POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Evaluation of encryption algorithm security in microcontroller-based platforms against differential power analysis attack

Relatore:

Prof. Stefano Di Carlo

Candidato:

Giulia Giordano

A.A. 2017-2018

Contents

Chapter 1

Introduction

Cyber security is a key aspect of today's everyday life. Data is stored on electronic devices and every kind of communications is performed through them. It is often necessary to cypher data in order to prevent attacks to sensitive information; despite that some attacks can still threaten security. Since the power consumed by a circuit varies according to the activity of its components, data is correlated with power consumption.

In this thesis differential and correlation power analysis are performed in order to discover the limits of AES 256 in CBC, ECB, CFB modes and AES 128 for the microcontroller-based platform considered. The goal is to test the security of a microcontroller equipped with a software implementations of the algorithms, to see if any information regarding the key can be retrieved by means of differential and correlation power analysis. These attacks take measurements of a target device's power consumption together with the plaintext (data to be encrypted) and ciphertext (the output of the encryption process) to extract secret keys.

A program with the selected encryption is loaded on a Nucleo F401RE board: each cyphering is enclosed between the switch on and off on a LED, used as reference. Power consumed by a Nucleo F401RE is captured by means of a wide bandwidth oscilloscope triggered by the LED and a current probe, so that one power trace corresponds to the encryption of one plaintext. The microcontroller is powered by means of the USB cable or with an external PIN named Vin: the best results are obtained with the the external power supply.

I used several tools for the analysis: the main ones are Pysca and Jlsca, which are open source tools created by Cees-Bart Breunesse and Ilya Kizhvatov, who was also the creator of the previously described Pysca. Pysca works in Python environment; Jlsca and its toolbox are written in Julia and they take advantage of Python libraries and Python Notebook. Jlsca receives in input a file containing the plaintext, one for the ciphertext and one with the power traces.

With the selected tool and instrumentations I could not retrieve the whole key, however interesting correlations do exist between couples of bytes or few correct bytes in the proposed solutions.

This indicates that there is a correlation between power leakages and cypher key: it may be used in order to have an idea on the most probable bytes and to reduce the whole set of possibilities.
There are many limits in the applications of this technique: instrumentation precision and influence of noise to name a few. Furthermore the tools used for this analysis depend a lot on the actual beginning and end of the round in the trace, so a little imprecision there could visibly change the results.

Moreover it can be challenging to realize the attack in a real world scenario: plaintexts and cypher texts are required and they must be perfectly aligned with the captured traces. The whole instrumentation setup is expensive if we consider the 350 MHz oscilloscope and the 50 MHz current probe, and the attacks is performed on a cheap microcontroller which should be leak a lot more than actual security devices.

List of Tables

List of Figures

Chapter 2

DPA State of Art

2.1 Side channel analysis

A side-channel attack is based on information retrieved from the implementation of a computer system. Sources of information to be exploited can be timing information, power consumption, electromagnetic leaks. [13]

Some side-channel attacks require technical knowledge of the internal operation of the system, although others such as differential power analysis are effective as black-box attacks.

Side channel attacks include, for example:

- Power-monitoring attack, meaning the attacks that make use of varying power consumption by the hardware during computation.
- Electromagnetic attack, that is to say attacks based on leaked electromagnetic radiation
- Differential fault analysis, in which secrets are discovered by introducing faults in a computation.

In all cases, the fundamental idea is that "*physical effects caused by the operation of a cryptosystem can provide useful extra information about secrets in the system*" [13], such as the cryptographic key, full or parts of the plaintexts and so on.

2.2 Introduction on Differential Power Analysis

The power consumed by a circuit varies according to the activity of its components. As a result, data is correlated with power consumption, meaning that a given operation will have a different consumption from another due to the charging and discharging of the capacitances involved. This will appear in the power traces, so that consumption measurements contain information about a circuit's operations.

Differential power analysis belongs to the family of side-channel attacks. In cryptography, a side-channel attack is any attack based on information gained from the physical implementation of a cryptosystem, rather than theoretical weaknesses in the algorithms or brute force. With DPA you can break the system without knowing the internal functioning of the device, seeing it as black box. These attacks take measurements of a target device's power consumption to extract secret keys, and they are effective against implementations of all major algorithms.

Differential Power Analysis is a statistical method to analyze sets of measurements in order to identify data-dependent correlations. The standard method requires to divide a set of traces into subsets, then doing the averages of these subset and then computing the difference. If the choice of which trace is assigned to each subset is uncorrelated to the measurements contained in the traces, the difference in the subsets' averages will approach zero as the number of traces increases. Otherwise, if the partitioning into subsets is correlated to the trace measurements, the averages will approach a non-zero value. Even if the difference of the means is very small, it will eventually become statistically

significant given a sufficient number of traces. In the years statistical tests and qualifiers for DPA has changed and improved:

- In the original DPA paper the "difference of means" method was introduced. The original paper on power analysis, published in 1998, analized a smart card cyphered with DES.
- All or nothing DPA, multi bit DPA and generalized DPA
- Maximum likelihood based DPA, correlation power analysis, mutual Information based distinguisher, stochastic methods and template attacks

The focus in such tests has been distinguishing the correct key from incorrect key guesses, in an attack setting. The research "All for one, one for all: Unifying univariate DPA attacks" by Mangard, S., Oswald, E., Standaert [28] indicates that the specific choice of technique does not have a major impact in the asymptotic sense.

2.3 DPA countermeasures

There has been much research on developing and implementing DPA-resistant logic styles. Several works have focused on:

- Masking: Randomize intermediate values of the algorithm or Mask logic gates
- Hiding: Randomize the execution of the algorithm or change the power consumption characteristics
- Dual-rail precharge logic
- Asynchronous logic styles

On the other hand it is possible to decrease the signal to noise ratio of the power side channel (either by decreasing signal or increasing noise) so that the number of traces required for a successful attack will increase. There are mainly two techniques to do that:

- Amplitude noise is added using circuit elements to perform calculations that are uncorrelated to the cryptographic intermediates being hidden. Only the spectral component of noise that is frequency-matched to the signal is relevant.
- Temporal noise is introduced by inserting variations in timing and execution order.

A factor of k reduction in the signal-to-noise ratio using leakage reduction techniques such as balancing increases the difficulty and number of traces required for DPA by k^2 [26]. However, sometimes signal processing techniques can detect and eliminate all or part of the additional noise: for example, filtering can eliminate amplitude noise that is not frequency matched to the signal.

A general approach for surviving leakage is to limit the number of transactions that can be performed with a key. For example, consider the case of a device with a 256-bit key that can be operated at most ten times, and which will destroy its keys after the 10^{th} transaction attempt. If, for example, the required security level is 192 bits, then this design can tolerate a total leak of 64 bits over its lifetime, which is achieved if the average leak is $<$6.4 bits per transaction.

A key update procedure is then performed at periodic intervals. The update frequency is chosen so that the number of uses for any one key value does not cross the design's security threshold.

Another basic method is called "power equalization" and it is performed introducing additional registers and complementary logic. It requires compensation of register and combinatorial logic switching: this causes an increase in power consumption that makes it cipher key independent, thus preventing DPA. Special libraries ensure power equalization on the gate or the transistor levels [37].

Recently there have been several successful power analysis attacks on Advanced Encryption Standard (AES) at different locations in the algorithm. These locations have been:

- Key scheduling
- Add roundkey
- SBOX output

SBOX attacks take place on raw hardware, while the other attacks have utilized additional hardware [15].

There are different implementations of DPA [29]

- Zero-offset DPA (ZODPA). This attack requires that masks and masked data data of the attacked device leak simultaneously and it uses squaring as a preprocessing step. It's a special case of second-order DPA and it exploits glitches to get information [39].
- A DPA attack can be based on a toggle-count power model of a masked S-box of the chip.
- A simplified attacked called zero-input DPA.

There are also High-Order Differential Power Analysis (HO-DPA), which is more complex and more effective but not largely used.

2.3.1 Example of DPA Attacks performed recently

The first successful power analysis attack against an FPGA was carried out by Örs et al. in 2003 as reported in [31]. They performed the attack against an elliptic curve cryptographic processor and were able to retrieve the secret key by simple visual inspection of the leakage traces. They mounted an FPGA on a hand-made board, in order to make the power-analysis attacks easier.

In 2012 [17] they performed a successful attack against the first round of an AES-128 decryption using the Hamming distance power model. The target device was a Xilinx FPGA on the Side-Channel Attack Standard Evaluation Board (SASEBO).

In 2014 a DPA attack had been performed on 32-bit ARM Cortex-M3 microprocessor [32]. Attacked algorithm is unprotected Advanced Encryption Standard (AES) with 128-bit key. The attack has been performed without syncronization, as they would be during a realistic DPA attack, using 150,000 samples.

In [6] the researchers from three Universities (Worcester Polytechnic Institute, Worcester, USA; Ruhr-Universitat Bochum, Germany; Florida Atlantic University, USA) present a successfull differential power analysis of an implementation of the McEliece cryptosystem. The target of this attack is a FPGA implementation of the QC-MDPC McEliece decryption operation. The presented cryptanalysis succeeds to recover the complete secret key after several analysis. It consists of a combination of a differential leakage analysis and an algebraic step that exploits the relation between the public and private key. Next they focused on how to recover the full key of QC-MDPC McEliece in a given scenario where the adversary has knowledge of several 1 bits of the key as well as several 0 bits of the key, possibly with few errors. They showed that the structure of the key can be used to successfully recover the remaining uncertain bits, or to detect remaining errors.

In China, McEvoy et al. described a DPA attack on an implementation of the HMAC algorithm that uses the SHA-2 hash function family; those DPA attacks are against multiple group operations, such as exclusive or and modulo addition, so they need various leakage models; moreover, it is difficult to recover the secret key with high noise levels. The SM3, which is an algorithm released in 2010 by China's Office of Security Commercial Code Administration, and certificated as the only standard hash algorithm of China, had been attacked in 2015. They used a new attack method, specific to modulo addition operation: they succeeded and they have shown that the attacker can recover the inner keyed state [24].

The Dept. of Electronics and Multimedia Communications in 2016 [33] demonstrated that a part of a private key can be recovered using differential power analysis. They attacked a software implementation of a secure bit permutation with a cryptosystem (McEliece PKC) implemented on a 32-bit ARM based microcontroller.

Chapter 3

Cryptography background and focus on AES

3.1 Introduction to Cryptography and terminology

Cryptography is the practice and study of techniques for secure communication in the presence of third parties called adversaries [34]. It is about constructing and analyzing protocols that prevent third parties or the public from reading private messages [16], focusing the attention on data confidentiality, data integrity and authentication.

Encryption is the conversion of information from a readable, ordinary state (called plaintext) to apparent nonsense (called ciphertext) [25]. The source of an encrypted message shares the decoding technique needed to recover the original information only with intended recipients, thereby precluding unwanted persons from doing the same.

A cipher (or cypher) is a pair of algorithms that create the encryption and the reversing decryption. The detailed operation of a cipher is controlled both by the algorithm and in each instance by a "key". The key is a secret (ideally known only to the communicants), usually a short string of characters, which is needed to decrypt the ciphertext. Formally, a "cryptosystem" is the ordered list of elements of finite possible plaintexts, finite possible ciphertexts, finite possible keys, and the encryption and decryption algorithms which correspond to each key.

There are two kinds of cryptosystems: symmetric and asymmetric. In symmetric systems the same key (the secret key) is used to encrypt and decrypt a message. Data manipulation in symmetric systems is faster than asymmetric systems as they generally use shorter key lengths. Asymmetric, also called public key, is any cryptographic system that uses pairs of keys: public keys which may be diffused widely, and private keys which are known only to the owner. The use of asymmetric systems enhances the security of communication.

Examples of asymmetric systems:

- RSA (Rivest-Shamir-Adleman)
- ECC (Elliptic Curve Cryptography)

Symmetric models examples:

- AES (Advanced Encryption Standard)
- DES (Data Encryption Standard)

Ciphertexts produced by a classical cipher (and some modern ciphers) will reveal statistical information about the plaintext, and that information can often be used to break the cipher. After the discovery of frequency analysis in the 9th century, [36] nearly all such ciphers could be broken by an informed attacker.

In practice all ciphers remained vulnerable to cryptanalysis using the frequency analysis technique until the development of the polyalphabetic cipher (around the year 1467).

Breaking a message without using frequency analysis essentially required knowledge of the cipher used and perhaps of the key involved. It was finally explicitly recognized in the 19th century that secrecy of a cipher's algorithm is not a sensible nor practical safeguard of message security; in fact, it was further realized that any adequate cryptographic scheme (including ciphers) should remain secure even if the adversary fully understands the cipher algorithm itself.

Security of the key used should alone be sufficient for a good cipher to maintain confidentiality under an attack. This fundamental principle was first explicitly stated in 1883 by Auguste Kerckhoffs and is generally called Kerckhoffs's Principle (it was also restated by Claude Shannon, the inventor of information theory and the fundamentals of theoretical cryptography, as Shannon's Maxim—'the enemy knows the system').

With the development of digital computers and electronics cryptanalysis improved and it was possible to create much more complex ciphers. Furthermore, computers allowed for the encryption of any kind of data representable in any binary format, unlike classical ciphers which only encrypted written language texts.

Many computer ciphers can be characterized by their operation on binary bit sequences (sometimes in groups or blocks), unlike classical and mechanical schemes, which generally manipulate traditional characters (i.e., letters and digits).

Open academic research into cryptography is relatively recent: it began only in the mid-1970s. In recent times, IBM personnel designed the algorithm that became the Federal (i.e., US) Data Encryption Standard; Whitfield Diffie and Martin Hellman published their key agreement algorithm [19]; the RSA algorithm was published in Martin Gardner's Scientific American column.

There are very few cryptosystems that are proven to be unconditionally secure. The one-time pad is one of those:

it cannot be cracked, but requires the use of a one-time pre-shared key the same size as, or longer than, the message being sent. The plaintext is paired with a random secret key and then each bit or character of the plaintext is encrypted by combining it with the corresponding bit or character from the pad using modular addition. If the key is truly random, is at least as long as the plaintext, is never reused in whole or in part, and is kept completely secret, then the resulting ciphertext is be impossible to decrypt or break [5].

Essentially, prior to the early 20th century, cryptography was primarily concerned with linguistic and lexicographic patterns. Cryptography now makes extensive use of mathematics, including aspects of information theory, computational complexity, statistics, combinatorics, abstract algebra, number theory, and finite mathematics generally.

3.1.1 Confidentiality

When we talk about confidentiality of information, we are talking about protecting the information and data from disclosure to unauthorized people. Encryption ensures that only the people who knows the key can read the information: if the data is confidential, it cannot be read or understood by anyone other than the intended recipient or recipients.

3.1.2 Block cyphers vs stream cyphers

A stream cipher is a symmetric key cipher where plaintext digits are combined with a pseudorandom cipher digit stream (keystream). In a stream cipher, each plaintext digit is encrypted one at a time with the corresponding digit of the keystream, to give a digit of the ciphertext stream. Since encryption of each digit is dependent on the current state of the cipher, it is also known as state cipher. In practice, a digit is typically a bit and the combining operation an exclusive-or (XOR).

The pseudorandom keystream is typically generated serially from a random seed value using digital shift registers. Stream ciphers represent a different approach to symmetric encryption from block ciphers. Block ciphers operate on large blocks of digits with a fixed, unvarying transformation.

A block cipher is a deterministic algorithm operating on fixed-length groups of bits, called a block, with an unvarying transformation that is specified by a symmetric key.

Claude Shannon analyzed product ciphers and suggested them as a means of effectively improving security by combining simple operations such as substitutions and permutations [35].

Iterated product ciphers carry out encryption in multiple rounds, each of which uses a different subkey derived from the original key. They are used, with different implementations, for DES and AES.

This distinction between the two categories is not always clear: in some modes of operation, a block cipher primitive is used in such a way that it acts effectively as a stream cipher. Stream ciphers typically execute at a higher speed than block ciphers and have lower hardware complexity. However, stream ciphers can be susceptible to serious security problems if used incorrectly; in particular, the same starting state (seed) must never be used twice. Binary stream ciphers are often constructed using linear-feedback shift registers (LFSRs) because they can be easily implemented in hardware and can be readily analysed mathematically. However, if used on their own, they are not secure enough The use of LFSRs on their own, however, is insufficient to provide good security.

3.2 Most used cryptosystems

3.2.1 RSA (Rivest-Shamir-Adleman)

RSA is a cryptosystem for public-key encryption, and is widely used for securing sensitive data, particularly when being sent over an insecure network such as the Internet. It was invented at MIT in 1977 by Ronald Rivest, Adi Shamir and Leonard Adleman. The public key in this cryptosystem consists of the value **n**, which is called the modulus, and the value **e**, which is called the public exponent. The private key consists of the modulus **n** and the value **d**, which is called the private exponent [10].

An RSA public-key / private-key pair can be generated by the following steps:

- Generate a pair of large, random primes p and q.
- Compute the modulus n as n = pq.
- Select an odd public exponent e between 3 and n-1 that is relatively prime to p-1 and q-1.
- Compute the private exponent d from e, p and q.
- Output (n, e) as the public key and (n, d) as the private key.

The encryption operation in the RSA cryptosystem is exponentiation to the e^{th} power modulo n: c = ENCRYPT (m) = $m^e mod n$.

The input m is the message; the output c is the resulting ciphertext. The actual message is encrypted with the shared key using a traditional encryption algorithm. This construction makes it possible to encrypt a message of any length with only one exponentiation.

Without the private key (n, d) (or equivalently the prime factors p and q), it's difficult to recover m from c. Consequently, n and e can be made public without compromising security, which is the basic requirement for a public-key cryptosystem.

The RSA Algorithm is utilized extensively for many practical applications, for example for "key exchange" and/or "digital signature". For example it is used by web browsers to validate the certificate for the remote server to which you have connected. Once it has validated the certificate it may use RSA to perform a secure key exchange with the server. RSA is also used for many Digital Rights Management (DRM) applications. Plus, it is not governed by any active patents, so anyone can use it in any private or commercial product.

3.2.2 ECC (Elliptic Curve Cryptography)

Elliptic-curve cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC requires smaller keys compared to non-ECC cryptography (based on plain Galois fields) to provide equivalent security.

Elliptic curves are applicable for key agreement, digital signatures, pseudo-random generators and other tasks. Indirectly, they can be used for encryption by combining the key agreement with a symmetric encryption scheme.

For elliptic-curve-based protocols, it is assumed that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible: this is the "elliptic curve discrete logarithm problem" (ECDLP).

The primary benefit promised by elliptic curve cryptography is a smaller key size, reducing storage and transmission requirements, i.e. that an elliptic curve group could provide the same level of security afforded by an RSA-based system with a large modulus and correspondingly larger key: for example, a 256-bit elliptic curve public key should provide comparable security to a 3072-bit RSA public key.

The U.S. National Institute of Standards and Technology (NIST) has endorsed elliptic curve cryptography in its Suite B set of recommended algorithms, specifically elliptic curve Diffie–Hellman (ECDH) for key exchange and Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signature. The U.S. National Security Agency (NSA) allows their use for protecting information classified up to top secret with 384-bit keys. However, in August 2015, the NSA announced that it plans to replace Suite B with a new cipher suite due to concerns about quantum computing attacks on ECC. While the RSA patent expired in 2000, there may be patents in force covering certain aspects of ECC technology. The use of elliptic curves in cryptography was suggested independently by Neal Koblitz and Victor S. Miller in 1985. Elliptic curve cryptography algorithms entered wide use in 2004 to 2005.

For current cryptographic purposes, an elliptic curve is a plane curve over a finite field (rather than the real numbers) which consists of the points satisfying the equation $y^2 = x^3 + ax + b$,

along with a distinguished point at infinity, denoted ∞. (The coordinates here are to be chosen from a fixed finite field of characteristic not equal to 2 or 3, or the curve equation will be somewhat more complicated.)

3.2.3 DES (Data Encryption Standard)

The Data Encryption Standard is a symmetric-key algorithm for the encryption of electronic data. It was developed in the early 1970s at IBM and, in a slightly modified version was later published in 1977 as an official Federal Information Processing Standard (FIPS) for the United States. DES is insecure, due to the 56-bit key size being too small: it was broken in January 1999 in 22 hours and 15 minutes by distributed.net and the Electronic Frontier Foundation.

This cipher has been superseded by the Advanced Encryption Standard (AES).

DES takes a fixed-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bitstring of the same length. In the case of DES, the block size is 64 bits. DES also uses a key to customize the transformation, so that decryption can supposedly only be performed by those who know the particular key used to encrypt. The key ostensibly consists of 64 bits; however, only 56 of these are actually used by the algorithm. Eight bits are used solely for checking parity, and are thereafter discarded. Hence the effective key length is 56 bits.

3.3 Focus on AES-256

AES cryptosystems exist in three different versions: the 128, 192, and 256-bit variants. They are base on the Rijndael block ciphers developed by Belgian cryptographers Vincent Rijmen and Joen Daemen. These three variants of AES are based on different key sizes (128, 192, and 256 bits).

These are the main advantages of AES:

- It's supported by most vendors, for various applications
- Fast for both hardware and software
- It is robust
- It is difficult to hack brute force (for 128 bit, 2^{128} attempts are needed to break the algorithm)

The three possible key lengths supported by AES allow users to pick a tradeoff between speed and security. Increased key length increases the execution time of both encryption and decryption. AES uses a single S-Box for all bytes in all rounds, while DES uses eight distinct S-Boxes, which increases implementation requirements.

The encryption phase of AES can be divided into three phases: the initial round, the main rounds, and the final round. They are combined as follows:

Initial Round

- AddRoundKey

Main Rounds (to be repeates 13 times for AES-256)

- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey

Final Round

- SubBytes
- ShiftRows
- AddRoundKey

The main rounds of AES are repeated a set number of times for each variant of AES. AES-128 uses 9 iterations of the main round, AES-192 uses 11, and AES-256 uses 13.

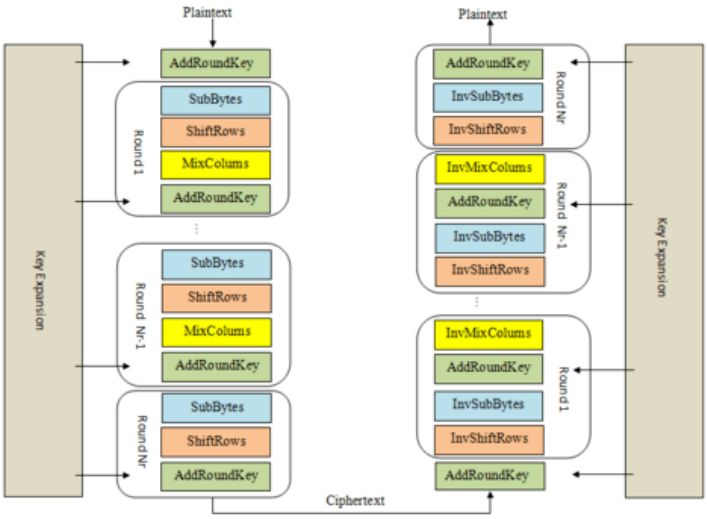

Figure 3.1: AES overview, according to [1]

AES operates on a 4×4 [[column-major order]] matrix of bytes, termed the "state", For instance, if there are 16 bytes,

$$b_0, b_1, ..., b_{15},$$

these bytes are represented as this matrix:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

The key size used for an AES cipher specifies the number of repetitions of transformation rounds that convert the input (called the plaintext), into the final output (called the ciphertext). The number of cycles of repetition for AES 256 (256-bit keys) are **14 cycles** of repetition.

Each round consists of several processing steps, each containing four stages, including one that depends on the encryption key itself. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key [2].

3.3.1 Description of the algorithm

- **KeyExpansions**: round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.
- **InitialRound** AddRoundKey: each byte of the state is combined with a block of the round key using bitwise xor.
- **Rounds** SubBytes: a non-linear substitution step where each byte is replaced with another according to a lookup table. ShiftRows: a transposition step where the last three rows of the state are shifted cyclically a certain number of steps. MixColumns: a mixing operation which operates on the columns of the state, combining the four bytes in each column. AddRoundKey
- **Final Round (no MixColumns)** SubBytes, ShiftRows and AddRoundKey.

Focus on a Round

This is a top level view of the four operations performed on each of the internal blocks that composes AES.

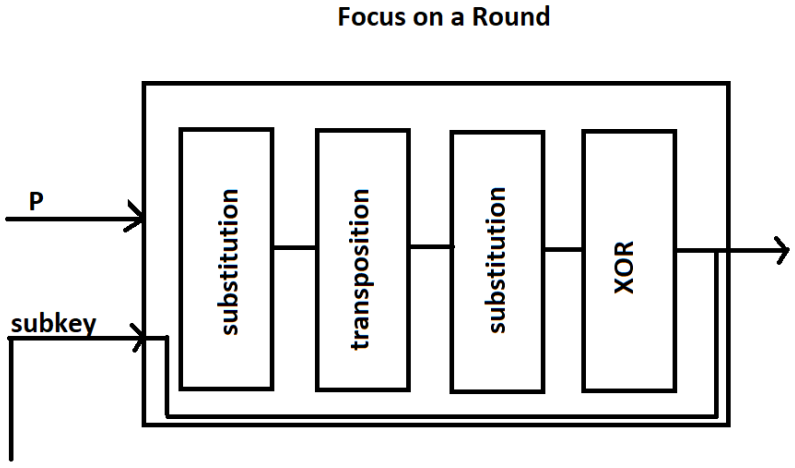

Figure 3.2: Focus on a round

In the first substitution step the 128k block is considered as 16 separated bytes, so the operation is performed byte-wise, each byte is replaced by a different one. For this operation a table is used. The

second step, transposition, is performed to change the order of those bytes. The second substitution is different from the first one because the operation is performed on blocks of 4 bytes, since the available substitutions are 2^{32}, which is a very large number, a formula is used instead of a table. Lastly, the XOR operation performs the xor of the result obtained with the subkey.

The SubBytes:

In the SubBytes step, each byte a_{ij} in the state is replaced with its entry in a fixed 8-bit lookup table Sbox (substitution box), which implements inverse multiplication in Galois Field 2^8:

$b_{ij} = \mathrm{S}(a_ij)$.

Sbox is an 8-bit substitution box, called the Rijndael S-box.

The byte input is broken into two 4-bit halves. The first half determines the row and the second half sets the column. This operation provides the non-linearity in the cipher, and it is constructed by combining the inverse function with an invertible affine transformation. During the decryption step, called the InvSubBytes step (the inverse of SubBytes) is used, which requires first taking the inverse of the affine transformation and then finding the multiplicative inverse.

	0	1	2	3	4	5	6	7	8	9	0a	0b	0c	0d	0e	0f
0	63	7c	77	7b	f2	6b	6f	c5	30	1	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	4	c7	23	c3	18	96	5	9a	7	12	80	e2	eb	27	b2	75
40	9	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	0	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	2	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	6	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	8
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	3	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3.3: SubBytes step [1]

ShiftRows:

The rows in this stage refer to the standard representation of the internal state in AES, which is a 4x4 matrix where each cell contains a byte. Bytes of the internal state are placed in the matrix across rows from left to right and down columns. The bytes in each row of the state are shifted cyclically to the left. The number of places each byte is shifted differs for each row [2].

For AES, the first row is not shifted. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. Row (n) is shifted left circular by (n − 1) bytes.

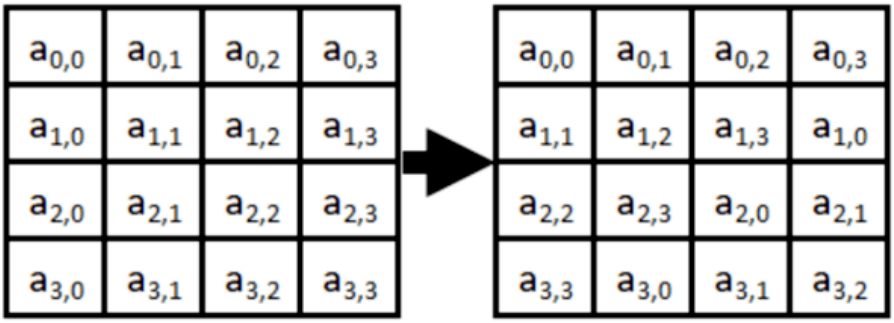

Figure 3.4: ShiftRows step [1]

MixColumns

In the MixColumns step, each column of the state is multiplied with a fixed polynomial c (x)$c(x)c(x)$. in order to increase diffusion.

In the MixColumns step, the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes.

During this operation, each column is transformed using a fixed matrix. Matrix multiplication is composed of multiplication and addition of the entries. Entries are 8 bit bytes treated as coefficients of polynomial of order x 7 x^7 x^7. Addition is simply XOR. Multiplication is modulo irreducible polynomial $x^8 + x^4 + x^3 + x + 1x^8 + x^4 + x^3 + x + 1$.

MixColumns step performs matrix multiplication as per Galois Field 2^8. it is important to note that this multiplication has the property of operating independently over each of the columns of the initial matrix, (eg the first column when multiplied by the matrix, produces the first column of the resultant matrix)

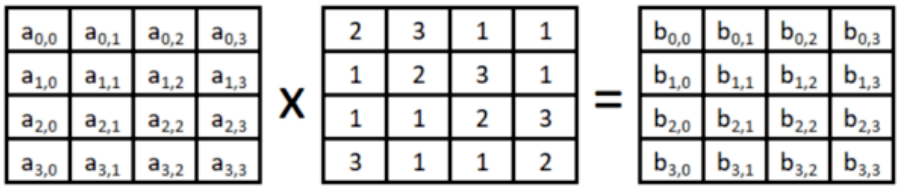

Figure 3.5: MixColumns step [1]

AddRoundKey

The AddRoundKey step directly operates on the AES round key: each byte of the state is combined with a byte of the round subkey using the XOR operation.

Speedup

On systems with 32-bit or larger words, it is possible to speed up execution of this cipher by combining the SubBytes and ShiftRows steps with the MixColumns step by transforming them into a sequence of table lookups. This requires four 256-entry 32-bit tables (together occupying 4096 bytes). A round can then be performed with 16 table lookup operations and 12 32-bit exclusive-or operations, followed by four 32-bit exclusive-or operations in the AddRoundKey step [2].

3.3.2 Key schedule

The AES Key Schedule is used to produce a set number of round keys from the initial key. In AES, the initial key is used in the initial round of AES as input to the AddRoundKey operation. From this

key, 10, 12, or 14 round keys are produced as input to the other AddRoundKey operations in the 128, 192, and 256-bit versions of AES.

Each word (32 bytes) of the previous round key is exclusive-ored with some value to produce the corresponding word of the current round key. In the case of words 1-3, the value used in the exclusive-or is the previous word (words 0-2) of the previous round key. For the first word of the round key, the value used in the exclusive-or is the result of passing the last word of the previous round key through the g function. The g function consists of three stages: a S-Box transformation, a permutation, and an exclusive-or. The S-Box operation used in the AES key schedule is identical to the one used in the encryption phase as described previously. In the permutation phase of the g function, each byte of the word is shifted one position to the left. Finally, the leftmost byte is exclusive-ored with a round constant. The output of the key schedule function is used as the round key input to the AddRoundKey operation in AES encryption. An identical transformation on the round key is performed to produce the next round key.

3.3.3 Decryption

To decrypt an AES-encrypted ciphertext, it is necessary to undo each stage of the encryption operation in the reverse order in which they were applied. This results in this sequence:

Inverse Final Round

- AddRoundKey
- ShiftRows
- SubBytes

Inverse Main Round

- AddRoundKey
- MixColumns
- ShiftRows
- SubBytes

Inverse Initial Round

- AddRoundKey

The AddRoundKey operation is its own inverse (since it is an exclusive-or). To undo AddRoundKey, it is only necessary to expand the entire AES key schedule (identically to encryption) and then use the appropriate key in the exclusive-or. The other three operations require an inverse operation to be defined and used. The first operation to be undone is ShiftRows. The Inverse ShiftRows operation is identical to the ShiftRows operation except that rotations are made to the right instead of to the left. The next operation to be undone is the SubBytes operation. The Inverse S-Box is shown in the Table below. It is read identically to the S-Box matrix. The last inverse operation to define is MixColumns. Like MixColumns, Inverse MixColumns can be defined as the matrix multiplication in Galois Field 2^8.

The specific values in both matrices are chosen in a way such that one multiplication is the inverse of the other in Galois Field 2^8.

3.4 Block cypher mode of operation

The earliest modes of operation, ECB, CBC, OFB, and CFB date back to 1981 and were specified in FIPS 81, DES Modes of Operation. In 2001, the US National Institute of Standards and Technology (NIST) included AES as a block cipher and adding CTR mode, while in January 2010 XTS-AES was

added, too. The block cipher modes ECB, CBC, OFB, CFB, CTR, and XTS provide confidentiality, but they do not protect against accidental modification or malicious tampering, which can be detected with a separate message authentication code or a digital signature. (It was observed that combining a confidentiality mode with an authenticity mode could be difficult and error prone, so they therefore began to supply modes which combined confidentiality and data integrity into a single cryptographic primitive (an encryption algorithm.)

The "easiest" and first block cypher mode is the ECB (Electronic Code Book). It's not a good idea to encrypt data directly using block ciphers like this one. The goal of encryption is to produce ciphertexts that look pseudo-random: there should be no visible patterns in the output. If we are using a block cipher directly as it's performed with ECB , encrypting the same plaintext multiple times will always result in the same ciphertext, so any patterns in the input will also appear in the output.

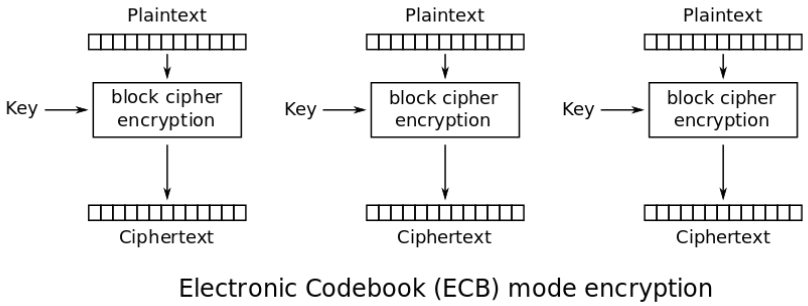

Figure 3.6: ECB algorithm [3]

ECB mode is not secure because when we observe a ECB- encrypted ciphertext some information from the plaintext can leak, for example we can see if two ECB-encrypted messages are identical, or whether they share a common prefix or common substrings, or detect whether (and where) a single ECB-encrypted message contains repeated data. There's a graphical demonstration of this on Wikipedia, where the same (raw, uncompressed) penguin image is encrypted using both ECB mode and a semantically secure cipher mode (such as CBC, CTR, CFB or OFB)

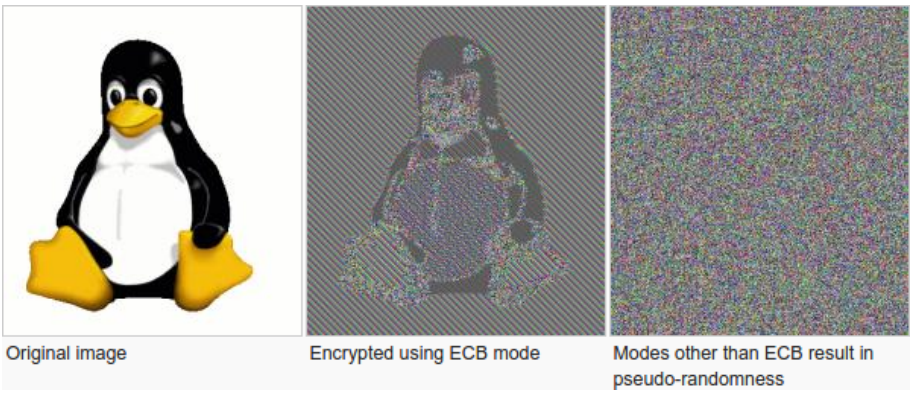

Figure 3.7: Image with two encryption comparison [3]

The ECB encryption mode also has other weaknesses, such as the fact that it is highly malleable: as each block of plaintext is separately encrypted, an attacker can easily generate new valid ciphertexts by piecing together blocks from previously observed ciphertexts.

However, the malleability is only an issue if ECB encryption is used without a message authentication code, and, in this situation, is shared (to some extent) by all other non-authenticated encryption

modes, like CBC, CTR, CFB and OFB. Thus, it cannot really be considered a specific weakness of ECB mode, even though it does tend to be an additional issue whenever ECB mode is used.

Semantically secure encryption modes, one the other hand, are safer even if they suffer from part of the same vulnerabilities of ECB. Stronger block cypher algorithms are:

- Cipher Block Chaining (CBC)
- Cipher Feedback (CFB):
- Output Feedback (OFB)
- Counter (CTR)

3.4.1 Cipher Block Chaining (CBC)

Cipher Block Chaining (CBC) mode of operation was invented in 1976 [40].

The plaintext is XORed with the previous ciphertext before being encrypted. There is no ciphertext before the first plaintext, so a randomly chosen initialization vector (IV) is used instead. An initialization vector is a block of bits that is used by several modes to randomize the encryption and hence to produce distinct ciphertexts even if the same plaintext is encrypted multiple times. An initialization vector has different security requirements than a key, so the IV usually does not need to be secret. However, in most cases, it is important that an initialization vector is never reused under the same key. While for CBC and CFB, reusing an IV leaks some information about the first block of plaintext, and about any common prefix shared by the two messages, for OFB and CTR, reusing an IV completely destroys security [27].

This can be seen because both modes effectively create a bitstream that is XORed with the plaintext, and this bitstream is dependent on the password and IV only, thus reusing a bitstream destroys security [14].

In CBC mode, the IV must, in addition, be unpredictable at encryption time; in particular, the (previously) common practice of re-using the last ciphertext block of a message as the IV for the next message is not secure.

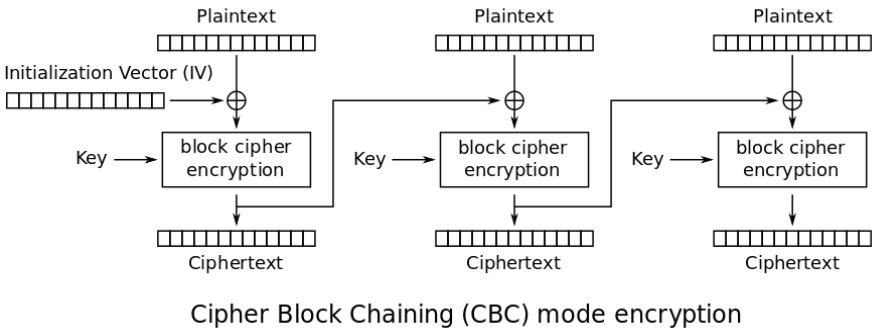

Figure 3.8: CBC algorithm [3]

CBC is a more advanced form of block cipher encryption. With CBC mode encryption, each ciphertext block is dependent on all plaintext blocks processed up to that point. This adds an extra level of complexity to the encrypted data and makes the encryption safer.

Since a block cipher works on units of a fixed size (the block size), but messages come in a variety of lengths. So some modes (namely ECB and CBC) require that the final block be padded before encryption. The simplest scheme is to add null bytes to the plaintext to bring its length up to a multiple of the block size, but care must be taken that the original length of the plaintext can be recovered. Other schemes, slightly more complex, is the original DES method, which is to add a single one bit, followed by enough zero bits to fill out the block. Otherwise there are also some more

sophisticated ones, CBC-specific, such as ciphertext stealing or residual block termination, which do not cause any extra ciphertext, at the expense of some additional complexity. Schneier and Ferguson suggest two possibilities, both simple: append a byte with value 128 (hex 80), followed by as many zero bytes as needed to fill the last block, or pad the last block with n bytes all with value n. The mathematical formula for CBC encryption is

$$C_i = E_K(P_i \oplus C_{i-1}), C_0 = IV \tag{3.1}$$

With the first block having index 1.
CBC has been the most commonly used mode of operation. However it has drawbacks:

- Encryption is sequential and it cannot be parallelized.
- The message must be padded to a multiple of the cipher block size
- A plaintext block can be recovered from two adjacent blocks of ciphertext

3.4.2 Cipher Feedback (CFB)

The previous ciphertext is used as the input for the cipher. Then, the plaintext is XORed with the result to get the new ciphertext. Again, an IV is used to replace the first (missing) ciphertext:

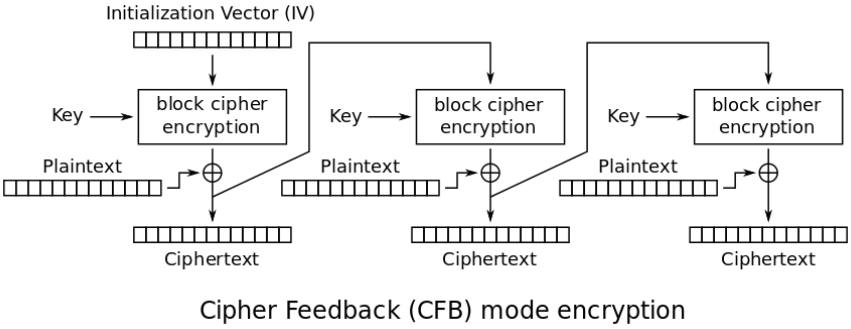

Figure 3.9: CFB algorithm [3]

Like CBC mode, changes in the plaintext propagate forever in the ciphertext, and encryption cannot be parallelized(but also like CBC, decryption can be parallelized).

3.4.3 Output Feedback (OFB)

An initialization vector is repeatedly encrypted to produce a pseudo-random sequence of blocks. Then, these encryption results are XORed with the plaintexts to produce the ciphertexts. Just as with other stream ciphers, flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. This property allows many error correcting codes to function normally even when applied before encryption. Because of the symmetry of the XOR operation, encryption and decryption are exactly the same.

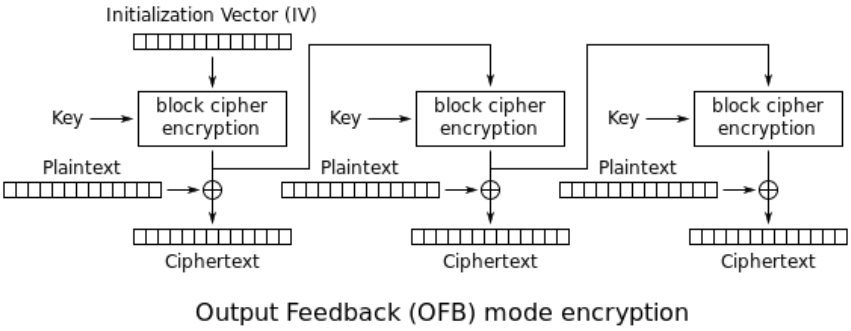

Figure 3.10: OFB algorithm [3]

Each output feedback block cipher operation depends on all previous ones, and so cannot be performed in parallel. However, because the plaintext or ciphertext is only used for the final XOR, the block cipher operations may be performed in advance, allowing the final step to be performed in parallel once the plaintext or ciphertext is available. It is possible to obtain an OFB mode keystream by using CBC mode with a constant string of zeroes as input. This can be useful, because it allows the usage of fast hardware implementations of CBC mode for OFB mode encryption.

3.4.4 Counter (CTR)

CTR mode was introduced in 1979 and like OFB, Counter mode turns a block cipher into a stream cipher. It generates the next keystream block by encrypting successive values of a "counter". An incrementing counter is encrypted to produce a sequence of blocks, which are XORed with the plaintexts to produce the ciphertexts:

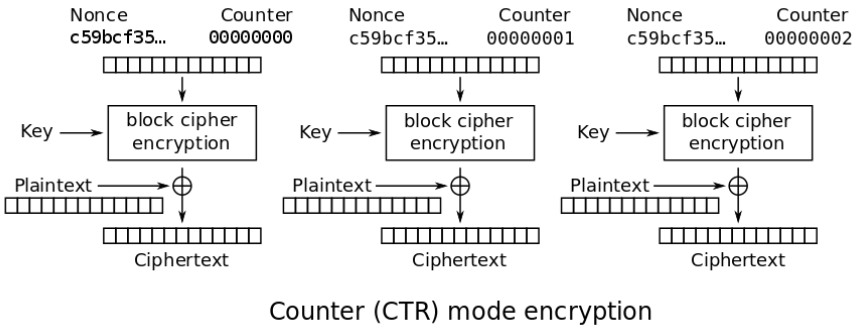

Figure 3.11: CTR algorithm [3]

3.4.5 Conclusions

All four of these modes share the same quality: if the same plaintext block is encrypted multiple times, the result will be different every time. CFB, OFB and CTR modes do not require any special measures to handle messages whose lengths are not multiples of the block size, since the modes work by XORing the plaintext with the output of the block cipher, so they do not require padding. The last partial block of plaintext is XORed with the first few bytes of the last keystream block, producing a final ciphertext block that is the same size as the final partial plaintext block.

For our analysis we will be using primarily CBC mode encryption, then we will repeat the process also on ECB and CFB for a comparison.

Chapter 4

DPA implementative steps overview: steps to test the strength of encryptions on a device

The main goal of this analysis is to test the strength of a cyphered data transmitted to/from a microcontroller and attacked with DPA. After we collected power spectra from the device and analyzed them will compare our results based on these details:

- How many samples are required to "break" the encryption, if the attacks will be successful, and how much time is required to collect them
- Which of the analyzed algorithms are more effective to protect data

4.1 Selection of the algorithm

First of all we select one encryption algorithm, the focus in our analysis will be at first AES-256 (see next chapter for further details on this topic) and then AES 128. Differential Power Analysis attacks examine the power consumption of an intermediate value (a register). This targeted register should leak some information about the cipher key value. In other words, the revealed information have a correlation with cipher key behaviour. For AES the cypher algorithm is organized in different steps that involve repeating a sequence of operations many times called *rounds* and using a lookup tables called *S-box*. DPA exploits leakages occurring during one or more of the encryption steps in order to retrieve the key. For example, output register of SBox has a characteristic affected by cipher key values. Where

$$RSbox = SBox(Plaintext \oplus CipherKey) \tag{4.1}$$

Usually the attack is performed on the first round or on the last round of encryption.

4.2 DPA main steps

As illustrated by Kocher et al [26]. an attack on AES-128 encryption can be based on the analysis of the distance-of-means. They attacked the output of the AES SBox in round 1, where the 16-byte secret key is XORed with the 16-byte of the plaintext. Afterwards, each byte is substituted by another using the SBox, which is an invertible lookup table. If the attacker knows the input plaintext and he is able to read one substituted byte in round 1, he can retrieve the key byte associated. The n_{th} byte I_n of the first round intermediate state after SubBytes is computed as:

$$I_n = Sbox[X_n \oplus K_n] \tag{4.2}$$

With K_{th} the n_{th} key byte and X_{th} the n_{th} input plaintext byte. Knowing X_n, the attacker has to test 256 possible values K_{in} (with i from 0 to 255). The analysis can be summarized with these steps:

- For each trace collected, the first byte I_0 of the intermediate state is computed, with the former equation, using a candidate K^i_{o0}
- Each trace is assigned to one of two subsets using the least significant bit of I_0 as the selection function.
- If the differential trace contains spikes, the two subsets are composed by correlated traces, so the candidate K^i_n is the correct one. If the difference is close to zero, it means that the two subsets are composed by random traces, so the candidate K^i_n is incorrect and the test must be restarted with another candidate.

The 16-byte secret key is then recovered repeating the test for each byte separately on the same set of traces.

4.3 Extending AES-128 Attacks to AES-256

According to literature AES substitution boxes are a good attack point. Since there are S-boxes operating on 1 byte each, we should be able to recover 16 bytes from the SubBytes() function. In the decryption code, this part of the algorithm corresponds to the first three lines [7]:

$$AesAddRoundKeyCpy(buf, ctx->deckey, ctx->key); \tag{4.3}$$

$$AesShiftRowsInv(buf); \tag{4.4}$$

$$AesSubBytesInv(buf); \tag{4.5}$$

This is enough for a complete AES-128 attack because there are only 16 bytes of key to look for. However, in AES-256, this only gets us half of the key, so we also need to attack the next round of the algorithm, which consists of one iteration through this loop [7]:

```
for (i = 14, rcon = 0x80; --i;)
        {
        if( ( i & 1 ) )
                                        {
                                        aes_expandDecKey(ctx->key, &rcon);
                                        aes_addRoundKey(buf, &ctx->key[16]);
                                        }
        else aes_addRoundKey(buf, ctx->key);
        aes_mixColumns_inv(buf);
        aes_shiftRows_inv(buf);
        aes_subBytes_inv(buf); <--- Attack the contents of buf at this point
        }
```

The critical difference between the initial round and this one is the addition of the mixColumns() operation. This operation takes four bytes of input and generates four bytes of output. Either we need to perform a guess over 4 bytes instead of 1 byte or we can apply an inverse equation. This is the state at the end of round 13:

$$X_{13} = SubBytes^{-1}(MixColumns^{-1}(ShiftRows^{-1}(X_{14} \oplus K_{13}))) \tag{4.6}$$

where X_{14} is the output of round 14, K_{13} is the 16 byte round key for round 13, and X_{13} is the output of round 13 (our attack point).

Since MixColumns() is a linear function we can simplify the expression tp get the probable key for round 13:

$$K'_{13} = MixColumns^{-1}(ShiftRows^{-1}(K_{13})) \tag{4.7}$$

and we use it to calculate X_{13} as

$$X_{13} = SubBytes^{-1}(MixColumns^{-1}(ShiftRows^{-1}(X_{14})) \oplus K'_{13}) \tag{4.8}$$

Using this hypothetical key, we can perform a regular attack to recover each subkey one byte at a time. Then, we can recover the actual round key with

$$K_{13} = MixColumns(ShiftRows(K'_{13})) \tag{4.9}$$

According to [7] "*Using these new equations, we can perform a full attack on AES-256 with the following steps*:

- *Perform a regular attack on the first S-box output to recover all 16 bytes of the* 14^{th} *round key.*
- *Using the known values of the* 14^{th} *round key, calculate the outputs of the second S-box. Use this attack point to recover all 16 bytes of the hypothetical* 13^{th} *round key.*
- *Transform the supposed round key into the actual value of the* 13^{th} *round key. Reverse the* 13^{th} *and* 14^{th} *round keys using the AES-256 key schedule to determine the 256 bit secret encryption key.*

Once this is done, we have successfully attacked an AES-256 implementation by looking at two separate AES S-boxes".

4.4 Oscilloscope connection and settings

Having reached this point the oscilloscope is needed in order to take the measurements while the microcontroller performs cryptographic operations.

After the successful connections of the oscilloscope 4.1 to the cryptographic device, adjusting the optimal settings and selecting the correct part of the sample for recording, there is a sequence of steps which should be followed in order to make the attack successful.

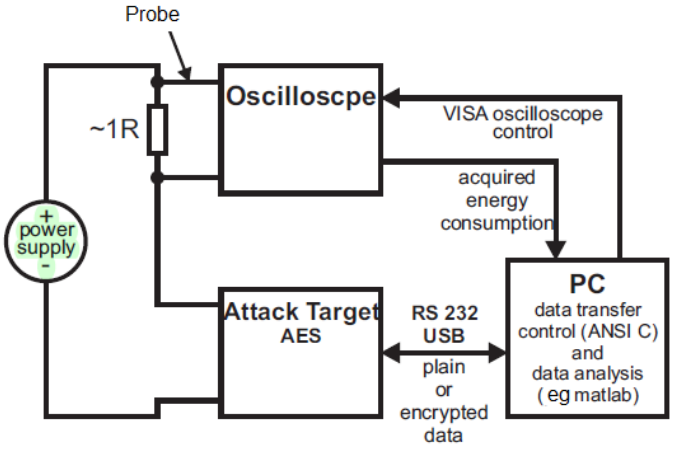

Figure 4.1: Measurements setup

The following are usually the minimum requirements for the oscilloscope [21]:

- Bandwidth of at least 50% of the device clock rate for software implementations and at least 80% of the clock rate for hardware implementations
- Capability to capture samples at 5x the bandwidth
- Enough storage capabilities to acquire the entire signal required for the test and analysis

In the first step, we need to prepare a sufficient amount of plaintext blocks, which are processed by the device. The oscilloscope has to be triggered so that it starts recording at the beginning of the encryption. When the encryption is finished, the recording stops and the captured trace is saved into a file. Then the next sample is taken. Traces may be unaligned due to high clock frequency jitter: this can be fixed next when preprocessing the acquired data.

Signal to noise ratio should be as good as possible in order to decrease number of power measurements during encryptions. For this reason it is better to use active differential probes, if available [30].

4.4.1 Oscilloscope settings and measurements hints

In order to obtain precise measurements the signal must be amplified or the gain setting on the scope or sampling card must be set to measure the full dynamic range (or as much as possible) of the signal being collected [22]. Furthermore at a minimum, the rate is set to five times the bandwidth, with an input bandwidth close to the device clock frequency.

A measurement trace usually must consist of: [22]

- An optional, specified fixed length header
- The raw measurements

The measurements should start with the beginning of the cryptographic operation or the specific location in the computation specified by the test. In case the measurements contain extra data, the actual start of the relevant measurement data must be specified [21].

One way to establish the start of the relevant data is to have the implementation generate a trigger signal when the cryptographic operation is started and storing the traces corresponding to the trigger.

Measurement hints:

- A resistor or a current probe in series with the device's power or ground lines can be used
- Measurements taken closer to the cryptographic component will usually be of better quality
- If a resistor cannot be inserted the device's internal resistance is often sufficient
- For triggering, the measurement system is typically connected to the device's I/O lines.
- Running a device near the edge of its operational parameters may enhance the leak being targeted (eg: lowering the input voltage)

4.4.2 Choice of the resistance

Being R_m the resistor between global supply and the supply pin of the controller, a bigger R_m would mean higher voltage swing across the resistor, which would be easier to measure. On the other hand we have to keep in mind that this voltage drop across R_m reduces the actual supply voltage of the controller which scales down the power consumption. Therefore it is clear that big values for R_m do not directly lead to the desired effect of higher voltage swing. Since power consumption itself also depends heavily on the amount of the supply voltage, to obtain better results one should run the device at the highest supply voltage possible [11].

4.4.3 Connection custom improvements

According to [18] in order to emulate a real attack, a custom connector is built to sample the power signal from the power supply line between the platform and the host PC. This special custom device is tailored for a generic USB device, so it can be employed also for other types of devices. It's composed of an USB male socket and a USB female connector soldered and linked together 4.2.

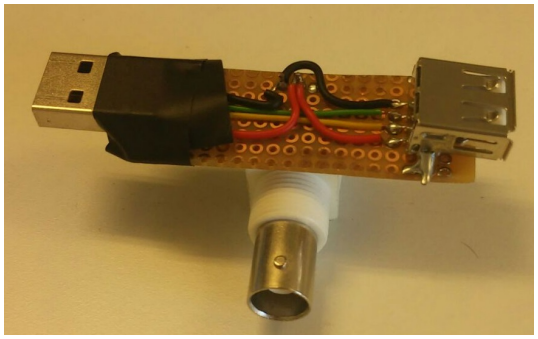

Figure 4.2: Custom connector as explained in [18]

The connection between the two ports is voluntarily left exposed, allowing the probes to be attached easily without introducing distortion. Furthermore they inserted a BNC connector in parallel between the supply line and ground line to connect the device directly to an oscilloscope avoiding the noise introduced by probes ohmic contacts, thus leading to more accurate measurements of the input voltage [18].

4.5 Cypher datum, attack with DPA and write down the measure

Power must be measured by having an external power supply attached to a resistor in series with the V_{CC} line supplying the device under test and measuring the voltage drop across the resistor R, (between A and B) or the voltage between B and the ground (after removing the DC bias).

It's suggested to use these values or similar for the value of resistor and the external bench voltage while performing the measurements:

Device	Nominal V	Max. Work Tolerance	Current Range	Sugg. R	Sugg. V in
μController	3.3V	$\pm$10 percent	50 to 144 mA	7 Ω	3.98V

Table 4.1: Suggested values

4.5.1 Algorithm for the attack

We select intermediate bit to analyze (XORed with final round key value) and align the signals so that different traces can be compared at the same point during the cryptographic calculation, that means aligning data points in time. Power traces are collected and corresponding ciphertext values. We can increase the number of traces used for the test to compensate for additional noise introduced due to misalignment. On the other hand we can also perform decimation in order to save DPA attack computation time [30]. We compute intermediate value, making a guess for key byte. We sort them into two different sets according to one bit in the serie. To compute that bit we use the selection function which needs the key (most of the time 6 bits of the key) and the cypher. After we have the two sets we compute the average of each set and then we want to compute the difference of the

two traces, and this will help to get the key. Once we get the confirmation our key was correct we repeat with different key and bits (repeating it for the other 255 key byte guesses using same power measurements). Then we calculate the master key of the encryption.

Sorting

The information revealed by a DPA test is determined by the choice of selection function. A selection function is used to assign traces to subsets and is typically based on an guess as to a possible value for one or more intermediates within a cryptographic calculation.

If the final DPA trace shows significant spikes, the cryptanalyst knows the selection function output is correlated to (or equals) a value actually computed by the target device. If no correlation is observed, then selection function output was not correlated. Selection functions (typically binary valued functions) may be, for example

- The predicted value of a single bit
- The predicted difference between the value of a bit in a register and the value of a bit that overwrites it
- Functions of multiple bits

For each analysis we need 6 bits on the subkey and a few bits for the cypher text [26].

The selection function, as denoted by D(C, b, Ks), computes value of target bit b, given ciphertext C and key guess Ks. M power traces of k samples each are collected, denoted by T 1:m[1:k] and corresponding ciphertext values C 1:m Sort data into two groups:

$$oD(C, b, Ks) = 0 \tag{4.10}$$

$$oD(C, b, Ks) = 1 \tag{4.11}$$

CMOS devices depend on the constant transitions of internal bits from zero to one and vice versa. The device requires variable amounts of power to perform these transitions. If there exists some correlation between the power model and power consumption there is a high probability that the key guess is the same as the actual key. The selection function determines how to group power traces. It depends on the power consumption model.

The power absorbed by a CMOS circuit is the sum of the static power (caused by device leakage currents) and the dynamic power (due to the switching activity)

$$P = Pstatic + Pdynamic \tag{4.12}$$

The dynamic power can be approximated as:

$$Pdynamic = \frac{1}{2} C_{load} V_{dd}^2 \tag{4.13}$$

It can be seen from the equation that power consumption of a computing device is related to its switching activity, and thus to the data handled. Several power models are possible, for example [18]:

- Hamming Distance : it is the measurement of the difference between two bit strings. It is a logical step forward from Hamming weight as it is a measure of the bits flipped from the previous value as opposed to the zero string. The Hamming distance and Hamming weight are the same if the previous value happens to be the zero string. It models the power consumption of a device by the transitions that occur on observed data. We XOR the the previously observed data value with he current data value add the number of 1's in the result string and that is the Hamming distance [38]. The main drawback of this model is the assumption that values that remain unchanged during the periods of measurement do not contribute to the overall power consumption of this device.

- Hamming Weight: one set contains the traces that are considered to generate the transition of the bit value 0 to 1 and 1 to 1. The other set contains the remaining traces; This would mean that all the appearances in the number '1' in the bit string of the intermediate value contributed to power consumption regardless of the value of that bit in the key guess. The use of this model assumes that the power consumption of the device is based on the number of bits switched from the zero bit string. This model has a relatively weak correlation to the power actually consumed by the device. Due to this generally weak correlation to the actual power consumption, this model often oversimplifies the circuit under attack and other power models should be considered. A major benefit of this model is the lack of implementation information needed to successfully implement this power model [38].
- Rising transition: the set PA contains the input transitions that make the target gate to commute from 0 to 1. The set PB contains all the other input transitions. This function models in an accurate way a CMOS circuit, whose power consumption is measured by means of a small resistor [23].

By construction, one of the two sets will contain a power consumption component not present in the other set. The choice of the selection function will change the components of the two sets, and as consequence also the averages change. In general, this will lead to different results in term of correct bits [18].

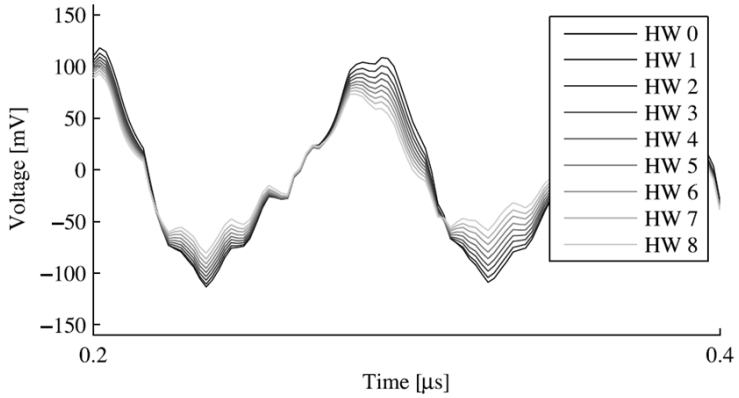

Figure 4.3: Hamming Weight dependancy between Power and Data. Source: Dpa Book [20]

4.5.2 Load of the algorithm on the microcontroller

The microcontroller generates a trigger signal to help the oscilloscope recognize the start of the encryption. First, we have to load an AES implementation in C language into the microcontroller.

4.5.3 Data collection

The first step is to collect one or more traces from the analyzed device. A trace is a sequence of measurements taken across a cryptographic operation or sequence of operations. The data trace can be captured by placing the resistor in series with the device's ground line, then using an oscilloscope to measure the voltage at the ground input. Accordingly, a larger measurement represent higher power consumption.

For each individual cryptographic operation that was part of the test, the measurement process must record:

- the key used, in our case it is the AES 256 and AES 128, so it is 256 bits of key for the first case and 128 bits for the former.
- the input data

- the result produced by the algorithm implementation in the DUT

Furthermore, the result must be checked with the algorithm for each trace in order to exclude there had been a malfunction in the device.

For AES implementations to be validated at a moderate security level resistance against unprofiled, first order attacks may be sufficient.

Average

The k-sample differential trace Δ[1..k] is performed to find the difference between the average of the traces for which D(C,b,Ks) is one and the average of the traces for which D(C,b,Ks) is zero.

For each sample, the differential trace Δ[j] is the average over the measured ciphertexts of the effect caused by the selector function D(C,b,Ks) on the power consumption measurement at the sample point If Ks is incorrect, the probability that D will yield the correct bit b is $\frac{1}{2}$, so the trace components and D are uncorrelated. The result is that Δ[j] approaches zero for large m.

If Ks is correct, the computed value for D will equal the actual value of the target bit b with probability 1, making the selection function correlated to the bit. The result will be spikes in the differential trace where D is correlated to the value being processed. da slides

We do the difference of the two sets if the assumption on the key is wrong the two will cancel. On the other hand if the assumption is correct we see peaks in the difference trace set that in the original were not present.

We establish a threshold for these peaks: If there is any point in time for which exceeds this threshold, the device fails hiding the cyphered data. Otherwise, the device is robust enough not to be attacked by DPA.

We do that on every different blocks of bits In DES the schedule is linear and we can compute the master key based on the subests we collected.

Difference

The differential trace Δ[j] is computed as the difference between the two average traces.
For an incorrect key guess K the Δ should approach zero, For a correct key guess K the Δ should approach the target bit's power contribution at the correlated sample(s).

$$\Delta_D[j] = \frac{\sum_{i=1}^{m} D(C_i, b, K_s)T_i[j]}{\sum_{i=1}^{m} D(C_i, b, K_s)} - \frac{\sum_{i=1}^{m}(1 - D(C_i, b, K_s))T_i[j]}{\sum_{i=1}^{m}(1 - D(C_i, b, K_s))} \tag{4.14}$$

$$\Delta_D[j] \simeq 2(\frac{\sum_{i=1}^{m} D(C_i, b, K_s)T_i[j]}{\sum_{i=1}^{m} D(C_i, b, K_s)} - \frac{\sum_{i=1}^{m} T_i[j]}{m}) \tag{4.15}$$

Evaluation

In simple cases, the results of DPA can be evaluated using visual inspection.

- Large peaks in the differential trace mean correct key guess
- Smaller peaks are signs of an incorrect key guesses

This is exploited for example by simple power analysis. For some algorithms, and for certain types of DPA attacks, the evaluation process is more complex, since there are other guesses ("harmonics" of the correct key) besides the correct key which have significant correlation to the target leak (and they may show as spikes in the differential trace).

Iteration is required for algorithms such as AES-256, where multiple round subkeys or multiple encryption keys must be found: new information about the key enables the generation of new selection functions.

Although harmonics can usually be ignored, they can provide useful information. The pattern of harmonic peaks can help identify the correct value. Harmonics are a function of the target algorithm and the selection function strategy employed. It is, therefore, possible to analyze the selection function

process, determine the pattern of harmonics it would generate, and then match this pattern against the amplitudes of the observed harmonics.

AES Test Vectors: a practical example

In literature there are some practical examples for the selection of clever test vectors. In [21] they suggest the tester must collect two data sets *DATASET1* and *DATASET2* from the core AES cryptographic block encryption with a specific, published key and a set of data as follows:

For DATASET1 the Key K for AES 256 is set to:

$$K = 0x0123456789abcdef123456789abcdef023456789abcdef013456789abcdef012 \tag{4.16}$$

They perform 2n encryptions with inputs: I0,I1,. . . ,I2n, where

- I0 = 0x00000000000000000000000000000000 (16 0 bytes)
- Ij+1 = AES(K, Ij) for 0 < j < 2n
- n is the number of distinct encryption samples chosen by the tester after the review of the documentation

The number of inputs and number of encryptions for this data set shall be twice the number of distinct samples deemed reasonable for an attacker to collect. In this case, the key K will perform 2n AES encryptions. The input for the first encryption shall be all zeros and each subsequent encryption uses the output of the previous encryption as its input, so it is a sort of CFB block cypher mode aes (see next chapters for details) with the initialization vector set to zero.

For DATASET2 the Key K is set to:

$$K = 0x0123456789abcdef123456789abcdef023456789abcdef013456789abcdef012 \tag{4.17}$$

And data J is set to 0xda39a3ee5e6b4b0d3255bfef95601895.
They perform n encryptions with input J. DATASET2 uses the same key as DATASET1, but repeatedly performs an encryption with a single fixed data value. Both DATASET1 and DATASET2 require the entire AES operation to be measured, recorded and checked. It is recommended that the collection of DATASET2 be interspersed with the collection of the first n traces from DATASET 1 in order to eliminate any time dependent bias between traces from DATASET1 and DATASET2.

DATASET1 was chosen to essentially provide a set of AES operations with a fixed key (for different key sizes) and a deterministic set of inputs which appear essentially random from a statistical perspective. On the other hand, DATASET2 was chosen to have the choice of key identical to DATASET1. The data J of the experiment was selected such that the following four conditions were met:

- In one middle round (outside first and last round) there is at least one data byte equal to zero.
- In one middle round, there is at least one byte of data XOR round key equal to zero.
- In one middle round, there is at least one S output byte equal to zero.
- In one middle round, there is at least one byte of round in XOR round out equal to zero.

To find that vector J, the following approach was used: they started with "J = first 121 bits of SHA1("TEST0") followed by 7, “0” bits" [21] and then they incremented J until the criterion is met.

4.6 DPA simulation: a case study

Since DPA can be performed during different stages of a device production and according to several paramenters a team of reserchears have evaluated a measuring and laboratory environment to simulate that [23].

The validation environment can be composed of two main parts:

- *synthesis and simulation environment* for acquisition of simulated power traces
- *DPA Suite* for performing DPA attack.

The synthesis and simulation environment takes the VHDL description of the circuit and automatically generates all the scripts required to synthesize it with a given target library. Then it generates the scripts required to perform the transistor level simulation of the circuit for a user defined vector set. The simulation is performed in such a way that it is possible to gather all the measures that an attacker could have obtained on a real device, for example the power traces are collected.

Usually a simulation involving circuits with hundreds of thousands transistors is performed in minutes.

They used a simulator that is an advanced circuit simulator for analogic, high performance digital and mixed-signal circuits, which provides an output that is then converted into a set of waveforms, one for each input vector.

This format is the same of the one usually generated by digital oscilloscopes, this way DPA Suite software can be used also in the case of real measurements on a physical device.

DPA Suite is composed of two main tools: *dpa* and *keyexplore*.

Starting from the waveforms and the input vectors, *dpa* tool generates the DPA traces for each key supposition, In particular classifying for each key supposition the waveforms based on the logical function implemented by the circuit and according to the chosen power consumption model.

DPA handles three power consumption models (Hamming weight, rising transition, and Hamming distance, previously described). and it is able to target one or more output bits of the circuit's SBox.

Furthermore they implemented also a method to calculate the multi-bit DPA trace [**?**] to detect the correct key. The selection function is applied independently to each target bit, in order to partition the waveforms in two sets for each target bit (PA1 and PB1 for bit b1, ..., PAn and PBn for bit bn). The final DPA trace is computed including waveforms in sets according to the selection function. Then the *keyexplore* tool allows detecting the value of the secret key starting from the DPA traces. It associates the secret key to the trace with largest power consumption. They used three different criteria:

- Amplitude: the curve with the highest absolute value is considered as the one corresponding to the correct key
- Integral: the curve with the highest average value is considered as the one corresponding to the correct key
- Time: the curve that has the longest time period during which the curve is the highest is considered as the one corresponding to the correct key.

To validate the resistance of a design against DPA attack it is necessary to compute the number of input vectors required to conduct a successful attack. For example, they used a sequence composed of 256 vectors. They showed that at least the first 56 patterns of this sequence are necessary to distinguish the curve corresponding to the right key; furthermore, the longer the input sequence, the more evident is the curve corresponding to the right key.

Chapter 5

Correlation power analysis

Correlation Power Analysis (CPA) is another attack, similar to DPA, that allows to find the secret encryption key. These are the main steps of the algorithm [4]:

- Evaluate a model for the victim's power consumption
- Encrypt several different plaintexts and record a trace of the victim's power consumption during each of these encryptions.
- Attack small parts (subkeys) of the secret key: for each guess and each trace, use the known plaintext and the guessed subkey to calculate the power consumption according to the model.
- Calculate the Pearson correlation coefficient between the modelled and actual power consumption and decide which subkey guess correlates best to the measured traces, then put together the subkey guesses to obtain the full key.

5.0.1 Power Consumption Models

The power consumption models used for CPA are the same described above when dealing with Dpa. So they are for example Hamming Distance or Hamming Weight models.

5.0.2 Pearson's Correlation Coefficient

In order to compare our power estimate to our measured traces we use the Pearson's correlation coefficient, which is

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sqrt{E[(X-\mu_X)^2]E[(Y-\mu_Y)^2]}} \tag{5.1}$$

This correlation coefficient will always be in the range [-1, 1]. It describes how closely the random variables X and Y are related: If Y always increases when X increases, it will be 1; If Y always decreases when X increases, it will be -1; If Y is totally independent of X , it will be 0. "*These equations are typically used to pick out patterns in noisy signals. In our attack, we'll be looking for a our model (a pre-calculated pattern) in measured power traces (noisy signals)*" [4] .

5.0.3 Using correlation

After taking our measurements, we'll have D power traces t, and each of these traces will have T data points. Using subscript notation,

$$t_{d,j} \tag{5.2}$$

will refer to point j in trace d.

We'll also estimate the power consumption in each trace using our model. We'll say that there are I different subkeys that we want to try. Then,

$$h_{d,i} \tag{5.3}$$

will refer to our power estimate in trace d, assuming that the subkey is i.

With this data, we can see how well our model and measurements match for each guess i and time j. We'll do this by finding how t and h correlate over the D traces. One way of calculating this is:

$$r_{i,j} = \frac{\sum_{d=1}^{D} \left[\left(h_{d,i} - \overline{h_i} \right) \left(t_{d,j} - \overline{t_j} \right) \right]}{\sqrt{\sum_{d=1}^{D} \left(h_{d,i} - \overline{h_i} \right)^2 \sum_{d=1}^{D} \left(t_{d,j} - \overline{t_j} \right)^2}} \tag{5.4}$$

There is an alternative form of the correlation equation to be used for "online" calculations - it allows us to add one trace at a time without re-summing all of the past data. This form is:

$$r_{i,j} = \frac{D \sum_{d=1}^{D} h_{d,i} t_{d,j} - \sum_{d=1}^{D} h_{d,i} \sum_{d=1}^{D} t_{d,j}}{\sqrt{\left(\left(\sum_{d=1}^{D} h_{d,i} \right)^2 - D \sum_{d=1}^{D} h_{d,i}^2 \right) \left(\left(\sum_{d=1}^{D} t_{d,j} \right)^2 - D \sum_{d=1}^{D} t_{d,j}^2 \right)}} \tag{5.5}$$

5.0.4 Subkey selection

The last step is to use the values of

$$r_{i,j} \tag{5.6}$$

to decide which subkey matches our traces most closely [4]. There are two steps to this:

- For each subkey i, find the highest value of

  $$|r_{i,j}| \tag{5.7}$$

  This will discard the time information.

- Looking at the maximum values for each subkey, find the highest value of

  $$|r_i| \tag{5.8}$$

  The location i of this maximum is our best guess: it correlated more closely with the traces than the other guesses.

5.0.5 Comparison between DPA and CPA

Correlation power analysis looks at correlation between all key guesses while DPA looks at difference of category averages for all key guess, furthermore CPA is faster and more accurate than DPA, it is less noisy and requires less traces to guess the correct key.

In order to have good hypothesis we have to pay attention to the dependency:

- to known plaintext
- to small amount of key bits
- to small changes, since it is non-linear
- to the model for power consumption mapping

Chapter 6

Step by step description of the "Cypher and Attack Process"

After reading the DPA and AES documentation cited in the pages above I set up the “cypher and attack” system. The goal of this work is to encrypt a text with AES-256 in CBC, ECB and CFB block cypher mode of operation, and try to get the correct key it by means of DPA and CPA. Then the operation will be repeated for AES-128 ECB too.

6.1 Program to be loaded on the microcontroller

6.1.1 Nucleo F401RE features

In the following there is an overview of the features of the Nucleo F401RE we are going to use.

- STM32F401RET6 in LQFP64 package
- ARM®32-bit Cortex®-M4 CPU with FPU
- 84 MHz max CPU frequency
- VDD from 1.7 V to 3.6 V
- 512 KB Flash
- 96 KB SRAM
- GPIO with external interrupt capability
- 12-bit ADC with 16 channels
- USB 2.0 OTG FS

6.1.2 Encryption program

First of all I used STMCubeMX software provided by STMicroelectronics in order to easily enable the libraries and utilities needed to run the software on the microcontroller. Then, with it, I produced an empty projects usable with STM32 Workbench and I wrote the code to perform the encryption by means of AES 256. This simple piece of code is written exploiting the AES256.h and AES256.c library functions provided by STMicroelectronics.

With the proper bits enabled the software sets the encryption block mode to CBC, ECB or CFB. After deciding a cypher key and an input vector (required for the CBC mode) the program cypher a given plaintext and repeats the procedure in a while(1) loop.

The encryption algorithm is enclosed between the power on and off of a LED, this is implemented to make automatic capture from the oscilloscope easier. The oscilloscope start capturing the traces when the LED goes on until the next rising edge (which is the next encryption).

In order perform the DPA analysis we keep the same plaintext except for the a byte, which goes from 0 up to 255 in a loop. Then another byte of the same plaintext does the same thing and so on. We keep the same encryption key and Initialization Vector at every encryption when the IV is required (CBC encryption).

For reference, these are the most useful data:

- Key: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
- Plaintext (reference one, subject to variations at each iteration):
  0xbd 0x0f 0x0f 0xff 0x3f 0x2a 0xf4 0x51 0xb4 0xc7 0x00 0x00 0xff 0x87 0x32 0x00
- IV: 0 1 1 2 3 2 1 1 0 0 0 0 0 0 6 5

```
for(last=0; last< 255; last++ ){
//I change the last byte of the 128 bit block
clrData[0]=last;
//I make sure the led starts as _ON

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

//I switch the pin Off when the encryption is finished

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}
```

Using the piece code reported above (the example is for ECB block cypher mode, the others vary a little, the full code is available in the Appendix at the end of the thesis) we test the algorithm with 4096 different plaintexts, this guide line will be use for most of the analysis reported in the next chapters.

6.2 Steps performed in the laboratory

6.2.1 Instrumentation

The instrumentations used to perform the analysis are reported in 6.1. The digital oscilloscope has been selected with the largest bandwidth available in order to perform the measurements with high accuracy, it is provided with an operating system which makes the acquisition step more user friendly.

Instrument	Specifics
Digital Oscilloscope	LeCroy WaveRunner 6030, 350 MHz (Quad 2.5 GS/s)
Current probe and Amplifier	Tektronix TM502A AM 503B
Multimeter	Hewlett Packard 34401A
DC power supply	Rigol DP832 Programmable

Table 6.1: Instrumentation used to perform the measurements

6.2.2 Connection

The device Nucleo F401RE equipped with the microcontroller STM32F401RE can be powered by means of the USB cable or with an external PIN named V_{in}. After loading the program on the microcontroller with the USB cable the F401RE we can unplug the device from the PC and it will not lose any information because the program and data are stored in a flash memory (512KB). DPA is usually performed evaluating supply current fluctuations either with a current probe or with a resistance and voltage measurement; in a first evaluation I added a 1 kΩ resistance in series between supply and V_{in} pin, and took voltage measurements on it. Then I tried with the current probe and noticed that the measurements were less subject to noise. The current probe has a 50 MHz bandwidth and the Nucleo device is working at 8 MHz, which is in band. The resistance has the function to enhance the leakage and make the measurements easier. Since current peaks were slightly more visible with the small resistance in series I took the measure with both probe and resistance. The measured voltage supply in input is within the working range of the device.

Since running a device near the edge of its operational parameters may enhance the leak being targeted I made sure the voltage arriving at the device is slightly above than the nominal 3.3 V supplied. On the other limit the available dynamic is not very flexible, because at 3.2 V there were problems with the LED blinking.

In 6.1 there is the connection setup for the measuring system. The AC power supply is connected to a breadboard and the Nucleo device through the resistance. The current probe reads the trigger from pin PB_5. The probe must be matched in terms of current/voltage divisions with the oscilloscope. In order to see the correct "numbers" on the oscilloscope the voltage-current/divisions of the two must be set to 10 mA-mV/div. Sometimes I used also the 5 mA-5mV/div in order to have a better oscilloscope representation: in that case the results looks as if multiplied for two, but this does not change the outcome for the DPA. In the following analysis two settings will be used:

Vertical Divisions	Coupling
10 mA/div	DC 1 MΩ
5 mA/div	DC 1 MΩ

Table 6.2: Current probe settings used

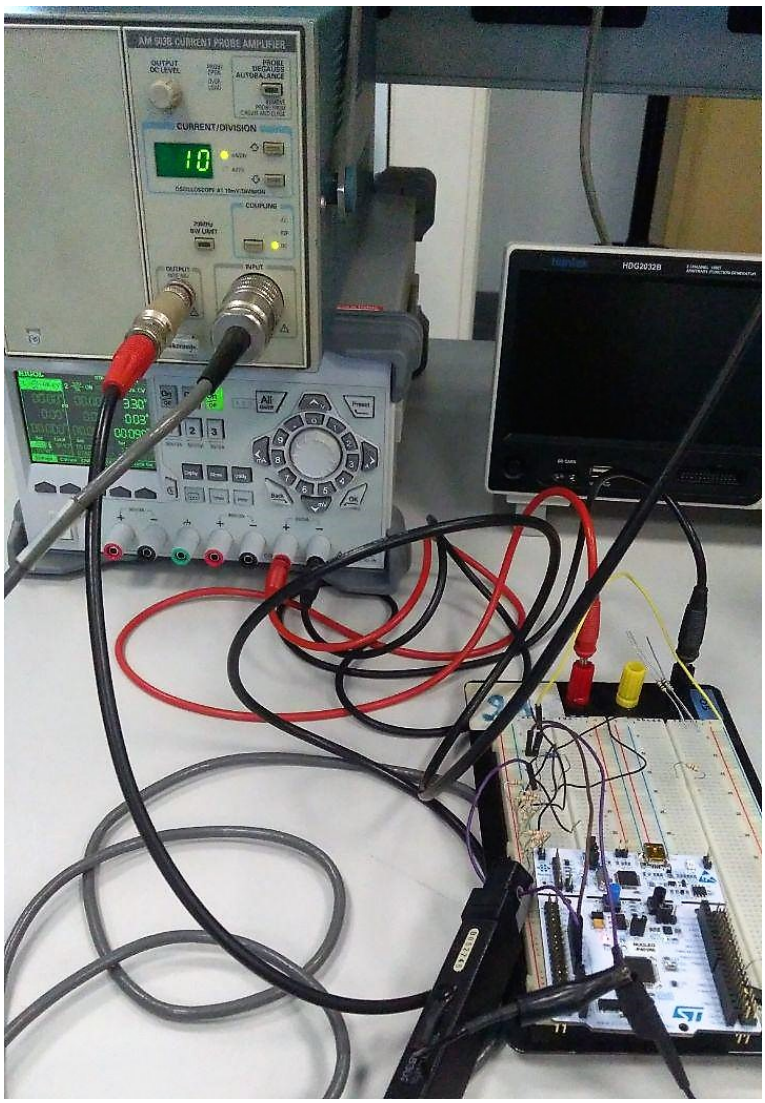

Figure 6.1: Hardware connections

6.2.3 Oscilloscope settings and trace acquisition

The traces have been taken with the LeCroy Oscilloscope using two channels (one for the trigger and one for the actual probe measure) with the following settings:

Setting	Value
Time division	10 μs/div
Trigger	Positive rising edge, value= -20 mV
Sampling	25 Ms/s, 100 MS/s, 250 MS/s or 1GS/s

Table 6.3: Oscilloscope settings used

This acquisition is triggered by the LED switch on on the microcontroller. The Nucleo microcontroller has a working frequency of 8 MHz, so the sampling is sufficient to perform the acquisition at

a frequency higher than:

$$F = 2 \cdot f0 \tag{6.1}$$

According to Nyquist-Shannon sampling theorem.

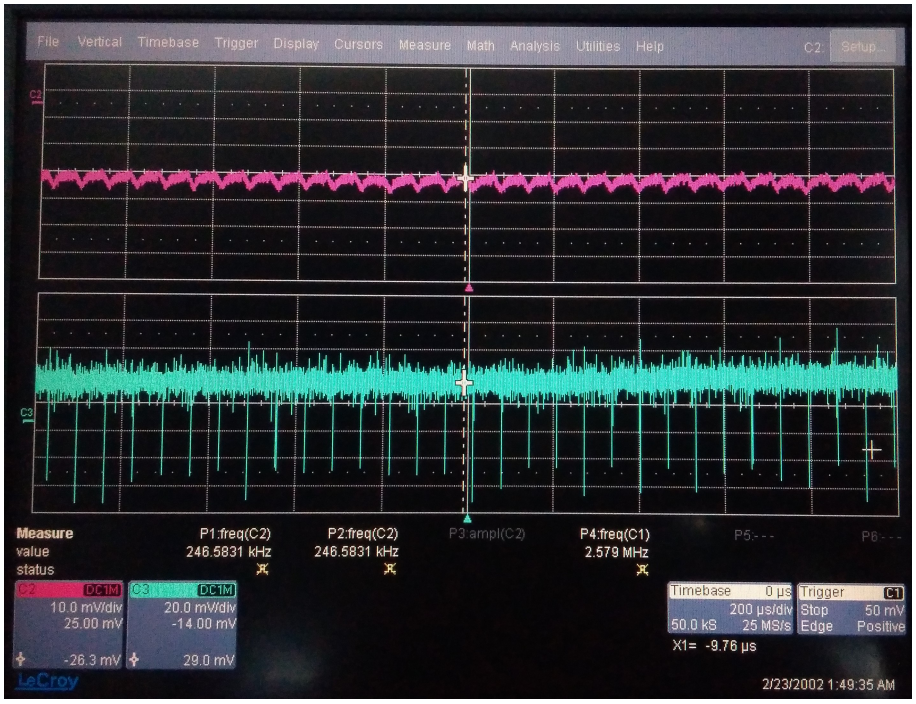

Figure 6.2: Oscilloscope traces

In figure 6.2 the two waveforms are represented: in pink the power supply and in blue the trigger.

In order to make sure the encryption starts with a known plaintext (in my case it is the plaintext with the last byte set to zero) the reset button is kept active on the device until the acquisition begins. The first acquired traces will contain only noise and will be manually discarded (they are easily detected because the acquired values are 1/100 of the expected ones).

The oscilloscope saves power traces (Amplitude Only) on an external USB device in .dat format, and at the end the acquisition is stopped manually. The power traces will be analyzed and elaborated by the DPA tool described in the next section.

6.2.4 Analysis with Tools for DPA and CPA

DPA and CPA Tools usually need, beside the acquired power traces, the set of plaintexts in input and their corresponding cyphered outputs. Since it is a large data set and it can't be evaluated real time during acquisition phase I implemented a parallel version of the microcontroller program, to be run on the PC, to evaluate ordered plaintexts and ciphertexts. These are saved in two .dat files and generally contain 4096 different vectors as plaintext, each divided in 16 bytes (Uint8 or HEX format depending on the tool used for the analysis).

In the next chapter a deeper insight of the used tools and analysis is performed and reported.

Chapter 7

Analysis in detail with AES-256 encryptions and AES-128 ECB

7.1 A priori evaluation with Matlab

After having captured every set of traces the first thing I did is to plot some traces on Matlab: this is to check they are sufficiently allineated and to discard the first "reset-state" captured traces.

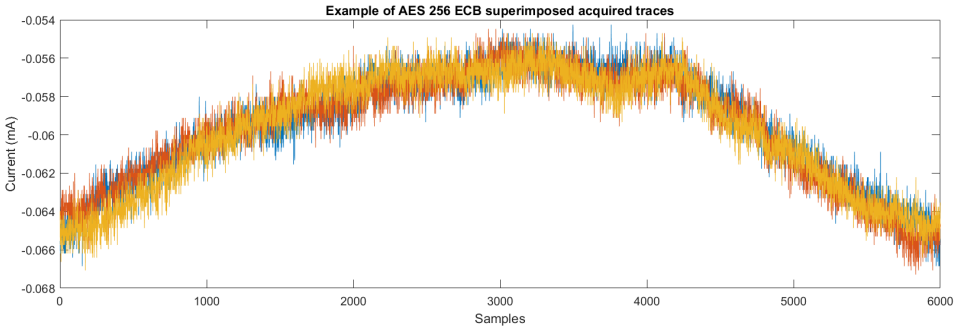

Figure 7.1: Superimposed traces evaluated with Matlab

As we can se from the picture above they are mostly aligned, but there are some differences due to the different operations of the microcontroller in distinct time instants.

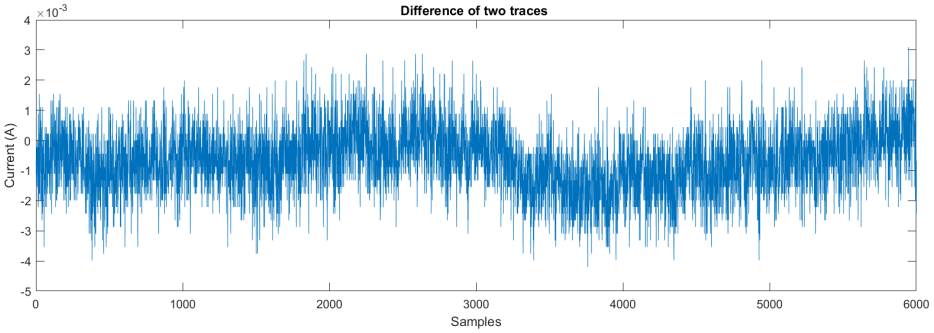

Figure 7.2: Difference between two traces evaluated with Matlab

Above, in fig 7.2, there is the difference between the traces of two encryptions for the same plaintext except for the last bit. We can see it differs from theory since noise and other interferences do not

make it appear "flat" with spikes: instead we have a wave similar to the plain wave but with enhanced spikes. This may be also due to slight misalignment which can be corrected via software in a future step.

7.2 Pysca

Pysca is a toolbox by Ilya Kizhvatov [12] designed to perform linear-regression analysis (LRA) based on DPA techniques and classical correlation power analysis (CPA), in particular targeting the AES S-box out. It required traces in .trs format, which is the format of a commercial tool named Riscure Inspector. Furthermore also the plaintexts are needed: since the implemented algorithm is CBC with a plaintext of 128 Kbyte (1 block only) but the tool has no options to select this particular version AES (most of the DPA tools work with ECB) I xored the actual plaintext with the IV to obtain a "corrected" input. The print on file of the plaintext and ciphertext at each step has been perfomed outside in another code in order not to waste speed and space of the microcontroller. Since the LeCroy oscilloscope provided the output traces in .dat, .bin or excel format I analized the Pysca code to convert the traces in a "trs like" format. Actually the toolbox has a built in converter that takes the .trs trace sets and convert them in numpy zipped (npz), which is a format containing a set of numpy (a python library) vectors.

```
#Dat_to_npz_converter

import argparse
import numpy as np
import struct
import Trace as trs

parser = argparse.ArgumentParser(description='Convert Inspector 4 traceset into numpy array')
parser.add_argument('-c', '--convertdata',  action='store_true', help='convert data from byte arra
to uint64 chunks')
parser.add_argument('filename', help='traceset file name without trs extension')

args = parser.parse_args()

samples=np.loadtxt(args.filename + ".dat", comments='#', delimiter=None, converters=None, skiprows
usecols=0, unpack=False, ndmin=0, encoding='bytes')

data_pre =[[232,224,222,162,250,205,250,170,239,207,140,20,231,1,240,216],
[165,2,125,19,106,43,111,40,133,116,245,250,157,161,227,119],
[107,241,81,90,54,121,111,59,4,107,90,70,89,98,5,112],
[171,223,123,10,106,123,85,12,29,246,66,202,230,206,61,35],
[61,179,81,30,103,167,200,35,139,215,91,88,240,118,72,167],
[56,25,125,189,62,30,214,84,63,31,68,18,41,52,239,177],
[18,217,209,232,91,63,47,181,170,1,241,80,60,56,10,106],
[80,175,128,87,85,165,112,102,234,114,183,210,66,122,85,92],
[53,227,145,82,149,189,168,52,195,210,186,137,96,175,....]

data_pre=data_clear[0:1023]
samples2=samples[0:1023]

traces = np.empty(shape=(1024, 25001), dtype = "float")
data = np.empty(shape=(1024, 16), dtype = "uint8")

for i in range(1023):
traces[i, :] = samples2[i]
```

```
data[i, :] = data_pre[i]

print "Saving file"
np.savez(args.filename, traces=traces, data=data)
print "Done"
```

The "1023" in the code above is the number of traces to be converted and it is manually set every time a new set of traces is transformed.

Pysca performs LRA and CPA and gives back each time the byte which is more likely to be that of the key, so it must be run 16 times in order to get the key for a AES-128 encryption.

At first I tried with a set of traces whose plaintexts were equal except for the last byte: this gave as a result a key composed of fifteen "0x00" bytes and a last byte different from zero. Then I ameliorate the code running on the microcontroller making each byte change (the 4096 vector version also reported in these pages), but still I did not get satisfying results. Pysca is designed for aes128, but I used in a first approximation to verify if I could obtain at least the first 16 bytes of the key. Since Pysca does not allow to do software alignment it is possible that a slight misalignment causes it to fail deciphering the key.

In a second moment I also tried Pysca with the AE128 traces. I used the .npy files created with Julia and created a .npz file directly with them. Since this set of traces and plaintext suffers even more from noise and flickering the key was not retrieved.

The attacks have been performed with these parameters:

Parameter	**Value**
sampleRange	(1, 1000)
N	4096
offset	5
evolutionStep	200
SboxNum	changing

Table 7.1: Pysca settings for the AES128 attack

The "sampleRange" setting is selected to identify the approximate first round of the ten total rounds, and "SboxNum" refers to the byte of the key under attack. The program is launched 15 times, each time focusing the attack on a different key byte.

The results of the attack are resumed in table 7.2. As we can see the results are not positive and the total distance of the retrieved key is: 58 . Which means 54.69 % correct.

In the figure below there's a plot Pysca makes each time it is launched.

- red trace is for known correct candidate
- blue trace is for the winning candidate (e.g. the one with maximum peak)
- grey traces are for all other candidates

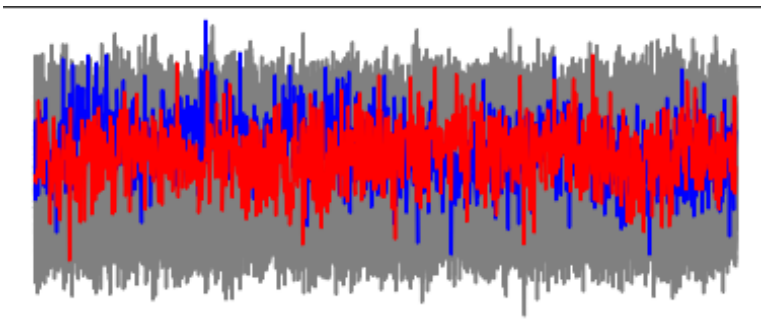

Figure 7.3: Correlations found in Pysca

Correct Byte	Retrieved Byte	H.W
00	a1	3
01	3d	4
02	15	3
03	e7	4
04	68	4
05	02	3
06	76	3
07	b6	4
08	16	4
09	d0	5
0a	5f	4
0b	50	5
0c	2a	3
0d	3b	4
0e	aa	3
0f	03	2

Table 7.2: Pysca key results compared with the correct ones

I also tried Pysca with the decryption process: I adapted the program running on the microcontroller and attacked it in decyphering mode on the first round. The recoveded key bytes were no correct as well.

7.3 Jlsca

Jlsca is another open source tool created by Cees-Bart Breunesse and Ilya Kizhvatov, who was the creator of the previously described Pysca. It supports several attacks, such as "Conditional averaging", for analog measurements, Correlation power analysis (CPA), non-profiled Linear regression analysis (LRA), AES128/192/256 encyrption, decryption, backward/forward S-box attacks and AES128 enc/dec chosen input MixColumn attack [8]. Jlsca and its toolbox are written in Julia and they take advantage of Python libraries and Python Notebook. It receives in input a file containing the plaintext, one for the ciphertext and of course one with the power traces. It supports .trs (Riscure Inspector) files or CWP (ChipWhisperer) format, so the first thing I did is to understand the shape of these vector files and created a converter.

```
using Jlsca.Trs
using DelimitedFiles
```

```
Name = "aes256cbc1round.dat"

numSamples = 2502
numTraces = 4096

samples=readdlm(Name)
datain=readdlm("outTOTcbc.dat")
dataout=readdlm("outTOTcbc.dat")
samples=samples'

# preallocate arrays
traces = Array{Float32}(undef, numTraces,numSamples)
textin = Array{UInt8}(undef, numTraces,16)
textout = Array{UInt8}(undef, numTraces,16)

n=1
for i in 1:numTraces

for j in 1:16
textin[i,j] = datain[1,n]
textout[i,j] =dataout[1,n]
n=n+1
end
end


# populate array forthe traces
for i in 1:numTraces
traces[i,:] = samples[2502*(i-1)+1:2502*(i-1)+2502]
end

(projectName, ) = splitext(Name)
exportCwp("$(projectName)-cwp", traces, textin, textout)
println("so far so good 4: the end")
```

This code produces a folder with the three files shaped as CWP requires.

In order to analyze the traces I mainly used as reference the "Still not Scary" Python notebook, which I modified according to my needs and parameters, which is provided in the Jlsca tutorials repository [9], which exploits a Python Notebook interface of the toolkit.

7.3.1 Main Steps

As a first thing after reading the files the Notebook provides a plot of few traces to see if they are sufficiently aligned.

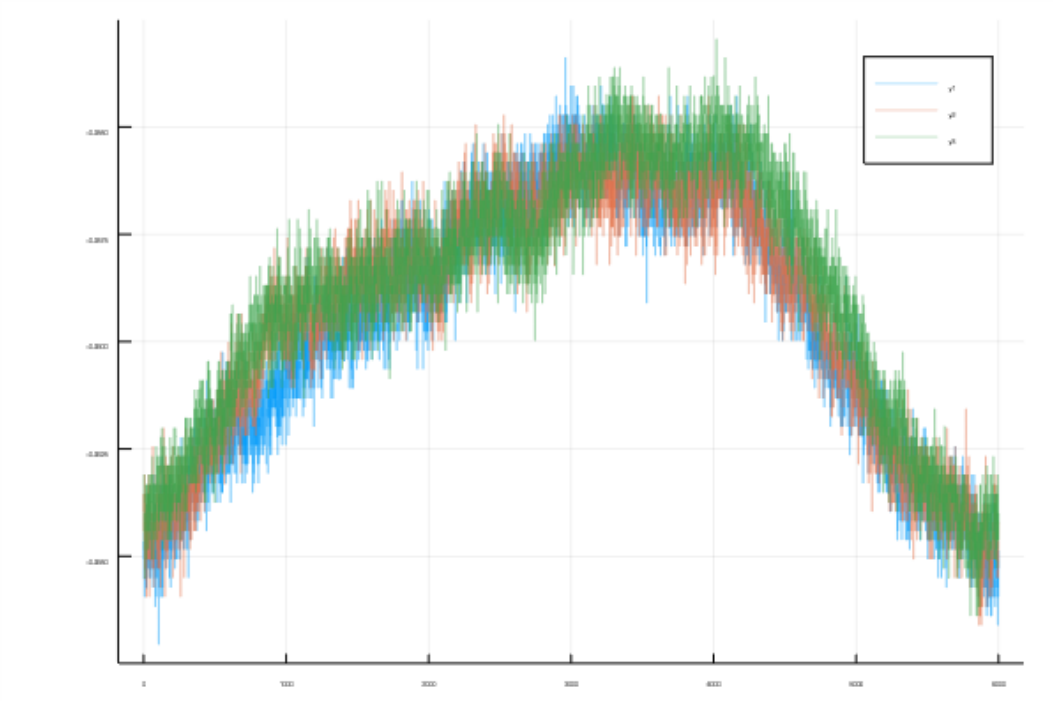

Figure 7.4: Jlsca plot of traces for ECB AES256 encryption

As we can see from 7.4 they are almost superimposed but not precisely, so in the next steps we can try to automatically adjust the samples in order to get maximum correlations.

```
# selecting the reference pattern in the first traces
referencestart = 1
referenceend = referencestart + 1000
reference = trs[1][2][referencestart:referenceend]

# in the search of alignment, traces will be shifted by max this amount of samples
maxShift = 2000

# the rejection threshold
corvalMin = 0.0

# create the alignment engine
alignstate = CorrelationAlignFFT(reference, referencestart, maxShift)
```

The rejection threshold allows the tool to discard some traces and samples if they can't be aligned.

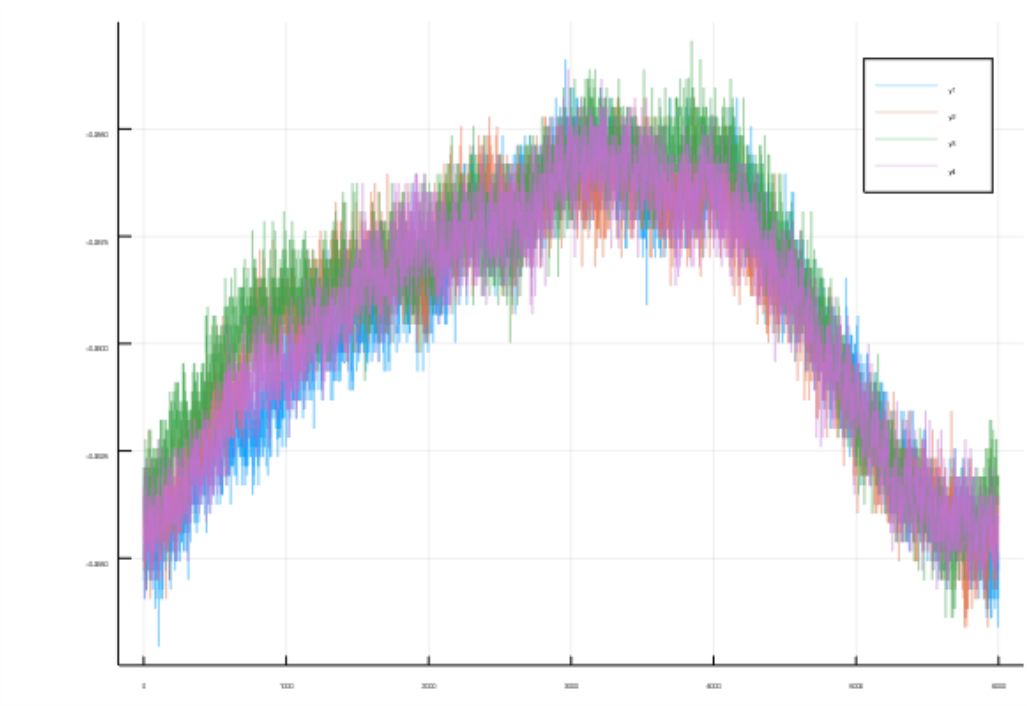

Figure 7.5: Jlsca plot of traces for ECB AES256 encryption after automatic alignment

In 7.5 there is the result after automatic alignment has been performed on the first 1000 samples. Then Jlsca runs the DPA/CPA attack according to the settings selected in the previous steps.

7.4 Encryption Modes

Here there are the waveforms for the different block cypher modes of operation. We can observe some differences just looking at the waveforms: the length of encryption of one block varies and also the shape of the power supply. Furthermore we can notice that the power supply shape seems to suffer from the switch on-off of the trigger led more than it does due to the encryption process.

Encryption and block cypher mode	**One block encryption time**
AES-256 CBC	67 µs
AES-256 ECB	60 µs
AES-256 CFB	64 µs
AES-128 ECB	52 µs

Table 7.3: One block encryption time comparison for the different AES encryptions

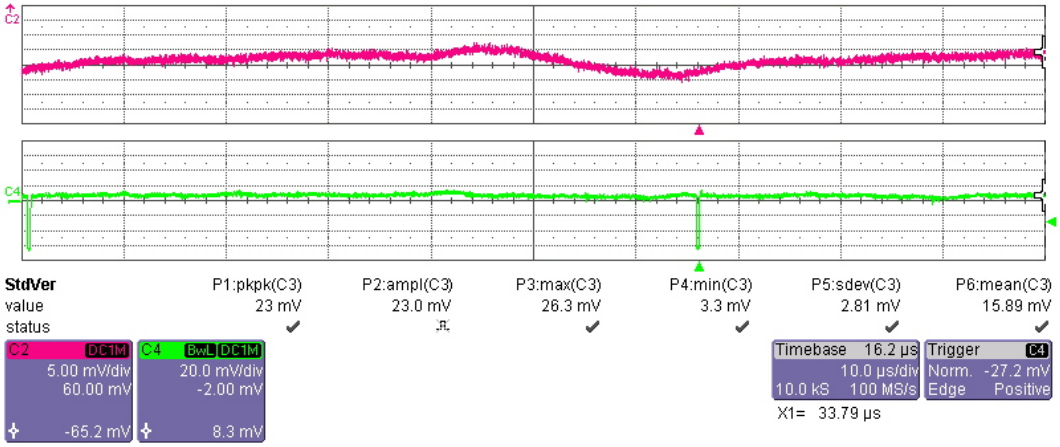

Figure 7.6: Oscilloscope trace for CBC AES256 encryption

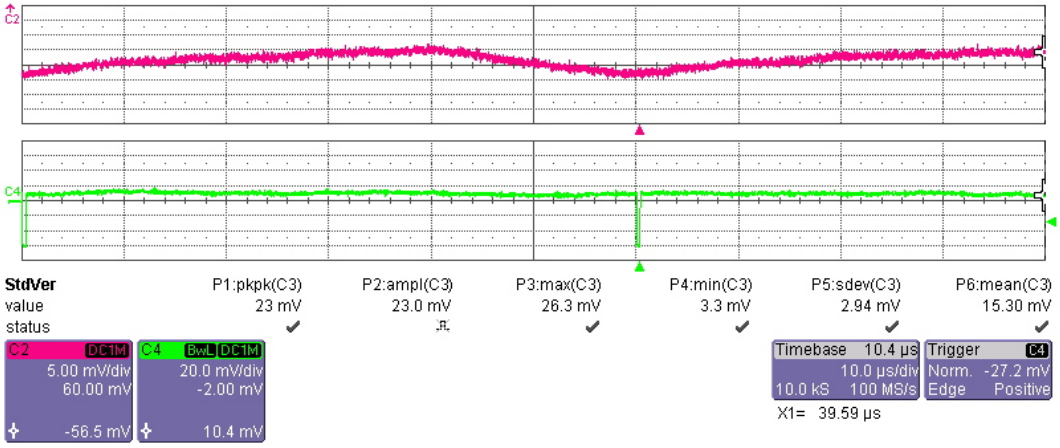

Figure 7.7: Oscilloscope trace for ECB AES256 encryption

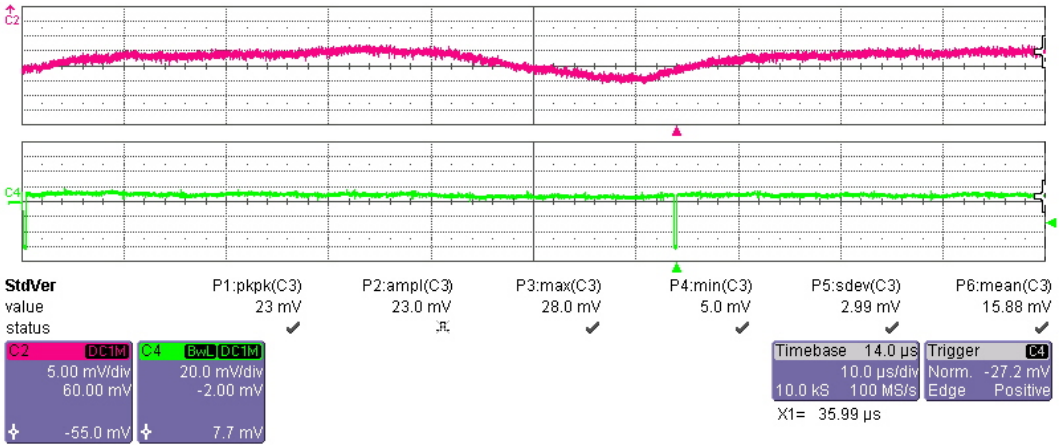

Figure 7.8: Oscilloscope trace for CFB AES256 encryption

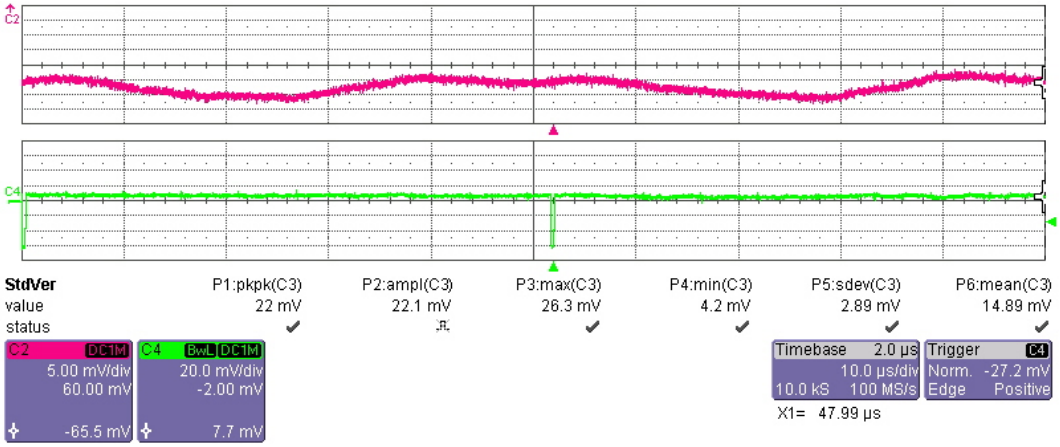

Figure 7.9: Oscilloscope trace for ECB AES128 encryption

7.5 Results AES 256 ECB

This analysis has been performed with 4096 traces and 10000 samples each.

As a first thing I run the "Piece of Cake" tool included in the Jlsca tutorials. The retrieved key is:

```
8c 19 05 86 8d ca d9 57 ed 99 de b2 03 27 c9 04 af 2f 04 2d 18 48
26 b3 bd eb dd ac d8 2b 57 17
```

Which has a hamming weight distance (the one used by Jlsca) of 123 bits from the correct key.

If we repeat the step of the previous paragraph we can select also second/third/fourth choice recovered bytes. The more similar to the correct key is:

```
2a 19 05 0e 29 32 06 0a 28 09 49 64 03 11 69 04 16 12 04 2d 18 04 26
 2b 3f 2a 22 11 03 16 34 1d
```

Here we can see there are two correct bytes at position seven and ten. The distance between this key and the actual one is now 95 bits, so we can observe an improvement.

The bytes in orange in table 7.5 are bytes present in the key but "recovered" in a wrong position, together with the settings of "Still Not Scary" script used.

DPA parameters	Value
attack	AES Sbox
mode	CIPHER
key length	KL256
direction	FORWARD
xor	false
analysis	Incremental CPA
leakages	HW
max shift	100

Table 7.4: DPA parameters for the attack

Settings	Recovered key
referencestart = 500 referenceend = 1500	c8 7f 03 6b 0b 4a d4 81 cc a2 51 9c a0 fb 26 3d de bd 00 9e 3b 4b 61 dd 3b 35 87 66 8e 5f d2 46
referencestart = 600 referenceend = 1600	56 20 33 ab 37 fe 28 54 11 fe ab b1 3f f9 a5 3d 63 df 0c e4 d0 5e bc 5a 88 ee 44 b7 4c 5d bb f7
referencestart = 700 referenceend = 1700	70 00 14 de a5 fb 03 94 8b 96 e1 9f d4 71 aa a1 7d ab 9c ae 31 0a 3b 5b 3e0310 27 38 e5 35 d8
referencestart = 800 referenceend = 1800	9d 10 49 de 55 2b 57 54 b9 a2 71 29 f8 77 18 cb eb 03 72 60 d8 ef 7c ca e9 56 b0 32 2c 72 ae 1d
referencestart = 900 referenceend = 1900	7c e5 94 fe ba 75 94 25 c4 e8 e4 b1 9b06 8c a0 08 5f d6 10 cc 88 28 59 aa 87 e9 e3 d1 56 44 90
referencestart = 1000 referenceend = 2000	b5 d4 40 94 62 e4 af 87 14 a6 71 cf 30 e7 91 3d a8 77 6b aa cd aa 6b 26 94 10 f1 a0 11 28 30 77
referencestart = 1100 referenceend = 2100	37 94 40 fe 8e 4a a6 06 64 37 16 9c17 b7 7d cd bd 88 44 73 af 8f 2e f3 e6 70 a4 07 f0 e3 a5 49
referencestart = 1200 referenceend = 2200	c7 26 41 fe ba 99 1e 60 fa 33 0f 66 ea f5 06 94 15 ab0c f1 40 d9 91 22 b2 5b cd 22 20 24 db 2a
referencestart = 1300 referenceend = 2300	b2 53 56 fe a3 c5 c4 ec 3a c4 ad 0a 92 8f 6f e5 7b 27 e9 51 1a 5f 97 d6 79 97 e8 90 7a db cc 3b

Table 7.5: Recovered key for AES-256 ECB encryption

As we can notice looking at the table above the recovered keys vary a lot depending on the reference start and its end. This is due to the fact that for an accurate analysis one should select exactly the wanted round (in this case, the first one). There is no clear distinction between the rounds, so I tried to recover the key in a reasonable range varying the bounds. It's still interesting to notice the correlations between recovered key bytes and actual ones, because in the proximity of the first round the incidences are higher (even though not enough to recover the actual key).

The same analysis can be performed on the last round, but that gave worse results. LRA gives result similar to CPA but the process is a lot slower.

In we evaluate the bit by bit distance between one of the recovered key and the correct one we find it is 127 for one of the "auto-aligned" case, one bit less than the distance given by selecting random numbers. For this case auto-alignment makes the result worse: indeed the traces should already be aligned due to the trigger.

If we repeat the analysis with LRA (Linear Regression Analysis, which is another approach to DPA) the retrieved key is:

```
e3 ec f2 d8 d2 20 7d e4 09 36 3a cd 0a 6f 6f fd 0b 86 7a ad 57 e1 c9 03 d7 6a 66 20 b4 e9 9a e5
```

Whose distance is 138 bits from the correct key, so these results are worse; furthermore, the analysis is slower.

7.6 Results AES 256 CBC

The same analysis performed for ECB was done for CBC.

As a first thing I run the "Piece of Cake" tool included in the Jlsca tutorials. The retrieved key is:

```
69 89 0c eb 6d 49 61 e5 d4 37 a4 5c b2 6e 54 92 11 68 db b0 f9 03 85 8b 2a 5a
 3d 31 fa b4 df 37
```

As we can see there are no correct bytes: in order to evaluate the bit by bit distance I converted the key vector in binary and compared it with the original key by means of a xor. The distance is 126 bits, that means 2 bits less than 128.

If we take also the 2^{nd},3^{rd},4^{th} and5^{th} most likely candidates we can arrange a key like this one:

```
69 89 0c 21 52 49 2e 30 2f 18 63 1c 02 1a 2e 2c 11 14 0f 2d 08 03 07 0f 11 5a
3d 1e 11 2a 1f 07
```

Which is more similar to the correct one: indeed the distance is only 94 bits.

DPA parameters	Value
attack	AES Sbox
mode	CIPHER
key length	KL256
direction	FORWARD
xor	false
analysis	Incremental CPA
leakages	HW

Table 7.6: DPA parameters for the attack

:

On a 10002 sample trace the first 6600 refers to our one block cypher process, according to the oscilloscope visual inspection with the help of the trigger. The rounds are not clearly distinguishable, so in order to get an idea where are the samples concerning the first round to be attacked I made a proportion dividing in 14 the interested samples.

Then I moved around this interval in order to find the most probable placement for the first round, but in every case we can notice the tool is very sample-dependant, meaning a shift of 10 evaluated samples changes (sometimes drastically) the computation.

Settings	Recovered key
referencestart = 1 referenceend = 500	e2 10 7a 66 b0 30 6f 5e 8b 84 9e 4b 0d f0 78 4d 83 ba 89 e8 b7 ff 19 35 82 c6 fc fb 5c 68 2b 08
referencestart = 100 referenceend = 570 corval min 0.2	e2 6d a2 66 11 4a 60 5e 76 53 3c b1 56 cf fe 55 e4 3f 35 6f 29 54 ad 9f b9 2c c0 21 c4 bb 78 65
referencestart = 100 referenceend = 570 corval min 0.0	f4 26 22 70 e8 5c 92 2b 64 a6 7b c4 64 71 02 11 3e cd e4 e1 4f 4b 80 5d ec 19 b4 48 49 e2 0e 7e
referencestart = 100 referenceend = 570 corval min 0.3	f4 26 22 70 e8 5c 92 2b 74 a6 7b c4 64 71 de 11 3e b7 e4 e1 21 e0 51 20 f0 d3 8a 6b 49 e2 0e cb
referencestart = 100 referenceend = 570 corval min 0.4	f4 26 22 70 e8 5c 92 2b 74 a6 7b c4 64 71 de 11 3e b7 e4 e1 21 e0 51 20 f0 d3 8a 6b 49 e2 0e cb
referencestart = 150 referenceend = 620 corv 0.4	2a 4a 30 ad 00 b1 6a 54 9b a2 cc 4b d3 d3 5f 55 33 4e 01 d9 84 84 5a 7d 81 ea 48 1c 10 95 ef 07
referencestart = 160 referenceend = 630	2a f4 dd 6b 7b 63 58 9b 64 6b 76 4b fd 64 02 ff da b8 61 7f 94 a5 f4 43 82 bc fb d9 0f da 55 14
referencestart = 170 referenceend = 640	9e b9 79 c0 b0 02 f3 69 74 a2 0a f4 56 02 a7 99 ca 7d 23 69 c3 7a f8 fd b9 8b 9f 59 bc 85 e2 86
referencestart = 180 referenceend = 650	a8 89 15 40 8c 91 bc 69 cf aa 16 8d 56 5d 3d cf dc 5b 54 8f 71 8a d9 f5 da f8 49 66 b0 c0 4b 16
referencestart = 100 referenceend = 472 corv 0.3	f4 26 29 66 2f 7d de b1 99 a2 27 ba 48 b1 d9 85 fb e3 8b b9 21 2b 56 b9 0e 7b c8 22 bf 0a bd ea
referencestart = 110 referenceend = 482 corv 0.3	8e 4b c5 94 3f 99 de a3 e4 84 16 e5 05 f0 ac 85 77 ee 59 f6 26 c0 7c f8 48 83 d9 5f 77 95 f4 4b
referencestart = 120 referenceend = 492 corv 0.3	8e 4b c5 94 3f 99 de a3 e4 84 16 e5 05 f0 ac 85 77 ee 59 f6 26 c0 7c f8 48 83 d9 5f 77 95 f4 4b
referencestart = 130 referenceend = 500 corv 0.3	8b 30 29 66 70 4a 9f 22 e2 84 16 40 7f 02 d5 6f eb 20 d8 20 2b e9 95 3b 8f b7 aa 8f bd 7a 54 28

Table 7.7: Jlsca outcome for a AES 256 analysis

When applying LRA analysis we get this result:

```
b7 a1 34 a3 3a 11 58 9e da 61 12 36 43 71 8d 03 69 ea 42 52 24
 60 b9 bb bf 43 63 ea 42 27 ac 0d
```

Whose distance is 126 from the correct key.

7.7 Results AES 256 CFB

AES 256-CFB applied for one block has no actual feedback, so it is not too different from ECB case. The plaintext is xored with the output of the Sbox to produce the ciphertext. The results are worse with respect to the ECB and CBC cases because the tool generally finds less correlations. Here's the result for the "Piece of Cake" script:

```
Retrieved key: 8c 19 05 86 8d ca d9 57 ed 99 de b2 03 27 c9 04 af 2f 04
2d 18 48 26 b3 bd eb dd ac d8 2b 57 17
```

The distance from the correct key is 123 bits.
The following is the key with the retrieved bytes from the 2^{nd} to the 5^{th} choices:

```
Retrieved key II: 2a 19 05 0e 29 32 06 0a 28 09 49 64 03 11 69 04 16
12 04 33 18 04 22 2b 3f 2a 22 11 03 16 34 1d
```

On the 7th and 10th position we get the correct byte and the distance is 92.
In the following lines there are some of the most interesting results with settings references.

```
maxShift = 500
referencestart = 100
referenceend = 500

target: 1, phase: 1, #candidates 256, "Sbox out, xor'ed w/ input"
rank:   5, candidate: 0x00, peak: 0.074556 @ 1823 -> correct

target: 12, phase: 1, #candidates 256, "Sbox out, xor'ed w/ input"
rank:   4, candidate: 0x09, peak: 0.080766 @ 1371 -> while it is 0a


maxShift = 500
referencestart = 100
referenceend = 600

target: 1, phase: 2, #candidates 256, "Sbox out, xor'ed w/ input"
rank:   4, candidate: 0x11, peak: 0.073947 @ 1955 -> while it is 10


target: 12, phase: 2, #candidates 256, "Sbox out, xor'ed w/ input"
rank:   4, candidate: 0x1b, peak: 0.073755 @ 7714 -> which is correct
```

If we repeat the analysis with Linear Regression Analysis the retrieved key is again:

```
e3 ec f2 d8 d2 20 7d e4 09 36 3a cd 0a 6f 6f fd 0b 86 7a ad 57 e1 c9 03 d7 6a 66 20 b4 e9 9a e5
```

Whose distance is 138 bits from the correct key.

7.8 Results AES 128 ECB

As far as the attack for AES 128 ECB is concerned we find more correlations with respect to AES 256 but still we cannot retrieve the full key.

Here's the retrieved key with no software alignment:

```
ef 64 fb d9 6b b3 28 6f 5e e6 54 6c ad b7 3e bf
```

Which differ 74 bits on a total of 128 bits, which is not interesting. And this is the one with also the other correlations:

```
b4 5d 06 9e 42 b3 28 1f 44 13 0d 0c 0b b7 3e 26
```

Which is 53 bits wrong compared to the correct one. So we have a $58\wedge$ of correct bits. These are some interesting results:

```
target: 11, phase: 1, #candidates 256, "Sbox out"
rank:   3, candidate: 0x0d, peak: 0.098824 @ 5146

target: 12, phase: 1, #candidates 256, "Sbox out"
rank:   2, candidate: 0x0c, peak: 0.112208 @ 5043

target: 13, phase: 1, #candidates 256, "Sbox out"
rank:   4, candidate: 0x0b, peak: 0.116519 @ 1370
```

This correlation looks peculiar because the correct sequence would have been: 0a, 0b, 0c. So the candidate for target 11 differs three from the actual one and the other two differ one. For these three consecutive bytes we have 5/24 bit wrong: that means a corrispondence of 80%.

Here there is reported an interesting piece of result for the "Still not Scary" modified script:

```
maxShift = 500
referencestart = 10
referenceend = 520


target: 2, phase: 1, #candidates 256, "Sbox out"
rank:   1, candidate: 0x03, peak: 0.161713 @ 5132 -> it should be 1


target: 3, phase: 1, #candidates 256, "Sbox out"
rank:   5, candidate: 0x06, peak: 0.156828 @ 1836 -> byte present, here it should be 2


target: 5, phase: 1, #candidates 256, "Sbox out" -> byte present, here it should be 4
rank:   5, candidate: 0x09, peak: 0.114908 @ 350


target: 7, phase: 1, #candidates 256, "Sbox out"
rank:   2, candidate: 0x07, peak: 0.076828 @ 7323 -> byte present, here it should be 6


target: 8, phase: 1, #candidates 256, "Sbox out"
rank:   5, candidate: 0x03, peak: 0.070111 @ 7174 -> byte present, here it should be 7

target: 11, phase: 1, #candidates 256, "Sbox out"
```

```
rank:   5, candidate: 0x08, peak: 0.099276 @ 6764 ->  byte present, here it should be a


target: 14, phase: 1, #candidates 256, "Sbox out"
rank:   4, candidate: 0x0d, peak: 0.166843 @ 4339 -> correct

maxShift = 1000
referencestart = 800
referenceend = 1000


target: 7, phase: 1, #candidates 256, "Sbox out"
rank:   3, candidate: 0x06, peak: 0.074441 @ 3255 ->correct

target: 10, phase: 1, #candidates 256, "Sbox out"
rank:   3, candidate: 0x0a, peak: 0.087192 @ 687  -> while it is 10

target: 12, phase: 1, #candidates 256, "Sbox out"
rank:   3, candidate: 0x0c, peak: 0.109326 @ 6330 -> while it is 0b
```

Also in this case the result varies a lot depending on the selected beginning and end of the round.

7.9 Results

HW distance (full key)	CBC	CFB	ECB
First choice	126	123	126
1,2,3,4,5 choices	94	92	94

Table 7.8: AES 256 result comparison on the full key, Hamming Weight

That means that the correct bits for the table above are:

HW distance (full key)	CBC	CFB	ECB
First choice	50.78 %	51.95 %	51.95 %
1,2,3,4,5 choices	63.28 %	62.89 %	64.06 %

Table 7.9: AES 256 result comparison on the full key, percentages

HW distance (full key)	CBC	CFB	ECB
Retrieved Key	126	138	138
Percentage	50.78 %	46.09 %	46.09 %

Table 7.10: AES 256 result comparison on the full key with LRA attack

As we can observe, LRA produces worse results with respect to CPA.

HW distance (full key)	AES 128
First choice	74
First choice %	42.19 %
1,2,3,4,5 choices	53
1,2,3,4,5 choices %	58.59 %

Table 7.11: AES 128 result comparison on the full key, Hamming Weight and percentages

7.10 Improvements: triggers to highlight only the first round

Since Jlsca tools seems to be really dependant on the presumed interval for the first (or last round) I tried to ameliorate the microcontroller program in order to have a more precise idea of where the first round is.

I placed the LED trigger only before and after the first round: in this way the beginning and the end of samples referring to the first round should be more precise. In order to do that I copied library functions in the main files so that they can borrow the functions for the microcontroller interface, namely the LED switch on and off.

In the following lines there is a piece of the code referring to the first round.

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

/* round 1: */
t0 = ctx->Te0[s0 >> 24] ^ ctx->Te1[(s1 >> 16) & 0xff] ^ ctx->Te2[(s2 >>  8) & 0xff] ^
 ctx->Te3[s3 & 0xff] ^ rk[ 4];
t1 = ctx->Te0[s1 >> 24] ^ ctx->Te1[(s2 >> 16) & 0xff] ^ ctx->Te2[(s3 >>  8) & 0xff] ^
 ctx->Te3[s0 & 0xff] ^ rk[ 5];
t2 = ctx->Te0[s2 >> 24] ^ ctx->Te1[(s3 >> 16) & 0xff] ^ ctx->Te2[(s0 >>  8) & 0xff] ^
 ctx->Te3[s1 & 0xff] ^ rk[ 6];
t3 = ctx->Te0[s3 >> 24] ^ ctx->Te1[(s0 >> 16) & 0xff] ^ ctx->Te2[(s1 >>  8) & 0xff] ^
 ctx->Te3[s2 & 0xff] ^ rk[ 7];

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
/* end of round 1: */
```

In order to get more samples and a higher accuracy for the following two cases I selected a sampling of 1 GS/s.

7.10.1 AES256 CBC Results

As we can se from figure tot the first round lasts 3.5 µs which is slightly less than 1/14 of the total one-block encryption process, which would be:

$$67\mu s/14 = 4.79\mu s \tag{7.1}$$

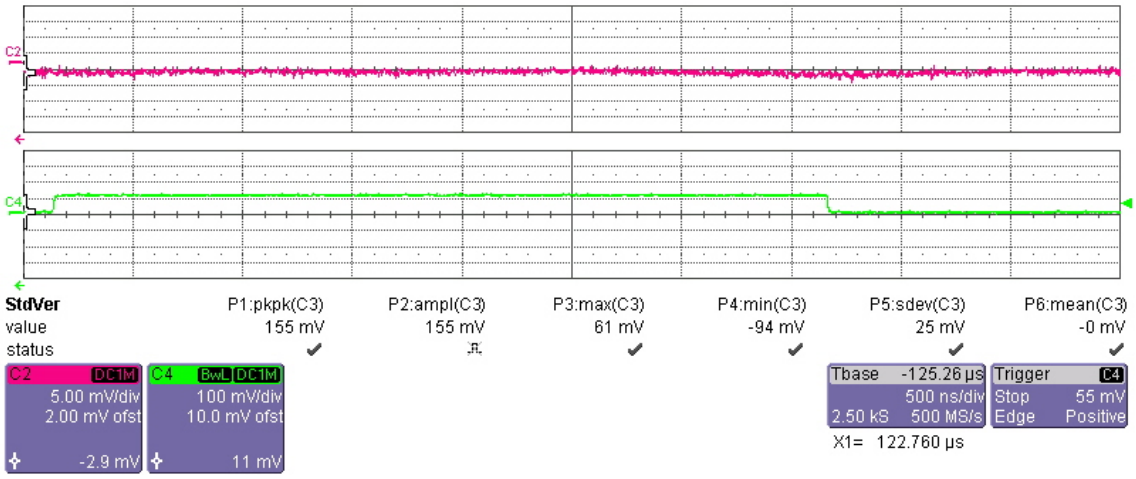

Figure 7.10: Trigger on the first round of AES-256 CBC

I tried to power the Nucleo both with DC power supply and USB cable to enlighten the differences between the two. The two traces below refers to the same conditions as far as sampling (1Gb/s) and number of samples is concerned.

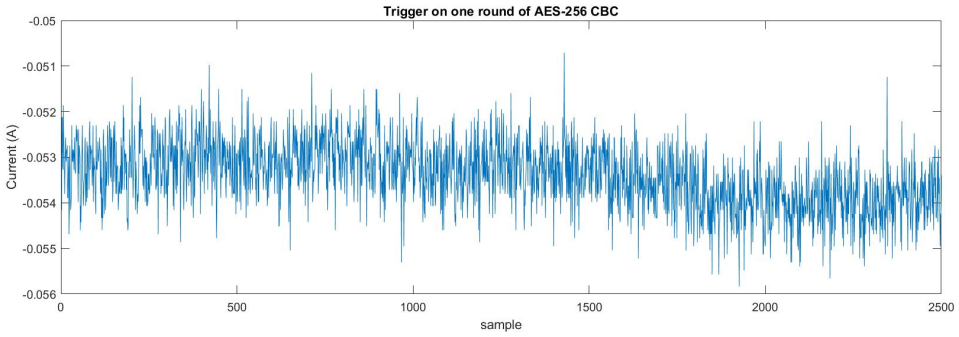

Figure 7.11: Trigger on the first round of AES-256 CBC, matlab evaluation

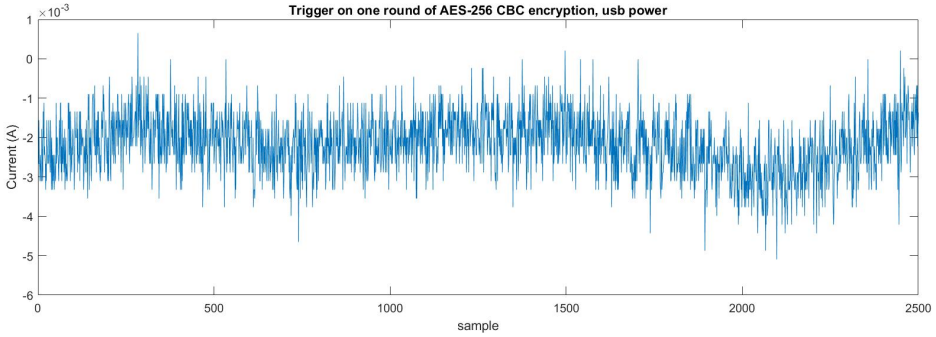

Figure 7.12: Trigger on the first round of AES-256 CBC, matlab evaluation, USB powered

As we can notice the current detected by the probe with the USB is less than 1/10 of the AC power supply mode. The first case appears more stable in time, while the second sometimes flickers. As in the previous cases we get some correlations between the actual key and the found bytes. It happens often to find correct byte correlations in the wring position. The example below refers to an attack

AES Sbox, CIPHER mode, Incremental CPA, raw data (software alignment non applied). Again data in orange is correct byte in a wrong position (let's recall that the key is: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f) Sometimes is interesting to see the distance between the recovered byte and the actual correct one: for example for target: 5, phase: 1, the candidate is 0x08, while the correct one is 0x04. For target: 16, phase: 2 at rank 5 we find the correct key byte: 0x1f.

```
target: 1, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x40, peak: 0.076331 @ 1813
rank:   2, candidate: 0x14, peak: 0.073827 @ 984
rank:   3, candidate: 0xe8, peak: 0.069951 @ 1540
rank:   4, candidate: 0x5c, peak: 0.068183 @ 1488
rank:   5, candidate: 0x3c, peak: 0.067658 @ 529
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 2, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x3b, peak: 0.066524 @ 525
rank:   2, candidate: 0x7a, peak: 0.066080 @ 392
rank:   3, candidate: 0x8d, peak: 0.064313 @ 2117
rank:   4, candidate: 0xc3, peak: 0.064127 @ 884
rank:   5, candidate: 0xe0, peak: 0.063592 @ 1569
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 3, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x0f, peak: 0.078546 @ 909
rank:   2, candidate: 0x22, peak: 0.069174 @ 1842
rank:   3, candidate: 0x65, peak: 0.068394 @ 2198
rank:   4, candidate: 0xd5, peak: 0.067086 @ 898
rank:   5, candidate: 0xc3, peak: 0.065177 @ 2073
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 4, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x72, peak: 0.080549 @ 1028
rank:   2, candidate: 0xe9, peak: 0.078617 @ 324
rank:   3, candidate: 0x4e, peak: 0.074003 @ 2187
rank:   4, candidate: 0xb5, peak: 0.071927 @ 948
rank:   5, candidate: 0xc5, peak: 0.071303 @ 1689
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 5, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x08, peak: 0.083292 @ 1234
rank:   2, candidate: 0x22, peak: 0.072705 @ 1836
rank:   3, candidate: 0x3c, peak: 0.071347 @ 1343
rank:   4, candidate: 0x60, peak: 0.070748 @ 1190
rank:   5, candidate: 0x68, peak: 0.070649 @ 924
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 6, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xeb, peak: 0.074876 @ 1258
rank:   2, candidate: 0x9c, peak: 0.073556 @ 253
rank:   3, candidate: 0x7d, peak: 0.063890 @ 659
rank:   4, candidate: 0xe4, peak: 0.063538 @ 859
rank:   5, candidate: 0x1f, peak: 0.063386 @ 430
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 7, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xd1, peak: 0.069580 @ 1790
rank:   2, candidate: 0x50, peak: 0.068700 @ 2189
rank:   3, candidate: 0x27, peak: 0.067378 @ 173
rank:   4, candidate: 0x7e, peak: 0.067253 @ 289
rank:   5, candidate: 0x34, peak: 0.066775 @ 1262
```

```
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 8, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x41, peak: 0.068454 @ 940
rank:   2, candidate: 0x91, peak: 0.067208 @ 1376
rank:   3, candidate: 0x3c, peak: 0.066144 @ 1790
rank:   4, candidate: 0x98, peak: 0.065850 @ 862
rank:   5, candidate: 0x55, peak: 0.065578 @ 1170
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 9, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x2a, peak: 0.070303 @ 1399
rank:   2, candidate: 0x87, peak: 0.067809 @ 426
rank:   3, candidate: 0xeb, peak: 0.067436 @ 381
rank:   4, candidate: 0xa1, peak: 0.066030 @ 19
rank:   5, candidate: 0xe5, peak: 0.064608 @ 386
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 10, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x24, peak: 0.074699 @ 609
rank:   2, candidate: 0x6b, peak: 0.071337 @ 1449
rank:   3, candidate: 0xb2, peak: 0.069252 @ 136
rank:   4, candidate: 0x33, peak: 0.067760 @ 1118
rank:   5, candidate: 0x57, peak: 0.067288 @ 1085
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 11, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xbd, peak: 0.068476 @ 1930
rank:   2, candidate: 0xb2, peak: 0.065253 @ 685
rank:   3, candidate: 0x25, peak: 0.063668 @ 1543
rank:   4, candidate: 0xae, peak: 0.062768 @ 731
rank:   5, candidate: 0x02, peak: 0.062503 @ 1884
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 12, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xce, peak: 0.071903 @ 437
rank:   2, candidate: 0x01, peak: 0.070497 @ 1415
rank:   3, candidate: 0x43, peak: 0.069378 @ 750
rank:   4, candidate: 0x38, peak: 0.066986 @ 31
rank:   5, candidate: 0x2d, peak: 0.066709 @ 641
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 13, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x5c, peak: 0.067198 @ 1637
rank:   2, candidate: 0xd5, peak: 0.065615 @ 1373
rank:   3, candidate: 0x38, peak: 0.064675 @ 2304
rank:   4, candidate: 0xc1, peak: 0.063604 @ 2363
rank:   5, candidate: 0x19, peak: 0.063188 @ 1005
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 14, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x9b, peak: 0.071603 @ 807
rank:   2, candidate: 0x00, peak: 0.069211 @ 849
rank:   3, candidate: 0xb0, peak: 0.067414 @ 1839
rank:   4, candidate: 0xd1, peak: 0.066919 @ 1798
rank:   5, candidate: 0x21, peak: 0.064934 @ 763
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 15, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xd8, peak: 0.072304 @ 1016
rank:   2, candidate: 0x8e, peak: 0.070994 @ 152
rank:   3, candidate: 0xf0, peak: 0.067461 @ 1086
```

```
rank:   4, candidate: 0x87, peak: 0.066384 @ 1287
rank:   5, candidate: 0xe1, peak: 0.064359 @ 1544
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 16, phase: 1, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xb3, peak: 0.070077 @ 876
rank:   2, candidate: 0x98, peak: 0.070002 @ 275
rank:   3, candidate: 0x1d, peak: 0.069569 @ 2039
rank:   4, candidate: 0x79, peak: 0.068650 @ 1122
rank:   5, candidate: 0x11, peak: 0.065661 @ 1973

target: 1, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xd8, peak: 0.078682 @ 131
rank:   2, candidate: 0xdb, peak: 0.070298 @ 705
rank:   3, candidate: 0xd9, peak: 0.067931 @ 1078
rank:   4, candidate: 0xf1, peak: 0.067675 @ 1354
rank:   5, candidate: 0xd6, peak: 0.066984 @ 695
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 2, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xa4, peak: 0.077450 @ 794
rank:   2, candidate: 0xcb, peak: 0.072137 @ 318
rank:   3, candidate: 0x77, peak: 0.071684 @ 1834
rank:   4, candidate: 0x30, peak: 0.069437 @ 189
rank:   5, candidate: 0xd1, peak: 0.068603 @ 2388
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 3, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x17, peak: 0.074362 @ 2471
rank:   2, candidate: 0x21, peak: 0.069385 @ 495
rank:   3, candidate: 0x4d, peak: 0.068242 @ 182
rank:   4, candidate: 0xb0, peak: 0.067735 @ 1770
rank:   5, candidate: 0xc7, peak: 0.067515 @ 397
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 4, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x85, peak: 0.080942 @ 1967
rank:   2, candidate: 0x0b, peak: 0.073288 @ 80
rank:   3, candidate: 0xb8, peak: 0.068727 @ 1331
rank:   4, candidate: 0x3d, peak: 0.065311 @ 2373
rank:   5, candidate: 0x45, peak: 0.065307 @ 325
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 5, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x83, peak: 0.071370 @ 1555
rank:   2, candidate: 0x0f, peak: 0.070817 @ 1227
rank:   3, candidate: 0xf4, peak: 0.070399 @ 318
rank:   4, candidate: 0x05, peak: 0.069980 @ 604
rank:   5, candidate: 0x2f, peak: 0.068993 @ 1020
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 6, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x7d, peak: 0.074512 @ 429
rank:   2, candidate: 0x5c, peak: 0.073216 @ 231
rank:   3, candidate: 0x33, peak: 0.071888 @ 290
rank:   4, candidate: 0x12, peak: 0.069266 @ 1741
rank:   5, candidate: 0xbd, peak: 0.067918 @ 285
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 7, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x90, peak: 0.075390 @ 2306
```

```
rank:   2, candidate: 0x2c, peak: 0.073455 @ 890
rank:   3, candidate: 0xb7, peak: 0.071578 @ 1234
rank:   4, candidate: 0x6c, peak: 0.069753 @ 652
rank:   5, candidate: 0xae, peak: 0.068757 @ 1603
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 8, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x58, peak: 0.075400 @ 89
rank:   2, candidate: 0xb8, peak: 0.072644 @ 1675
rank:   3, candidate: 0x93, peak: 0.070598 @ 277
rank:   4, candidate: 0xfd, peak: 0.067528 @ 523
rank:   5, candidate: 0x1f, peak: 0.067030 @ 423
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 9, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xd3, peak: 0.073313 @ 2248
rank:   2, candidate: 0x8b, peak: 0.072055 @ 570
rank:   3, candidate: 0xdd, peak: 0.066431 @ 478
rank:   4, candidate: 0xde, peak: 0.065555 @ 265
rank:   5, candidate: 0x32, peak: 0.065465 @ 1900
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 10, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xcc, peak: 0.068776 @ 1686
rank:   2, candidate: 0x8c, peak: 0.067763 @ 1950
rank:   3, candidate: 0x89, peak: 0.066524 @ 1349
rank:   4, candidate: 0xfa, peak: 0.065700 @ 1993
rank:   5, candidate: 0x90, peak: 0.062993 @ 1562
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 11, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xd3, peak: 0.076003 @ 2263
rank:   2, candidate: 0x66, peak: 0.073474 @ 236
rank:   3, candidate: 0x84, peak: 0.073393 @ 735
rank:   4, candidate: 0x57, peak: 0.073308 @ 2348
rank:   5, candidate: 0x98, peak: 0.070739 @ 2245
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 12, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x59, peak: 0.076616 @ 1830
rank:   2, candidate: 0xf4, peak: 0.072336 @ 1513
rank:   3, candidate: 0x15, peak: 0.071034 @ 496
rank:   4, candidate: 0xcb, peak: 0.069721 @ 993
rank:   5, candidate: 0x3f, peak: 0.068980 @ 261
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 13, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xaf, peak: 0.070363 @ 559
rank:   2, candidate: 0xd9, peak: 0.069397 @ 109
rank:   3, candidate: 0x40, peak: 0.068957 @ 555
rank:   4, candidate: 0x9b, peak: 0.066210 @ 333
rank:   5, candidate: 0xbd, peak: 0.066045 @ 474
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 14, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0xc8, peak: 0.074328 @ 347
rank:   2, candidate: 0x1a, peak: 0.070985 @ 2443
rank:   3, candidate: 0x27, peak: 0.067891 @ 271
rank:   4, candidate: 0x93, peak: 0.066847 @ 132
rank:   5, candidate: 0xf6, peak: 0.066496 @ 762
Results @ 4096 rows, 2502 cols (4096 rows consumed)
```

```
target: 15, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x18, peak: 0.067600 @ 687
rank:   2, candidate: 0xb7, peak: 0.066849 @ 143
rank:   3, candidate: 0xbb, peak: 0.065304 @ 1672
rank:   4, candidate: 0x9e, peak: 0.064509 @ 2338
rank:   5, candidate: 0x0b, peak: 0.063015 @ 1606
Results @ 4096 rows, 2502 cols (4096 rows consumed)
target: 16, phase: 2, #candidates 256, "Inverse sbox out"
rank:   1, candidate: 0x9e, peak: 0.070338 @ 753
rank:   2, candidate: 0x13, peak: 0.067659 @ 2178
rank:   3, candidate: 0xe9, peak: 0.067495 @ 2315
rank:   4, candidate: 0x4b, peak: 0.066682 @ 2344
rank:   5, candidate: 0x1f, peak: 0.066274 @ 1954
```

Similar partial results can be obtained with forward attack.

After having better aligned the traces the results improve a little bit, we obtain the same slight correlations as before but there are also more interesting results:

```
Attack Parameters:
attack:       AES Sbox
mode:         CIPHER
key length:   KL256
direction:    FORWARD


maxShift = 800
referencestart = 500
referenceend = 1000

target: 9, phase: 1, #candidates 256, "Sbox out"
rank:   1, candidate: 0x09, peak: 0.075995 @ 794 while it is 8

target: 1, phase: 2, #candidates 256, "Sbox out"
rank:   2, candidate: 0x13, peak: 0.066296 @ 1659 while it is 10


target: 7, phase: 2, #candidates 256, "Sbox out"
rank:   3, candidate: 0x16, peak: 0.068016 @ 1157   while it is 15

target: 14, phase: 2, #candidates 256, "Sbox out"
rank:   3, candidate: 0x1d, peak: 0.064449 @ 321   which is correct



maxShift = 800
referencestart = 500
referenceend = 1300


target: 4, phase: 1, #candidates 256, "Sbox out"
rank:   4, candidate: 0x02, peak: 0.069746 @ 1431 while it is 3

target: 14, phase: 1, #candidates 256, "Sbox out"
rank:   2, candidate: 0x0b, peak: 0.069724 @ 219  while it is 0d
```

```
target: 3, phase: 2, #candidates 256, "Sbox out"
rank:   1, candidate: 0x16, peak: 0.065731 @ 2089 while it is 13

target: 10, phase: 2, #candidates 256, "Sbox out"
rank:   1, candidate: 0x1a, peak: 0.072510 @ 2238  while it is 19


maxShift = 500
referencestart = 1000
referenceend = 2100

target: 1, phase: 1, #candidates 256, "Sbox out"
rank:   4, candidate: 0x00, peak: 0.066555 @ 914 which is correct

target: 3, phase: 2, #candidates 256, "Sbox out"
rank:   3, candidate: 0x11, peak: 0.069395 @ 623 while it is 12

target: 11, phase: 2, #candidates 256, "Sbox out"
rank:   5, candidate: 0x1c, peak: 0.065039 @ 1064 while it is 1a

target: 13, phase: 2, #candidates 256, "Sbox out"
rank:   5, candidate: 0x1c, peak: 0.066707 @ 1228 which is correct
```

Usually the traces referring to USB power supply gave worse results because of the probable non alignment between traces and input texts. On the other hand they are more stable in terms of flickering.

7.10.2 AES128 ECB Results

The same program of the previous case is adapted to perform AES 128.

As we can se from figure 7.13 the first round lasts 3.5 µs which is less than 1/10 of the total one-block encryption process, which would be:

$$52\mu s/14 = 5.2\mu s \tag{7.2}$$

Unfortunately it presents many drawbacks:

- Noise is an important component and the power trace flickers a lot. Sometimes captured traces are aligned in terms of samples but the DC component varies, so there are differences on the vertical axe.
- LED switching at a higher frequency partially hides encryption variations due to supply distortion.
- On one hand high sampling is required to detect useful power traces differences, on the other hand with a higher bandwidth it samples also more noise.

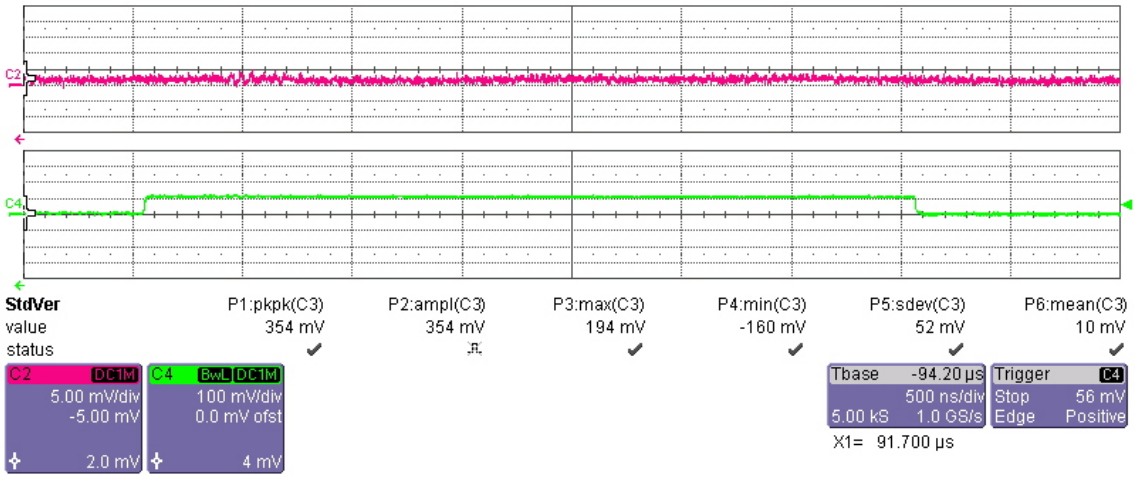

Figure 7.13: Trigger on the first round of AES-256 CBC

Due to these reasons the results are not so satisfying compared with those of the AES-256 case: we still get correlations but no significant improvements.

In 7.14 we can see the trace are not vertically superimposed:

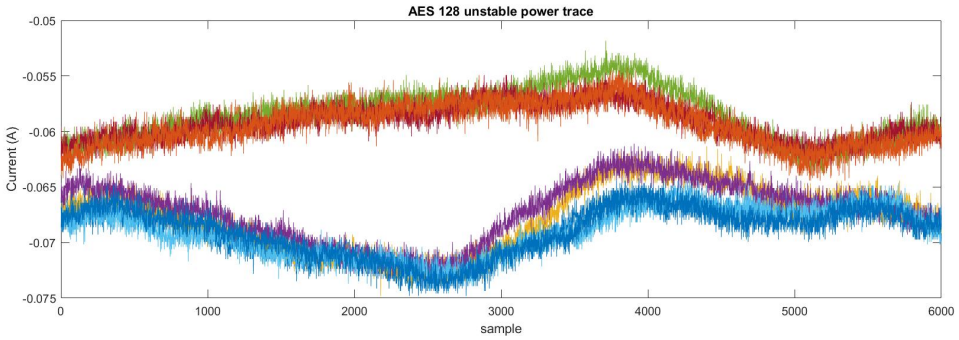

Figure 7.14: Trace with AES-128

7.11 Last round

Similarly to the previous case, where the trigger was surrounding the first round, I enclosed the last round with the trigger. Then I set the parameters to perform the analysis on the last round for AES-256 CBC.

DPA parameters	Value
attack	AES Sbox
mode	CIPHER
key length	KL256
direction	BACKWARD
xor	false
analysis	Incremental CPA
leakages	HW

Table 7.12: DPA parameters for the attack

No auto-alignment is performed.

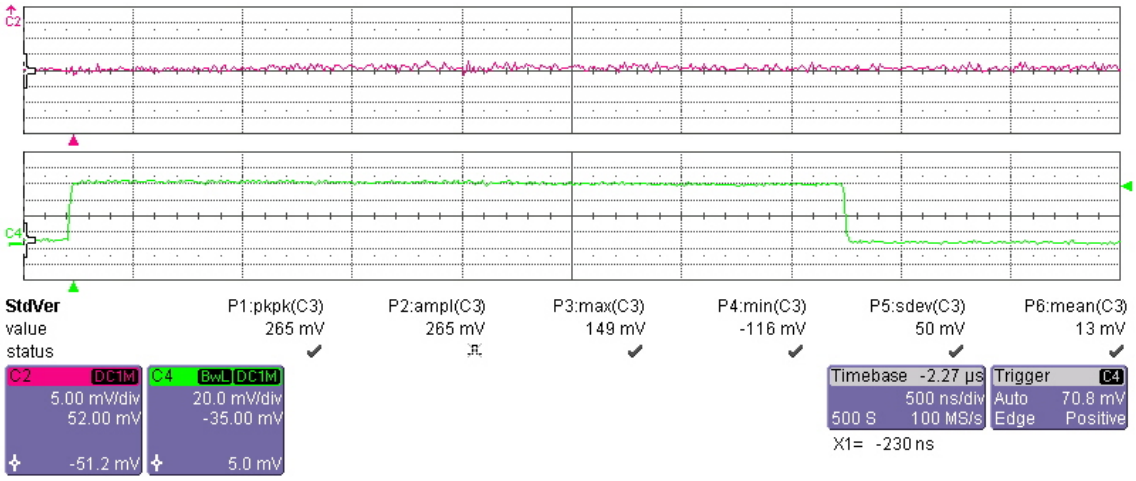

Figure 7.15: Trigger on last round of AES-256 CBC

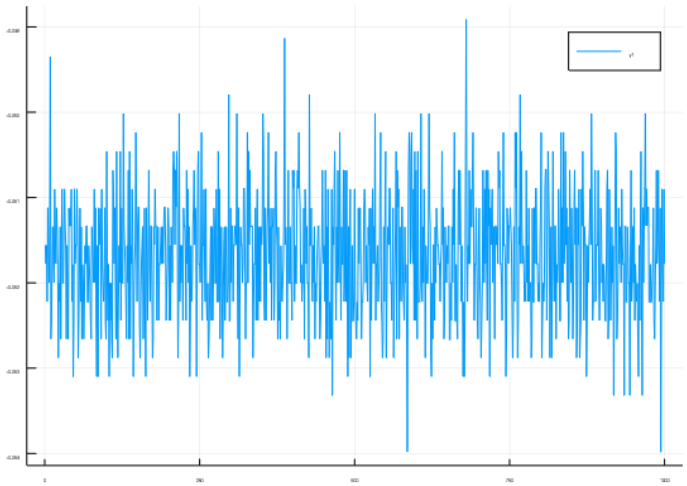

Figure 7.16: Julia plot of one trace, trigger on last round of AES-256 CBC

The retrieved key is:

```
89 6a ee 21 08 fb 36 ca 64 5f dc b6 23 05 5e e8 61 f7 3c a0 1e e8 ce a4 da fd a9 bf da e1 11 6f.
```

Which is far from the correct one. A comparison can be performed on the binary representation of the two, so that we can see the effective distance of the two vectors.

```
Found key= 10001001 01101010 11101110 00100001 00001000 11111011 00110110
 11001010 01100100 01011111 11011100 10110110 00100011 00000101 01011110
 11101000 01100001 11110111 00111100 10100000 00011110 11101000 11001110
 10100100  11111101 10101001 10111111 11011010 11100001 00010001 01101111.

Correct key= 00000 00001 00010 00011 00100 00101 00110 00111 01000 01001
01010 01011 01100 01101 01110 01111 10000 10001 10010 10011 10100 10101
10110 10111 11000 11001 11010 11011 11100 11101 11110 11111
```

Performing the xor of the two we get a total of 132/256 bits of difference between the two. Considering each bit can be 0 or 1 we should have a 50% possibility to guess each bit and so we expect something like 128 bits of difference if we try to guess the key without any analysis. No improvements with the DPA attack. We can conclude that attack on the last round looks a lot less effective than the one performed on the first round.

7.12 Fewer averaged traces, higher resolution

Since the noise component is affecting the result I tried another approach to solve this problem. In literature there are experiences of physical filters placed inside DPA acquisition systems. For these kind of measurements this is a risky operation since we cannot discriminate very well noise components from useful components in the power trace. In the post processing phase we can try to "filter out" noise averaging the traces.

For this study the program to be loaded on the microcontroller is AES-256 CBC with the last byte of the plaintext varying from 0 up to 255, so we focus the investigation only on a section of the encryption key (16th byte).

The first attempt is performed with sampling at 100 MS/s, and five sets of 256 traces are captured. Then they are then averaged in Julia to obtain one file only.

This is first evaluated by means of Jlsca.

The suggested bytes are:

```
target: 16, phase: 1, #candidates 256, "Sbox out"
rank:   1, candidate: 0x82, peak: 0.306443 -> distance 4
rank:   2, candidate: 0x8c, peak: 0.286547 -> distance 3
rank:   3, candidate: 0xa8, peak: 0.285264 -> distance 5
rank:   4, candidate: 0x51, peak: 0.280699 -> distance 5
rank:   5, candidate: 0x52, peak: 0.280012 -> distance 5
```

For a comparison, the detected byte for byte 16 of the key for AES-256 CBC is 5 bit distant, so we can see a slight improvement.

The second set of traces is captured at 1 GB sampling. I had four sets of 256 traces which I averaged to obtain one, each trace is 100002 samples long.

```
target: 16, phase: 1, #candidates 256, "Sbox out"
rank:   1, candidate: 0x48, peak: 0.319484 @ 33555  -> distance 4
rank:   2, candidate: 0xb6, peak: 0.318753 @ 37542  -> distance 5
rank:   3, candidate: 0xf1, peak: 0.314857 @ 53991  -> distance 7
rank:   4, candidate: 0x7e, peak: 0.313918 @ 57335  -> distance 4
rank:   5, candidate: 0xf5, peak: 0.307775 @ 59045  -> distance 6
```

The key byte is still wrong and there are no improvements notwithstanding the increased precision and averaging.

The next focus will be AES-128 in the same settings mode. The byte under attack is the same, the 16^{th}, which is the last byte of the 128 bit key. 5 sets of 256*100002 samples are acquired and averaged before Jlsca analysis. The outcome of the forward attack is the following:

```
target: 16, phase: 1, #candidates 256, "Sbox out"
rank:   1, candidate: 0xa4, peak: 0.312349 @ 68680 -> distance 5
rank:   2, candidate: 0x22, peak: 0.266270 @ 66597 -> distance 4
rank:   3, candidate: 0x91, peak: 0.260956 @ 69505 -> distance 5
rank:   4, candidate: 0x93, peak: 0.255759 @ 3646  -> distance 4
rank:   5, candidate: 0xe8, peak: 0.253886 @ 5777  -> distance 6
```

Which is worse in terms of distance compared with the previous analysis: this result can be due to the visible trace flickering. The last byte suggested by backward analysis is "20", which is 5 bit distant in therms of Hamming Weight. We can conclude that for AES 128 we see no improvements in terms of Hamming Weight Distance when applying averaging prior to trace analysis, while for AES 256 it can be an improvement.

7.13 Result comparison II

Encryption algorithm and method	Last byte (rank 1) H.W.
AES 256 CBC	5
AES 256 CBC higher sampling, averaged traces	4
AES 128 ECB	3
AES 128 ECB higher sampling, averaged traces	5

Table 7.13: Result comparison for the 16th byte retrieved

In table 7.13 there is the comparison between the last byte evaluated with a sampling of 1 GS/s with the 256 averaged traces and the ones with sampling at 25 MS/s, 4096 non averaged traces. As we can see the two results for the two cases are comparable: there is not much improvement from averaging. For the case of AES 128 the averaged result is even worse.

Chapter 8

Final Conclusions

8.1 Observations and Criticalities

The tools used for this analysis depend a lot on the actual beginning and end of the selected round, so a little imprecision there could visibly change the results. At the same time it looks like that capturing the power supply only of one round does not ameliorate the outcome. Furthermore, the led switch on and off causes variances of power higher than the encryption does; this was visible on the oscilloscope, since the power trace had bigger spikes in correspondence of the LED switching, and it might affect the result. For the same reason a faster led blink affects the result more deeply, since for example the traces referring to AES 128 encryption with the trigger around just one round were visibly less stable. On the other hand, there might be also problems on the oscilloscope side:

- During acquisition sometimes it buffers to save data on the USB and the image of the screen "freeze", so it might be possible that it loses some data. This occurred several times in between the capture of large data sets.
- Data alignment with traces: for the reason described above we could have also a non alignment between data in input/output and power traces. This problem may also occur due to the initial reset-state traces: with USB power supply they are non distinguishable from the block encryption traces.
- Traces alignment: although with oscilloscope trigger and software adjustment we can obtain a good result there might still be problems due to noise.

Although the minimum requirements for the oscilloscope according to [21] have been fulfilled:

- Bandwidth of at least 50% of the device clock rate for software implementations: the bandwidth of the LeCroy oscilloscope is 350 MHz and the Nucleo F401RE is running at 4 MHz.
- Capability to capture samples at 5x the bandwidth: the LeCroy can sample at 1 GS/s and 2.5 GS/s.
- Enough storage to acquire the entire signal required for the test and analysis: I used an external USB to have a larger storage available.

Active differential probes could not be because they were unavailable in the laboratory equipment, these may be useful to reduce input noise [21].

The conclusion of this thesis work is that nowadays DPA on microcontroller side can be used to recover partial results or reduce the possibility of keys, so that for AES 256 instead on 2^{256} possibilities we have less, but probably not for key recovery. Indeed the best results I got it is a 64% of correct occurrences obtained in 7.11. We have to note that this result is obtained taking the first five correspondences of the recovered results, and the 16 numbers are selected "by similarity" because I

knew the key. If I had to try all the possible choices in order to find that combination I had to try a total of:

$$totalcombinations = 5^{32} = 2.3283064 \cdot 10^{22} \tag{8.1}$$

To get that percentage of correctness, which is of course better than random numbers, but still far from the 100% correct bytes. So DPA it is still an interesting analysis to perform but with today's low power/low leakage microcontroller it is difficult to have good leakage power traces. Furthermore we have to consider the cost/time/benefit curve: the instrumentation I used it is expensive and with relatively high accuracy and the Nucleo device a standard, cheap microcontroller, so it is probably more leaky than other devices. I believe that given these assumptions and considering the number of traces analyzed it is no possible to retrieve better results.

If I had to compare the obtained results with a similar case I can see the results are comparable and sometimes there are some improvements comparing with the results obtained in [18]. They attacked the last byte of the key for a AES-256 encryption on a Nucleo equipped with ARM Cortex M4 low-power processor, and found a Hamming Weight between 5 and 3 and a Hamming Distance of 2 for a number of traces between 3000 and 5000. They showed sometimes the number of wrong bits decreases increasing the number of traces but sometimes they increase. In average, the found a distance of 4 and 5 for the Sbox AES and AES attack. It means that, extending this reasoning to the full key, the average wrong bits for a 32 byte key is between:

Extended wrong bits lower limit:

$$32 * 4 = 128 \tag{8.2}$$

Extended wrong bits higher limit:

$$32 * 5 = 160 \tag{8.3}$$

The results obtained in this thesis are slightly better, sometimes they are comparable (distance of 127, 126) but considering the special cases 7.8 the results are closer to the correct key.

The evaluation is interesting in terms of academic work but if we really had to break the key it would not be successful. In fact AES 256 is considered secure in most applications and widely used, and the whole analysis confirms it is.

Chapter 9

Appendix

```
/**
******************************************************************************
* @file           : main.c
* @brief          : Main program body
******************************************************************************
** This notice applies to any and all portions of this file
* that are not between comment pairs USER CODE BEGIN and
* USER CODE END. Other portions of this file, whether
* inserted by the user or by software development tools
* are owned by their respective copyright owners.
*
* COPYRIGHT(c) 2018 STMicroelectronics
*
* Redistribution and use in source and binary forms, with or without modification,
* are permitted provided that the following conditions are met:
*   1. Redistributions of source code must retain the above copyright notice,
*      this list of conditions and the following disclaimer.
*   2. Redistributions in binary form must reproduce the above copyright notice,
*      this list of conditions and the following disclaimer in the documentation
*      and/or other materials provided with the distribution.
*   3. Neither the name of STMicroelectronics nor the names of its contributors
*      may be used to endorse or promote products derived from this software
*      without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
* SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
* CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*
******************************************************************************
*/
/* Includes ------------------------------------------------------------------*/
```

```
#include "main.h"
#include "stm32f4xx_hal.h"


/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <stdlib.h>
#include "aes256.h"
/* USER CODE END Includes */

/* Private variables ---------------------------------------------------------*/
UART_HandleTypeDef huart2;
static GPIO_InitTypeDef  GPIO_InitStruct;


/* USER CODE BEGIN PV */
/* Private variables ---------------------------------------------------------*/

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);

/* USER CODE BEGIN PFP */
/* Private function prototypes -----------------------------------------------*/

/* USER CODE END PFP */

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
* @brief  The application entry point.
*
* @retval None
*/
int main(void)
{
/* USER CODE BEGIN 1 */

/* USER CODE END 1 */

/* MCU Configuration--------------------------------------------------------*/

/* Reset of all peripherals, Initializes the Flash interface and the Systick. */
HAL_Init();

/* USER CODE BEGIN Init */

/* USER CODE END Init */
```

```
/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART2_UART_Init();




/* -1- Enable GPIOA Clock (to be able to program the configuration registers) */
__HAL_RCC_GPIOA_CLK_ENABLE();

/* -2- Configure PA05 IO in output push-pull mode to
drive external LED */
GPIO_InitStruct.Pin = GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FAST;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);



//user code for the aes256
//////////AES Encryption
const uint8_t cipherKey[B5_AES_256]= {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b
0x0c,0x0d,0x0e,0x0f,0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,0x1b,0x1c,0x1d,
0x1e,0x1f};


uint8_t clrData[16]={0xbd,0x0f,0x0f,0xff,0x3f,0x2a,0xf4,0x51,0xb4,0xc7,0x00,0x00,
0xff,0x87,0x32,0x00};




int i=0, last=0;
B5_tAesCtx ctx; //aes context called ctx
int32_t init, setiv, update;


uint8_t encData[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; //encrypted data
int16_t nBlk=1 ; //number of blocks to be processed by AES.
//const uint8_t IV[B5_AES_IV_SIZE] = {0,1,1,2,3,2,1,1,0,0,0,0,0,0,6,5}; //init vector

//I initialize the struct
ctx.mode =  B5_AES256_ECB_ENC;
ctx.Nr   =  14 ;
```

```
while (1)
{

for(last=0; last< 255; last++ ){
//I change the last byte of the 128 bit block
clrData[0]=last;
//I make sure the led starts as _ON

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

// HAL_Delay(3000);
///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
// setiv=   B5_Aes256_SetIV(&ctx, IV);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

//I switch the pin Off when I'm finished

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);

// i do a glitch to stop the trigger
//         HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
//        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
// HAL_Delay(2000);
}


clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[1]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
```

```
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}


clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[2]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}

clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
```

```
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[3]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}


clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[4]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}


clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
```

```
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[5]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}


clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[6]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption
```

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}

clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[7]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}


clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){
```

```
clrData[8]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}

clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[9]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}

clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
```

```
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[10]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}



clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[11]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}
```

```
clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[12]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}


clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[13]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
```

```
///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}



clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[14]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}



clrData[0]=0xbd;
clrData[1]= 0x0f;
clrData[2]=0x0f;
clrData[3]=0xff;
clrData[4]=0x3f,
clrData[5]= 0x2;
clrData[6]=0xf4;
clrData[7]=0x51;
clrData[8]=0xb4;
clrData[9]=0xc7;
```

```
clrData[10]=0x00;
clrData[11]=0x00;
clrData[12]=0xff;
clrData[13]=0x87;
clrData[14]= 0x32;
clrData[15]= 0x00;

for(last=0; last< 255; last++ ){

clrData[15]=last;

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);

///AES Encryption
init=  B5_Aes256_Init(&ctx,cipherKey, B5_AES_256 , B5_AES256_ECB_ENC);
update= B5_Aes256_Update(&ctx, encData, clrData, nBlk);
//END of AES encryption

HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
}
}//questa è la parentesi del while1
}



/**
* @brief System Clock Configuration
* @retval None
*/
void SystemClock_Config(void)
{

RCC_OscInitTypeDef RCC_OscInitStruct;
RCC_ClkInitTypeDef RCC_ClkInitStruct;

/**Configure the main internal regulator output voltage
*/
__HAL_RCC_PWR_CLK_ENABLE();

__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE2);

/**Initializes the CPU, AHB and APB busses clocks
*/
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue = 16;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLM = 16;
RCC_OscInitStruct.PLL.PLLN = 336;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
RCC_OscInitStruct.PLL.PLLQ = 7;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
```

```
_Error_Handler(__FILE__, __LINE__);
}

/**Initializes the CPU, AHB and APB busses clocks
*/
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
|RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
{
_Error_Handler(__FILE__, __LINE__);
}

/**Configure the Systick interrupt time
*/
HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

/**Configure the Systick
*/
HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

/* SysTick_IRQn interrupt configuration */
HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
}

/* USART2 init function */
static void MX_USART2_UART_Init(void)
{

huart2.Instance = USART2;
huart2.Init.BaudRate = 115200;
huart2.Init.WordLength = UART_WORDLENGTH_8B;
huart2.Init.StopBits = UART_STOPBITS_1;
huart2.Init.Parity = UART_PARITY_NONE;
huart2.Init.Mode = UART_MODE_TX_RX;
huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart2.Init.OverSampling = UART_OVERSAMPLING_16;
if (HAL_UART_Init(&huart2) != HAL_OK)
{
_Error_Handler(__FILE__, __LINE__);
}

}

/** Configure pins as
* Analog
* Input
* Output
* EVENT_OUT
* EXTI
```

```
*/
static void MX_GPIO_Init(void)
{

GPIO_InitTypeDef GPIO_InitStruct;

/* GPIO Ports Clock Enable */
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOH_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOB_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);

/*Configure GPIO pin : B1_Pin */
GPIO_InitStruct.Pin = B1_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : LD2_Pin */
GPIO_InitStruct.Pin = LD2_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
* @brief  This function is executed in case of error occurrence.
* @param  file: The file name as string.
* @param  line: The line in file as a number.
* @retval None
*/
void _Error_Handler(char *file, int line)
{
/* USER CODE BEGIN Error_Handler_Debug */
/* User can add his own implementation to report the HAL error return state */
while(1)
{
}
/* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
* @brief  Reports the name of the source file and the source line number
*         where the assert_param error has occurred.
```

```
* @param  file: pointer to the source file name
* @param  line: assert_param error line source number
* @retval None
*/
void assert_failed(uint8_t* file, uint32_t line)
{
/* USER CODE BEGIN 6 */
/* User can add his own implementation to report the file name and line number,
tex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
/* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/**
* @}
*/

/**
* @}
*/

/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

Bibliography

[1] The advanced encryption standard (aes) algorithm. 3, 11, 13, 14

[2] Advanced_encryption_standard, wikipedia. 12, 13, 14

[3] Block cipher mode of operation, wikipedia. 3, 16, 17, 18, 19

[4] Correlation power analysis, wiki.newae. 30, 31

[5] Cryptology and data secrecy : The vernam cipher. 8

[6] *Differential Power Analysis of a McEliece Cryptosystem.* 6

[7] Extending aes-128 attacks to aes-256, wiki.newae.com. 21, 22

[8] Jlsca. 40

[9] Jlsca tutorials. 41

[10] The mathematics of the rsa public-key cryptosystem. 9

[11] Power analysis tutorial. 23

[12] Pysca. 38

[13] Side channel attack, wikipedia. 4

[14] *"Stream Cipher Reuse: A Graphic Example".* 17

[15] J. A. Ambrose, N. Aldon, A. Ignjatovic, and S. Parameswaran. Anatomy of differential power analysis for aes. In *2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 459–466, Sept 2008. 6

[16] Phillip Bellare, Mihir; Rogaway. *Introduction.* 7

[17] N. Benhadjyoussef, H. Mestiri, M. Machhout, and R. Tourki. Implementation of cpa analysis against aes design on fpga. In *2012 International Conference on Communications and Information Technology (ICCIT)*, pages 124–128, June 2012. 6

[18] M. Bollo, A. Carelli, S. Di Carlo, and P. Prinetto. Side-channel analysis of secube x2122; platform. In *2017 IEEE East-West Design Test Symposium (EWDTS)*, pages 1–5, Sept 2017. 3, 24, 25, 26, 65

[19] Martin Diffie, Whitfield; Hellman. *New Directions in Cryptography" (PDF).* 8

[20] Thomas Popp Elizabeth Oswald, Stefan Mangard. *Power Analysis Attacks - Revealing the Secrets of Smartcards.* 3, 26

[21] Josh Jaffe Pankaj Rohatgi Gilbert Goodwill, Benjamin Jun. A testing methodology for sidechannel resistance validation. 22, 23, 28, 64

[22] Josh Jaffe Pankaj Rohatgi: Cryptography Research Inc. Gilbert Goodwill, Benjamin Jun. A testing methodology for sidechannel resistance validation. 23

[23] Bruno ROUZEYRE Giorgio DI NATALE, Marie-Lise FLOTTES. An integrated validation environment for differential power analysis. 26, 28

[24] L. Guo, L. Wang, D. Liu, W. Shan, Z. Zhang, Q. Li, and J. Yu. A chosen - plaintext differential power analysis attack on hmac - sm3. pages 350–353, Dec 2015. 6

[25] David Kahn. *The Codebreakers.* 7

[26] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 5, 20, 25

[27] Jung-Hui Chiu1 Kuo-Tsang Huang1 and Sung-Shiou Shen. International journal of network security and its applications (ijnsa). 17

[28] Oswald E.-Standaert Mangard, S. All for one, one for all: Unifying univariate dpa attacks. 5

[29] Stefan Mangard1, Institute for Applied Information Processing Kai Schramm, and Communciations (IAIK). Pinpointing the side-channel leakage of masked aes hardware implementations. 6

[30] Milos DRUTAROVSKY Michal VARCHOLA. The differential power analysis laboratory setup. 23, 24

[31] Elisabeth Preneel Bart Ors, SB × Oswald. Power-analysis attacks on an fpga - first experimental results. 6

[32] M. Petrvalsky, M. Drutarovsky, and M. Varchola. Differential power analysis attack on arm based aes implementation without explicit synchronization. In *2014 24th International Conference Radioelektronika*, pages 1–4, April 2014. 6

[33] M. Petrvalsky, T. Richmond, M. Drutarovsky, P. L. Cayrel, and V. Fischer. Differential power analysis attack on the secure bit permutation in the mceliece cryptosystem. In *2016 26th International Conference Radioelektronika (RADIOELEKTRONIKA)*, pages 132–137, April 2016. 6

[34] Ronald L. Rivest. *Cryptography.* 7

[35] Claude Shannon. *Communication Theory of Secrecy Systems.* 9

[36] Simon Singh. *The Code Book.* 7

[37] M. Strachacki and S. Szczepanski. Implementation of aes algorithm resistant to differential power analysis. In *2008 15th IEEE International Conference on Electronics, Circuits and Systems*, pages 214–217, Aug 2008. 5

[38] Christopher Kevorkian Jacob Tanenbaum. Advanced cryptographic power analysis. 25, 26

[39] Jason Waddle, Marc Joye David Wagner, and Jean-Jacques Quisquater, editors. *Towards Efficient Second-Order Power Analysis.* 6

[40] John L. Smith Walter L. Tuchman William F. Ehrsam, Carl H. W. Meyer. *"Message verification and transmission error detection by block chaining".* 17