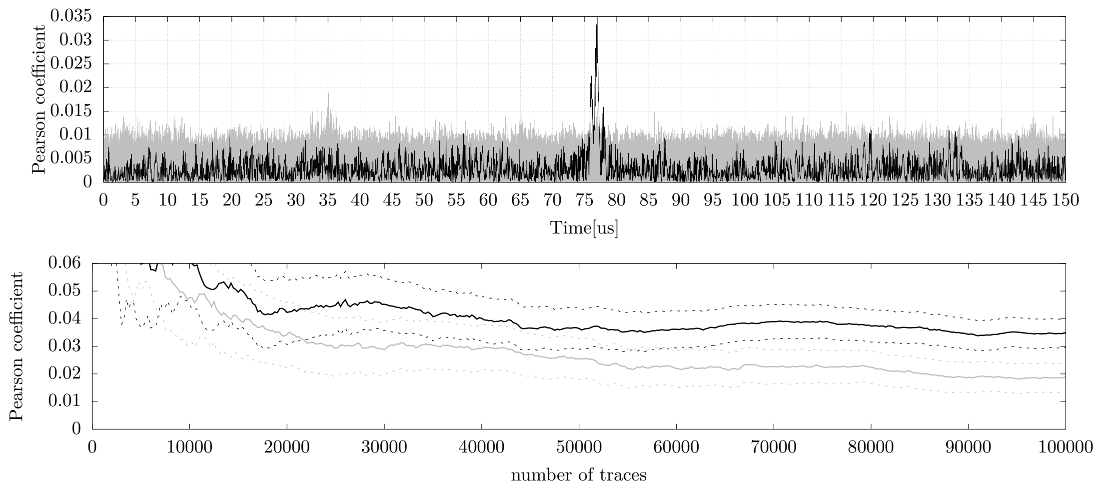


Figure 5.29: CPA attack using SDR with filters applied ($f_{clk} = 32$ MHz)

window function made of two Hann windows placed symmetrical with respect to the clock frequency, centered in 31.80 MHz $\pm$ 0.55 MHz and where each sidelobe has Bandwidth $bw = 0.40$ MHz. After the trace-set had been processed, we performed another CPA attack with the same parameters as before and the results are depicted in Figure 5.29. After the processing, we see that the Pearson correlation coefficient of the correct key is increased from 0.02 to 0.035 and the amplitude of the correlation of any other key (depicted as the grey-colored background noise) is unmodified, with peaks at 0.015.

With the filters proposed in this work, the CPA attack becomes feasible using $\alpha = 0.1$ with 60000 unaveraged traces.

5.4.2 Attacks at higher frequencies

To check the validity of the filters we have found, we performed CPA attacks with the target board clocked to higher frequencies. We collected 100000 traces both with the CPU clocked to $f_{clk} = 64$ MHz and the CPU clocked to $f_{clk} = 128$ MHz. For each trace-set, we compared the execution of two CPA attacks: one before and one after applying the filters found in the previous section.

64 MHz The first test at higher frequencies is performed when $f_{clk} = 64$ MHz. Figure 5.30 depicts GQRX while the board tuned to the correct frequency while the board is running the firmware which creates the alternate pattern described before. We noticed that the clock peak is located at $f = 63.9$ MHz, instead of 64 MHz.

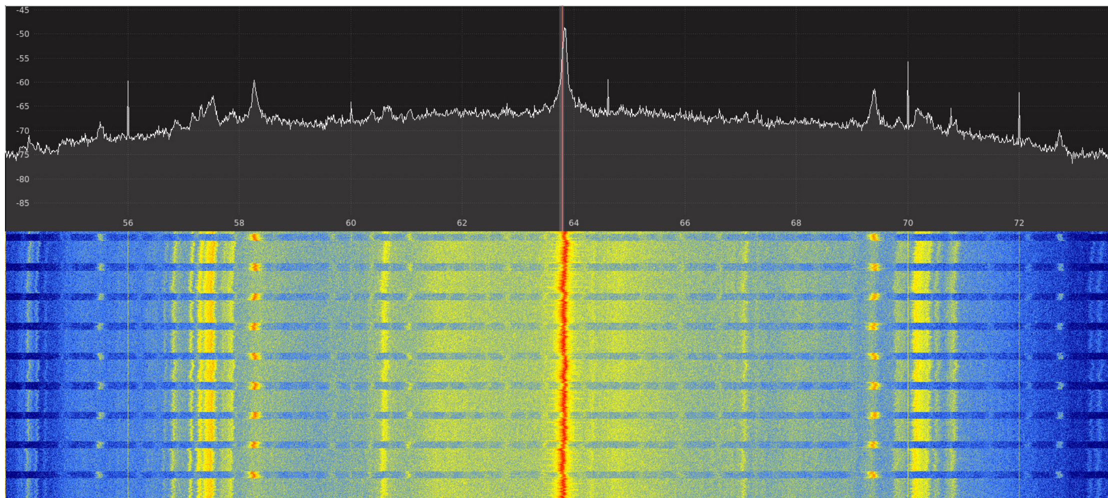
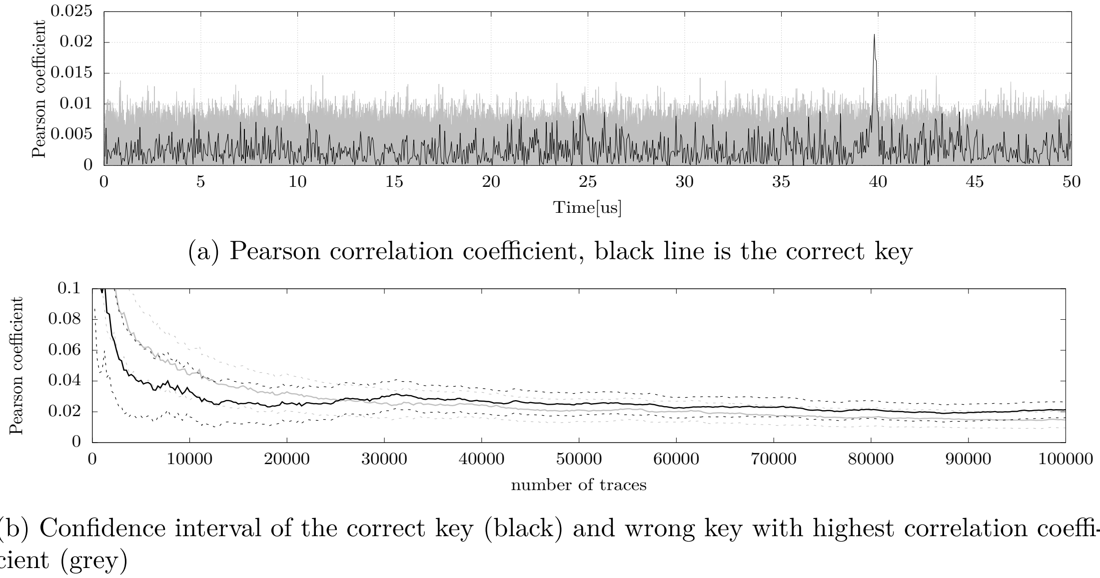


Figure 5.30: GQRX tuned to the clock frequency, when $f_{clk} = 63.9$ MHz

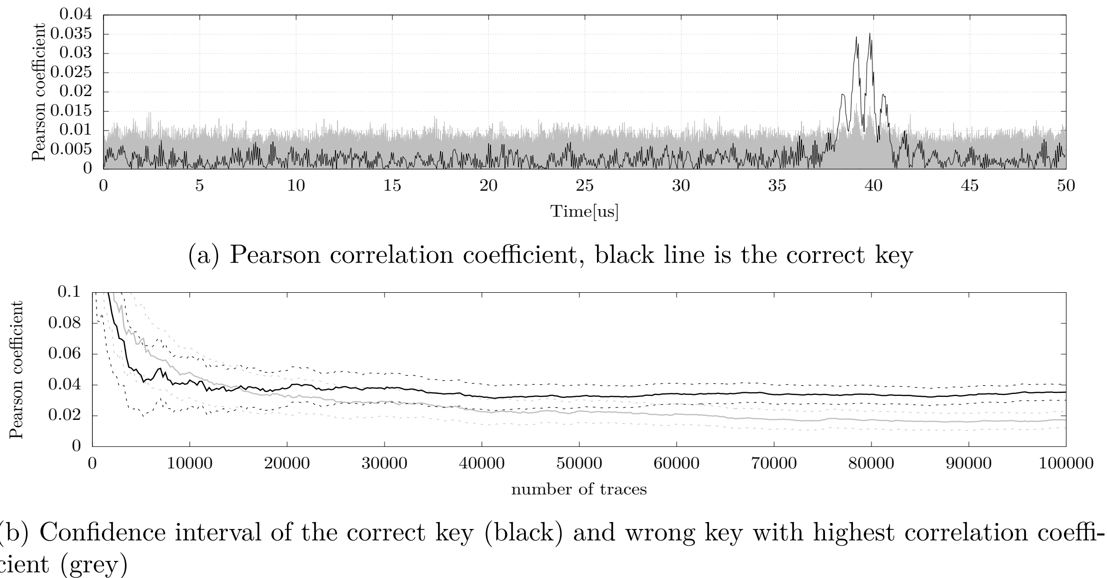


(a) Pearson correlation coefficient, black line is the correct key

(b) Confidence interval of the correct key (black) and wrong key with highest correlation coefficient (grey)

Figure 5.31: CPA attack using SDR without filtering ($f_{clk} = 64$ MHz)

We collected a trace-set made of 100000 traces and we performed a CPA attack with the p-value $\alpha = 0.1$, which turned out to be ineffective: Figure 5.31a depicts the Pearson correlation coefficients. Even if the peak corresponding to the correct key is visible, the confidence interval of the correct key overlaps with the confidence interval of another



(a) Pearson correlation coefficient, black line is the correct key

(b) Confidence interval of the correct key (black) and wrong key with highest correlation coefficient (grey)

Figure 5.32: CPA attack using SDR with filtering ($f_{clk} = 64$ MHz)

key for the whole trace-set, making the attack impossible with the chosen p-value.

We performed filtering of this trace-set by applying a double-sideband window with the parameters described in Table 5.25. The attack executed after filtering showed an improvement: after filtering, multiple correlation peaks appeared in Figure 5.34a. As depicted in Figure 5.34b, the two confidence intervals starts being disjoint right after 40000 traces.

128 MHz The last attack has been performed by tuning $f_{clk} = 128$ MHz. At first we ran, as before, a firmware performing the alternate pattern described before and we tuned the SDR using this.

Figure 5.33 depicts GQRX while showing this pattern in the waterfall, where the center peak is found at $f = 127.5$ MHz. We collected 100000 traces, aligned them and performed a CPA attack without filtering (depicted in Figure 5.34), obtaining no correlation between the power model generated using the correct key and the traces. Due to the extremely high environmental noise, the chosen p-value is lowered to $\alpha = 0.85$.

We filtered the trace-set using the double-sideband window function described in Table 5.25 and, after the filtering, we performed again the CPA attack, obtaining in this way the correlation diagram depicted in Figure 5.35. There are two points in Figure 5.35b where the two confidence intervals are disjoint: when 60000 traces are processed and when 100000 traces are processed.

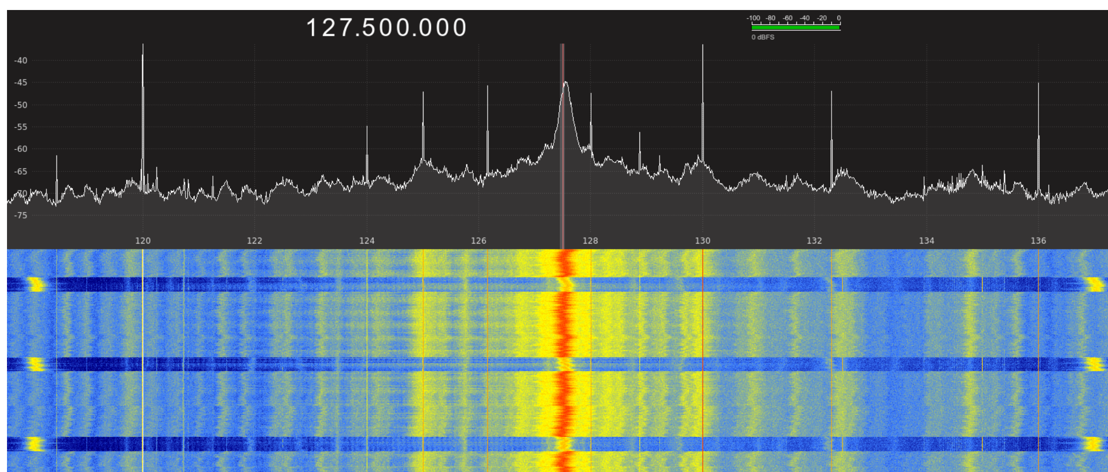
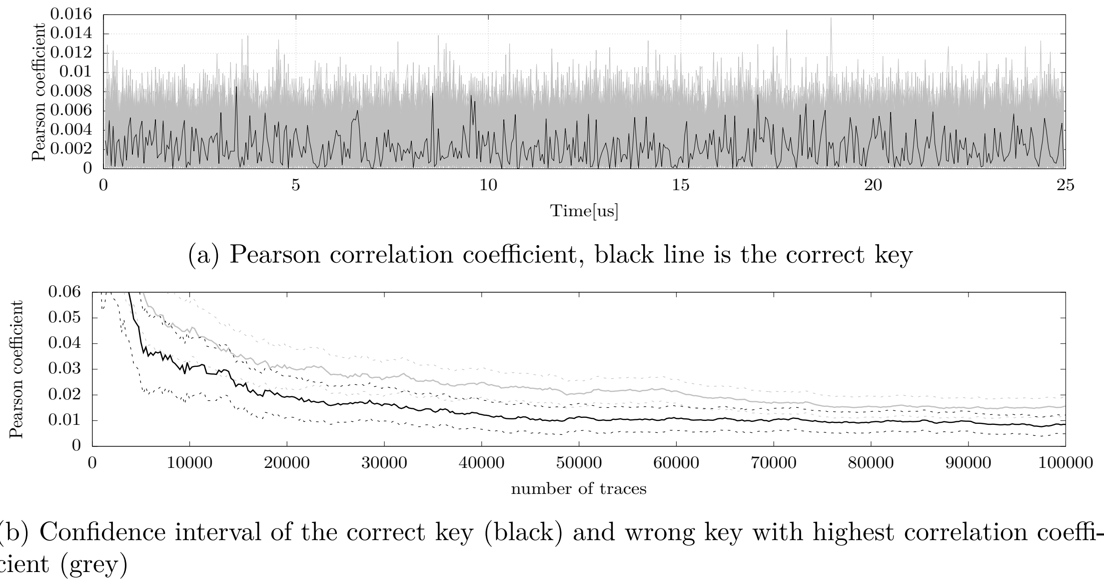


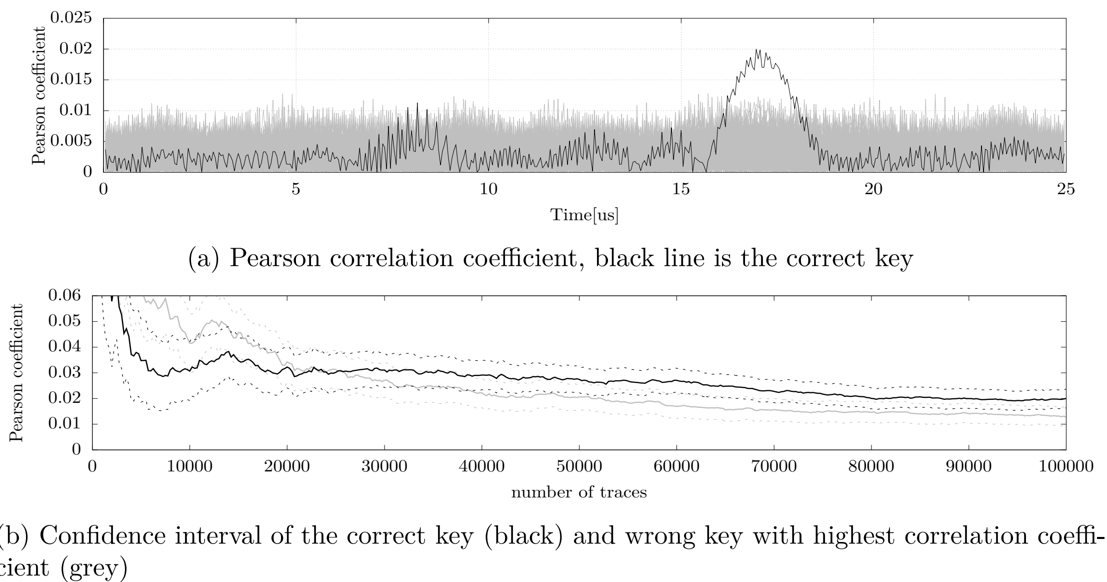
Figure 5.33: GQRX tuned to the clock frequency, when $f_{clk} = 127.5$ MHz



(a) Pearson correlation coefficient, black line is the correct key

(b) Confidence interval of the correct key (black) and wrong key with highest correlation coefficient (grey)

Figure 5.34: CPA attack using SDR without filtering ($f_{clk} = 128$ MHz)



(a) Pearson correlation coefficient, black line is the correct key

(b) Confidence interval of the correct key (black) and wrong key with highest correlation coefficient (grey)

Figure 5.35: CPA attack using SDR with filtering ($f_{clk} = 128$ MHz)

5.5 Architectural Reverse Engineering

In this section we analyze the architecture of an ARM Cortex-M7 using automatic tools which help us to discover which instructions can run in dual issue at any time and which one not. Our analysis has been done in two steps: at first we have gathered online information about the architecture, both from the official documentation and from the constraints written in the GCC backend (known as machine-description); then we have refined our model using of the method employed in [2] which exploits the information leaked by the Cycles Per Instruction (CPI) measure, achieved on a given instruction sequence.

5.5.1 Public information

This section contains information gathered online, in the official documentation and obtained by reading the GCC backend.

Online We have found an accurate description of the Cortex-M7 pipeline online [7], where we noticed that despite the fact that the Cortex-M7 is classified as microcontroller, there are two instruction decoders, it features a branch prediction unit and there are two Arithmetic-Logic Unit(s) (ALU), where one has also support for DSP instructions. The ALU without it is skewed, so that there is zero penalty for Integer, Floating Point (FP) or LOAD operations followed by a STORE. Also, skewing allows the issuing of two ALU operations with dependencies simultaneously. The Integer Register File (RF) has six read and four write ports, the FP RF instead has only four read and two write ports. There are two LOAD pipelines and just one STORE pipeline even though the bandwidth is the same. This is because the LOAD pipes can handle a 32-bit LOAD each one while the STORE pipeline can only handle a single 64-bit STORE. There is also a forwarding path in the Load-Store Unit (LSU) to minimize the latency of load-use operations.

GCC backend We have also gathered information by reading the constraints written by ARM in the GCC backend. These constraints are encoded in a LISP-like language called Machine-Description and can be found in the directory `gcc/config` inside the GCC source code. All the information in this subsection are taken from `arm/cortex-m7.md`. The following units are specified: 2 issue units, 2 ALU, 1 LSU, 1 MAC unit, 1 FPU, 1 Branch unit, 1 Write-Back buffer, 1 Shifter, 1 Extender. There are

no forwarding paths (a.k.a. bypass) specified. Part of the file is dedicated to fetching and branch prediction, this part is called "frontend". We are not going into details of the frontend and FPU because it is not interesting for our purposes. Regarding ALUs, the first ALU is the only one with DSP support (`smlaxy`, `smlalxy`, ...); both of them, instead, are able to execute all other operations. These operations are divided in two categories: Simple OPs, OPs with shifting. Either the first ALU or the second on can be used for operations with shifting but there is just one shifter inside the CPU, so theoretically two operations with shifting can not be issued at the same time. There is also just one unit dedicated to data extension: the rule for shifting is applied also here. Memory LOAD/STORE of 8 (or more) bytes uses all units for the whole execution; Loading 12/16 bytes, instead stalls the pipeline for 2 cycles. The most time-consuming operations are Integer division, Double-precision MAC and Float Division (FDIV) which cause a stall for 4 cycles.

5.5.2 CPI Analysis

The aim of this part is to discover which instructions can be issued simultaneously and which one not. To do that, we exploit the same method used in [2], which consists in measuring and comparing the average CPI index of a sequence of instructions both with and without Read After Write (RAW) hazards. If the CPI index is 0.5, the corresponding instructions are being executed in dual-issue (two per clock-cycle). At the moment, the best average CPI we get is $0.6(\frac{3}{5})$, which means that 5 instructions are executed every 3 clock-cycles, instead of 6 (pure dual-issuing), hence there is a 1-cycle stall after 3 instructions.

During this analysis we found out a necessary condition to issue two instructions at the same time: both of them should be written using 16-bit Thumb opcodes. This is because the flash chip installed into NUCLEO-F476ZG is able to read up to 4 bytes per clock-cycle at its maximum speed (0 wait-states and with the CPU clocked at 16MHz).

Benchmarks All these benchmarks were performed using a PC(host) which writes a custom firmware into the test-board (NUCLEO-F476ZG). This firmware uses the UART protocol to communicate with the host-machine through the USB cable (115200 BPS) and is able to perform tests as described below.

[1] -0.5 CPI if loaded from `[PC]`, except for `LOAD;LOAD` which goes from 8.0 to 6.78

[2] 1.0 if the `ADDS` operation changes the destination register

	nop	mov	adds	addsi	add w/shf	mul	lsl w/imm.	lsl	load[1]	store[2]
nop	0.60	0.60	0.60	0.60	1.00	1.00	0.60	0.87	4.00	0.60
mov	—	0.60	0.60	0.60	1.00	1.00	0.60	0.87	4.00	0.60
adds	—	—	0.60	0.60	1.00	1.00	0.60	0.87	4.00	0.60
addsi	—	—	—	0.60	1.00	1.00	0.60	0.60	4.00	0.60
add w/shift	—	—	—	—	1.15	1.15	1.00	1.15	4.50	1.00
mul	—	—	—	—	—	1.15	1.00	1.15	4.50	1.00
lsl w/imm.	—	—	—	—	—	—	0.60	0.87	4.00	0.60
lsl	—	—	—	—	—	—	—	1.15	4.00	0.87
load	—	—	—	—	—	—	—	—	8.00	1.00
store	—	—	—	—	—	—	—	—	—	1.00

(a) CPI obtained executing instructions without RAW hazards

	nop	mov	adds	addsi	add w/shf	mul	lsl w/imm	lsl	load[1]	store[2]
nop	—	—	—	—	—	—	—	—	—	—
mov	—	1.00	1.00	1.00	1.00	1.50	1.00	1.00	4.00	0.60
adds	—	—	1.00	1.00	1.00	1.50	1.00	1.00	4.00	0.60
addsi	—	—	—	1.00	1.00	1.50	1.00	1.00	4.00	0.60
add w/shift	—	—	—	—	1.15	1.50	1.00	1.15	4.50	1.00
mul	—	—	—	—	—	2.00	1.50	1.50	4.50	1.00
lsl w/imm.	—	—	—	—	—	—	1.00	1.00	4.00	0.60
lsl	—	—	—	—	—	—	—	1.15	4.00	0.87
load	—	—	—	—	—	—	—	—	8.00	1.00
store	—	—	—	—	—	—	—	—	—	1.00

(b) CPI obtained executing instructions with RAW hazards

Table 5.1: Comparison between CPI without(a) and with(b) RAW hazards

All these tests, except for the Load-Store test (described below) were performed after disabling the Adaptive Real-Time Accelerator (ART) accelerator and the instruction prefetcher. Due to flash issue, we carefully chose each instruction so that it has a 16-bit opcode (except for ADD w/SHIFT, which can only be encoded with 4 bytes). We built Table 5.1a with all the CPI obtained by executing the instructions without any RAW hazards. In this way, the ARM microcontroller will execute them with the best performance as possible. After we had executed the first test-suite, we changed all the instructions so that they could have RAW hazards between them. Table 5.1b reports the CPI of this new test-suite. All the tests where at least one instruction is `nop` have been omitted.

RF ports and bypass We designed another test-suite to have a confirmation about the number of Read/Write ports in the RF and to check if some forwarding paths (sometimes known as bypass) are present or not. Each test for bypass has been designed by following a simple rule: at first we measure the CPI of two unrelated instructions,

then we measure the CPI of two instructions where the first one writes a value in a register and the second one uses that value. If a bypass is present, there should be almost no difference between the two CPI. Table 5.2 reports the results of this test-suite. Please keep in mind that the corresponding CPI is the cost of both instructions together, divided by two. For example, if “Command 1” is “`A;B`”, “Command 2” is “`C`” and $CPI = 4$ then the instruction “`C`” takes 4 cycles to be executed and “`A;B`” (together!) take also 4 cycles. This is useful because (as you can see in the next paragraph), the sequence `LDR rA,[sp], STR rA,[sp]` is executed in just 1 CPI; which is less than a single LOAD (e.g. `LDR rA,[pc]`) which is executed in 3.5 CPI.

Command 1	Command 2	CPI
RF Ports		
UMULL rA,rB,rC,rD	nop	0.85
UMULL rA,rB,rC,rD	UMULL rB,rA,rE,rF	1.15
UMULL rA,rB,rC,rD	UMULL rC,rB,rA,rD	2.00
Bypass EX->EX		
ADDS rA,rA,rB	ADDS rC,rC,rD	0.6
ADDS rA,rA,rB	ADDS rB,rB,rA	1.0
ADDS rA,rA,rB	ADDS rB,rB,rC	0.6
Bypass EX->MEM		
ADDS rA,rC,#0	LDR rB,[rA]	4.5
ADDS rA,rC,#0	LDR rB,[rC]	4.0
ADDS rA,rC,#0; nop	LDR rB,[rC]	4.3
ADDS sp,sp,n	STR rB,[sp]	1.0
ADDS rA,rA,n	STR rA,[sp]	0.6
ADDS rA,rA,n	STR rB,[sp]	0.6
Bypass MEM->MEM		
LDR rA,[rB]	STR rA,[rB]	4.0
LDR rA,[rB]	STR rC,[rB]	4.0
Bypass MEM->EX		
LDR rA,[pc]	ADD rB,rB,rA	3.5
LDR rA,[pc]	ADD rB,rB,rC	3.5
LDR rA,[sp];STR rA,[sp]	ADD rB,rB,rA	1.25
LDR rA,[sp];STR rA,[sp]	ADD rB,rB,rC	1.25

Table 5.2: Various CPI w/ and w/o RAW hazards

Command 1	Command 2	CPI w/o ART	CPI w/ ART
`LDR r0, [sp]`	`LDR r0, [sp]`	8.00	8.05
`LDR r0, [sp]`	`STR r0, [sp]`	1.00	1.00
`LDR r0, [r1]`	`STR r0, [r1]`	1.00	1.00
`LDR r0, [r1]`	`STR r2, [r1]`	1.00	1.00
`LDR r0, [pc]`	`STR r0, [sp]`	3.50	3.52
`LDR r0, [lr]`	`STR r0, [sp]`	8.50	8.60
`LDR r0, [pc]`	`LDR r0, [pc]`	3.75	3.75
`LDR r0, [r1]`	`LDR r0, [pc]`	6.77	6.77
`LDR r0, [pc]`	`STR r0, [sp]`	4.00	4.07

Table 5.3: Load/Store CPI

Load/Store CPI At last, we have designed a small set of benchmarks to discover patterns in data/instruction cache and flash-memory usage while loading or storing data. All the results of these tests have been reported in Table 5.3. They have been performed at first with the ART and the Prefetcher disabled and then with both of them enabled. The results show that there is no improvement in enabling these features.

Latencies Each operation described in the GCC backend has an associated latency; sometimes these are not correlated to the real latency; for example, the `STORE` operation seems to be an operation without any latency, or if these are relative and not absolute (as it seems), it has less latency than any ALU operation, which seems impossible. These values are reported for completeness in Table 5.4.

Command	**Latency**
ALU,DSP,MUL,MAC	2
Integer DIV	4
LOAD(4,8,16)	2
BRANCH	0
STORE(4,8,16)	0
FP-MCR	1
FPU (Single Precision)	
ALU	3
MAC	6
DIV	16
LOAD	2
STORE	0
FPU (Double Precision)	
MUL	6
MAC	10
DIV	31
LOAD	3
STORE	0

Table 5.4: Cortex-M7 Unit Latencies reported in the GCC Back-end

	1	2	3	4	5	6
ADDS rA,rA,rB	IF	D1	E1	WB		
ADDS rC,rC,rD	IF	D2	E2	WB		
ADDS rA,rA,rB		IF	D1	E1	WB	
ADDS rC,rC,rD		IF	D2	E2	WB	
ADDS rA,rA,rB			IF	D1	E1	WB
ADDS rC,rC,rD			IF	D2	E2	WB

(a) Pipeline schema of ADDS-ADDS ops w/o RAW, $CPI = 0.5$

	1	2	3	4	5	6	7	8	9
ADDS rA,rA,rB	IF	D1	E1	WB					
ADDS rB,rB,rA	IF	D2	--	E2	WB				
ADDS rA,rA,rB		IF	D1	--	E1	WB			
ADDS rB,rB,rA		IF	--	D2	--	E2	WB		
ADDS rA,rA,rB			IF	--	D1	--	E1	WB	
ADDS rB,rB,rA				IF	--	D2	--	E2	WB

(b) Pipeline schema of ADDS-ADDS ops w/ RAW, $CPI = 1.0$

	1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61	65	69	73	77	81
LDR rA, [sp]	FDE.	....	.W																		
LDR rA, [sp]	FD..	....	.E..	....	.W																
LDR rA, [sp]	FD.	....	....	....	.E..	....	.W														
LDR rA, [sp]	F..	....	.D..	....	....	....	.E..	....	.W												
LDR rA, [sp]			F..	....	.D..	....	....	....	.E..	....	.W										
LDR rA, [sp]			F..	....	....	....	.D..	....	....	....	.E..	....	.W								
LDR rA, [sp]							F..	....	.D..	....	....	....	.E..	....	.W						
LDR rA, [sp]							F..	....	....	....	.D..	....	....	....	.E..	....	.W				
LDR rA, [sp]											F..	....	.D..	....	....	....	.E..	....	.W		
LDR rA, [sp]											F..	....	....	....	.D..	....	....	....	.E..	....	.W

(c) Pipeline schema of LOAD ops, $CPI = 8.0$

	1	5	9	13	17	21	25	29	33	37	41
LDR rA, [sp]	FDE.	....	..W								
NOP	FDE.	....	..W								
LDR rA, [sp]	FD.	....	..E.	....	..W						
NOP	FDE	....	....	....	..W						
LDR rA, [sp]	FD	....	....	....	..E.	....	..W				
NOP	F.	....	..D.	....	..E.	....	..W				
LDR rA, [sp]			FD	....	....	....	..E.	....	..W		
NOP			F.	....	..D.	....	..E.	....	..W		
LDR rA, [sp]					FD	....	....	....	..E.	....	..W
NOP					F.	....	..D.	....	..E.	....	..W

(d) Pipeline schema of LOAD-NOP pair, $CPI = 4.0$

Table 5.5: Reverse-engineered ARM Cortex-M7 pipeline schematics

The fact that these latencies are wrong or not fully completed pushed us to compute manually the latency of each useful pair of instructions. The way we have chosen to reconstruct the latencies has been to draw the CPU pipeline for a pair of instructions so that both the CPI index without RAW hazards is equal to the one written in Table 5.1a and the CPI index with RAW hazards is equal to the one written in Table 5.1b. Table 5.5a and Table 5.5b show the pipeline stages while the CPU is executing a sequence of `ADDS` respectively without and with a RAW hazard. The `ADDS` instruction has been chosen because can be encoded with a 16-bits Thumb-2 instruction and the flash memory can fetch two instruction per clock-cycle. The pipeline schema of a sequence of `LDR` instructions (`LDR` loads a value from the flash to a register) is represented in Table 5.5c and shows why it has CPI index 8.0. Table 5.5d shows, instead, what happens when a sequence of `LDR` is interleaved with a sequence of `NOP`: as predicted, each `NOP` is executed in dual-issue with the `LDR` which takes 4 cycles to be completed, showing 4 as average CPI.

5.5.3 Results

We report here a summary of our deductions made by analyzing information we have obtained and reverse-engineered.

Dual-issuing Using the differences between Table 5.1a and Table 5.1b we have been able to create the dual-issue table (Table 5.6), which reports (with an `x`) which instructions can be issued at the same time and which one not. Some instructions are reported as $\sim$, which means that if there is no RAW hazard, these instructions have $0.6 < CPI < 1.0$ and then any hazard slows them down to $CPI >= 1.0$.

	nop	mov	adds	add w/shf	mul	lsl w/imm	lsl	load
nop	x	x	x			x	~	???
mov	—	x	x			x	~	???
adds	—	—	x			x	~	???
add w/shift	—	—	—					???
mul	—	—	—	—				???
lsl w/imm.	—	—	—	—	—	x	~	???
lsl	—	—	—	—	—	—		???
load	—	—	—	—	—	—	—	???

Table 5.6: Cortex-M7 dual issue table

Flash memory speed It is impossible for the NUCLEO-F476ZG to issue at the same time two instruction composed of 4 bytes each; even if the documentation online says that the LOAD unit is able to fetch up to 64-bit per clock-cycle. The only explanation we have found is because of the flash memory, which is able to fetch at most 32-bits per clock-cycle.

Read/Write ports As specified online, the Radio Frequency (RF) is composed of six read ports and four write ports. We ran a test to confirm this by issuing a sequence of `UMULL` with two destination registers. Table 5.2 shows that these instructions can be issued at the same time, this is possible if and only if the number of write ports is greater or equal than 4.

Forwarding paths Between al possible forwarding paths, we are interested in the following one: `EX->EX`, `EX->MEM`, `MEM->EX`, `MEM->MEM`. Table 5.2 shows that there is an `EX->MEM` bypass only between the execution phase and the value that is going to be stored. The forwarding path is not used if the value computed is used as address.

There is just one `EX->EX` bypass in this CPU: the execution of two instructions with a dependency is done in dual issue; the execution of four instructions where each one depends on the previous one is done in single issue.

ALU This CPU is powered by two ALUs, called (by GCC backend) `ALU0` and `ALU1`. Both ALUs are able to execute any common operation (Sum, Subtraction, Logical/Arithmetic Shifts, Multiplication, …). `ALU0` is more powerful than `ALU1` because is the only one able to execute DSP instructions. On the other hand `ALU1` is skewed, which helps while dual-issuing instructions. The GCC backend describes the CPU with an extra shifter (external of the ALUs). The “add w/shf” column in Table 5.1a shows that two ADD operations with shifting cannot run at the same time even without RAW hazards. On the other hand, both ALUs are able to run a shift operation, so two LSLs(Logical Shift Left) can run in parallel without problems.

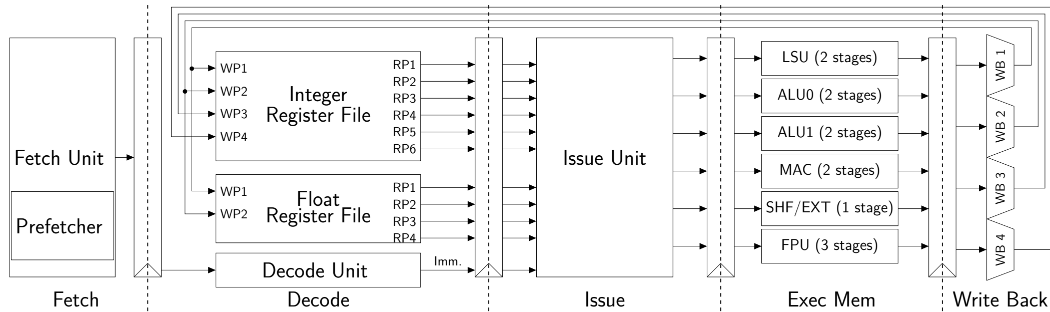


Figure 5.36: Reverse-engineered ARM Cortex-M7 internal structure

Load/Store A sequence of LOAD or STORE operations referring to a random address have CPI 4. If the requested address is near the Program Counter (PC), the CPI index shows that the cost is 0.5 less than any other register (from 4 to 3.5). As shown in Table 5.3, a sequence of interleaved LOAD and STORE operations having the same value and referring to the same address has CPI 1; this is probably due to caches. On the other hand, a sequence of just LOAD operations referred to the same address has CPI index 8 (or 6.78 if the address is the PC).

CPU Pipeline All the findings on the microarchitecture of the Cortex-M7 reported in this section helped us in reverse-engineering an accurate logic scheme of the CPU pipeline, which is depicted in Figure 5.36.

The forwarding paths are not drawn even if they are present, the shifter/extender is drawn as it is a single unit, even if in the real CPU these are probably two distinct components that can be even used in parallel.

Chapter 6

Conclusions

With this work we developed a set of tools which allowed us to discover how side-channel leakages are spread around the spectrum, which Digital Signal Processing (DSP) techniques can be applied to increase the Signal-to-Noise Ratio (SNR) and how much information is leaked. We found a linear correlation between the target CPU clock frequency and the location of the leakage.

During the first part of this work, a set of measurements were collected using the oscilloscope and analyzed. Once that the frequency-oriented leakage model has been found and validated, we used it to perform a set of attacks using a Software Defined Radio (SDR) as capture device. We showed how to collect, align and filter traces using a SDR, so that a Differential Power Analysis (DPA) side-channel attack becomes feasible using less than 100000 unaveraged traces.

A set of DPA attacks were performed with the target board clocked from 32 MHz up to 128 MHz by aligning the traces using a rigid-alignment pipeline and by correcting the clock-drifting using an elastic-alignment pipeline based on the FastDTW algorithm. To reduce the size of the traces and speed-up the attacks, we developed a leakage-invariant transform which reconstructs the original signal on a different carrier.

We also provided an innovative way to collect measures used to perform architecture reversing via Universal Asynchronous Receiver-Transmitter (UART) connection. In this context, we have developed a technique to minimize UART jittering to the point that it becomes negligible and it does not interfere with the analysis.

Bibliography

[1] ARM. Product Backgrounder. https://web.archive.org/web/20071203000700/http://www.arm.com/miscPDFs/3823.pdf, 2005. [Online; accessed 14-oct-2018].

[2] Alessandro Barenghi and Gerardo Pelosi. Side-channel security of superscalar cpus: evaluating the impact of micro-architectural features. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 120:1–120:6. ACM, 2018. ISBN 978-1-5386-4114-9. doi: 10.1145/3195970.3196112. URL http://doi.acm.org/10.1145/3195970.3196112.

[3] BusinessWire. Worldwide Smartphone Population Tops 1 Billion in Q3 2012. https://www.businesswire.com/news/home/20121017005479/en/Strategy-Analytics-Worldwide-Smartphone-Population-Tops-1, 2012. [Online; accessed 14-oct-2018].

[4] Zhimin Chen, Ambuj Sinha, and Patrick Schaumont. Using virtual secure circuit to protect embedded software from side-channel attacks. *IEEE Trans. Computers*, 62 (1):124–136, 2013. doi: 10.1109/TC.2011.225. URL https://doi.org/10.1109/TC.2011.225.

[5] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[6] ARM Cortex. M3 technical reference manual. *Rev. r1p1*, 2006.

[7] Charlie Demerjian. ARM goes into great detail about the M7 core - SemiAccurate.com. https://web.archive.org/web/20150524193658/https://semiaccurate.com/2015/04/30/arm-goes-great-detail-m7-core/, 2015. [Online; accessed 15-lug-2018].

[8] Hassan Eldib, Chao Wang, Mostafa M. I. Taha, and Patrick Schaumont. Quantitative masking strength: Quantifying the power side-channel resistance of software code. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(10):1558–1568, 2015. doi: 10.1109/TCAD.2015.2424951. URL `https://doi.org/10.1109/TCAD.2015.2424951`.

[9] Vinnie Falco. A Collection of Useful C++ Classes for Digital Signal Processing - GitHub. `https://github.com/vinniefalco/DSPFilters/`, 2017. [Online; accessed 30-ago-2018].

[10] G.B. Folland. *Fourier Analysis and Its Applications*. Pure and applied undergraduate texts. American Mathematical Society, 2009. ISBN 9780821847909.

[11] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[12] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. *IACR Cryptology ePrint Archive*, 2013:857, 2013. URL `http://eprint.iacr.org/2013/857`.

[13] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from pcs using a radio: Cheap electromagnetic attacks on windowed exponentiation. *IACR Cryptology ePrint Archive*, 2015:170, 2015. URL `http://eprint.iacr.org/2015/170`.

[14] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[15] Auguste Kerckhoffs. *La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef*. Librairie militaire de L. Baudoin, 1883.

[16] Johan Kirkhorn. Introduction to iq-demodulation of rf-data. *IFBT, NTNU*, 15, 1999.

[17] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, Apr 2011. ISSN 2190-8516. doi: 10.1007/s13389-011-0006-y. URL `https://doi.org/10.1007/s13389-011-0006-y`.

[18] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999. ISBN 3-540-66347-9. doi: 10.1007/3-540-48405-1_25. URL https://doi.org/10.1007/3-540-48405-1_25.

[19] Markus G. Kuhn. Electromagnetic eavesdropping risks of flat-panel displays. In David M. Martin Jr. and Andrei Serjantov, editors, *Privacy Enhancing Technologies, 4th International Workshop, PET 2004, Toronto, Canada, May 26-28, 2004, Revised Selected Papers*, volume 3424 of *Lecture Notes in Computer Science*, pages 88–107. Springer, 2004. ISBN 3-540-26203-2. doi: 10.1007/11423409_7. URL https://doi.org/10.1007/11423409_7.

[20] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007. ISBN 978-0-387-30857-9.

[21] *2.3GHz to 2.7GHz Wireless Broadband RF Transceiver*. Maxim Integrated Products, Inc., 7 2015. Rev. 2.

[22] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from xilinx virtex-ii fpgas. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 111–124. ACM, 2011. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046722. URL http://doi.acm.org/10.1145/2046707.2046722.

[23] *32-bit ARM Cortex-M4/M0 flashless MCU*. NXP Semiconductors, 3 2016. Rev. 6.

[24] C. Prati. *Segnali e sistemi per le telecomunicazioni*. Collana di istruzione scientifica. McGraw-Hill Companies, 2003. ISBN 9788838660931. URL https://books.google.it/books?id=ABcrAAAACAAJ.

[25] Stan Salvador and Philip Chan. Fastdtw: Toward accurate dynamic time warping in linear time and space. In *KDD workshop on mining temporal and sequential data*. Citeseer, 2004.

[26] Hermann Seuschek and Stefan Rass. Side-channel leakage models for RISC instruction set architectures from empirical data. *Microprocessors and Microsystems - Embedded Hardware Design*, 47:74–81, 2016. doi: 10.1016/j.micpro.2016.01.004. URL `https://doi.org/10.1016/j.micpro.2016.01.004`.

[27] Hermann Seuschek, Fabrizio De Santis, and Oscar M. Guillen. Side-channel leakage aware instruction scheduling. In Mats Brorsson, Zhonghai Lu, Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi, editors, *Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC 2017, Stockholm, Sweden, January 24, 2017*, pages 7–12. ACM, 2017. ISBN 978-1-4503-4869-0. doi: 10.1145/3031836.3031838. URL `http://doi.acm.org/10.1145/3031836.3031838`.

[28] Bristol University. OpenSCA - A Matlab-based open source framework for side-channel attacks. `http://opensca.sourceforge.net/`, 2009. [Online; accessed 30-ago-2018].

[29] Nicolas Veyrat-Charvillon and François-Xavier Standaert. Mutual information analysis: How, when and why? In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 429–443. Springer, 2009. ISBN 978-3-642-04137-2. doi: 10.1007/978-3-642-04138-9_30. URL `https://doi.org/10.1007/978-3-642-04138-9_30`.