UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Countermeasure implementation and effectiveness analysis for AES resistance against side channel attacks

A Master's Thesis

Submitted to

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Asier Matias Zubeldia Otaegui

In partial fulfilment

of the requirements for the degree of

MASTER IN ELECTRONIC ENGINEERING

Advisors: David Hernández García & Isidro Martín García

Barcelona, June 2021

Title of the thesis: Countermeasure implementation and effectiveness analysis for AES resistance against side channel attacks

Author: Asier Matias Zubeldia Otaegui

Advisor: David Hernández García, Isidro Martín García

Abstract

Side channel analysis (SCA) is composed of a bunch of techniques employed to extract secret information from hardware operations through statistical analyses of execution data. For instance, the secret key of a crypto-algorithmic implementation could be targeted and its value could be retrieved. The data is obtained by measuring the power consumption or electromagnetic radiation of a device while performing an operation, due to the linear relationship between the currents flowing through the circuitry during the execution of chip operations. Side channel is one of the most widely used attack methods in cryptanalysis.

In order to avoid such attacks, the algorithmic implementations can be protected from side channel leakage with the use of different countermeasures. These countermeasures can be built on either software or hardware. The objective is to reduce, or even completely eliminate, the leakage of the device related to confidential data. Generally speaking, there are two main approaches to do so. The first aims to reduce the side channel observability, while the second intends to undermine the predictability of the data.

This project focuses on designing and implementing different countermeasures that protect cryptographic implementations from side channel attacks, and test and analyze them afterwards. The countermeasures will be implemented in software and then tested though Correlation Power Analysis in a hardware device.

The Advanced Encryption Standard (AES) algorithm will be used as a base structure, in order to improve its cryptographic security with the different countermeasures designed. However, the election of AES does not reduce the scope of this project since the implemented countermeasures could be applied to other cryptographic algorithms as well.

Acknowledgements

First of all, I would like to thank David Hernandez, advisor of this project, for all his help and support during the realization of this project. I would also like to thank the members of the IT department of Applus+ Laboratories.

In addition, I would like to thank the many friends from my years in UPV/EHU, who were part of my first steps in university. Thanks specially to Jon Martinez, who supported me and stayed next to me along the road.

Regarding this last years in the UPC, I would also like to thank Isidro Martín García, master coordinator, professor and advisor of this thesis, for his help and consideration. Additionally, I would like to thank Marina Martí, Romà Macario and Marcel Palets for the great moments we shared and for their support and patience as well.

Finally, I would like to thank my family, who have given me the opportunity to build my future , who have given me the means to grow as a man, who have showed me how to persevere in life and who have always stayed close to me.

Revision history and approval record

Revision	Date	Purpose
0	12/10/2021	Document creation

Written by: Asier Matias Zubeldia Otaegui		Reviewed and approved by: David Hernández García	
Date	12/10/2021	Date	13/10/2021
Name	Asier Matias Zubeldia Otaegui	Name	David Hernández García
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables

1. Introduction

1.1. Cybersecurity

Together with the growth of computer based systems over the last century, security has become an essential part of the modern electronic world. A world in which data is being constantly generated and processed for many purposes. Often, this data carries sensitive or confidential information and, thus, its protection is required. Payment information or personal identification data are some of the many examples of this sensitive information.

With the objective of securing this kind of information, the protection of the computer systems and networks is a must. This security scenario, known as **cybersecurity,** intends to protect users, systems and networks from the malicious intentions of an attacker willing to unveil confidential data.

Cybersecurity is an increasingly challenging field since the amount of interconnected devices is growing from day to day. The exchange of data is also spreading, which is translated into multiple new opportunities for malicious attackers. Consequently, hacking methods are being improved and attackers are becoming wiser on how to hack devices. In parallel, developers are also improving their defenses against cyberattacks.

For most of the people, cybersecurity is related to malicious malware known as “virus” or spam, or phishing though electronic mail. There has been some popular attacks along the last decades, driven to the massive filtering of sensitive user data from big servers; e.g. Yahoo (2013-2014), Facebook (2019) and LinkedIn (2021). However, the world of cybersecurity is much bigger than what is commonly thought. Cybersecurity includes hardware protection as well as software and network protection. For instance, additional related aspects to cybersecurity are the analyses of components and devices before reaching the market and the company departments for monitoring and responding to cyber-threats.

While the various security threats increase, cybersecurity is also in constant change and development in order to fight them.

1.2. Software, Hardware and Security

Regarding modern computing systems, three main fields of software cybersecurity shall be considered. **Network security** focuses on the attacks on a network connecting multiple computer systems, and the mechanisms to ensure its availability, usability and integrity under potential attacks. **Software security** focuses on malicious attacks on software applications and operating systems, often exploiting different implementation bugs such as inconsistent error handling and buffer overflows. In addition, techniques to ensure reliable software operation in presence of potential security risks are also a part of this field. **Information security** focuses on the general practice of providing, among many other security attributes, confidentiality and integrity of information through protection against unauthorized access, use, modification, or destruction.

Historically, data security has been an issue of paramount concern for system designers and end users. Consequently, protection of systems and networks against various forms of attacks, targeting corruption or leakage of critical information and unauthorized access, have been widely investigated over the years. Information security, primarily based on cryptographic measures, has been analyzed and deployed in a large variety of applications. Software attacks in computer systems have also been extensively analyzed, and many diverse solutions have been proposed.

Study of **hardware cybersecurity**, on the other hand, has attracted little attention as opposed to network or software security, due to the higher complexity of protecting the hardware and the difficulty of breaking systems with attacks against hardware devices. In comparison with hacking devices through e.g. a buffer overflow vulnerability, a hardware attack requires a lot more effort and resources, i.e. expensive equipment capable of manipulating the hardware circuitry of the devices. Nevertheless, it is a field inside cybersecurity that has been considered since the early '90s when hardware attacks were used to hack payTV systems.

Hardware security really became a trend when credit cards moved from magnetic stripe payment to chip payment. Based on chips specialized in security and authentication applications, these devices were considered impossible to crack until the first fault injection demonstrations in 1996.

Hence, hardware security focuses on attacks and protection of hardware itself. More accurately, it deals with the security of electronic hardware, encompassing its design, architecture, implementation, and validation. It forms the foundation of system security, providing trust anchor for other components of a system that closely interact with it. Hardware should enable a secure and reliable operation of the software stack. If the hardware is not secure the full system can be vulnerable.

Over the years, many hardware attacks have been crafted in order to steal or compromise sensitive information from implementations. The targets of these attacks, known as assets, are typically the secrets stored inside the hardware components, e.g. cryptographic keys, digital rights management (DRM) keys, sensitive user data, firmware code, configuration data etc.

In order to ensure that hardware devices are secure, a set of security requirements and testing specifications must be defined, implemented by product developers, and tested by cybersecurity evaluation Labs.

1.3. Security evaluations: White and black box approaches

With the objective of evaluating the security of a hardware, the evaluation Labs must consider all the applicable attacks, their associated vulnerabilities, the root causes for these vulnerabilities, and the countermeasures implemented by developers in their devices.

An evaluation can be "black box" or "white box" type. It is said to be a black box evaluation when the testing is made without any knowledge, or few knowledge, of what is happening inside the DUT (Device Under Test). An evaluation in these conditions is always complex and extensive, since the scope of security breaches that need to be covered is broad. A black box evaluation must consider every possible attack in order to assess the resistance of the DUT from all perspectives. In other words, if the Lab has no information about the DUT design, the only way to prove its resistance is by attacking.

On the contrary, when the evaluation Lab has access to the design and implementation features of the DUT, it is said to be a white box evaluation scenario. The developers can facilitate, for instance, hardware design code (Verilog/VHDL), schematics, firmware code, or application code. This way, the evaluators are able to accurately analyze the security architecture of the DUT and conduct a complete vulnerability assessment for all the assets against all kind of attacks, thus, limiting the testing campaign only to the identified vulnerabilities.

Normally, when performing an evaluation of a hardware device in a white box scenario, the next steps are followed:

- Firstly, a review of the hardware design and its security architecture is conducted.
- Based on it, a vulnerability assessment is carried out. This analysis identifies what vulnerabilities endanger any security attribute related to the assets.
- Next, an attack scenario is defined for each identified vulnerability. Depending on how feasible is to execute such attack, i.e. how easy is to exploit the vulnerability, the attack is executed or not. The attack is dismissed only when considered too complex for the given assurance level of the evaluation (some DUTs are evaluated in more depth and others with a more superficial assurance level).
- Lastly, attacks results are analyzed in order to give a verdict whether the DUT is vulnerable or resistant to the attacks.

Nowadays, security evaluations are always performed in a white box scenario due to the impracticability of performing the hundreds of existing hardware attacks in each evaluation. This scenario reduces the required effort for a security evaluation but, on the other hand, requires a deep knowledge on the effects of the countermeasures implemented on the DUT. In the end, the conclusions about the security of the DUT are based in the quality of the vulnerability assessment that the Lab performs.

As far as the development of this project is concerned, the implemented countermeasures were evaluated following a white box approach since both, developer and evaluator figures, were carried by myself. In these circumstances, a thorough analysis on the protective features of the countermeasures could be done. The flexibility of modifying the different implementations offered many points of view about what is happening when a developer introduces these countermeasures into their implementations.

1.4. Overview of hardware attacks

The main difference between software and hardware attacks is that when performing a hardware attack, the attacker needs to interact physically with the device. This interaction is done through hardware tools and equipment, leading to a much more costly task in comparison with software attacks.

For instance, hardware attack scenarios may require oscilloscopes, real-time pattern recognition devices, function generators, *xyz*-positioning stages, signal conditioning hardware, filters and data processing techniques, etc. All of these are specialized instruments, which require an accurate and precise performance.

The most influential hardware attacks nowadays are grouped into three main types: Physical Attacks, Side Channel Attacks and Fault Injection Attacks.

1.4.1. Physical Attacks

An attacker that performs physical attacks will substantially manipulate the device aiming to access its information. Reverse engineering is the first technique that falls into this group. The attacker analyzes in depth the chip layout from high resolution images taken with a Scanning Electron Microscope (SEM), after an etching process of the chip. The

analysis follows with the layer interconnections, the physical distribution of the memories, the analog and logic parts of the circuitry etc. If the reverse engineering is done well enough, an attacker could even illegally reproduce the device. In addition, the attacker can probe the data buses and observe the information that is traveling through them.

Another possibility is to introduce extra hardware to cause a malfunction of the device. This can be done by connecting some lines to ground or supply line in order to disable some security functionalities of the device. These tasks require high precision tools able to manipulate the chip's circuitry at a nanometer scale. For example, Focused Ion Beam (FIB) equipment is used to mill or deposit material. The milling of e.g. a power line will leave open circuit the power source of a hardware module, while the deposition of a conducting material between a circuit line and ground will generate a short-circuit permanently disconnecting the target module.

Alternatively, the attacker can focus directly on reading the memories from the high resolution images (SEM images). For instance, some ROM memories are often the easiest to read due to their physical construction that makes it possible to identify which cells are a '1' and which ones are a '0'. The simplest ROM cells can be read merely by observation of the physical presence or absence of a transistor in the cell. For this reason, it is recommended not to store critical information in these memories.

In general, Physical attacks are very powerful, allowing the physical tampering of the device and data in it. However, a great effort must be made and a deep knowledge on the chip is required in order to apply these techniques.

1.4.2. Fault Injection Attacks

Fault injection (FI) is an invasive attack technique where an operating device is perturbed in order to inject a fault along its normal execution. This fault intends to either introduce a temporary malfunction or modify certain data stored inside the hardware device.

When the objective is to produce a malfunction of the device, an attacker usually targets the CPU. As the device operation continues, the fault can propagate to other locations and can generate a faulty chain effect in the execution of the device. For example, the objective of injecting a fault could be to jump a line of the firmware code of a device. Considering the case of a credit card, if a fault is injected exactly when the PIN number is verified, an attacker could proceed to purchase something with a stolen credit card, without actually knowing the PIN.

Alternatively, a register or memory position can be targeted in order to modify its content, e.g. set a register all to ones ("1") or all to zeros ("0"). As an example, it is known that UV light can be used to erase EEPROM cells. Hence, a target could be the memory position that stores the secret key of an algorithm which, after attacked with UV light, will be forced to be set all to zeros. If achieved, the attacker would know the value of the key.

There are different energy sources that can be used to inject faults into a device. One of the most used fault injection sources are laser beams. The main advantage of the laser beam in comparison with other sources of perturbation is the ability to focus in very specific areas of the chip to inject the fault.

Electromagnetic fault injection (EM-FI) is another efficient source of perturbation. A coil that generates EM pulses can induce computational faults without any physical contact with the device.

Another fault source is the voltage glitching. In this case, a transient voltage spike causes the malfunction of the system. This glitch is coupled to the device's power supply and is enough to disrupt the normal cycle of a firmware execution, without causing permanent damage to the circuitry. This is one of the first sources of perturbation used to hack the old payTV cards.

1.4.3. Side Channel Attacks

Side channel analysis (SCA), unlike physical and fault injection attacks, is a non-invasive attack that uses statistical analyses of data in order to unveil sensitive information from a device. The analyzed data can be obtained through measurement of information that the device generates while operating, such as the variations of the power consumption, the signal propagation delay, and/or the electromagnetic emissions.

In order to apply SCA, an attacker first needs to acquire the data samples. For the power consumption analysis, a small resistor is usually placed between the power supply line and the cryptographic device in order to measure the voltage drop across it. In the case of EM radiation analysis, the signal is obtained using dedicated antennas.

Once the power or EM signal is available to the attacker, it can be taken into an oscilloscope in order to sample and record it. The attacker needs to gather big amounts of data, all proceeding from the same operation performed by the hardware DUT. Once this is done, the recorded data can be analyzed.

In order to infer useful information from the attack, the attacker will need to model the power consumption of the device and then apply statistics involving both measured and expected data. It is therefore, the relation between the measured data and the modelled data which establishes an analyzable and exploitable link for the attack.

Countermeasures against SCA can be implemented aiming to break that link or at least intending to debilitate this relationship between the power consumed and the data processed by the hardware device. Since this is the attack method chosen to conduct the countermeasure testing of this project, the topic will be more deeply addressed later in section 2.2 of this document.

1.5. Project definition

1.5.1. Motivation

Side cannel analysis is probably one of the most challenging attack methods nowadays. For this reason, it is widely-used against hardware devices. An increasing number of attack techniques are being developed and published every day and many crypto-algorithmic implementations are shown to be vulnerable to such attacks.

Together with the development of SCA, engineers are trying to introduce countermeasures into their implementations in order to compensate for their vulnerabilities. However, each countermeasure has a distinct protective effectivity, each of them introduces a different complexity level into the design, each of them requires a different memory allocation and each of them adds a different overhead into the firmware

execution time. In overall, every countermeasure has some positive and negative aspects and, hence, some of them can be more or less useful depending on the context. In order to be capable of providing assurance on the security of a device, a deep and thorough knowledge on the different aspects involving algorithmic implementations and its countermeasures is required.

For the various countermeasures that exist nowadays, gaining knowledge about their fundamental effects is essential, including their protective effectivity and other improvable facets.

1.5.2. Objectives, procedure and experimental set-up

The objectives of this project are to design, implement, test, analyze and compare firmware secure cryptographic countermeasures against side channel attacks.

The Advanced Encryption Standard (AES) algorithm is chosen as base structure to implement the countermeasures onto. The Correlation Power Analysis (CPA) is chosen as the side channel technique to test the various implementations developed. The results of the CPA are used as a metric to compare the effectiveness of the countermeasures.

For the development of this project, theoretical and practical works were conducted.

For the theoretical part, a brief description of the AES algorithm is given. Afterwards, SCA basics are introduced, in addition to the side channel attacks available nowadays. Lastly, an insight on side channel countermeasures is given to the reader.

For the practical part, the next experimental steps were followed:

- *Chipwhisperer* platform was chosen for the AES implementation and the development of the side channel countermeasures.
- A leakage assessment was carried out on an unprotected implementation of AES, where a CPA was performed. The results were used as reference point for future comparisons with protected implementations of AES.
- Different countermeasures were designed and implemented on the cryptographic algorithm.
    - Dummy round insertion
    - Shuffling
    - Random delay
    - Masking
- For each countermeasure implemented, a new leakage analysis was done in order to assess its protective effectivity.
- The different implementations were analyzed in terms added overhead (i.e. performance) with respect to the original unprotected AES.
- Results were gathered and compared.

2. State of the art

2.1. Advanced Encryption Standard

The AES symmetric algorithm was chosen as base implementation for the development of this project. The Rijindael cypher, later renamed as Advanced Encryption Standard, is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001 [1]. Two Belgian cryptographers, Vincent Rijmen and Joan Daemen, developed this algorithm that was used to replace the DES as the official encryption standard for protecting sensitive information.

The AES encryption finds applications in Mobile Phones, Smart Cards, Intel Core Processors Family, Automated Teller Machines (ATM), WWW servers, SSD Devices, IPSec and SSL Protocols, etc. It is massively spread and can be found in practically all security cryptosystems nowadays.

This cryptosystem is an iterative symmetric block cypher. It processes individual data blocks, having a fixed length of 128 bits, with a cipher key of variable lengths. The key length has to be chosen independently as 128, 192 or 256 bits. Hence, this algorithm can be used with three different key lengths, which result in three distinct formats referred to as AES-128, AES-192 and AES-256. It is an iterative cypher because the steps involved in this algorithm are repeated a fixed number of rounds The total number rounds of the cypher depends on the size of the key used. Table 1 shows the relation between key size and the total number of rounds for each AES format.

AES format	AES-128	AES-192	AES-256
Number of rounds	10	12	14

Table 1: Number of rounds for each AES format

In order to encrypt data, the 128-bit data block is divided into 16 bytes and correspondingly mapped into an array of size 4x4 known as the State matrix. All the internal operations (*SubBytes*, *ShifRows*, *MixColumns* and *AddRoundKey*), repeated round by round, are performed on the State matrix (Figure 1).

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

Figure 1. State matrix

When using AES, and also with every other symmetric algorithm, we look for an avalanche effect, where one single bit change influences as many output bits as possible, offering high diffusion and confusion to the message encryption. The diffusion and confusion of the message are obtained through the round internal operations of the AES, that perform permutations and substitutions on the state matrix.

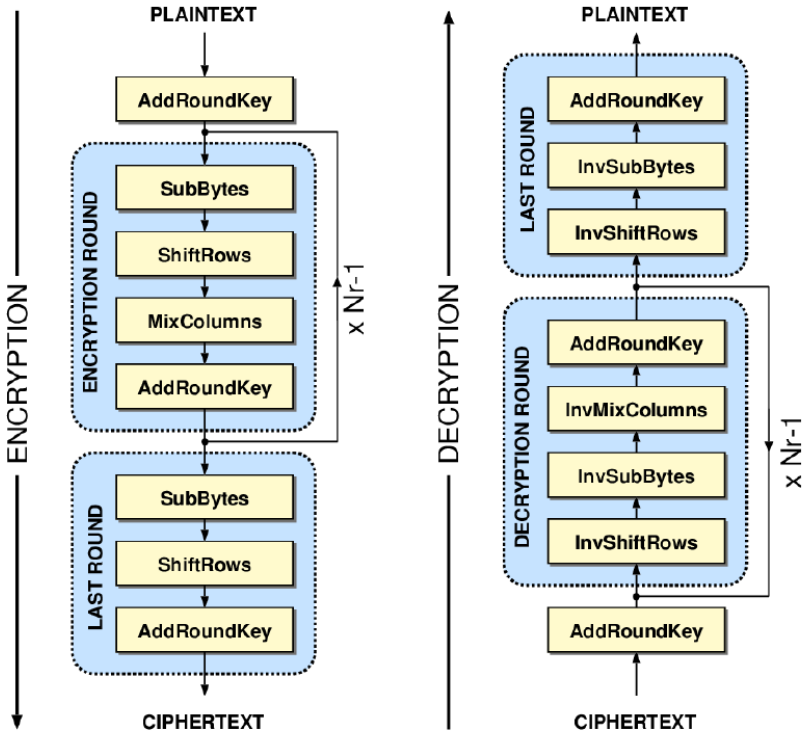

Figure 2. AES encryption and decryption

As shown in Figure 2, the algorithm starts with an initial *AddRoundKey* step. It is then followed by each of the rounds with the next internal transformations, in the following order: *SubBytes*, *ShifRows*, *MixColumns* and *AddRoundKey*. Note that, there is no *MixColumns* step in the last round. The same happens when decrypting that there is no *InvMixColumns* step in the last round. From a high-level perspective, each transformation step can be explained as:

- *SubBytes*: non-linear substitution step where each byte is replaced by another according to a LUT (Look-Up Table).
- *ShiftRows*: byte permutation step where the last three rows of the state are left-shifted cyclically a certain number of positions. The shifting applied is equal to the row number (from 0 to 3).
- *MixColumns*: linear algebraic mixing operation that operates on the columns of the state, combining the four bytes in each column through a multiplication with a constant matrix.
- *AddRoundKey*: recombination step where each byte of the state is merged with a byte of the round key through a bitwise XOR operation

In addition, AES uses a key schedule to expand the secret key into a number of separate round keys. The algorithm requires a separate 128-bit round key for each round plus one more, due to the initial *AddRoundKey* step. Thus, the key schedule produces all the required sub-keys from the initial cypher key.

As far as the decryption is concerned, all the transformation steps are reversed (*InvSubBytes, InvShiftRows and InvMixColumns)* in order to modify the ciphertext back to plaintext. Note that *AddRoundKey* is just an XOR, so it is its own inverse.

In case the reader was interested, a more detailed description of the AES algorithm and its internal operations is given in Appendix A, considering both mathematic and implementation aspects.

2.2. Side Channel Analysis (SCA)

2.2.1. Side Channel basics

Side channels are unintended sources of information that can be exploited by any attacker in order to extract secret data. Therefore, side-channel attacks take advantage of unexpected leakages of information. The designer of any implementation does not intentionally add this information sources into their design; however, the raw functionality of an electronic design carries, inevitably, analyzable side channels. For instance, an electronic circuit always generates a characteristic proportional to the power consumption and it always emanates electromagnetic radiation as the currents flow through the circuitry.

It is important to remark that unlike other attack methods, side channel analysis only requires to "listen" to the target device while it operates. We are talking about a non-invasive attack, where there is no need to influence or modify the target device in order to obtain information from it.

The simplest side channel attack technique is the timing attack. Timing attacks are based on the idea of analyzing different operations that take different computation times. Every logical operation in a computer takes certain time to execute, and these times can differ based on the inputs for example. In 1996, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems* [2] was one of the first side channel based attack ever published. Paul Kocher described the methodology to compromise keys of RSA, DSS and other cryptosystems, by measuring the execution time for the overall cryptographic operations.

In a similar way, sound can also be a useful side channel source. Acoustic emissions occur in coils and capacitors because of small movements when a current passes through them. Capacitors in particular change diameter slightly, generating sound, as their many layers experience electrostatic forces. One example is the first Side Channel attack performed in 1965 where the sound of a router was used to extract secret information [3].

Nevertheless, power consumption is the most widely used side channel source. As a general concept, it lays on the simple idea of some operations consuming more power than others. By measuring variations in the instantaneous power consumption of a device, it is possible to learn a considerable amount of information about the data being manipulated. For example, transistors, the most used elements in digital electronics, generate dynamic power consumption when changing from low to high state and vice-versa.

Therefore, power analysis attacks exploit the fact that the power consumption of a device depends on the operations it performs and on the data it processes. Usually, the total power consumption of the device is measured just by inserting a small resistor (1Ω - 50Ω) between the supply line and the cryptographic device. The voltage drop across this resistor is then proportional to the current that is flowing trough the device.

Electromagnetic radiation coming from the target device is another powerful side channel source. The analysis is equivalent to power consumption analysis with one main particularity: an EM prove (e.g. antenna) is used to receive the radiation, which can be placed accurately on the DUT surface (e.g. on top of the cryptographic co-processor to precisely acquire the targeted signal).

2.2.2. Power consumption and leakage models

In CMOS circuits, the power consumption is derived from two parts, the static and dynamic power consumption. In the context of side channel attacks, the latter is generally the main source of exploitable power consumption as it is both operation and data dependent. It can be calculated as:

$$P_{dyn} = \alpha \cdot C \cdot {V_{dd}}^2 \cdot f \quad (1)$$

The parameter α is the switching factor, C is the load capacitance, V_{dd} is the supply voltage and f denotes the clock frequency.

Since the power consumption at a given point in time is related to the number of transistors that change state, it is also related to the data being processed. If the same point in time over many acquisitions of the same operation is targeted, any operation dependent power consumption can be viewed to be part of the static power consumption and, thus, can be ignored.

Any side channel attack proceeds by using some statistical distinguisher, such as Pearson's correlation coefficient, to compare the hypothetical leakage model and the acquired power traces. A power trace refers to a set of power consumption measurements $s(t)$ taken across the target operation. Considering this, the leakage model is constructed to estimate the power consumption of algorithmic intermediate values during the operation of a cryptosystem. In other words, the leakage model is a simplified model that describes the leaking signal (power consumption or EM radiation) in a workable manner, e.g.:

$$s(t) = f(algorithm, data, time) \quad (2)$$

The simpler the model, the easier to work with, but more distant to reality. When enough traces are available, and the hypothetical leakage model is accurate enough, the secret can be retrieved from the acquired power consumption. Therefore, the leakage model is required to approximate the actual power consumption as much as possible. When the acquisition of traces is limited by available equipment and limited access to the target device, a well-built leakage model can significantly enhance the performance of the attack method.

In the case of a DPA attack, the simplest of the models is considered: it is a single bit model that relies on the elementary idea of ones ('1') consuming more than zeros ('0'). However, the most commonly used models to estimate the power consumption are the linear models of Hamming weight and Hamming distance.

The Hamming weight model corresponds to the number of bits set to 1 in a binary data element.

$$B = \{b_m, b_{m-1}, b_{m-2}, b_{m-3}, , \dots, b_0\}_2 \rightarrow Hw(B) \in [0, m] \quad (3)$$

For example, bytes considered, all the possible resulting Hamming weights range from 0 to 8. The following equations show two byte examples and their related Hamming weight values:

$$B_1 = \{10000100\}_2 \rightarrow Hw(B_1) = 2 \quad (4)$$

$$B_2 = \{10110001\}_2 \rightarrow Hw(B_2) = 4 \quad (5)$$

Therefore, the Hamming weight model states a linear relationship between the power consumption and the number of ones travelling through the circuitry:

$$s(t) = a(t) \cdot Hw(B) + b \quad (6)$$

s denotes our power estimate, while a is the scalar gain between the Hamming weight and s. All the remaining aspects in the power consumption of a chip are assigned to a term denoted b which is assumed independent from the other variables. b encloses offsets, time dependent components and noise (typically Gaussian noise).

However, the Hamming weight model has a limitation. As stated in [4]: "*It is generally assumed that the data leakage through the power side channel depends on the number of bits switching from one state to the other at a given time. This seems relevant when looking at a logical elementary gate as implemented in CMOS technology. The current consumed is related to the energy required to flip the bits from one state to the next*" [4].

Consequently, it makes sense to define a leakage model as the Hamming distance. The basic power consumption model for the data dependency can be written as:

$$s(t) = a(t) \cdot Hd(R \oplus M) + b \quad (7)$$

This model represents the transition from a reference state R to a modified state M, where some bits, or all bits, have already been flipped. It is assumed that the switching of a bit from 0 to 1 or from 1 to 0 requires the same amount of energy and that all the machine bits handled at a given time are perfectly balanced and consume the same. This can be seen as a limitation, but considering a chip as a large set of elementary electrical components, this linear model fits reality quite well from a statistical point of view.

The Hamming distance can be easily calculated as the Hamming weight of the XOR-ed values of both bytes, i.e. the reference byte and the modified one. Applied to the example bytes of equations (4) and (5) and considering them, respectively, the reference and modified states:

$$Hd(B_1, B_2) = Hw(B_1 \oplus B_2) = 4 \quad (8)$$

Looking at the equation (8), it is easy to deduce that the Hamming weight is just a particular case of the Hamming distance where the reference state is set to 0.

In many cases, the Hamming weight or Hamming distance models will not be an optimal leakage model for a given device. However, both linear models do provide an easily computable and robust approximation of the leakage, which is applicable in a wide range of scenarios.

Non-linear models have also been widely investigated, trying to improve the approximation of the actual leakage. In [4], the authors proposed a switching distance leakage model to improve the attack performance. In their work they suggest a model where the transistor switching from 1 to 0 consumes less power than from 0 to 1, i.e. different power is consumed in the charging and discharging phases. The experimental results showed that the attack performance is improved for a particular setup; however, some sort of profiling of the specific DUT is required.

In summary, a leakage model is always assumed in side channel attacks in order to estimate the relationship between the signal acquired with the oscilloscope and the data processed by the device. If the leakage model is accurate to reality, as Hamming weight and distance models are, we can find a linear relationship between the data processed by the device (sensitive data such as secret keys) and the power traces acquired.

2.2.3. Side Channel attacks

Historically, many powerful side channel attacks are based on statistical methods pioneered by Paul Kocher. In 1995 Paul Kocher stated: "*Chip power consumption is somehow clearly linked to the manipulated data*". Later in 1999, an efficient side channel attack was introduced by himself: *Differential Power Analysis* [5]. The power consumption turned out to include deterministic data dependent parts, which could be exploited by simple and differential power analysis. Later, Eric Brier published *Correlation Power Analysis with a Leakage Model* [6], where correlations between power measurements and data were used to improve the previously introduced differential attacks through the Hamming distance leakage model.

As an evolution of first order attacks, in which the power consumption is analyzed in a time-independent manner, the SCA attacks took a more general perspective that resulted in higher order SCA, as originally proposed by Messerges [9] and Chari et al. [10]. Higher order SCA considers various time instants within a power trace to combine and build more complex leakage models.

Finally, one of the most powerful SCA techniques are the profiling attacks, in which the power consumption of a "twin" device is characterized creating templates that are later used against the victim's device to extract sensitive data. Besides Template Attacks and its variants [11 ,12, 13], the SCA community started using machine learning (e.g. Artificial Neural Networks) to conduct profiling attacks [14] [15].

2.2.3.1. Simple Power Analysis (SPA)

SPA, known as SEMA in the case of electromagnetic radiation analysis, is the side channel analysis that involves the visual inspection of one, or a few number, of power traces. SPA is based on the identification of recognizable patterns, which may correspond to the target operation to be analyzed (e.g. crypto-operation or the load of an asset from memory). The objective is to identify instructions, security mechanisms or countermeasures, or directly read sensitive data as proposed by Kocher in his original work.

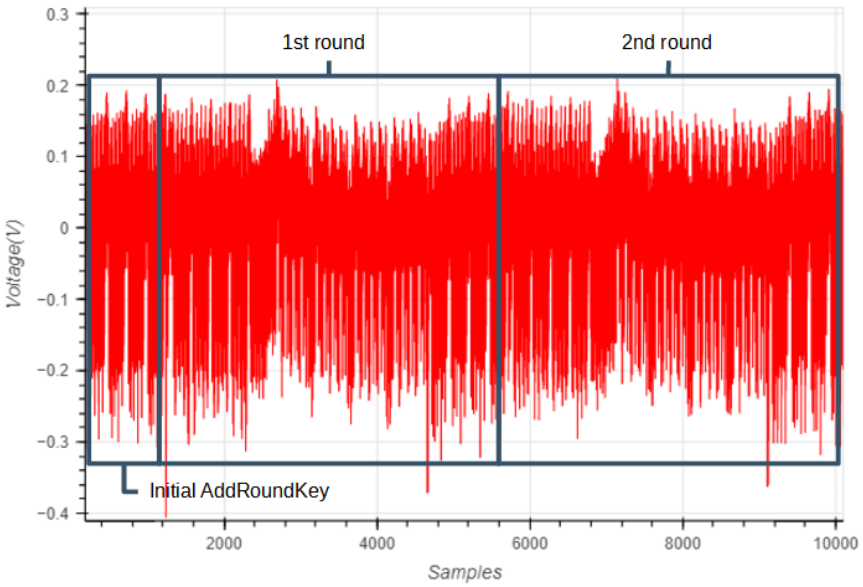

Figure 3. SPA on AES encryption

Figure 3 shows an example of SPA. The region of the initial *AddRoundKey* step in AES can be seen in the signal, followed by the power consumption of the first and second rounds of the algorithm.

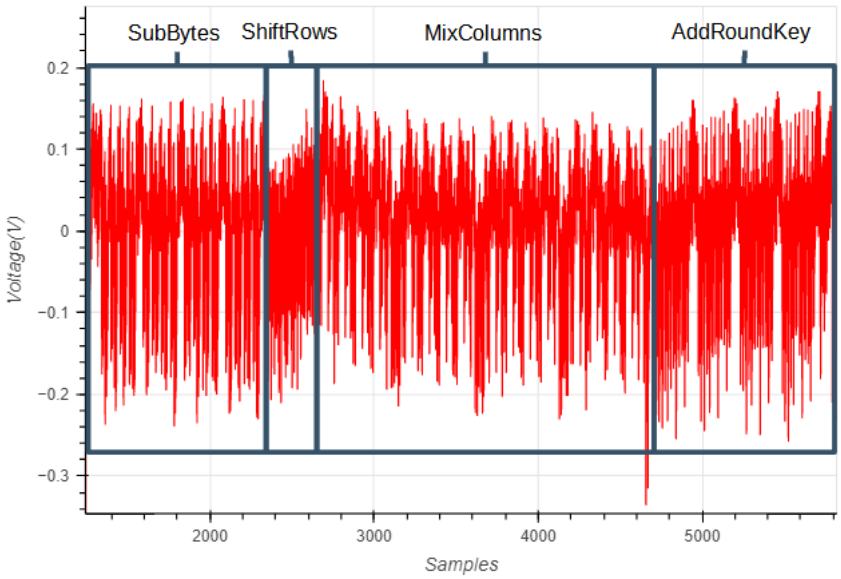

Figure 4. SPA on AES first round

In Figure 4 the first round of AES encryption is shown from a closer view, were the different internal operations can be distinguished thanks to the distinguishable patterns that can be observed in the signal.

Performing an SPA is always an interesting initial step for extracting relevant features of the algorithm, such as the position of the rounds or the power consumption and duration of the internal operations of each round. Note that apart from the power consumption, timing also gives us valuable information in a SPA.

2.2.3.2. Differential power analysis (DPA)

DPA is a statistical analysis that involves working with a high number of power traces. The attack focuses on those small power variations and follows a "*divide and conquer"* strategy: find a point in the algorithm that works with smaller pieces of the key to be able to compute all possible values of the key for that smaller pieces.

If we consider an AES-128 that has a 128 bits key length, all the possible keys add up to 2^{128} possibilities. However, let's consider the *SubBytes* operation as the target of the attack. The *SubBytes* operation is performed byte by byte 16 consecutive times, meaning that there will be different instants of time within the power consumption signal corresponding to the management of the 16 bytes independently. When attacking the SubBytes operation, instead of 2^{128} possibilities we only have $16 \cdot 2^8$ possible key guesses to assess in the power traces.

With this in mind, in order to carry out a DPA the next steps need to be followed:

- Obtain an amount of power traces of a known cryptographic operation, where the related inputs or outputs (plaintext or ciphertext) are random and known, but not the key, which has to be fixed (this is the secret to guess).
- Select the attack point in the algorithm where the differential attack will be carried out (e.g. the SubBytes operation of AES algorithm).

- Compute the reduced sub-key set and calculate the intermediate values at the attack point, for each sub-key, based on the inputs or outputs. In other words, guess the key is 0, then guess the key is 1, then guess it is 2, and so on, and for each possible key guess, calculate the data at the attack point.
- Select a bit of that intermediate data and observe if it is a 1 or a 0 in each case. Then, classify the traces into two groups depending on the value of its related bit.
- Lastly, obtain the difference among the averages of the two groups of traces.

When a wrong key is guessed, wrong intermediate data are calculated, hence wrong classification of traces is performed. In plain words, the group of '1's will contain traces related '1' and also traces related to '0', and the same for the group of '0's. Therefore, the subtraction of the average traces of the two groups will be noise.

However, when the correct key is guessed, the right intermediate data are calculated and traces are properly classified, meaning that in a specific instant of time, all traces in the group of '1's work with data that have one bit to '1' in all those traces (same for the group of '0's and data having a '0' in all traces). As a consequence, the average trace of the ones group will present a mean value higher than the average trace of the zeros group at the time instant that the tracked bit is handled in the power traces. The subtraction of the two averages will result in a peak in the differential signal.

From the mathematical point of view, the DPA can be represented with equation (9).

$$\Delta_D[j] = \frac{\sum_{i=1}^{m} D(C_i,b,K_s)\boldsymbol{T}_i[j]}{\sum_{i=1}^{m} D(C_i,b,K_s)} - \frac{\sum_{i=1}^{m}\left(1-D(C_i,b,K_s)\right)\boldsymbol{T}_i[j]}{\sum_{i=1}^{m}\left(1-D(C_i,b,K_s)\right)} \quad (9)$$

m represents the total amount of encryption measurements and $T_i[j]$ denotes the sample j at each of the related power traces T_i . The selection function $D(C_i, b, K_s)$ is defined as the computation, at the attack point, of the selected bit b .This computation starts from the plaintext or ciphertext C_i and considers the key guess K_s .

Note the differential signal has to be computed for each one of the key guesses. The highest peak among the differential signals per each key guess will be related to the right guess of the secret key.

2.2.3.3. Correlation Power Analysis

The main drawback of DPA is that it only focuses on one single bit of the whole data, while the power consumption of the device is not proportional to a single bit, but to the whole data manipulated by the device. So, despite DPA works, a more efficient attack results when analyzing the whole data and not a single bit. Continuing with the previous example in AES, a more efficient attack results when considering the whole byte value at the SubBytes operations rather than a single bit of this byte.

2.2.3.3.1. The Pearson's correlation coefficient

Once we have a way to model our power consumption, we need a way to compare our power estimations to our measurements. A helpful tool to find this relationship is Pearson's correlation coefficient, which is:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\bar{x})(Y-\bar{y})]}{\sqrt{E\left[(X-\bar{x})^2\right]E\left[(Y-\bar{y})^2\right]}} \quad , \quad \rho_{X,Y} \in [-1,1] \tag{10}$$

The covariance of two random variables, X and Y, divided by the multiplication of their respective standard deviations, gives us the linear correlation value between them. A coefficient of 1 represents direct proportionality between the random variables, while a coefficient of -1 denotes an inverse proportionality. Independency between the variables is, thus, represented with a coefficient value of 0.

2.2.3.3.2. CPA attack

CPA targets the correlation between the power traces and the estimated values of the handled data in the algorithm.

Compared to the DPA, the CPA is better in terms of efficiency and robustness. On the one hand, DPA requires more sample curves since all the unpredicted data bits penalize the signal to noise ratio. CPA can use the Hamming weight or distance models, which consider bit groups as a whole.

The procedure for CPA attack is equivalent to the one described for DPA. The only difference resides on the metric, being the Pearson's coefficient in each sample the metric for the CPA. Each of the correlation factors, related to a key guess, can be obtained applying equation (11):

$$\rho_{i,j} = \frac{\sum_{d=1}^{D}\left[(h_{d,i}-\overline{h_i})(t_{d,j}-\overline{t_{d,J}})\right]}{\sqrt{\sum_{d=1}^{D}(h_{d,i}-\overline{h_i})^2 \sum_{d=1}^{D}(t_{d,j}-\overline{t_{d,J}})^2}} \tag{11}$$

After taking our measurements, we have a total of D power traces and each of these d traces has J data points. Using subscript notation $T_{d,j}$ refers to the point j in trace d. There are I different key guesses that we have to try. Then, $h_{d,i}$ refers to our power estimate in trace *d*, for the subkey guess *i*. With these data we can calculate how well our model and measurements match for each subkey guess through time. This will be done by finding how *t* and *h* correlate over the D traces.

In other words, for each key guess it is calculated the data at the attack point and, instead of looking at the value of one bit, we apply a leakage model, e.g. the Hamming weight model, and correlate the power consumption traces with the Hamming weight of the intermediate data calculated. The highest correlation peaks will be the ones related to the right key. For wrong keys, wrong intermediate data is calculated which will result in no correlation with the power traces.

2.2.3.4. Higher order attacks

In order to protect devices against DPA/CPA, one can break the correlation between power traces and calculated intermediate data by randomizing the data manipulated by the DUT. This is known as data masking. This topic will be addressed in more depth in section 2.3. The idea is to conceal intermediate data through addition or multiplication with random values [16], which might be impossible to predict for an attacker. However, the so-called first order masking countermeasure succumbs to second order DPA/CPA attacks as originally proposed in [9] and [10].

The mounting point for second order attacks is the fact that the side channel leakage of a masked value depends on a predictable value (the original data) and an unpredictable

one (the mask). The core idea is to jointly analyze the leakage of the masked value and the leakage of the mask to establish a relationship within the power consumption with the two values. These attacks are based on the joint statistical properties of multiple aspects of the signal, i.e. joint analysis of the power consumption at two (or more) points in time [17, 18, 19].

Higher order attacks imply bigger costs in terms of number of samples and computational complexity. In addition, the identification of the points in time at which to take the signals is a hard problem.

2.2.3.5. Profiling attacks

Nowadays, profiling attacks are probably the most powerful and most widely used type of side channel attacks due to its high effectivity. The attack consists of two stages: the profiling stage and the extraction stage. The goal of the first is to fully characterize the operation of a given device with “profiles” for all the possible values that the operation can work with. A “profile” is essentially a set of probability distributions that describe how similar power or EM traces look for all different inputs.

Once the characterization is made, the developed profiles can be applied to the same device or to executions of the same operation from other “twin” devices, in order to rapidly extract the sensitive data. This application is made by comparing the power consumption of the victim’s device with the obtained profiles. The *maximum likelihood* estimator is often used as a metric of similarity for this purpose.

In order to succeed, an attacker needs to gather a huge quantity of data related to the target operation intended to characterize. We are talking about data sets reaching usually more than a million traces. On the other hand, when the template is applied to the victim’s device, only a few traces of the target operation are required to complete the attack.

Taking our example of the AES128, 256 possible values exist per each byte of the 16 that the AES master key has. Therefore, 4096 (256×16) profiles need to be created. Note that if the Hamming weight model is used, only 9 possible values exist per each byte (9 Hamming weights in a byte) and the profile amount required is reduced to 144.

The classical technique for applying profiling attacks is known as Template Attacks (TA). This methodology, based on a Gaussian assumption [5] for the characterization of the templates, can offer robust and accurate results.

Nevertheless, as machine learning keeps gaining strength in the modern era, profiling attacks are also turning into this field [14]. Profiling by deep learning using Artificial Neural Networks as analyzed in [15] and [22] has been reported to be a more a powerful tool that the others, with a huge potential still to be discovered.

2.3. Countermeasures against side channel attacks

Every algorithmic implementation can succumb to attacks by power analysis methods if it is not properly protected. In general, the solution is to re-implement cryptosystems

taking into account a wide range of countermeasures, even if the cost in terms of performance could be high.

The followings are the two general approaches for cryptographic countermeasures against side channel:

- Data hiding to reduce the side channel observability.
- Data masking to undermine the intermediate variable predictability.

2.3.1. Hiding Countermeasures

Power analysis attacks work because the power consumption of cryptographic devices depends on intermediate values of the executed algorithm. Therefore, the goal of countermeasures is to avoid, or at least to reduce, these dependencies. In the case of data hiding, this is done by breaking the link between the power consumption of the devices and the processed data values. There are two options: one is to hide power consumption in amplitude and the other is to hide it in time.

Hence, cryptographic devices that are protected by hiding execute cryptographic algorithms in the same way as unprotected devices. In particular, they calculate the same intermediate values. Yet, the hiding countermeasures make it difficult for an attacker to find exploitable information in power traces.

2.3.1.1. Amplitude hiding

The objective is to directly change the power consumption characteristics of the performed operations. These techniques lower the leakage of a cryptographic device by lowering the SNR of the performed operations. It can be done in two ways: Increasing the noise or reducing the measured signal.

On the one hand, the most obvious way of increasing the noise is introducing any kind of noise in parallel, either performing several operations in parallel or using dedicated noise engines.

On the other hand, the most commonly used strategy for signal reduction is to employ dedicated logic styles for the cells of cryptographic devices. The overall power consumption of a cryptographic device is the sum of the power that is consumed by its cells. If each cell is built in such a way that its power consumption is constant, the overall power consumption will also be constant.

For instance, a practical example of amplitude hiding is the replacing of critical assembler instructions with ones whose “consumption signature” is hard to analyze. Another example is the process of re-engineering the critical circuitry which performs arithmetic operations and memory transfers.

In software, the options to alter the consumption of a cryptographic device are very limited. The power consumption characteristics of the instructions that are executed on a device are defined by the underlying hardware. Since this project is based on firmware implementations, this type of countermeasures were disregarded.

2.3.1.2. Hiding in the time dimension

An important characteristic of power analysis attacks is that they need to acquire power traces that are aligned in time. If single points between power measurements belong to different time moments (i.e. distinct operations of the device), the statistical analysis of this point cannot be performed efficiently. In case of a CPA attack, the better alignment, the higher correlations that could be obtained.

2.3.1.2.1. Dummy executions

This technique is based on inserting dummy executions in a random basis. These dummy executions must use dummy inputs and must never act on the real data that the algorithm is working with. As a result the output of the algorithm will not be affected by the countermeasure.

If any statistical analysis is applied to the measured traces, there will be random data that will obfuscate the results, while the real data will be displaced in the time axis.

It is important to make sure that every added dummy operation is undistinguishable from the real operations. Otherwise the attacker could simply identify the pattern of the fake operation within the power signal and filter it out.

In the case of an AES, dummy operations can be inserted in many formats, from high to low abstraction level. A full dummy encryption could be executed just before or after the real one. However, this option is usually not considered since it doubles the throughput. Alternatively, extra rounds could be added within a single AES execution, dummy operations can be inserted inside the rounds or individual dummy instructions, such as register data assignations, can be introduced into the algorithm.

2.3.1.2.2. Randomizing or shuffling

Another option is to randomize the order of execution inside the algorithm. Usually, there are some executions inside an algorithm that have no order dependency between them, which means they could be randomized without influencing the final result.

Shuffling is a countermeasure that randomizes the power consumption in a similar way as the random insertion of dummy operations. However, shuffling does not affect the throughput as much as the insertion of dummy operations. The two countermeasures differ in the fact that the first one is adding extra data which enlarges the total execution time, while the other is only shuffling the data that is already there.

In the case of an AES, the most obvious part to be shuffled is the AES SBOX, which performs 16 independent fetches from the LUT. In the same manner, the *AddRoundKey* function can be shuffled in the order that establishes to XOR the state bytes with the round key bytes. Apart from that, the *ShiftRows* function internally acts independently on three rows and the *MixColumns* performs its operations in each independent column as well. Therefore, either the row order or column order can be shuffled as well in each function.

When an attacker tries to correlate data with the power traces, he will face difficulties because at the same instant of time, in different power traces it will be handled different data bytes.

The disadvantage of shuffling is that it can only be applied to a certain extent. The number of operations that can be shuffled in a cryptographic algorithm are limited, e.g. we can only shuffle the 16 bytes of the SBOX or the 4 columns of the MixColumns operation in AES. This number depends on the algorithm and on the architecture of the implementation.

In practice, both shuffling and the random dummy insertion are often combined.

2.3.1.2.3. Random time delay

One of the most common countermeasures against SCA is the introduction of random delays. Instead of executing all the operations sequentially, the CPU interleaves the code's execution with that of dummy instructions so that the corresponding operation cycles do not match between different power traces because of the time shifts. These

time delays must be randomly generated along each execution and their effect can be considered as additive noise that worsens the SNR.

In general, random delays consist of a dummy loop where a random value is generated and then decremented until the accumulator reaches zero before executing any further code. Usually, the value generated and then decremented is uniformly distributed across all the values it can take.

An example of three power consumption acquisitions that include random delays is shown in Figure 5. The three acquisitions belong to the first round of an AES and perform exactly the same operations, resulting in the exact same power profile. However, delays of 102 and 192 measured samples have been applied to trace 2 and trace 3, respect to the non-delayed trace 1. As a result, the acquisitions are no longer synchronous.

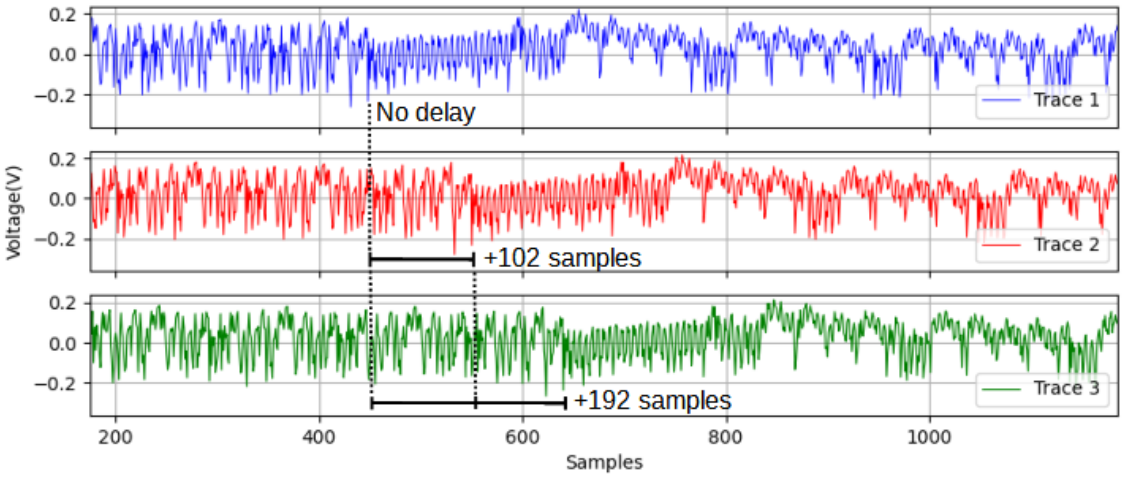

Figure 5. Power acquisitions with random delays

As the size of the random delay increases, an attacker is obliged to acquire more samples, so it is of interest to maximize the length of the delays, which on the contrary penalizes the performance of the execution in terms of added overhead.

Delays are rarely used in one single place. A single delay is easy to identify for an attacker and, therefore, its effect is easy to correct. This can be seen in the previous image where the high frequency pattern in the middle of the signals is only shifted some positions to the right in the case of the red and green signals. Therefore an attacker simply needs to re-align the traces to make that pattern match again. This is why random delays are usually implemented with short lengths and placed at different points through the whole algorithm. The objective is to break the trace with relatively short delays in multiple places so that it is undistinguishable where the attack point is.

Hence, it is usually the cumulative effect of several random delays what protects an implementation from a SCA attack. Following the Central Limit Theorem, when the sum of random delays is generated from uniformly distributed random variables, the sequence of themselves rapidly becomes binomial, which approximates to a normal Gaussian distribution. This is why the cumulative delays are usually measured in terms of the mean μ, variance σ and standard deviation σ^2.

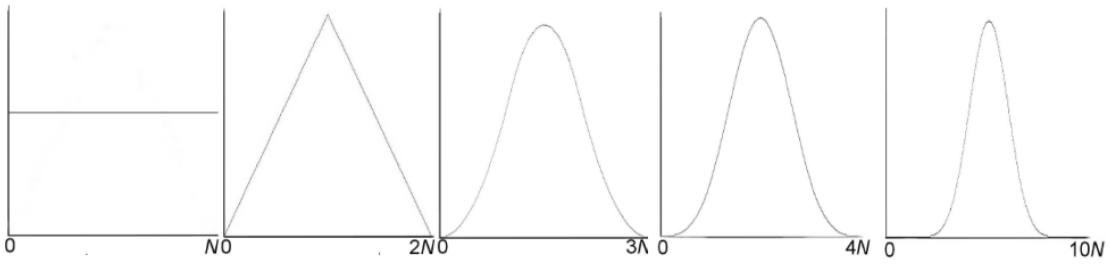

Figure 6. p.d.f.-s for the cumulative cases of 1,2,3,4 and 10 plain uniform delays

Figure 6 displays the distribution for the cumulative effect of random delays when 1,2,3,4 and 10 sums of uniformly distributed delays are considered. It can be seen how the sum of two delays is not enough for the approximation to a Gaussian curve to be acceptable. However, from three delays on, the p.d.f. shape resembles well enough that of the normal Gaussian distribution.

From [26], we know that the variance and standard deviation are closely related to the protective effectivity of the random delays. The bigger the standard deviation is the better misalignment that will be obtained. On the other side, the lower the mean the lower the total overhead added by the random delays. Therefore, the cumulative effect of random delays is more efficient for high standard deviations and low mean values [25] [26] [27].

2.3.2. Masking countermeasures

There is a different kind of countermeasure that must be always considered: data masking. Data masking intends to break all correlation between the power consumption and the actual intermediate data. In other words, masking allows making the power consumption independent from the intermediate data, even if the device has a data-dependent power consumption.

An advantage of this approach is that it can be implemented at the algorithmic level without changing the power consumption characteristics of the cryptographic device.

Hence, a masked algorithm is an algorithm which with given inputs will produce the same output than the non-masked version, with the only difference that all intermediate computations will be masked with random values. The masks are internally generated by the device for each algorithmic execution.

But what does masking mean? Data being masked means that the real value of the data is somehow mixed with more data. This mixing is usually done through XOR and AND operations. Since the data manipulated by the algorithm is not strictly the real data, the power consumption generated will not be the same and the leakage will be strongly reduced. If the masking is well implemented the leakage should disappear completely. In other words, if an attacker does not know to which values correlate the power traces (because the masking values are random and secret) the attack cannot be mounted.

Typically, the masks are directly applied to the plaintext or the key. The implementation of the algorithm needs to be slightly modified in order to process the masked intermediate values and in order to keep track of the masks. The result of the encryption is also masked. Therefore, the masks need to be removed at the end of the computation in order to obtain the ciphertext. It is important to make sure that every intermediate value is masked all the time, otherwise correlations could be recovered between unmasked intermediate values and the power traces.

2.3.2.1. Boolean masking vs Multiplicative masking

In Boolean masking schemes the intermediate value is always concealed by XOR-ing it with the mask, while multiplicative masking uses AND operations for the masking.

Let's consider only Boolean masking for now. Regarding most of the cryptographic algorithms, the masking has to be applied to both linear and non-linear transformations. A linear function f always complies with the following equation, where m stands for any mask and x represents any intermediate value:

$$f(x \oplus m) = f(x) \oplus f(m) \tag{12}$$

Boolean masking is suitable for linear operations since they modify the mask in an easily computable way. However, if a non-linear function g is considered, the previous relation does not hold anymore:

$$f(x \oplus m) \neq f(x) \oplus f(m) \tag{13}$$

Since Boolean masking cannot properly suit how non-linear functions operate, more complex perspectives have to be contemplated in order to compute the mask modifications.

Now, let's consider the case of an AES algorithm. All the functions used in it are linear functions except for one, the *SubBytes* transformation. This function is based on computing the multiplicative inverse inside the Rijndael's finite field, which is compatible to multiplicative masking since:

$$(x \otimes m) \Rightarrow inverse \Rightarrow (x \otimes m)^{-1} \tag{14}$$

Note that instead of an XOR the above function operates with an AND. In order to implement this masked multiplicative inverse, the Boolean masking coming from the previous linear transformation has to be modified into a multiplicative masking. The opposite happens with the output of the inverse that has to be modified back into Boolean masking. As a result, a mix between Boolean and Multiplicative masking can be applied to solve the problem of the non-linear function in AES.

Nevertheless, changing from one type of masking to the other is not trivial and often requires a significant amount of additional operations.

2.3.2.2. Higher order masking

Closely related to the higher order attacks discussed in section 2.2.3.4, higher order masking schemes can be implemented as a direct protection against HODPA/HOCPA.

A hardware device with a first order masking scheme is vulnerable to second order attacks. If a second order masking is implemented in the device, this will become resistant to first and second order attacks, but it will still be vulnerable to third order attacks. Following this line of thought, the resistant-vulnerable relation between higher order masking schemes and higher order attacks could be, theoretically, escalated to any level.

We already know that higher order attacks target multiple points in time for the side channel analyses. The question now is: how is the order of a masking increased? This is done by introducing more shares into the equation, i.e. for a first order masking we have:

$$x_m = x_i \oplus m \tag{15}$$

Where m is the mask, x_i is the intermediate value to be protected and x_m is the masked intermediate value. Now, if a n^{th} order masking is considered:

$$x_m = x_i \oplus (m_{n-1} \oplus \dots \oplus m_2 \oplus m_1 \oplus m_0) \tag{16}$$

As shown in (16), for higher order masking, the intermediate value is XOR-ed with more than one mask. Second order masking requires two masks, third order masking requires three masks and so on.

2.3.3. Countermeasure effectivity

Hiding countermeasures reduce the effectivity of any side channel attack. These countermeasures introduce uncertainty into the attack and achieve to reduce the dependency between power consumption and intermediate values of the cryptographic algorithm. However, this dependency does not totally disappear. In other words, these countermeasures make the attack effort increase in order to obtain any useful result, but they do not avoid the attack.

In this situation, the attacker needs to increase the number of power traces, resulting in an increase of employed time and resources. Alternatively, the attacker could try to improve the attack conditions by processing the measured power data to increase the SNR or by applying alignment data processing techniques [24]. Anyway, the attack effort still increases in many ways.

In order to be able to measure how efficient hiding countermeasures are, it will be considered that the attacker does not have any further knowledge on how to process or align the acquired power traces. Therefore, the only available option is to increase the quantity of traces captured for the attack.

In [7] they pointed out that the number of needed power traces grows quadratically with the number of randomized operations. In order to obtain the minimum trace quantity needed for a protected implementation, the minimum trace quantity of the unprotected implementation has to be multiplied by a factor that scales up quadratically:

$$N' = k^2 \cdot N \tag{17}$$

N is the minimum number or power measurements required for a successful CPA attack on an unprotected implementation, while N' represents the same in the case of a protected implementation. k will be denoted as protective effectiveness factor. This parameter is affected by many variables that correspond to the device and trace acquisition environment. Among these, we have the leakage characteristics of the device, its power consumption and electronic noise, in addition to the sampling rate and SNR of the acquired signal.

In [8] Mangaard proposed an approximation of the protective effectiveness factor k based on the correlations obtainable for each implementation:

$$N' = \left(\frac{\rho_{unprotected}}{\rho_{protected}}\right)^2 \cdot N \tag{18}$$

The factor can be approximated as the division of the correlations resulting from the attack on each of the implementations.

Considering the hiding countermeasures implemented for this thesis, the protection level that they offer will be measured and compared using the k factor defined in (17). For the unprotected implementation k will be of one.

Regarding the masking countermeasures, their effectivity is quite straightforward to measure. If there is still leakage, the masking is not effective. On the contrary, if the leakage is eliminated, the data dependency is not exploitable anymore, meaning that the masking scheme is efficient.

2.3.4. Importance of unpredictability in randomization

System designers are typically more concerned with the power consumption and bit generation speed, than with the actual randomness of the bits generated.

This is strange, considering that in most, if not all, cryptographic systems, the quality of the employed random numbers directly determines the security strength of the system. In other words, the quality of the random number generator directly influences how difficult is to attack the cryptographic system.

What happens if we start with key material that is partly predictable to the attacker? Immediately, the security of the system is weakened, regardless of the algorithm or protocol used. Take, for example, the effective strength of an AES128 key. If your 128-bit key contains 16 predictable bits, using it in AES-128 does not give you a 128-bit protection (i.e. 2^{128} possibilities to cover by brute force), but only gives you 112 bits of protection, making the effective security of your system lower than the actual key length it uses.

Now, if we think about the cryptographic countermeasures, all the countermeasures need randomness in order to be effective. Indeed, every countermeasure that has been introduced would be useless if they were applied deterministically, or if the attacker could guess the “random” values applied.

Considering the approach of hiding in time, the attacker could unmake the trace desynchronizations. In the case of masking, it would be even easier to extract the real values of the intermediate variables. If the values of the random masks are known, the double unknown variable problem is reduced to a single unknown variable problem, making the implementation vulnerable to first order attacks again.

3. Methodology and project development

3.1. Experimental setup

3.1.1. Chipwhisperer - Lite

After consideration of various possibilities, it was decided to work with a Chipwhisperer-Lite board. Chipwhisperer is an open source platform for hardware embedded security research, specially designed for side channel analysis and glitching based fault injection. The Chipwhisperer kit typically comes with two main parts: a multi-purpose power analysis capture instrument, and a target board. The target board is the device under test (DUT), which is basically a standard microcontroller were you can implement algorithms onto.

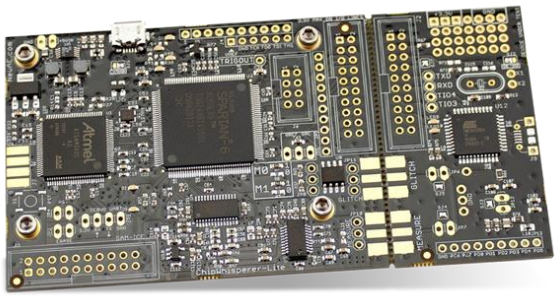

Figure 7. Chipwhisperer lite capture and target boards

The capture board uses an ATSAM3U2CA-AU microcontroller, which is a 32-bit high performance MCU based on ARM cortex-M3 RISC processor, in addition to a FPGA that belongs to the SPARTAN-6 family. The microcontroller has a USB controller interface implemented in C and is mainly used for communication purposes between the capture board, target device and the computer at the other end of the USB (see Figure 8). The FPGA is employed for high-speed capture purposes, in addition to other useful features such as clock or voltage glitching and triggering.

All communication with the capture board is done via USB through the Chipwhisperer's Python API. However, any language that could talk to *libusb* (C library that provides access to usb devices) should be compatible.

The communication between the target and capture board is done using the *Simpleserial* protocol based on the C library. The master-slave serial protocol begins every communication from the capture device sending data packets to the target. Whether the target device sends data back or not, it has to answer with an ACK to tell the capture board that the communication was successfully performed.

Some of the main features of the Chipwhisperer capture board are:

- 10-bit ADC with a maximum sampling rate of 105MS/s for capturing power traces.
- ADC clock that can work synchronously at target clock frequency (x1) or four times faster than the target clock (x4).
- AC-coupled LNA with adjustable gain from -6.5dB to 55dB
- Maximum sample buffer size of 24.573

It should be noted that, both, capture and target devices work with the same clock offering synchronous capture features. This ensures sample points are directly related to the digital clock that generates the signals of interest. As a result, the power

consumption of the target DUT can be successfully analyzed, even if lower sampling rates are used compared to a regular oscilloscope.

Regarding the target device, an ATXmega128D4-AU 8-bit RISC microcontroller of AVR architecture (Harvard modified architecture developed by ATMEL) is employed. The device is of low power performance at up to 32 MHz speed and has the following memory capabilities:

Memory type	Space (KB)
Flash (program memory)	128
EEPROM	2048
SRAM	8

Table 2. Target MCU memory

3.1.1.1. Capture configuration

A fixed capture configuration was set in order to perform the power measurements of the different implementations. These are gathered in Table 3.

Parameter	Value
Gain	25dB
f (target)	7.38MHz
f (ADC)	29.53MHz
Baud rate	38400 bits/s

Table 3. Capture configuration

Regarding the sampling rate, the chosen configuration uses a sampling frequency four times bigger than the target signal to be measured. This way, the resulting sampling rate doubles the minimum limit required by the Nyquist theorem, ensuring that the obtained signal can be completely reconstructed.

3.1.2. Implementation environment

The implementations were coded in C and programmed into the target XMEGA device. Eclipse IDE environment was employed for their development. The only libraries used were the standard C library, from which *stdlib.h* and *stdint.h* were included.

All the Chipwhisperer firmware is bare-metal, meaning that no operating system is supporting it and the same path was followed for the development of the AES implementation and the countermeasures.

Regarding the random number generation, the *secrets* built-in Python module was used, together with the *rand()* function of the standard library of C. Note that two independent random number generations were used in the experimental set-up, one external to the target DUT and the other internal.

On the one hand, we have the Python code running on a windows computer that uses the *secrets* module. This module is based on a cryptographically secure PRNG that employs various OS data as randomness source (PRNG seeding).On the other hand, we have the C code implemented inside the DUT, that uses the *rand()* function and has to be internally seeded.

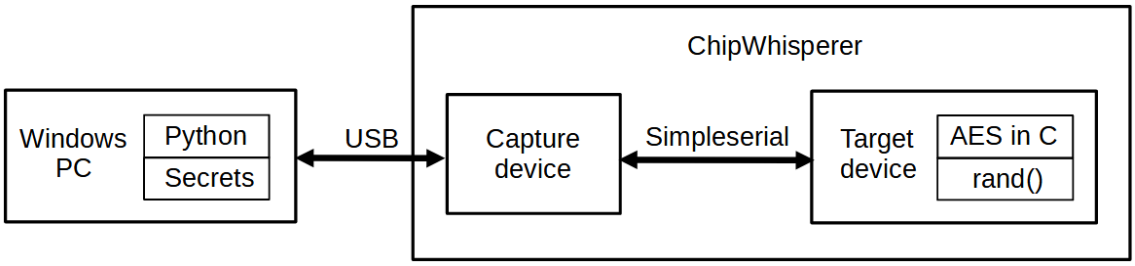

Figure 8: Experimental set-up

Each plaintext input to encrypt was generated randomly using a python script and sent to the target DUT trough the capture board. Regarding the secret key, this was also sent through a python script, only that it was fixed to the following hexadecimal value.

$$Secret\ key = \{CAFE\ ABBA\ CAFE\ ABBA\ CAFE\ ABBA\ CAFE\ ABBA\}_{HEX} \quad (19)$$

Actually, the secret key is sent once and kept into the memory of the target device, unless a new key is sent to replace it.

Apart from the plaintext and key, a random word of 32 bits is also sent before each encryption, which is used to seed the *rand()* function that will internally generate any needed random number for the implementation of the countermeasures.

Once the AES encryption is done and its power consumption is acquired by the capture device, the data is sent to the windows computer where the traces can be saved in order to apply SCA on them.

3.2. Analysis metrics

In order to conduct the project and be able to fairly compare the effectiveness of the implemented countermeasures, different figures of merit and metrics will be used. Hereunder, these figures of merit and metrics are introduced, before using them in the following sections of the document.

3.2.1. Attack point: SBOX output

First to define is the attack point within the AES encryption algorithm. The countermeasures developed in this project are meant to protect the DUT against side channel analysis. As side channel analysis can be done in different parts of the AES algorithm. It has been chosen, as attack point, the data output at the SBOX of the first round.

The rationale behind this attack point is the following: the first operation conducted by AES algorithm is an XOR between the input data and the secret key. The second operation is the SBOX. Hence, the data at this SBOX output is the first point in the algorithm where we have a variable related to the secret key that is treated byte by byte. Hence, it is a nice target point for an attacker that would like to retrieve the value of the secret key by SCA. Note that after the first XOR operation we also have a variable related to the secret key, however, the XOR operation is bit-wise while the SBOX operation is performed byte-by-byte in sequential order.

The following diagram shows a schema of the attack point.

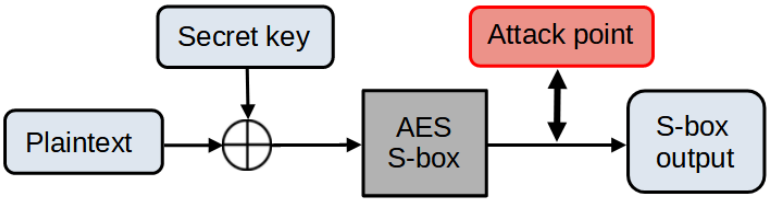

Figure 9: Attack point: SBOX output

The attack process is the following:

- Acquire a set of power traces (hundreds or thousands depending on the experiment) with random input data, but fixed key.
- For each key guess (from 0x00 to 0xFF) of the first byte of the AES key, calculate the values of the first byte at SBOX output.
- Compute the Pearson coefficient between the power traces and that calculated values at SBOX output. This must be done for all key guesses in order to identify the one retrieving the highest Pearson coefficients (which will be the one corresponding to the correct key).
- Repeat the process for all other key bytes.

3.2.2. Attack technique

In order to analyze how good the implemented countermeasures are, we are going to use correlation analyses (CPA as explained in section 2.2.3.3). The correlation analysis will compute the Pearson correlation coefficient between the power traces acquired from the DUT with the data at SBOX output, per each key guess, byte by byte.

As explained in previous point, the data at SBOX output depends on the input data to the algorithm, which is known, and also on the secret key, which is unknown by an attacker. Therefore, in order to compute the Pearson coefficient at SBOX output, an attacker needs to make hypothesis of the value of the key. When the attacker correctly guesses the key value, the correlation analysis will correlate the correct data at SBOX output with the power traces, and correlation peaks will be seen as a result of the attack. On the other hand, when the guessed key value is incorrect, the attack will try to correlate the power traces with incorrect data and the Pearson correlation will be very low.

Note the importance of choosing as attack point the output of the SBOX. If an attacker would target the XOR, he would need to compute 2^{128} key hypothesis (because AES key is 128 bits long, or 16 bytes equivalently). However, as the SBOX is conducted byte-by-byte, the attacker only needs to compute 2^8 key hypothesis per each of the 16 key bytes independently. So in order to recover the full AES key, an attacker would need to repeat 16 times the CPA at SBOX output, one per each key byte, which is completely affordable because only 16×2^8 key guesses are required to recover the full key.

Hence, as it can be observed, the objective of this project is to implement countermeasures that make the CPA retrieve nothing, even when the attacker correctly guesses the right key. Accordingly, the obtained Pearson coefficients will be lower in all the protected cases. This way we can conclude that the countermeasures are efficiently protecting the AES algorithm against side channel attacks.

3.2.3. Figures of merit and comparison metrics

In order to assess and compare the effectiveness of the different implemented countermeasures the following metrics will be used:

- In an unprotected AES implementation, we are going to obtain which is the minimum number of traces required to retrieve the AES key by CPA, and which are the Pearson coefficients found.
- Per each implemented countermeasure, we are going to, first, assess whether the correct key can be retrieved or not; and second, if can be retrieved, which is the minimum number of traces needed (which might be much higher than the ones from the unprotected implementation) and which are the Pearson coefficients (which might be much lower than for the unprotected implementation).
- The protective factor k is defined as the relationship between the number of traces N required to retrieve the correct key in the protected implementation vs. the traces N' needed for the unprotected implementation.

$$N' = k^2 \cdot N \tag{19}$$

  Hence, if the countermeasure implemented is not effective, k will be close to 1, while in case the countermeasure is really effective, k will tend to ∞.

Therefore, the main figures of merit used to assess the protective effectivity of the countermeasures are:

- ❖ The amount of retrievable key bytes, if any.
- ❖ The minimum trace quantity for the key retrieval.
- ❖ The protective factor k.
- ❖ The maximum Pearson correlation coefficient.

Besides, each countermeasure implementation will be analyzed in terms of duration of encryption (i.e. performance). Therefore, a secondary figure of merit is defined:

- ❖ Time overhead respect to the duration of the unprotected AES (measured in clock cycles)

3.3. Implementation development and analysis

3.3.1. AES without countermeasures

3.3.1.1. Implementation aspects

When implementing an AES in software, there are various options to consider. In this section some of them will be introduced and some of their pros and cons will be discussed (The reader is encouraged to have a look at Appendix A for the better understanding of the following section, where AES is explained in more depth).

Since the AES works strictly with bytes in an 8-bit platform, the employed data type was *uint8_t* (unsigned 8bit) in order to keep the implementation as light as possible.

The developed implementation uses some variables defined as static and global, such as the state matrix, which is implemented as a two dimensional array, and a single dimensional array used to store all the keys derived from the key schedule. The static

variables remain in memory while the program is running, regardless of the functions that call or influence them. The round constants were defined as global, static and constant variables, since they are never modified through the program execution. The same was done for the SBOX and reverse SBOX LUTs. The rest of the variables used in the algorithm are instantiated locally inside the functions that require them.

Regarding the *SubBytes* function, most of the times the AES SBOX and its reverse are implemented in form of LUTs of 256 components (Appendix B1). However, there is also the possibility of implementing them as the inverse function in the Rijndael's finite field followed by an affine transformation (Appendix A.2.1). When no complexity is required, the LUT tables work perfectly, only that taking extra 512 bytes of memory space. Nevertheless, some data masking schemes prefer to implement the inverse and affine transformations in order to mix the intermediate values with random masks.

The *ShiftRows* function, instead, offers no option at all. The bytes are shifted as required by the algorithm and that is all. However, since the shifting is done circularly some byte buffers are needed, which has a big impact on the leakage generated.

The *MixColumns* function is probably the one that offers more implementation possibilities. As a first option, the Galois multiplications (Appendix A1 and A.2.3) for encryption, and decryption, can all be implemented in form of LUTs (Appendix B2). Then, equation (44) is applied and the transformation has concluded. This option is quite straightforward, but notice that a total of 1536 bytes of memory are required in order to store the six LUTs.

A more efficient way is to implement the function based on the independent multiplications by x that the *MixColumns* transformation performs in the Galois field. The multiplication by x can be implemented in a function as follows:

$$f(x) = \Big((x \ll 1) \text{ ^ } \big((x \gg 7 \,\&\, 1) \oplus 0x1b\big)\Big), \;\; where\; x \in GF(2^8) \qquad (21)$$

These function reflects the equations shown in (46) and (47). Then equation (40) can be directly applied.

As a last consideration, there is an implementation technique worth mentioning called bit slicing. Bit slicing considers single bits and not byte structures. A bit sliced algorithm can perform N encryptions in parallel on a microprocessor with N-bit register width, resulting in a significant performance boost. Applied to the XMEGA target, 8 encryptions could be carried out in parallel. However, this was considered to be out of the scope of this project and, accordingly, it was not implemented.

The final implementation of the AES128 can be found in Appendix C1. Even if both encryption and decryption were implemented, only encryption will be considered for future analyses on the countermeasures.

3.3.1.2. Leakage assessment of AES

SPA on AES

Before analysing the effectiveness of any countermeasure, it is worth analysing the power consumption characteristics of the unprotected implementation of AES. A SPA was carried out in order to identify the power consumption patterns of the implementation and locate the first rounds of the AES.

As can be seen in next figures, when we execute the programmed AES algorithm we obtain a trace where a repetitive pattern can be identified. The FPGA was triggered just

before executing the AES encryption process in order to start recording the samples just before computing the first step of the encryption. We can see that the power consumption corresponds to the AES-128 algorithm because each of the 10 rounds conforming it can be identified as a repetitive pattern. As expected, the last round was found to be shorter than the others due to the absence of the *MixColumns* function in it.

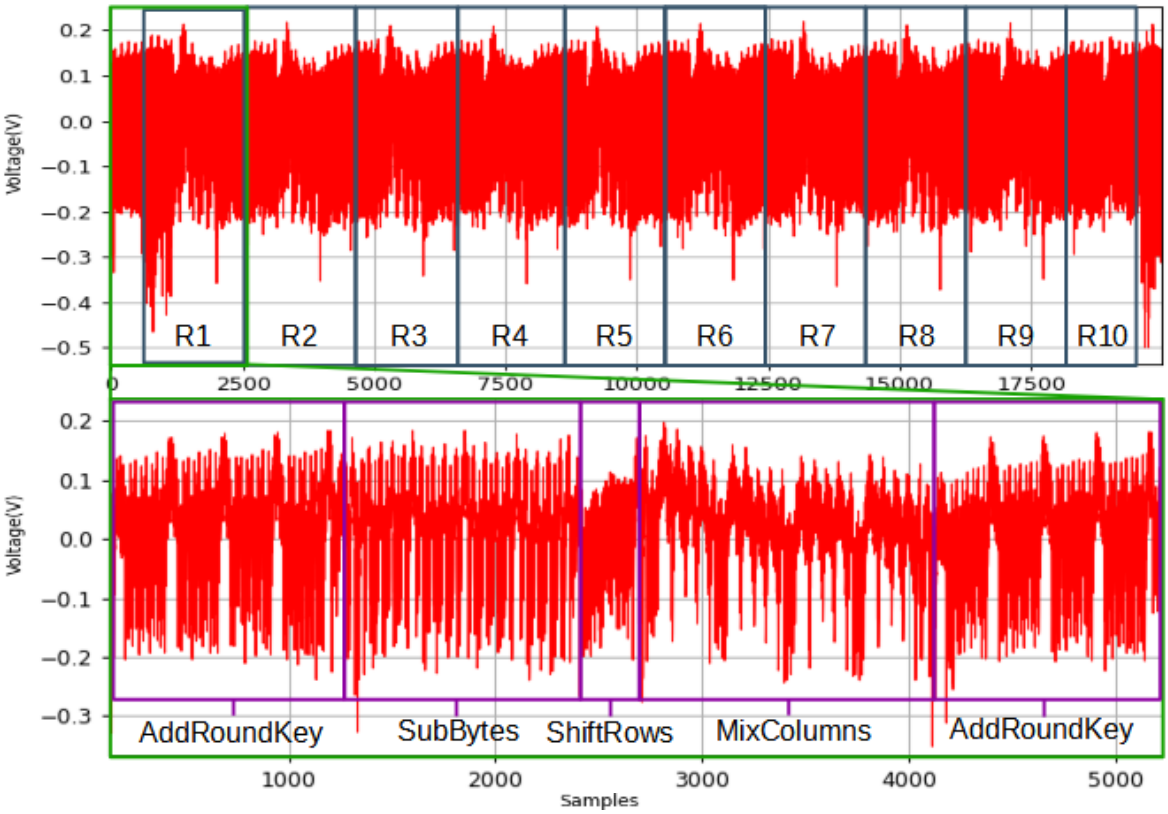

Figure 10. SPA on AES128 implementation

By zooming into one of these rounds, different patterns can also be observed, allowing for the identification of the different transformations that AES conducts within its rounds: *AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns*. This is the sample space where the CPA will be performed.

If we look at the power profile of each transformation, the power shapes are assembled in groups of four (except for *ShiftRows*), actually grouped row by row or column by column. This behaviour is a direct consequence of implementing the AES state as a two dimensional 4x4 matrix. Moreover, inside each row/column group, the smaller operations on the bytes can be differentiated (i.e. in the *SubBytes* region each of the four row processing regions can be differentiated and the power consumed by each of the SBOX look-ups can be observed inside them).

The sample amount needed to capture all the encryption process was bigger than the buffer size of the capture tool. This is why a decimation of 2 had to be applied to the ADC clock, setting a sampling rate of 14.765MHz. For the zoomed view the decimation was undone, restoring the sampling rate to 29.53MHz, in order to offer the highest resolution possible (i.e. the samples between first and second graph of Figure 10 do not match because sampling rate was doubled for the augmented view).

As the reader will see in the future analyses done, this sampling rate modifications were applied to most of the performed SPAs (with different decimations), because it was the

only option to overcome the sample buffer limit of the capture tool in order to obtain a good resolution. However, it had no impact in the analyses as this decimation was only done for SPA and undone before CPA was conducted.

As a next step for the SPA, a timing analysis was done on the different distinguishable operations inside the encryption of the AES. The time spans were measured in samples in each case and, considering the sampling rate that was used, their related clock cycles were calculated.

The next table gathers the measured time periods for the different operations of the AES:

	Clock cycles (cc)	Time
Full encryption	9780	1.325ms
Round	985	133.42µs
Last round	635	86.01µs
SubBytes	285	38.60µs
ShiftRows	70	9,48µs
MixColumns	350	47,41µs
AddRoundKey	280	37,92µs

Table 4. Time measurements on the unprotected AES

The AES encryption had a duration of approximately 19.560 samples, which is to say that 9780 clock cycles or 1.327ms were required by the target DUT in order to compute the encryption of a plaintext.

Since this was the first timing analysis that was done, the measurements were taken for all the internal operations of the AES. However, in the future analyses, only the full encryption length will be considered because this is the figure of merit chosen for the time overhead assessment of the countermeasures.

CPA on AES

After conducting a CPA with 200 traces on the unprotected AES implementation on the attack point identified in section 3.2.1, the secret key was recovered successfully. The correlation results obtained for the first 5 positions can be seen in Table 5:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	CA 0.745	FE 0.724	AB 0.723	BA 0.783	CA 0.733	FE 0.918	AB 0.721	BA 0.796	CA 0.740	FE 0.908	AB 0.817	BA 0.915	CA 0.765	FE 0.823	AB 0.810	BA 0.915
1	C1 0.239	0B 0.244	DE 0.266	4F 0.292	72 0.209	F5 0.285	DE 0.247	4F 0.278	BF 0.293	0B 0.273	DE 0.234	E6 0.283	7E 0.238	A9 0.255	DE 0.238	4B 0.259
2	3C 0.233	31 0.230	2B 0.240	13 0.271	A4 0.208	CB 0.270	14 0.240	0E 0.245	80 0.242	CB 0.261	22 0.222	02 0.272	48 0.236	CB 0.248	5E 0.219	F6 0.241
3	3F 0.219	A2 0.225	14 0.215	26 0.216	E1 0.207	0B 0.233	37 0.219	28 0.225	7E 0.226	46 0.260	98 0.218	4F 0.267	C7 0.218	C1 0.237	5D 0.214	49 0.227
4	FF 0.219	46 0.221	29 0.210	B1 0.216	BA 0.204	E3 0.224	DB 0.214	1D 0.216	72 0.224	67 0.233	A4 0.206	13 0.245	5A 0.209	48 0.232	94 0.212	E4 0.220

Table 5. CPA on first round results for unprotected AES

As explained, the CPA attack is performed per each key byte, making 2^8 hypothesis of the value of this key byte. The Pearson correlation coefficient is computed for all key hypotheses and the one resulting in the highest correlation is the one that is correct. Therefore, as can be seen, the CPA returns a ranking of key values sorted by the value of the Pearson coefficient. Each column represents each of the 16 key bytes to be

guessed (this enumeration will be used from now on to reference each key byte) and the rows indicate the ranking position of the byte hypothesis. The correct key byte guesses are highlighted in red.

If we observe the first byte (key byte 0), we can see that the first position is occupied by the value “0xCA” with a 0.745 Pearson correlation, and the second position is occupied by the value “0xC1” with a 0.239 Pearson correlation. With this numbers, it is clear that the value “0xCA” is the correct value for the first byte of the key. Note that the other bytes of the key are retrieved following the same procedure.

Figure 11 shows the correlation value over time for the key byte 1:

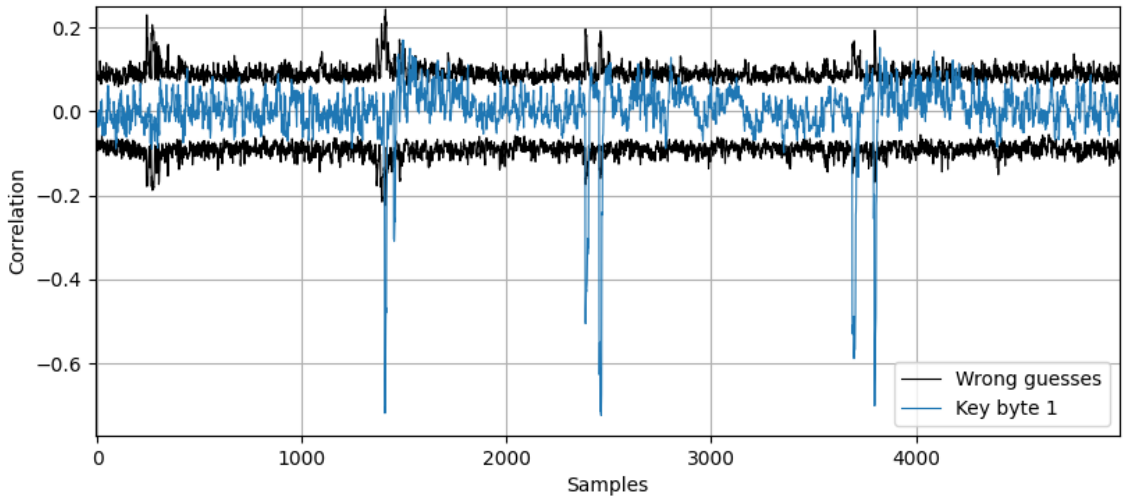

Figure 11. Correlation vs time for key byte 1

In the figure above, the Pearson correlations obtained for the key byte 1 were plotted (column 1 of Table 5). The plotting was carried out drawing a curve, for each byte hypothesis, with the correlations obtained at each sample.

Remember that the CPA performed correlates the measured power consumption to the power estimates at the attack point (i.e. Hamming weight of the SBOX output). When a correct byte hypothesis is made, the correlation curve shows peaks at the time instants where the power estimate matches the measured power. As a result, the correct byte hypothesis (blue) shows observable correlation peaks at three different time regions of the first AES round. On the contrary, when the wrong hypothesis are made, the power estimates do not approximate well enough to the real data and, therefore, the correlations for the wrong guesses (black) do not stand out.

In relation to the time instants where the blue peaks were obtained (each double peak will be considered as one peak), the first one corresponds to the SBOX operation performed in the *SubBytes* transformation. The second was generated by the shifting of the same byte made in the *ShiftRows* transformation. The third spike was produced by the XOR-ing made in the *MixColumns* transformation (when C is traduced to machine code, the bytes are loaded from memory, leaking their value, before XOR-ing them). Hence, we can observe that the byte value at the SBOX output is processed by the algorithm at three different operations during the first round of the AES.

Figure 12 shows the correlations obtained at the *SubBytes* transformation for all the key bytes:

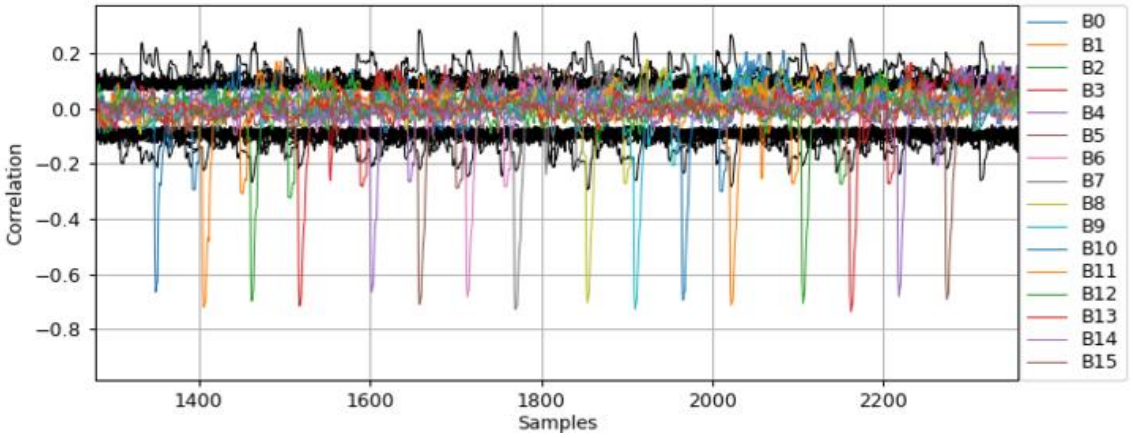

Figure 12. Correlation vs time for all key bytes

This time, the correlation results were plotted considering all of the key bytes, but only for the *SubBytes* transformation (not all the round like in Figure 11). The correct subkey guesses were highlighted in colours (i.e. 16 colours for 16 correct key bytes), while the wrong guesses were represented all in black. Each of the 16 spikes were generated due to the 16 SBOX substitutions, one for each key byte, performed sequentially by the implemented algorithm.

We can see that all the key bytes show the same leaking behaviour. However, as a matter of fact, some key bytes show bigger peaks than others do. This indicates that not all the samples leak the same amount of information. This is probably due to the power consumption of the DUT not being deterministic.

The important thing is that each correlation from each byte is found in a different time position. This is what makes the attack feasible because in each time position we can analyse a single byte making 2^8 hypothesis of the key value as explained before (now it is clearly seen graphically).

As a next step, we will focus on the evolution of the correlations for the key bytes in function of the number of traces. As stated at the beginning of this section, the CPA was performed with 200 power consumption traces. The results shown in Table 5, Figure 11 and Figure 12, correspond to the values obtained when all of the 200 traces were added into the CPA attack. However, the CPA can be performed trace by trace (i.e. adding one more trace to the attack in each iteration) in order to analyse the evolution of the attack. As a result, the correlations evolve each time a new trace is added into the analysis.

Figure 13 shows the correlations obtained for the key byte 1 in function of the number of traces:

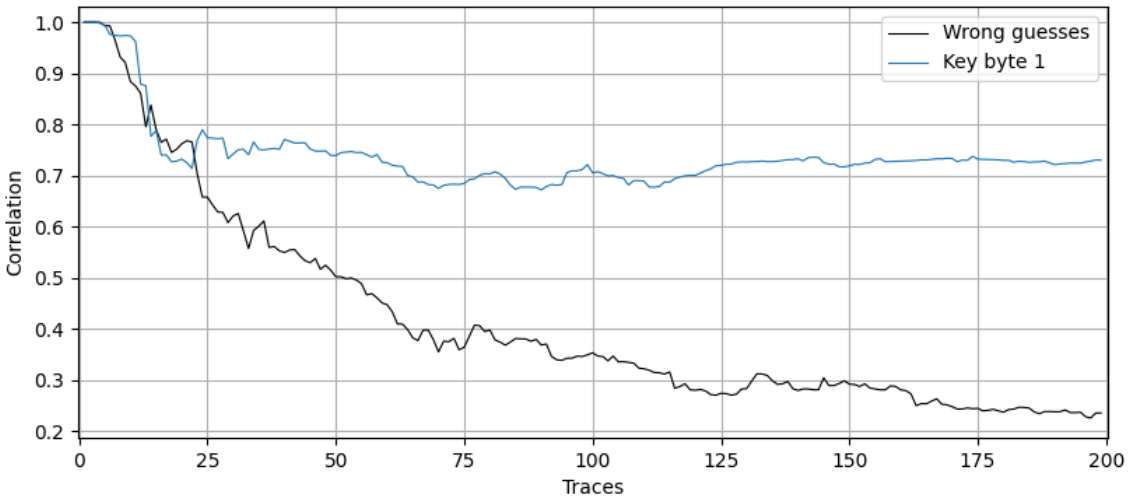

Figure 13. Correlation evolution during the CPA for the key byte 1

Once again, the correlation for the correct key byte 1 was plotted in blue, while the maximum correlation for the wrong guesses was plotted in black. Looking at the graph, we can see that approximately for the first 25 traces, both curves show similar correlation values, meaning that the correlations of the correct key guess were similar to the ones of the wrong guesses at first. If the attack had stopped at this stage, it would be impossible to know which one is the right key guess since this would be mixed with the wrong ones. However, as the attack keeps on considering more traces, the correct key guess stands out respect to the incorrect ones.

A divergence between both curves is generated slightly before adding the trace 25. Observe the more accurate view of the divergence shown in Figure 14. It can be seen that the divergence was generated when trace number 23 was added into the attack. After this divergence, the difference between both curves gets bigger, increasing the difference of correlations between the correct byte guess and the wrong ones.

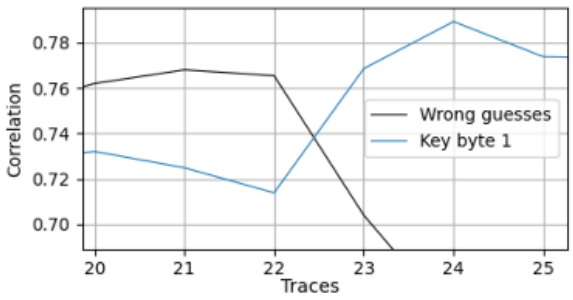

Figure 14. Zoomed view of the divergence

As a result, it can be considered that 23 traces were needed in order to retrieve the correct value of key byte 1 (i.e. 23 traces were needed in order to rank the correct value of key byte 1 at first position).

The analysis that was made for key byte 1, was equally made for all the other key bytes. The leaking behaviour is the same, or very similar at least, for all of them. The only difference resides in the exact correlation values obtained during the CPA in each case and the number of traces to rank the correct key value at first position.

Even if most of the key bytes were successfully retrieved with less than 30 traces, the key byte number 6 was the slowest (worst case key byte) needing a total of 46 traces to rank first. Remember that the objective of the attack was to extract the complete value of the secret key (all of the 16 key bytes). Consequently, it will be considered that a minimum quantity of 46 traces were needed in order to retrieve the secret key and break the AES for this initial unprotected implementation of the algorithm.

In order to compare the performed attack and results with the future countermeasure implementations, the reference figures (worst case considered) are gathered in Table 6:

CPA region	Correlation	Traces	K	Retrieved
First round	0.918	46	1	Full key

Table 6. CPA results on first round for the unprotected AES

Note that the maximum correlation obtained in the attack was included in the table considering all operations conducted within the first round of the algorithm (i.e. 0.918 is the maximum correlation obtained no matter whether it is found in the SubBytes, MixColumns, AddRoundKey or ShiftRows transformations). Considering that the countermeasures aim to reduce the correlations obtainable for this implementation, the worst case will be the maximum correlation that could be obtained in comparison with this one.

As a last step, two more CPA attacks were conducted with the same trace set, but focusing only on the sample spaces of the *SubBytes*, and *MixColumns* transformations (not all the first round like before). Table 7 gathers the results.

CPA region	Correlation	Traces	K	Retrieved
SubBytes	0.753	81	1	Full key
MixColumns	0.812	61	1	Full key

Table 7. CPA results on different operations of the unprotected AES

The values in Table 7 will be used to assess the protective effectivity of the randomized SBOX-es and the randomized *MixColumns* analysed in section 3.3.2.2. Considering the results in Table 7, it is clear that different operations leak the different amounts of information.

3.3.2. AES with hiding countermeasures

3.3.2.1. Dummy rounds

The dummy rounds are additional rounds to the AES algorithm that operate on a dummy state, with random inputs and random keys that are completely uncorrelated to the actual encryption process, leaving the real state matrix untouched. Their objective is to create confusion to the attacker who cannot distinguish which rounds of the algorithm work with real data and real key, and which ones work with fake data and fake key.

The implementation developed has two variations. The first variation adds a dummy round before or after each real round. The second variation adds two dummy executions per each round, where both of them could come before or after the real round, or one

before and the other after. The positioning of the dummy rounds is done through random bytes generated before each encryption process.

For the experimental process, CPA attacks were carried out for the single dummy round implementation focusing on the first two rounds. Statistically, the probability of finding real data in the first round of each trace is of $1/2$. Every sample containing real data improves the resulting correlation value, while the samples containing dummy data affect negatively on the correlation results.

For the double dummy round implementation the CPA targeted the initial three rounds. The probability of finding real data in the first round decreases to $1/3$ in this case.

Before every CPA attack, a SPA was done in order to define the attack region (i.e. the sample space targeted for the attack) and also to measure the length of the AES encryption with the implemented countermeasures.

The dummy rounds implementation is shown in Appendix C2.

SPA on dummy round implementations

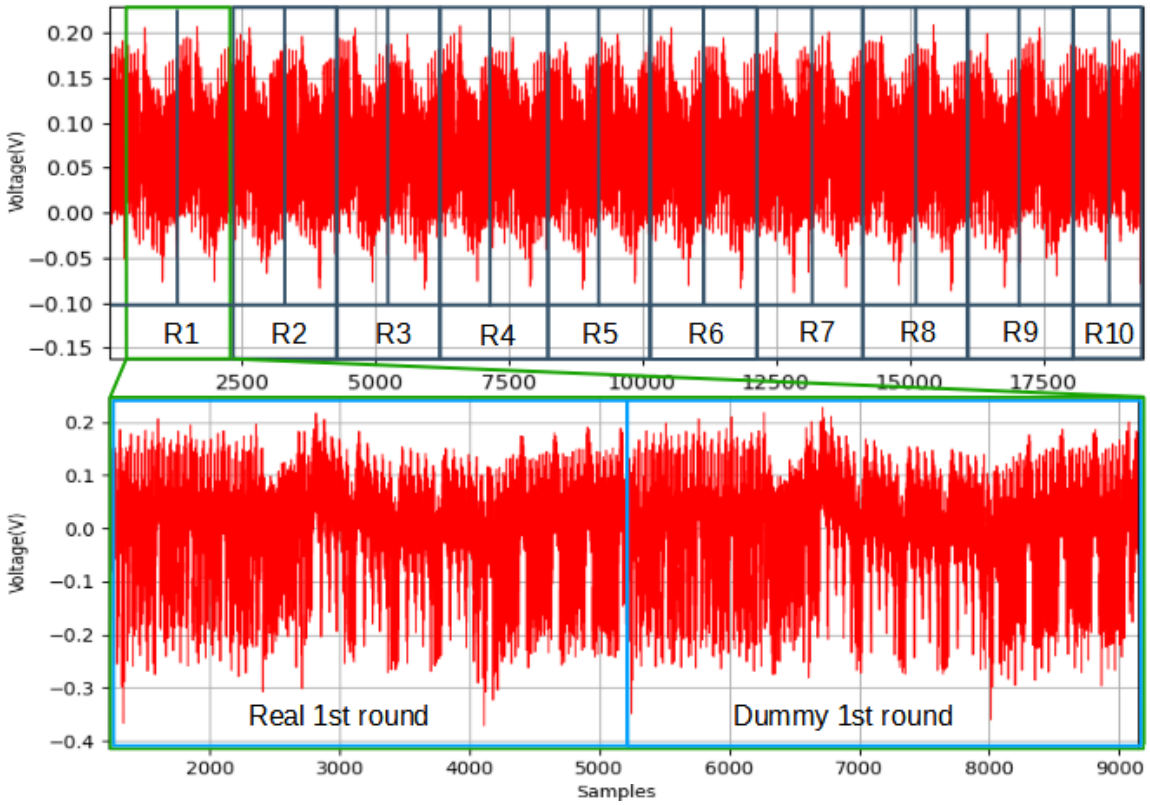

Figure 15. SPA on the single dummy round implementation

Figure 15 shows the SPA carried out on the single dummy round implementation. The implementation executes 20 consecutive rounds, 10 of which are real and the rest are dummy. As can be seen, the AES rounds were differentiated in the figure. Every two rounds a real round and a dummy round is processed. In fact, an augmented view of the first pair is offered, where the power consumption of the real round and the dummy round

are shown to be identical. This zoomed region is where the CPA will be carried out for the single dummy round implementation.

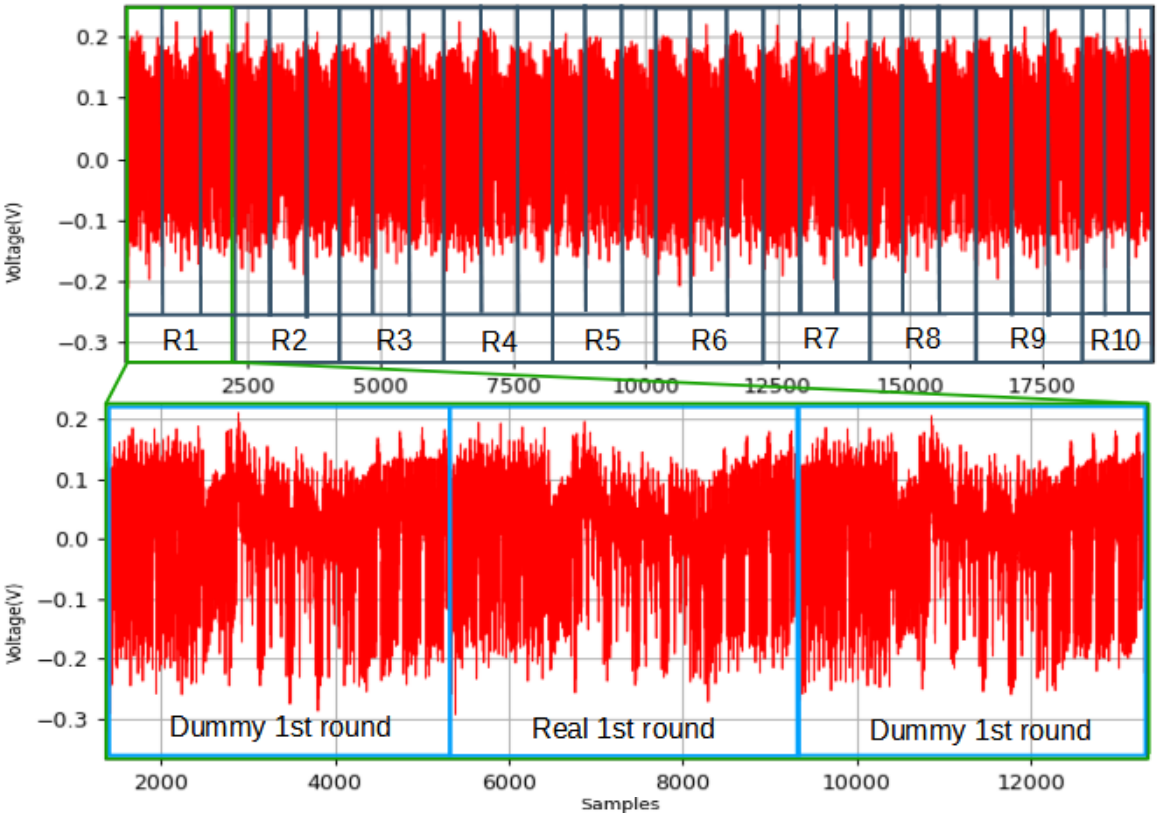

Figure 16: SPA on double dummy round implementation

Figure 16 shows the SPA done for the double dummy round implementation. With the double dummy round protection, a total of 30 rounds (20 dummy and 10 real) are executed during each encryption. Every three rounds, two dummy executions are processed, while the resting one contains real data. The zoomed view of the first three rounds shows how the real round located in the middle has the same power profile compared to the dummy rounds located before and after it. This is the sample space where the corresponding CPA will be performed.

As a last step, the full encryption of the single and double dummy round implementations, together with the encryption of the unprotected AES, were captured again. However, this time the power acquisitions were made with a common ADC frequency of 4.92MHz. The captures are shown in Figure 17:

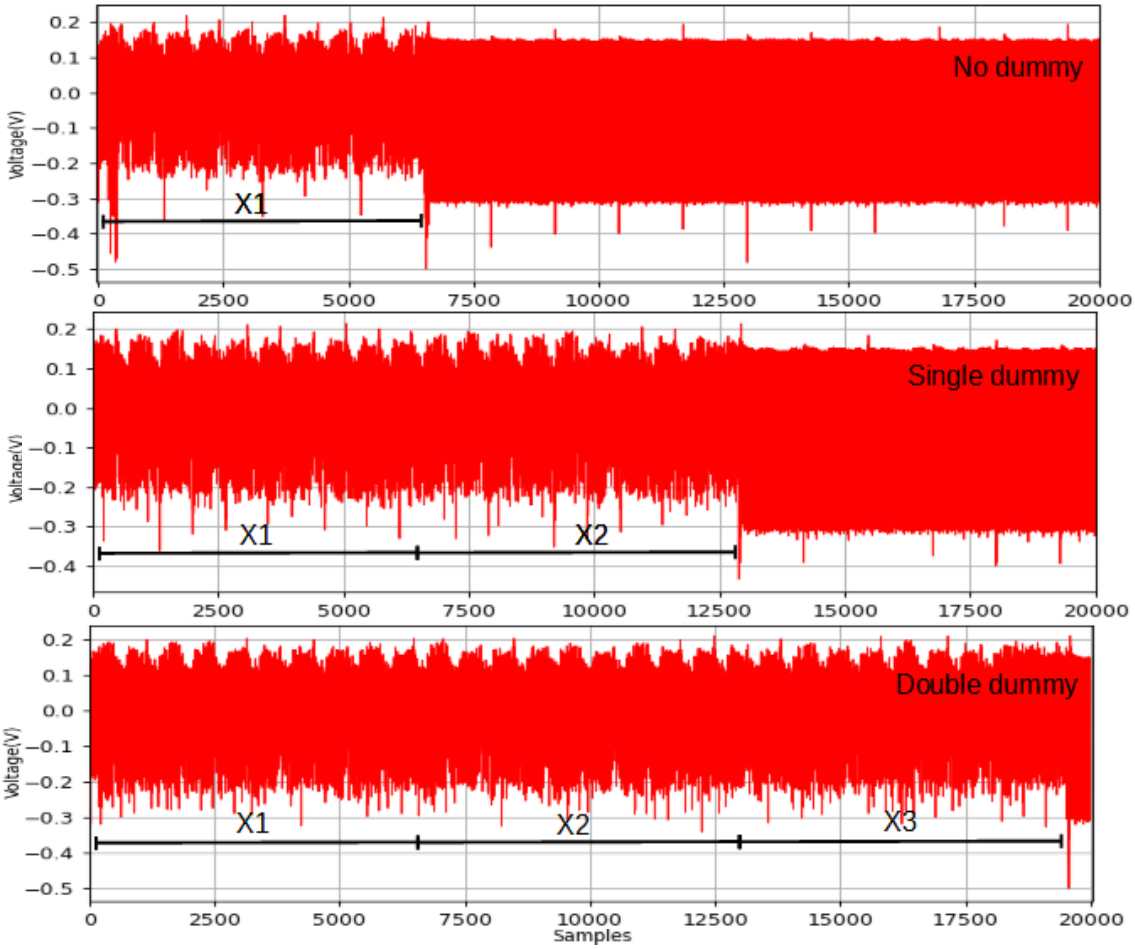

Figure 17. AES length comparison for the unprotected, single dummy and double dummy

As expected, compared to the unprotected AES case, the single dummy round implementation roughly doubles the required time for encryption (if the initial *AddRoundKey* had a dummy pair the duration would be exactly the double). In the case of the double dummy round, the encryption length is almost triplicated. The measured encryption lengths are gathered in Table 8, given in DUT clock cycles and milliseconds.

Implementation	Clock cycles (cc)	Time (ms)
Unprotected	9780	1.325
Single dummy	19280	2.612
Double dummy	28780	3.899

Table 8. Encryption length for unprotected, single dummy and double dummy

CPA on dummy round implementations

A CPA with 1000 traces was conducted for the single dummy implementation and another with 3000 traces for the double dummy implementation.

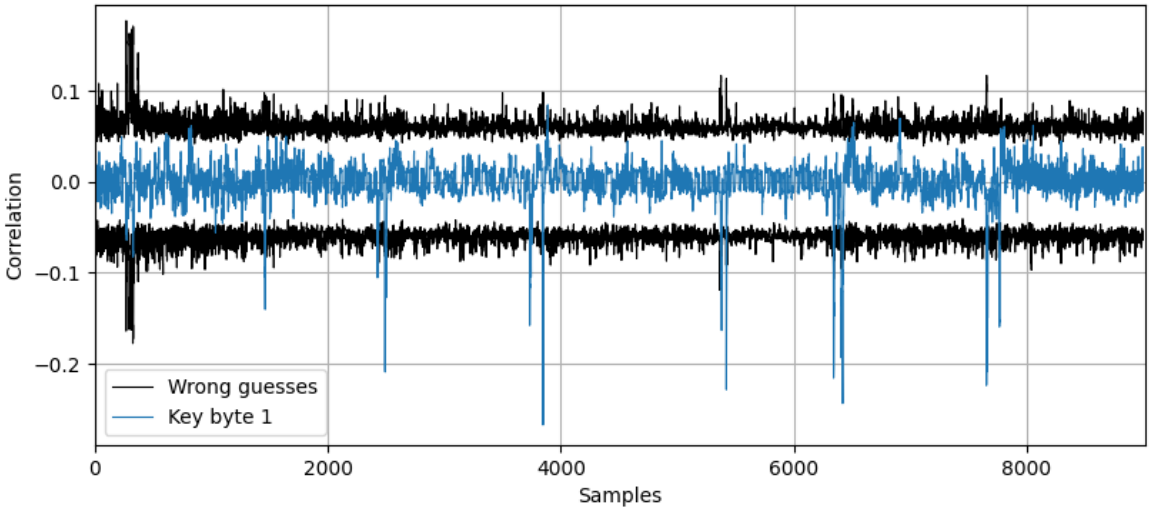

Figure 18: Correlation vs. time for key byte 1 on single dummy round implementation

Figure 18 shows the correlations obtained for key byte 1 when the single dummy implementation was attacked. In comparison with Figure 11, where the same data was shown for the unprotected implementation case, the single dummy implementation resulted in the double amount of peaks. As expected, the correlated data was found both in the first and second rounds of the AES because the order of execution of the real and fake rounds is mixed. The first three peaks correspond to the first round, while the last three to the second.

For the double dummy round implementation, the leaking behavior was the same, but extended to three rounds. Nine correlation peaks were found for the key byte 1, meaning that the correlated data was found in all of the initial three rounds.

Table 9 gathers the figures of merit for the dummy round insertion implementations:

Implementation	Correlation	Traces	K	Retrieved
Unprotected (first round)	0.918	46	1	Full key
Single dummy	0.341	580	3.551	Full key
Double dummy	0.203	1670	6,025	Full key

Table 9. CPA results for single dummy and double dummy implementations

In comparison with the unprotected AES, the single dummy round required at least 13 times more traces for the key retrieval, while this number increased to 36 when two dummy rounds were inserted. Moreover, seems that the value of the protective factor obtained grows linearly with the amount of dummy rounds inserted. Following this line of thought, if three dummy rounds were inserted per round, one could expect to obtain a k value of approximately 9. Therefore, the number of traces needed to break the algorithm with that countermeasure could be predicted.

Regarding the correlations obtained, there is a leakage reduction with the dummy rounds respect to the unprotected implementation. Indeed, this is what makes the CPA attack slower. The dummy data inserted obfuscates the correlation analysis results and therefore, more traces are needed in order to differentiate the correlations of correct byte hypothesis from the wrong ones. This leakage reduction is still bigger with the double dummy round protection, resulting in a slower CPA attack (i.e. more traces).

Another thing to take under consideration is the sample amount used in each attack. Since the attacker does not know where the real data is located, the single dummy and double dummy implementations had to be attacked with two times and three times more samples, compared to the unprotected case. Twice amount of samples, make the CPA twice slower. While three times more samples, make the CPA three times slower.

3.3.2.2. Shuffling

As a first approach, the SBOX-es of the *SubBytes* function were randomized. Similarly to the dummy round insertion, the SBOX shuffling is generated independently for each encryption of the AES. The objective is to desynchronize the target attack point in each trace and, thus, a higher number of traces will be required to correlate back the traces with the correct key value.

The implementation uses a function that generates a shuffling array where the order of the 16 SBOX executions is stored for each AES round. This array is precomputed before each encryption process. As a result, the probability of two traces having the same SBOX operation at the same instant of time is of $1/16$.

As a next step, one more possibility of AES randomization was implemented following the same rules: the column order on which MixColumns operates. The case of the MixColumns randomization is similar to the one of the SBOX-es, only that with the following two differences:

- The attack region for the CPA was limited to the *MixColumns* operation of the first round, not *SubBytes*.
- The randomized feature was the column order in which the *MixColumns* function operates. Therefore, the probability of the applied randomization is of $1/4$, instead of $1/16$.

The final implementation can be found in Appendix B3. This implementation also includes the shuffling of the *AddRoundKey* operation, applied identically respect to the SBOX randomization.

SPA on shuffled implementation

In Figure 20, the total number of the 10 rounds of the algorithm can be seen, same as in the case of the unprotected implementation. Considering that only shuffling was applied (no dummy operations were added), one could expect to see a really similar power profile compared to the unprotected AES implementation. However, in the zoomed view offered of the first round, the power profile looks quite different to the ones obtained for previous implementations.

The reason for this difference is that the function used for the shuffling required the state matrix to be an unidimensional array of length 16, unlike the two dimensional 4x4 array we had before. As a result, the power profile of the round operations, except for *ShiftRows* (which is the same), is not grouped in rows or columns anymore. On the contrary, the byte level operations, inside each transformation, now can be seen as 16 sequential patterns. Besides, this modification resulted in a slower processing of the round operations.

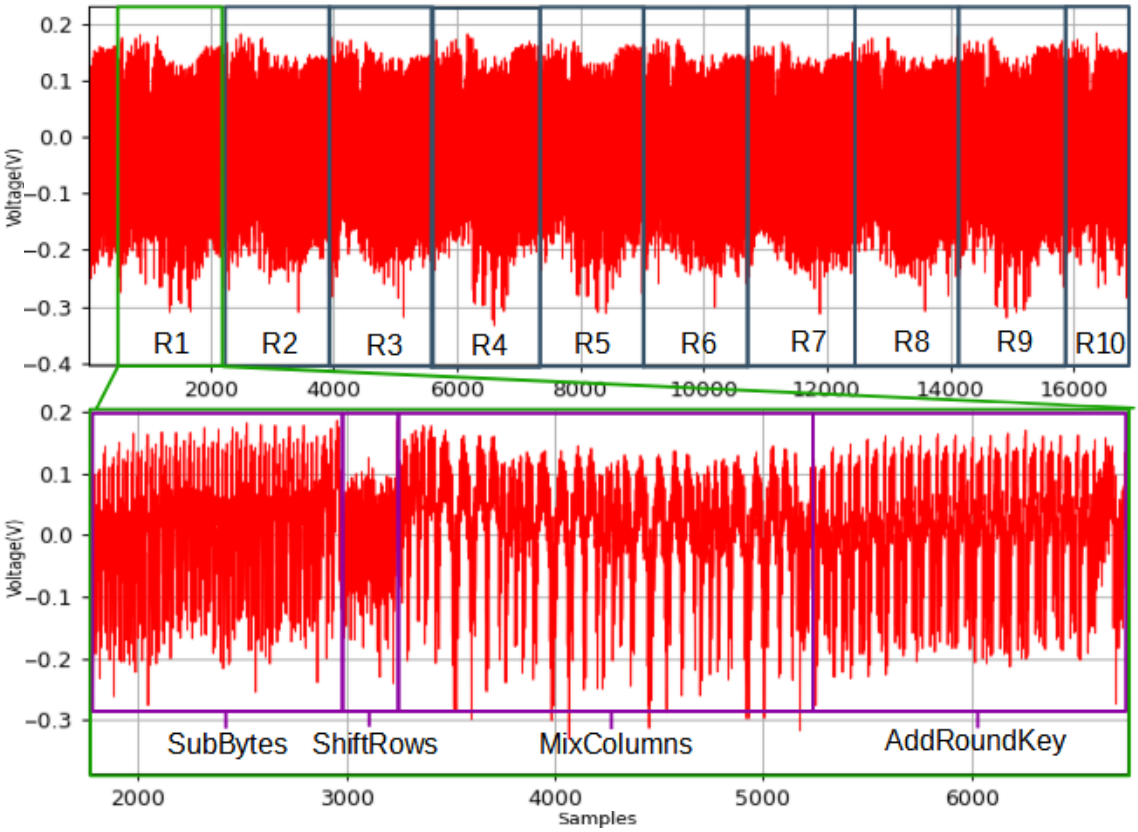

Figure 20. SPA on randomized implementation

The duration of the encryption was measured and is shown in Table 10:

Implementation	Clock cycles (cc)	Time (ms)
Unprotected	9780	1.325
Shuffled	12705	1.721

Table 10. Encryption length for the unprotected and randomized implementations

CPA on shuffled SBOX and MixColumns

A CPA with 30.000 power traces was performed on the shuffled SBOX-es. The correlations plot obtained for the key byte 1 can be seen in Figure 21:

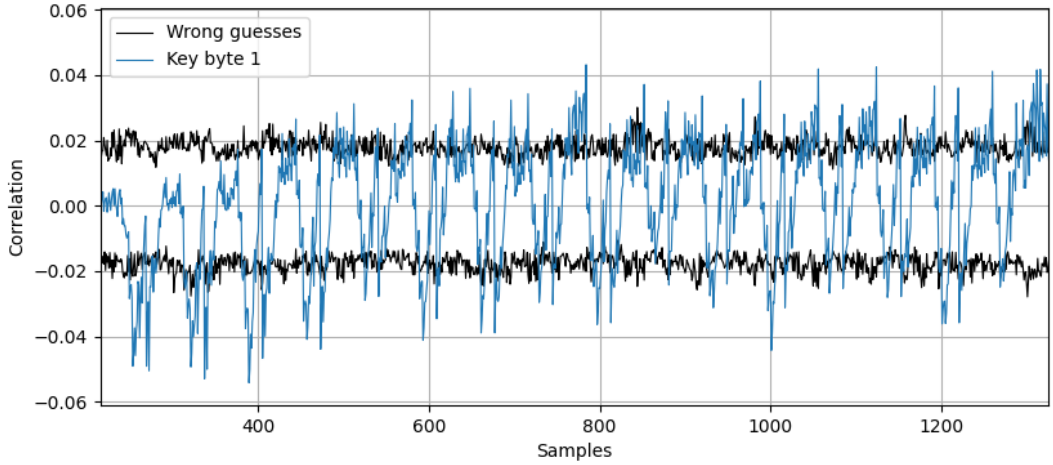

Figure 21. Correlation vs time for key byte 1 with SBOX shuffling

Compared to the non-randomized case, the Pearson coefficients obtained are really small indicating that the protection against SCA added by the shuffled SBOX is considerable. Table 11 gathers the figures of merit obtained for the CPA on the randomized SBOX-es, where the maximum correlation is given with a value of only 0.05:

Implementation	Correlation	Traces	K	Retrieved
Non-shuffled SBOX	0.753	81	1	Full key
Shuffled SBOX (1/16)	0.05	21110	16.14	Full key

Table 11. CPA results for the shuffled SBOX-es

A CPA with 2000 traces was conducted on the randomized *MixColumns* operation and Table 12 gathers the results obtained:

Implementation	Correlation	Traces	K	Retrieved
Non-shuffled MixColumns	0.812	61	1	Full Key
Shuffled MixColumns (1/4)	0.229	940	3.92	Full Key

Table 12. CPA results for the shuffled MixColumns

From the obtained results, we can deduce that the factor k is proportional to the inverse probability of the applied randomization (i.e.$1/p$). In the case of the shuffled SBOX-es the probability of two traces having the same SBOX operation at the same time spot is of $1/16$ and k approximates to 16. In the case of the MixColumns randomization, for a probability of $1/4$, k approximates to 4.

Note that the shuffled countermeasure only protects the shuffled operation. If we only randomized the *SubBytes* (SBOX-es) operation, but we attacked the entire first round, the leakage from the *ShiftRows* and *MixColumns* operations would be the same, resulting in the retrieval of the key in those operations as well as in the unprotected implementation. On the contrary, the dummy round insertion analyzed before, offered protection at round level of the AES.

3.3.2.3. Random delays

The random delay countermeasure aims at desynchronizing the time instant where data is manipulated in each trace. This is done is by adding random "*sleep()*" functions in the code. Note this is a countermeasure that adds less overhead to the AES execution in comparison with the dummy rounds, as we only insert small delays instead of adding full rounds to the execution.

The implemented delay function consists of a dummy loop where a random value is introduced and then decremented until the accumulator reaches zero. No further code of the AES encryption is executed until the sleep loop is done. Therefore, each delay adds a variable time shift depending on the random values introduced into the function.

Regarding the specific time shifts generated by the delays, the function adds a constant delay of two clock cycles, plus three cycles per each value decremented. This granularity could not be further reduced for the target MCU (despite AVR assembly instructions were used to optimize it).

For instance, if trace one had a delay where 4 is decremented to 0 and trace two had a delay (introduced at the same point in time) were 8 is decremented to zero, both traces would suffer a desynchronization of 12 clock cycles (3x4=12) between them. Hence, the desynchronization between traces is proportional to the difference of the decremented values.

Regarding the constant delay of two cycles added by the delay function, it does not contribute with any misalignment between the traces because it is applied equally to all of them. As a result, the constant part of the delay is just adding an unavoidable overhead to the encryption duration (2 clock cycles per delay inserted).

The experimental analysis for this countermeasure was done in two steps:

- Analysis of the protection offered by a single uniformly distributed random delay.
- Analysis of the protection offered by multiple random delays.

The implementation for the random delays can be found in Appendix C4.

3.3.2.3.1. Single delay analysis: plain uniform delay

As far as the random value generation is concerned (i.e. random values that are introduced in the delay function), the classical and straightforward method is to generate individual delays independently with values uniformly distributed in the interval $[0, a]$ for some $a \in N$. We refer to this method as plain uniform delays.

Considering the implemented delay function, the plain uniform delays are generated by precomputing some random values $x \in [1, a]$ and then passing them one by one to the delay function every time this is called. The bigger that a is, the more desynchronization that the power traces will suffer.

As a first step, one single random delay was placed before the initial *AddRoundKey* of the AES encryption. In order to analyze its protective effectivity, increasing values of a were fixed for the random variable generation (i.e enlarging the window of the uniform distribution) and, for each case, a CPA was carried out on the first round.

For the following experiments, not all the key was possible to retrieve in many cases. Therefore, in order to have a comparable minimum amount of traces, this quantity was targeted for the key byte 10 (which could be retrieved in the majority of the experiments).

SPA on desynchronized traces by a single random delay

Figure 22 shows three overlapped power traces, corresponding to the first two rounds of the AES128. The power profile is the same for of all three traces plotted. However, each of them has a different delay length applied, making each of them to be desynchronized respect to the others. When a CPA is conducted in this situation, the samples do not match between traces, making it more difficult for the attacker to correlate the power consumption.

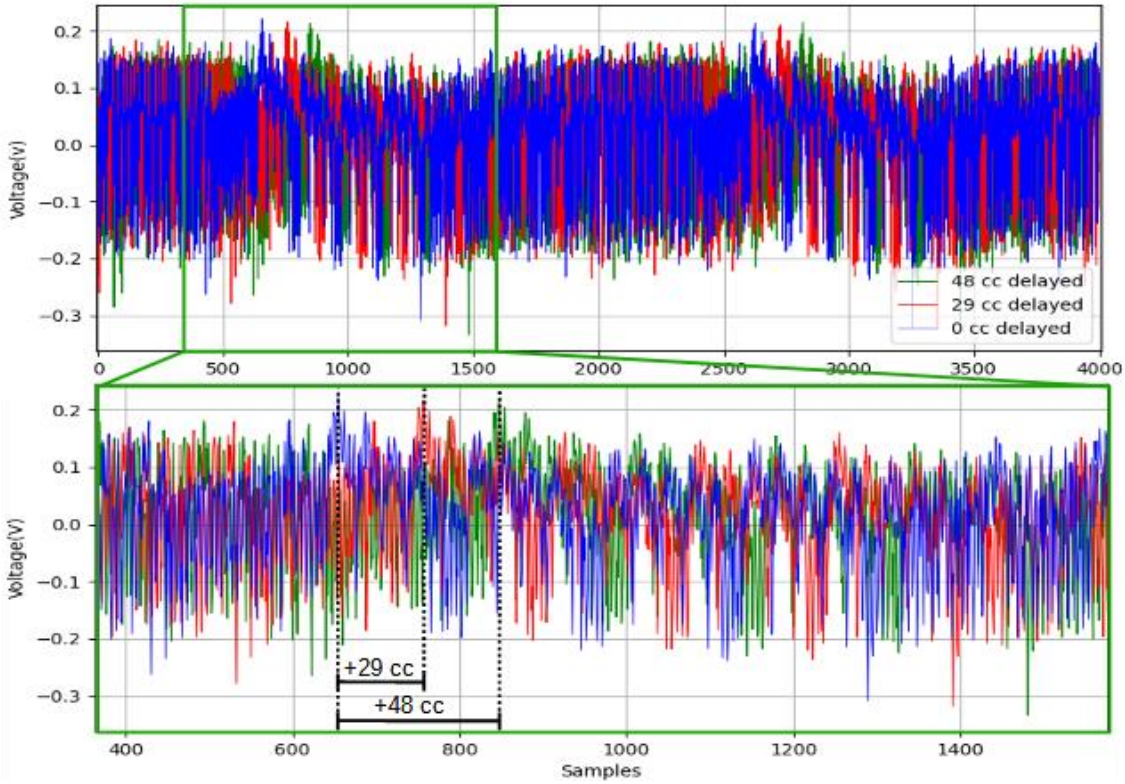

Figure 22: SPA on desynchronized traces by plain uniform delay

Table 13 gathers the total overhead added:

Uniform distribution	Clock cycles (cc)	Time (ms)
No delay	9780	1.3252
[1,2]	9788	1.3262
[1,3]	9791	1.2366
[1,4]	9794	1.3271
[1,5]	9797	1.3275
[1,6]	9800	1.3279
[1,7]	9803	1.3283
[1,8]	9806	1.3287
[1,9]	9809	1.3291
[1,10]	9812	1.3295
[1,63]	9974	1.3514
[1,127]	10166	1.3775
[1,255]	10547	1.4291

Table 13. Encryption length for AES128 with a single plain uniform delay

It can be seen that the maximum overhead added by the random delay depends on which is the distribution length used. However, in general, the total added overhead to the encryption is quite small in all cases.

CPA on desynchronized traces by a single random delay

The CPA results for the single plain uniform random delay are shown in Table 14:

Uniform distribution	Correlation	Traces	K	Retrieved	Abnormal behaviour
No delay	0.918	46	1	Full key	None
[1,2]	0.534	280	2.467	Full key	None
[1,3]	0.437	700	3.901	Full key	None
[1,4]	0.242	1560	5.823	15 key bytes	Key byte 3
[1,5]	0.218	2090	6.740	12 key bytes	None
[1,6]	0.168	3500	8.723	11 key bytes	Key byte 3
[1,7]	0.112	6600	11.978	12 key bytes	None
[1,8]	0.088	7500	12.768	11 key bytes	Key byte 3
[1,63]	0.016	100000	46.62	1 key byte	Key byte 3

Table 14. CPA results for single plain uniform delay analysis

It can be observed that the random delay works really well even if only one delay is inserted per encryption. Already for the [1,8] distribution, the amount of required power traces is 163 times bigger than for the unprotected implementation and besides, not all the key bytes could be retrieved with this amount.

When the [1,5] distribution was attacked, retrieving the key bytes related to the first row of the AES started to be more and more difficult. This is why a maximum amount of 12 key bytes were retrieved in many cases. In each case, k was calculated for the last retrieved key byte.

Apart from that, the experiment for the distribution [1,63] resulted in only one key byte ranked at first position. Note that this CPA was performed with 100.000 traces, denoting that the protective effectivity is huge for this countermeasure.

Moreover, the only key byte that was found at first position had a maximum correlation of 0.016, while the wrong guesses had correlations of 0.015. In other words, the correlations obtained for the wrong guesses were quite similar to the correlation of the right guess. In this situation, an attacker that does not know which the value of the right guess is (unlike in this project where the key is known), could not be sure of having retrieved the correct key byte.

In overall, the protective power of random delays as hiding countermeasure surpasses that of the dummy rounds or shuffling. In addition, with a correct positioning of the delays, the encryption can be protected from its beginning to the end.

However, there is a problem related to this results. A single random delay placed at the beginning of the encryption is easy to identify and, therefore, easy to correct. The attacker can use pattern recognition techniques in order to resynchronize all the traces. As explained in the following section 3.3.2.3.2, multiple delays can be placed at different points in time in order to make the resynchronization of the traces more difficult.

Note that in Table 14, an extra column was added named “Abnormal behavior”. During the different experiments made, an abnormal leaking behavior was observed for key byte 3. With this behavior, either for the correct and wrong guesses, the resultant correlations were much higher than for the rest of the key bytes.

In Figure 22, the correlations obtained for key byte 1, with normal leaking behavior, can be seen when a single plain uniform delay of distribution [1,63] was introduced:

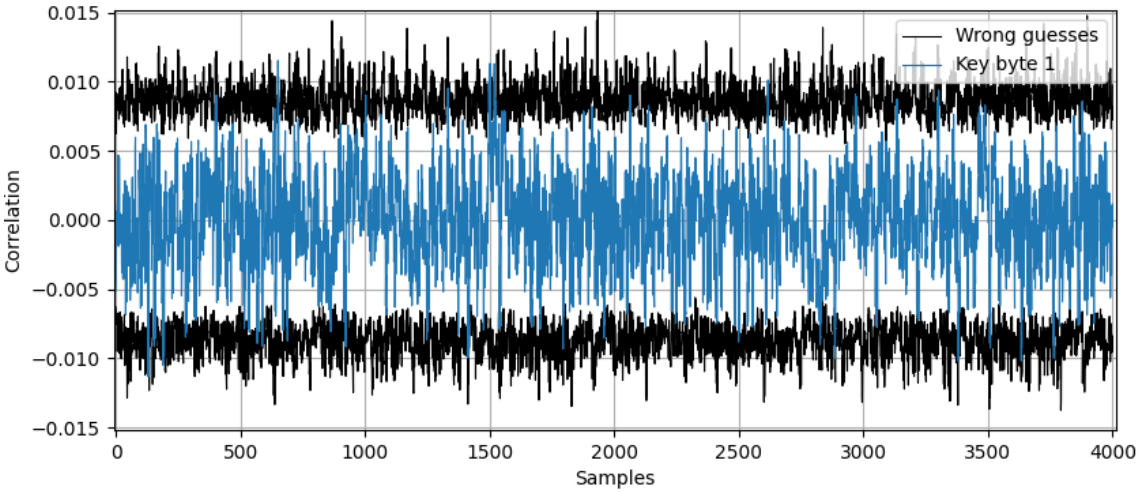

Figure 22: Correlation vs. Time for key byte 1 with normal leaking behavior

In Figure 23, for the same delay experiment, the correlations obtained for key byte 3 with abnormal leaking behavior can be seen:

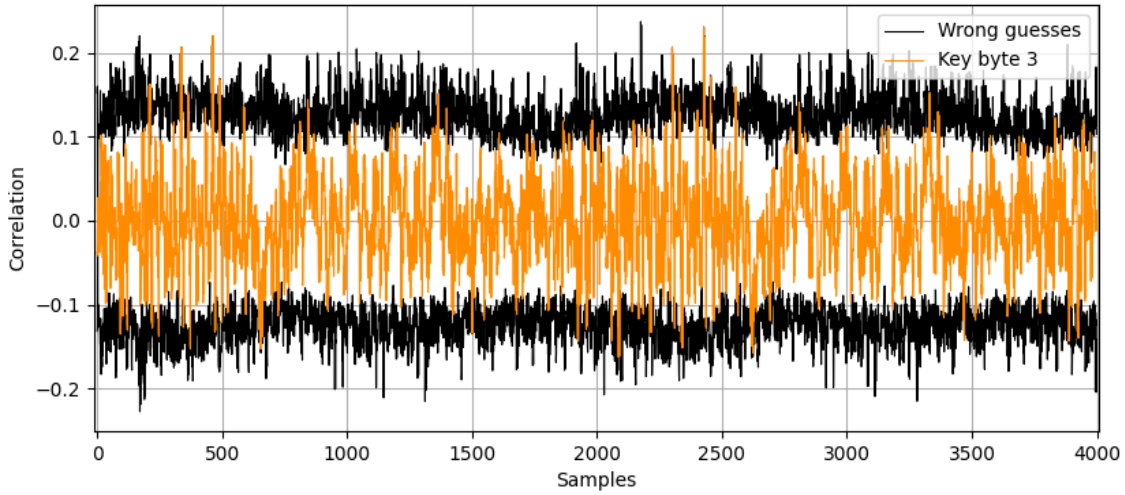

Figure 23. Correlation vs. Time for key byte 3 with abnormal leaking behavior

It can be seen that the correlation values in the abnormal case are much higher than those of the normal case. Most of the times, when the key byte showed this behavior, it could not be retrieved, or was the last being retrieved at least. Consequently, it is possible to say that, even if higher correlations are obtained, the abnormal behavior does not bring any advantage to the attacker.

The behavior was related to key byte 3 in all the experiments; however, the reason behind this conduct could not be cleared out.

3.3.2.3.2. Multiple delay analysis

We have seen that a single and large delay placed at the beginning of the encryption works really well for the trace desynchronization. However, from an attacker's point of view, this protection is easy to unmake, since there is only one desynchronization point to be considered. In other words, it is easy to identify the point where delays are introduced and, as a result, its effect is easy to correct.

For this reason, delays are rarely used in one single place. Random delays are usually implemented with short lengths and placed at different points of the algorithm. Therefore, in each execution run, variable time shifts will be applied at all those points.

The objective is to break the trace with relatively short delays in multiple places. This way it is more difficult for an attacker to resynchronize the traces, since the total desynchronization is a combination generated by multiple desynchronization points.

Consequently, an attacker usually faces the sum of several random delays. The delays that will affect the SCA attack are the ones placed between the triggering of the power measurement (i.e. beginning of AES encryption) and the attack point (i.e. SBOX output).

CPA on desynchronized traces by multiple random delays

For the experiments carried out, the uniform distribution was fixed at [1,3]. The CPA attacks were performed in the case-scenarios where 3, 4, 5, 6 and 7 delay functions were added. The delay functions were inserted before, during and after the initial *AddRoundKey* transformation.

Note that in a real implementation the delays would be placed strategically across all the encryption of the AES. However, considering the attack point chosen and the analytic purpose of this experiment the case-scenarios defined are adequate enough.

In each case, the CPA was conducted targeting the first round. The results are shown in Table 15:

N	Correlation	Traces	K	Retrieved	Abnormal beavior
0 (no delay)	0.918	46	1.00	Full key	-
3	0.141	3200	8.34	15 key bytes	Key byte 3
4	0.122	5600	11.03	11 key bytes	Key byte 3
5	0.109	8200	13.35	11 key bytes	Key byte 3
6	0.091	9700	14.52	11 key bytes	Key byte 3
7	0.061	10500	15.108	11 key bytes	Key byte 3

Table 15. Results for multiple plain uniform delay analysis

It can be seen how the the minimum amount of traces grows together with the number N of delays in the cummulative sum. However, seems that the more delays that are added into the sum, the less that the protective effectivity grows.

3.3.3. AES with Boolean masking

The masking countermeasure pursues a very different objective than previous ones. The objective is not to desynchronize the manipulation of sensitive data among different executions but to completely decorrelate them by modifying these sensitive data to something which cannot be guessed by an attacker in a CPA.

As explained in the State of the Art section 2.3.2, a masking countermeasure consists basically in XORing the input data of the algorithm with a random unknown value. Then, the algorithm has to be modified to keep the same encryption process:

$$AES(input, key) = AES'(input \oplus mask, key) \quad (22)$$

Now, in order to mount the CPA attack, the attacker has two unknown variables, the mask and the key, and the first one is randomly changing in each execution so it cannot be guessed. With this countermeasure, the power consumption of the AES' at the attack

point is proportional to $SBOX(input \oplus mask \oplus key)$ and not to $SBOX(input \oplus key)$ as before. Since the mask value is random and unknown, the attacker does not know to which data correlate the power traces.

Note that, as the attacker does not know the mask value, he simply will try to correlate the power traces with $SBOX(input \oplus key)$, like before, and hope that the masking implementation is badly done so that the Pearson correlation could still be found and the secret key could still be recovered with some number of traces. Therefore, the objective of this part of the project is to implement correctly the masking process in order to completely hide the leakage.

3.3.3.1. Boolean masking implementation

The Boolean masking scheme that was implemented needs only 10 masks to work. There are 6 masks ($m_1, m_2, m_3\ m_4$, m_5 and m_6) that are randomly generated before the encryption of the AES and four masks (m_1', m_2', m_3' and m_4') that derive from the first four.

Regarding the random mask generation, each mask produced is a random value ranging from 0x01 to 0xFF. The mask 0x00 was avoided due to this value hacing no effect when XOR-ed with any byte.

Since the masking is done row by row, the *ShiftRows* transformation does not have any effect on the masks. On the contrary, the *MixColumns* transformation combines the bytes from different rows and acts linearly on the masks. This is why, before the encryption, the *MixColumns* function is applied to four of the initial masks obtaining:

$$MC(m_1) = m_1' \qquad MC(m_2) = m_2' \qquad MC(m_3) = m_3' \qquad MC(m_4) = m_4' \tag{23}$$

The two remaining masks (m_5 and m_6) are used to compute the masking of the *SubBytes* transformation. A LUT of 256 components has to be constructed in order to store every byte value that meets equation (25):

$$SB(x \oplus m_4) = SB(x) \oplus SB(m_5) \tag{24}$$

The masking principle used for the implementation is quite simple, but has every intermediate value masked for all the encryption process. Remember that this is the indispensable requirement for a masking scheme to be effective.

The implemented masking scheme is better explained in the block diagram of Figure 24 where the mask modifications for each step of the AES encryption are shown.

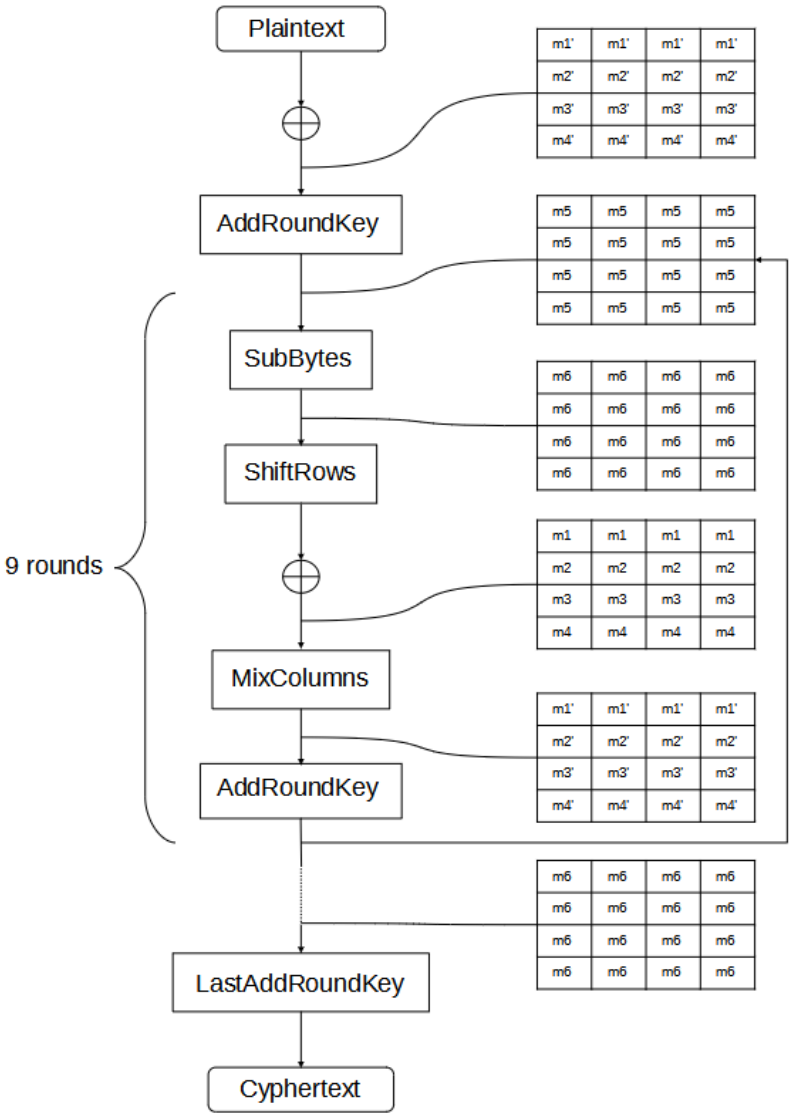

Figure 24. Simple Boolean masking scheme

In Figure 24, we can see how the *AddRoundKey* functions are used to transform the masks, in addition to their original function. Every round key is mixed with the corresponding masks before the encryption to achieve this. The masked round keys, except for the last one, remove the masks coming from the output of the *MixColumns* transformation and add the input mask for the *SubBytes* transformation. The last round key is the only one that is different from the others. This key unmakes the masking proceeding from the last round *SubBytes* function and thus, reveals the ciphertext.

Moreover, two XOR-ing functions have been defined in order to manipulate the masks. The first XOR is used to mask the plaintext bytes, in each of the rows of the state, before starting the encryption process. The second XOR allows the proper tracking of the masks and the intermediate variables through the algorithmic process. With this purpose, it is used in each round driving the transition from the output of the *SubBytes* function into the *MixColumns*.

The implementation for the Boolean masking is given in Appendix B5.

SPA on Boolean masking implementation

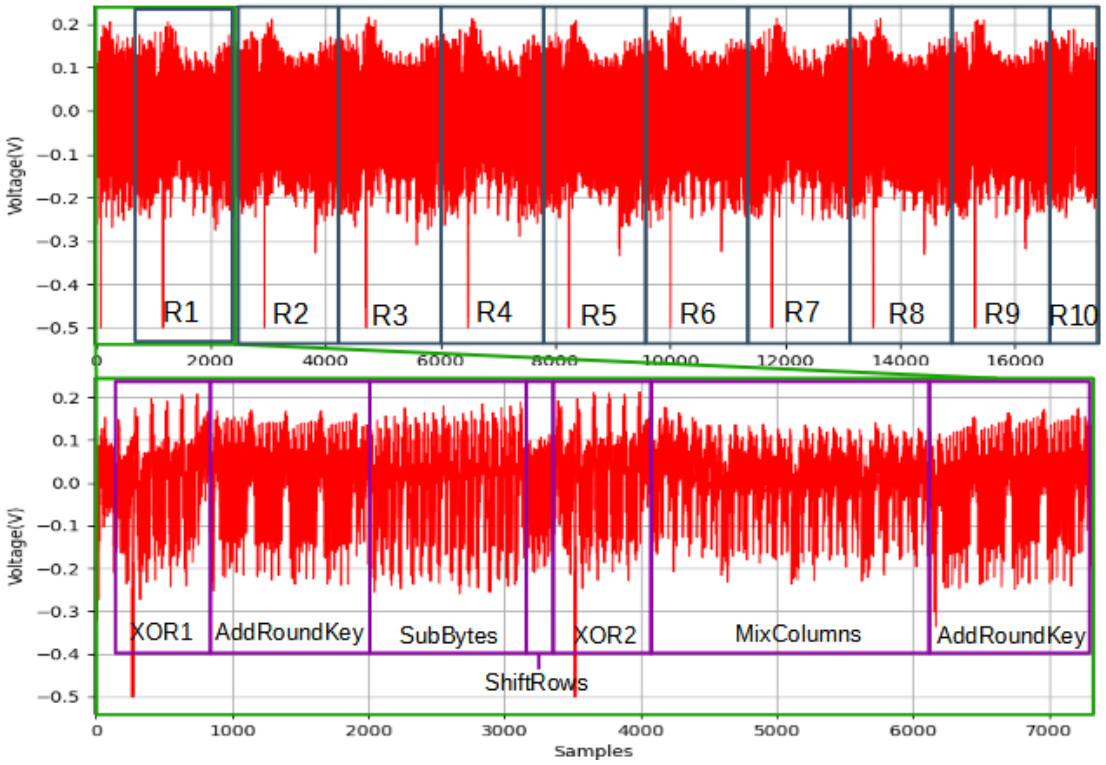

Figure 25. SPA on Boolean masking implementation

Figure 25 shows the SPA on the AES with Boolean masking implementation. The encryption is performed, like in the case of the unprotected AES, in 10 rounds. However, in the zoomed view, some differences can be found.

Before the initial *AddRoundKey* operation, the power profile of the first XOR can be seen, where the plaintext data is mixed with the masks for the first time. Then, one more identical power profile can be seen corresponding to the second XOR, where the output masks of the *SubBytes* operation are modified into the input masks of the *MixColumns* operation.

The encryption duration was measured and is given in Table 16:

Implementation	Clock cycles (cc)	Time (ms)
Unprotected AES128	9780	1.325
AES128 with Boolean masking	9935	1.346

Table 16. Encryption length for the Boolean masked AES128 implementation

The performance cost for the masking countermeasure is minimal in comparison with some hiding countermeasures previously analyzed.

CPA on Boolean masking implementation

A CPA with 100.000 traces was conducted on the masked implementation. The CPA resulted in a non-successful attack for most of the key bytes, meaning that the masking was being effective in most cases. However, one key byte could be retrieved. Results

showed that the key byte 11 leaked information at some point of the encryption process. The CPA results are gathered in Table 17:

Implementation	Correlation	Traces	K	Retrieved
Unprotected AES128	0.918	46	1	Full key
AES128 with Boolean masking	0.067	100000	∞	1 key byte

Table 17. CPA results for unprotected and Boolean masked implementations

Figure 26 shows the correlations plot for the perfectly masked key byte 1:

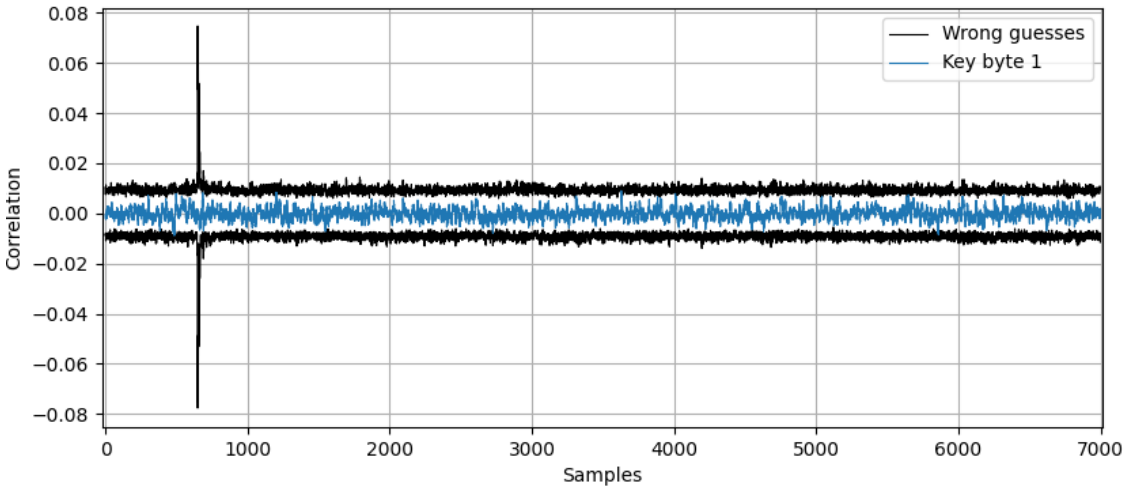

Figure 26. Correlation vs. time for a perfectly masked key byte 1

It can be seen how the correlations for the right guess (blue) never exceed the values of the wrong guesses (black). Hence, no leakage was found for the correct byte guess meaning that the masking is effective for this key byte.

Nevertheless, a black correlation peak (positive and negative) appeared for the wrong guesses. These kind of peaks generated by wrong guesses were found for all the key bytes during this experiment with the masked implementation. Maybe, for same reason, adding the masks into the algorithm modified the power consumption of the device in such a way that the CPA started to obtain peaks for the wrong guesses. However, there is no evidence to support this theory from the experiments carried out.

Similarly to the abnormal behavior reported in section 3.3.2.3.1, the reason behind these peaks could not be cleared out.

We have already seen the correlations obtained for a perfectly masked key byte in Figure 26. The following Figure 27 shows the correlation plot for the leaking key byte 11 (note that the correlation peaks for the wrong guesses appear again):

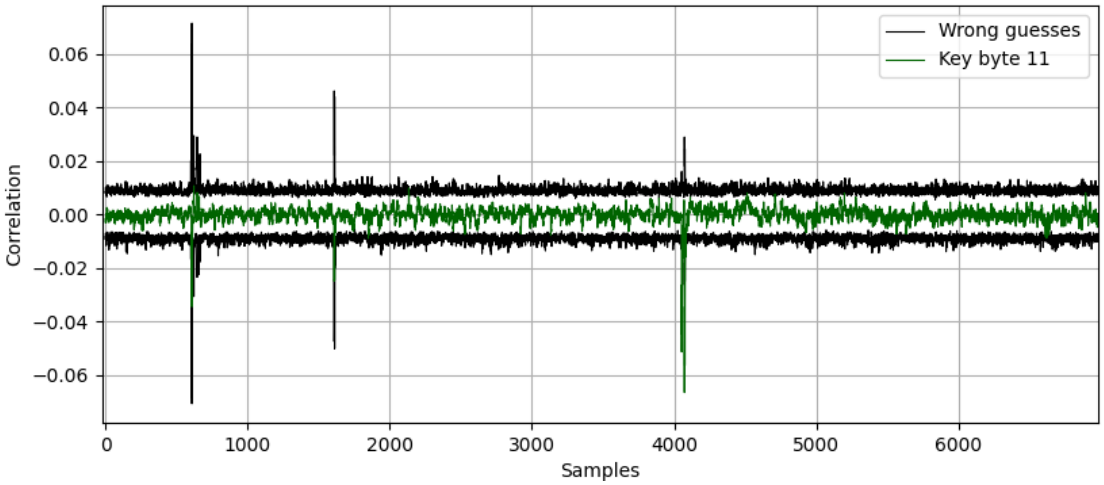

Figure 27. Correlation vs. time for the only leaking key byte 11

In this case, the CPA for the key byte 11 resulted in a leakage peak for the correct guess located around sample 4000. Seems that the key byte leaked information before the *MixColumns* transformation when the masks m_6 are exchanged for the masks m_1, m_2, m_3 and m_4 through an XOR operation. Nevertheless, if the problem was the XOR-ing operation, this behavior should be seen for all the key bytes, which is not the case. The implementation that was developed treats every key byte in the same way; however, for some reason only the key byte 11 showed this leaking behavior.

One possibility is to consider that there is an intermediate value, or mask, of 0x00 that falls more than once at the point in time where the information is leaked. However, the plaintext values from which the intermediate values derive are random and the masks generated are also random (no 0x00 mask produced).

Figure 28 and Figure 29 show the correlation evolution of key byte 1 and 11 during the CPA:

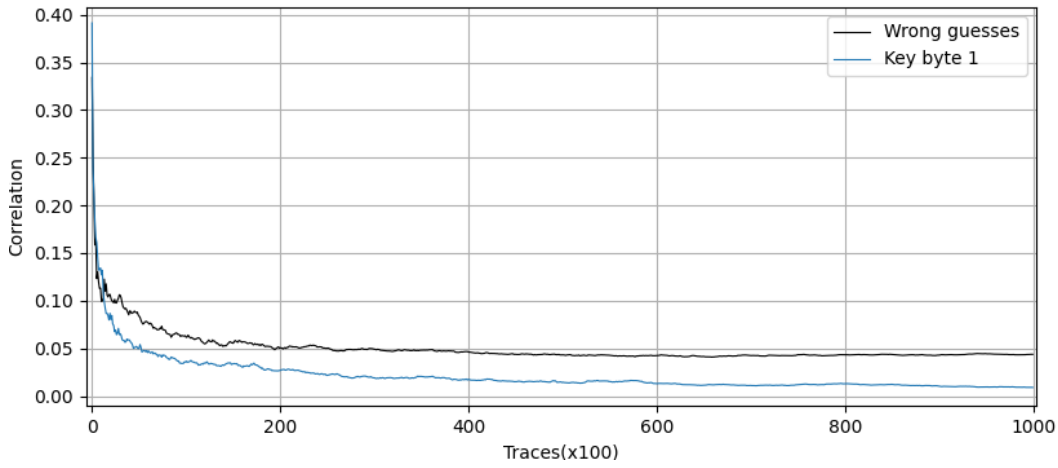

Figure 28. Correlation evolution during the CPA for perfectly masked key byte 1

The correlations obtained for the perfectly masked value are always below the correlations of the wrong guesses. This is what the masking achieves. It is not possible to correlate the correct byte value to the power consumption anymore because of the masks that were added. This same behavior (masked behavior) was observed for all key bytes, except for the key byte 11 as seen below:

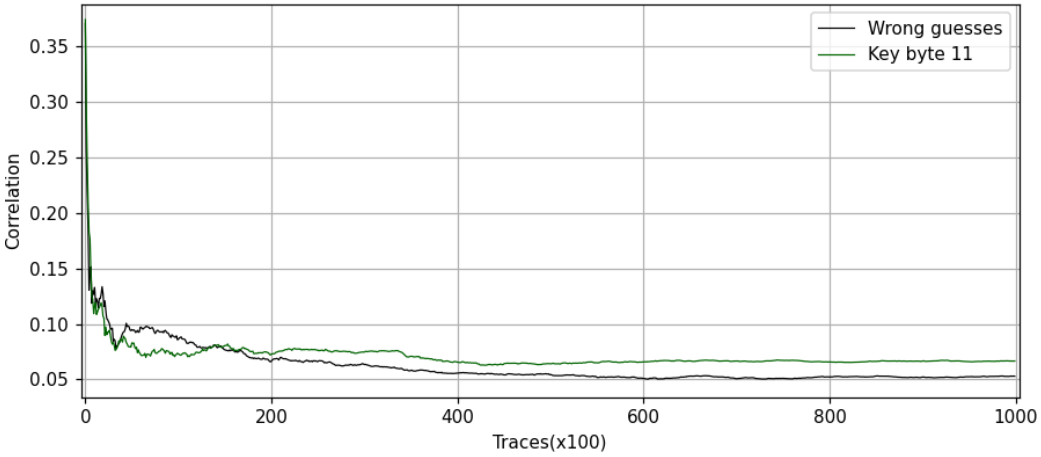

Figure 29. Correlation evolution during the CPA for leaking key byte 11

Key byte 11, on the contrary, already with 20000 traces resulted in correlations for the right guess over those of the wrong guesses. Now, even if this is true, in comparison with the unprotected implementation case, we can see that the correlations obtained are quite small and the difference between the right and wrong guesses is small as well. Consequently, the masking is not perfect for this key byte case, but it is still partially protective.

Therefore, it can be stated that the Boolean masking scheme that was implemented offers an almost perfect protection against first order SCA.

4. Conclusions and future work

4.1. Conclusions

Side Channel attacks have been proven as the most effective and powerful hardware attacks nowadays. With the right equipment and knowledge, all type of confidential information can be extracted from operating devices. However, if the underlying nature of SCA is understood, diverse types of countermeasures can be designed in order to thwart these attacks.

This work was focused on implementing and analyzing side channel countermeasures in order to assess their protective effectivity. In addition, their related overhead drawback was considered as well.

Implementation	Correlation	Traces	K	Retrieved	Overhead
Unprotected AES	0.918	46	1	Full key	+0%
Single dummy	0.341	580	3.05	Full key	+97.12%
Double dummy	0.203	1680	6	Full key	+294.26%
SBOX shuffling	0.05	21110	16.14	Full key	+29.81%
MixColumns shuffling	0.229	940	3.92	Full key	+29.81%
Random delay [1,63]	0.016	100000	46.62	1 key byte	+1.96%
Boolean masking	0.067	100000	∞	1 key byte	+1.58%

Table 18. Comparison of overall countermeasure results

Table 18 summarizes the most relevant results obtained in section 3. All the implemented countermeasures improved the cryptographic security of the AES algorithm and all of them showed a related performance cost.

The dummy round insertion offers a relatively low protection, in comparison with the big performance overhead added to the execution of the device. The shuffling, otherwise, can offer decent protective features with low performance costs; however, its protection is limited to the shuffled operation.

The random delay insertion can offer a great protection, together with low performance drawbacks. Either implementing large variable delays or short and multiple delays, resulted in a noticeable desynchronization of the power traces. However, unlike the rest of countermeasures, the effect of the random delays can be corrected through the resynchronization of traces.

Lastly, the implemented Boolean masking scheme was shown to be a countermeasure that offers an almost perfect protection against first order SCA. There was a single key byte that could be retrieved, while the rest of the key bytes were perfectly masked. Moreover, its related cost in performance is totally negligible considering the protection offered.

4.2. Future work

Countermeasures are an essential tool for protection against Side Channel analysis nowadays. In order to properly implement them, a developer must understand what is exactly happening with his or her implementation, how is the countermeasure offering protection and why does this protection avoid SCA from extracting sensitive information. For this reasons, many lines of future work exist related to the work done.

As a direct consequence of the results obtained from the experiments with the random delays, the investigation of the abnormal behavior that was reported is another possible future line of work. In the same manner, the correlation peaks obtained for the wrong guesses with the masked implementation could be investigated as well. Experienced workers in Applus+ have reported this same behaviors in some products tested in the Lab. However, none of them could tell what are the fundamental reasons behind them.

Regarding the shuffling, work could be done on finding an implementation that does not require the state modification from a 4x4 matrix to a unidimensional array of 16 elements. This way, the protective effectivity would be kept, while reducing the total time overhead added by the countermeasure to zero.

As far as the masking implementation is concerned, this project has covered only first order SCA attacks. Further implementations and experiments could be carried out in order to investigate second order and higher order masking schemes able to thwart higher-order CPA attacks.

Lastly, there is one more possibility for future work: hardware countermeasure implementation and analysis. This would allow the investigation of countermeasures dedicated to hide signal in amplitude dimension. The analysis done for software countermeasures should be valid for hardware designs; however, there is still a wide possibility of research in the field of hardware countermeasures that cannot be implemented in software. For instance, the implementation of the 16 SBOX operations in parallel could be done in hardware, for the processing of all bytes at once. This is a countermeasure whose effectivity would be very interesting to analyze.

5. Budget

This project was developed in Applus+ IT Laboratories. The company has extensive experience in the R&D sector in addition to a high budget for these activities. A research project can be divided into staff hours and equipment hours. Staff hours count the time that the employee has been carrying out research tasks and equipment hours are counted those that the devices have been working on in the programmed experiments.

	Staff hour	Equipment hours
Nº of hours	300	520
Total	820	

Table 20. Hours dedicated to the project

Table 20 states the total amount of hours dedicated to this project. The company considers that the costs per hour of research ascend to 45€/H. Here, only the Staff hours are considered. In this price, the costs of energy, amortization of equipment's and other expenses are considered.

Nº of staff hours	300
Price/hour	45€/h
Chipwhisperer	280€
Total	1630€

Table 21. Cost of the project

Table 21 shows the total cost of this project. As can be seen, the price of the *Chipwhisperer* board was included and as a result, a total of13.500€ has been invested to develop this project.

Bibiliography

[1] FIPS 197, Advanced Encryption Standard (AES). https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf

[2] Kocher P.C. (1996) Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz N. (eds) Advances in Cryptology — CRYPTO '96. CRYPTO 1996. Lecture Notes in Computer Science, vol 1109. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-68697-5_9

[3] P. Wright. Spy Catcher: The Candid Autobiography of a Senior Intelligence Officer. Viking Press,1987

[4] Q. Tian, M. O'neill and N. Hanley, "Can leakage models be more efficient? non-linear models in side channel attacks," *2014 IEEE International Workshop on Information Forensics and Security (WIFS)*, 2014, pp. 215-220, doi: 10.1109/WIFS.2014.7084330.

[5] Kocher P., Jaffe J., Jun B. (1999) Differential Power Analysis. In: Wiener M. (eds) Advances in Cryptology — CRYPTO' 99. CRYPTO 1999. Lecture Notes in Computer Science, vol 1666. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-48405-1_25

[6] Brier E., Clavier C., Olivier F. (2004) Correlation Power Analysis with a Leakage Model. In: Joye M., Quisquater JJ. (eds) Cryptographic Hardware and Embedded Systems - CHES 2004. CHES 2004. Lecture Notes in Computer Science, vol 3156. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-28632-5_2

[7] Messerges, Thomas & Dabbish, Ezzy & Sloan, Robert. (1999). Investigations of Power Analysis Attacks on Smartcards.

[8] Mangard, S., Oswald, E., & Popp, T. (2010). *Power analysis attacks: Revealing the secrets of smart cards*. New York, NY: Springer.

[9] Messerges T.S. (2000) Using Second-Order Power Analysis to Attack DPA Resistant Software. In: Koç Ç.K., Paar C. (eds) Cryptographic Hardware and Embedded Systems — CHES 2000. CHES 2000. Lecture Notes in Computer Science, vol 1965. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-44499-8_19

[10] Chari S., Jutla C.S., Rao J.R., Rohatgi P. (1999) Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Wiener M. (eds) Advances in Cryptology — CRYPTO' 99. CRYPTO 1999. Lecture Notes in Computer Science, vol 1666. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-48405-1_26

[11] Chari S., Rao J.R., Rohatgi P. (2003) Template Attacks. In: Kaliski B.S., Koç .K., Paar C. (eds) Cryptographic Hardware and Embedded Systems - CHES 2002. CHES 2002. Lecture Notes in Computer Science, vol 2523. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-36400-5_3

[12] Schindler W., Lemke K., Paar C. (2005) A Stochastic Model for Differential Side Channel Cryptanalysis. In: Rao J.R., Sunar B. (eds) Cryptographic Hardware and Embedded Systems – CHES 2005. CHES 2005. Lecture Notes in Computer Science, vol 3659. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11545262_3

[13] Choudary O., Kuhn M.G. (2014) Efficient Template Attacks. In: Francillon A., Rohatgi P. (eds) Smart Card Research and Advanced Applications. CARDIS 2013. Lecture Notes in Computer Science, vol 8419. Springer, Cham. https://doi.org/10.1007/978-3-319-08302-5_17

[14]Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. Side-Channel Attacks: an Approach Based on Machine Learning. In Second International Workshop on Constructive Side-Channel Analysis and Secure Design, pages 29–41. Center for Advanced Security Research Darmstadt, 2011.

[15] Zotkin, Y., Olivier, F., & Bourbao, E. (2018). Deep Learning vs Template Attacks in front of fundamental targets: experimental study. *IACR Cryptol. ePrint Arch., 2018*, 1213.

[16] Coron JS., Goubin L. (2000) On Boolean and Arithmetic Masking against Differential Power Analysis. In: Koç Ç.K., Paar C. (eds) Cryptographic Hardware and Embedded Systems — CHES 2000. CHES 2000. Lecture Notes in Computer Science, vol 1965. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-44499-8_18

[17] Oswald E., Mangard S., Herbst C., Tillich S. (2006) Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In: Pointcheval D. (eds) Topics in Cryptology – CT-RSA 2006. CT-RSA 2006. Lecture Notes in Computer Science, vol 3860. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11605805_13

[18] Joye M., Paillier P., Schoenmakers B. (2005) On Second-Order Differential Power Analysis. In: Rao J.R., Sunar B. (eds) Cryptographic Hardware and Embedded Systems – CHES 2005. CHES 2005. Lecture Notes in Computer Science, vol 3659. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11545262_22

[19] Waddle J., Wagner D. (2004) Towards Efficient Second-Order Power Analysis. In: Joye M., Quisquater JJ. (eds) Cryptographic Hardware and Embedded Systems - CHES 2004. CHES 2004. Lecture Notes in Computer Science, vol 3156. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-28632-5_1

[20] Gierlichs B., Batina L., Preneel B., Verbauwhede I. (2010) Revisiting Higher-Order DPA Attacks:. In: Pieprzyk J. (eds) Topics in Cryptology - CT-RSA 2010. CT-RSA 2010. Lecture Notes in Computer Science, vol 5985. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-11925-5_16

[21] Gierlichs B., Batina L., Tuyls P., Preneel B. (2008) Mutual Information Analysis. In: Oswald E., Rohatgi P. (eds) Cryptographic Hardware and Embedded Systems – CHES 2008. CHES 2008. Lecture Notes in Computer Science, vol 5154. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-85053-3_27

[22]Kim, J., Picek, S., Heuser, A., Bhasin, S., Hanjalic, A.: Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems 2019(3) (May 2019) 148–179

[23] Clavier C., Coron JS., Dabbous N. (2000) Differential Power Analysis in the Presence of Hardware Countermeasures. In: Koç Ç.K., Paar C. (eds) Cryptographic Hardware and Embedded Systems — CHES 2000. CHES 2000. Lecture Notes in Computer Science, vol 1965. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-44499-8_20

[24] Homma N., Nagashima S., Imai Y., Aoki T., Satoh A. (2006) High-Resolution Side-Channel Attack Using Phase-Based Waveform Matching. In: Goubin L., Matsui M. (eds) Cryptographic Hardware and Embedded Systems - CHES 2006. CHES 2006. Lecture Notes in Computer Science, vol 4249. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11894063_15

[25] M. Tunstall and O. Benoit. Effcient use of random delays in embedded software. In D. Sauveron, K. Markantonakis, A. Bilas, and J.-J. Quisquater, editors, WISTP 2007, volume 4462 of LNCS, pages 27{38. Springer, Heidelberg, 2007.

[26] J.-S. Coron and I. Kizhvatov. An e_cient method for random delay generation in embedded software. In C. Clavier and K. Gaj, editors, CHES 2009, volume 5747 of LNCS, pages 156{170. Springer, Heidelberg, 2009.

[27] Coron JS., Kizhvatov I. (2010) Analysis and Improvement of the Random Delay Countermeasure of CHES 2009. In: Mangard S., Standaert FX. (eds) Cryptographic Hardware and Embedded Systems, CHES 2010. CHES 2010. Lecture Notes in Computer Science, vol 6225. Springer, Berlin, Heidelberg.

Appendices

Appendix A: AES in depth

A.1 Rijndael's finite field

In order to properly understand how each of the AES internal transformation steps are performed, a little insight into Rijndael's finite field is needed. AES performs its mathematical operations in the characteristic 2 finite field with 256 elements, which can also be named Galois Field or $GF\ (2^8)$. As it may seem obvious, the field dimension order of 8 makes it suitable for working with bytes (1 byte = 8 bits).

To get started, a generic Galois Field with p^n elements would be denoted as $GF(p^n)$, where p is a prime number, meaning simply a ring of integers modulo p. As a result, operations such as addition, subtraction and multiplication can be performed as usual, followed by a reduction modulo p. For instance, in $GF(6)$, the result of adding 4 to 6 would be reduced to 4 modulo 6.

A particular case is $GF(2)$, where addition is performed through the exclusive OR (XOR) and multiplication with an AND. Since the only invertible element is 1, division is the identity function.

Elements of $GF(p^n)$ may be represented as polynomials of degree strictly less than n over $GF(p)$. Operations are then performed modulo R where R is an irreducible polynomial of degree n over $GF(p)$. When the prime characteristic number is 2, it is conventional to express elements of $GF(p^n)$ as binary numbers, with each term in a polynomial represented by one bit in the corresponding element's binary expression. An example of equivalences is shown in:

Polynomial	x7 + x4 + x3 + x + 1
Binary	{10011011}2
Hexadecimal	{9B}16

Table 22. Equivalent representations

The Rijndael's finite field is represented as:

$$GF(2^8) = \frac{GF(2)[x]}{(x^8+x^4+x^3+x+1)} \tag{25}$$

$GF\ (2)[x]$ is the set of polynomials with coefficients in $GF\ (2)$, which, as stated before, has a binary quotient ring limited to $\{0,1\}$. The irreducible R in $GF\ (2)[x]$, is the polynomial in the denominator making equation (25) a finite field. Every element inside the field will have a binary representation, always between the possible values inside a byte, e.g. $\in [00_H, FF_H]$.

In every finite field with characteristic 2 happens that addition modulo 2, subtraction modulo 2 and XOR are identical operations. Multiplication in any finite field is multiplication modulo the irreducible polynomial R used to define the finite field. In other words, in $GF(2^8)$ polynomial multiplication is done as usual followed by a division, using the irreducible polynomial R as the divisor. The remainder of the division is the product. The next example equations show how a multiplication between polynomials, $P(x)$ and $Q(x)$ would work in $GF(2^8)$:

$$P(x) = x^7 + x^5 + x^4 + 1 \in GF(2^8) \tag{26}$$

$$Q(x) = x^4 + x^3 + x + 1 \in GF(2^8) \quad (27)$$

$$P(x) \cdot Q(x) = (x^7 + x^5 + x^4 + 1) \cdot (x^4 + x^3 + x + 1) \quad (28)$$

$$P(x) \cdot Q(x) = x^{11} + x^{10} + x^8 + x^7 + x^9 + x^8 + x^6 + x^5 + x^8 + x^7 + x^5 + x^4 + x^4 + x^3 + x + 1 \quad (29)$$

$$P(x) \cdot Q(x) = x^{11} + x^{10} + x^9 + x^8 + x^6 + x^3 + x + 1 \; mod(x^8 + x^4 + x^3 + x + 1) \quad (30)$$

$$P(x) \cdot Q(x) = x^7 + x^6 + x^4 + x = \{11010010\}_2 = \{D2\}_{16} \quad (31)$$

The division made from equation (30) to (31) can be demonstrated through long polynomial division.

Therefore, to sum up, in the Rijndael's finite field we can work with all values inside a byte, represented with 8 bits, and we get to perform two operations: XOR (addition) and AND (multiplication).

A.2 AES round internal operations

A.2.1 SubBytes

SubBytes is the function that provides confusion through non-linearity to the AES. This function is a combination of two transformations: the inverse function in the Rijndael's finite field and an invertible affine transformation.

In the next examples, the single bit values will be represented with lowercase letters, while uppercase letters refer to bytes.

Consider the polynomial $B(x)$ and its coefficients in vector B, as well as the polynomial $V(x)$ and its coefficients in vector V:

$$B(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \quad (32)$$

$$B = b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0 \quad (33)$$

$$V(x) = v_7x^7 + v_6x^6 + v_5x^5 + v_4x^4 + v_3x^3 + v_2x^2 + v_1x + v_0 \quad (34)$$

$$V = v_7\ v_6\ v_5\ v_4\ v_3\ v_2\ v_1\ v_0 \quad (35)$$

Vector V will be the invert of B in $GF(2^8)$ if the next condition is satisfied:

$$B(x) \cdot V(x) \equiv 1 \; mod(x^8 + x^4 + x^3 + x + 1) \quad (36)$$

Hence, the remainder of the polynomial division between $B(x)$ and the irreducible polynomial will be the multiplicative inverse of $B(x)$, denoted as $V(x)$. Once the invert is obtained, an affine transformation shown in equation (37) is applied to V:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad (37)$$

Vector Y keeps the output byte of the *SubBytes* function, related to the input byte B. Following this logic and computing the output for every possible value, the AES SBOX is built (Appendix B1)

When deciphering, the inverse operation *InvSubBytes* is performed. Firstly, the inverse affine transformation (38) is applied and then, the inverse of the result is computed as in (36).

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (38)$$

Once again, computing every possible output the AES inverse SBOX is built (Appendix B1)

A.2.2 ShiftRows

ShiftRows is, basically, a byte permutation inside the state matrix. The permutation is done row by row shifting the bytes to the left in 0, 1, 2 or 3 positions respectively. The first row is kept, no shifting is applied to it. The second row is shifted 1 position to the left, the second is shifted 2 positions to the left and the last row is shifted 3 positions to the left.

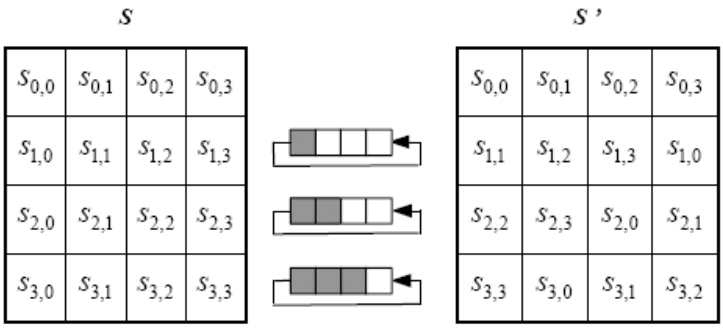

Figure 30. ShiftRows permutation

Regarding decryption flow, *InvShiftRows* works exactly the same way but shifting in the opposite direction.

A.2.3 MixColumns

MixColumns is the step where the four bytes of each column are combined using an invertible linear transformation. *MixColumns,* together with *ShiftRows*, provides diffusion to the AES algorithm. Hence, at this step each column of the state is multiplied by a fixed matrix as shown in the following equations. Remember that both multiplication and addition are performed in the Rijndael's finite field:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{39}$$

$$\begin{cases} b_0 = x \cdot a_0 + (x+1) \cdot a_1 + a_2 + a_3 \\ b_1 = a_0 + x \cdot a_1 + (x+1) \cdot a_2 + a_3 \\ b_2 = a_0 + a_1 + x \cdot a_2 + (x+1) \cdot a_3 \\ b_3 = (x+1) \cdot a_0 + a_1 + a_2 + x \cdot a_3 \end{cases} \tag{40}$$

a_0, a_1, a_2 and a_3 represent the four input bytes of the same column of the state, while b_0, b_1, b_2 and b_3 are their corresponding outputs. Therefore, we only need to perform multiplications by 1, x and $x+1$.

On the one hand, multiplying by 1 does nothing. On the other hand, multiplying by x+1 is done by multiplying by x, then by 1 and applying an XOR to both results. When multiplying any byte by x in $GF(2^8)$ we have two possibilities. Consider any byte B:

$$B = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \ \ where \ \ b_i \in \{0,1\} \tag{41}$$

If the MSB of B is 0, the operation is a simple bit shift to the left:

$$B \ll 1 \tag{42}$$

If the MSB of the byte is 1, the operation includes a shift plus an XOR with $0x1B$:

$$(B \ll 1) \oplus 0x1B \tag{43}$$

An interesting remark is that a bit shift to the left can be traduced as a multiplication by 2 in the Rijndael's finite field. If this multiplication exceeds the limit value of 256 (FF_H) the result has to be XOR-ed with $0\text{x}1\text{B}$ (which works as the division by the irreducible polynomial $0\text{x}11\text{B}$ in the particular case where the dividend is not greater than $1FF_H$)

Taking this under consideration, equation (39) and (40) can be written as:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{44}$$

$$\begin{cases} b_0 = 2a_0 + 3a_1 + a_2 + a_3 \\ b_1 = a_0 + 2a_1 + 3a_2 + a_3 \\ b_2 = a_0 + a_1 + 2a_2 + 3a_3 \\ b_3 = 3a_0 + a_1 + a_2 + 2a_3 \end{cases} \tag{45}$$

From this point of view, we get equation (42) and (43) translated to equation (46) and (47):

$$2a_i = \begin{cases} a_i \ll 1 & if\ a < FF_H \\ (a_i \ll 1) \oplus 0x1B & if\ a \geq FF_H \end{cases} \quad (46)$$

$$3a_i = 2a_i \oplus a_i \quad (47)$$

It is feasible to compute all the input-output possibilities for the $2a$ case multiplication and the same can be done for the $3a$ case multiplication. This way, a look-up table can be constructed for each multiplication as we did with the AES SBOX in the *SubBytes* step (Appendix B2).

Regarding decryption, *InvMixColumns* works the same way, only that the multiplication matrix is the inverse of (44) and can be written as:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad (48)$$

$$\begin{cases} b_0 = 14a_0 + 11a_1 + 13a_2 + 9a_3 \\ b_1 = 9a_0 + 14a_1 + 11a_2 + 13a_3 \\ b_2 = 13a_0 + 9a_1 + 14a_2 + 11a_3 \\ b_3 = 11a_0 + 13a_1 + 9a_2 + 14a_3 \end{cases} \quad (49)$$

All possibilities for these four multiplication cases can also be gathered in four LUT tables (Appendix B2).

A.2.4 AddRoundKey

AddRoundKey is performed once at the beginning of encryption and once more at the end of each round. The initial *AddRoundKey* mixes the secret key with the plaintext by XOR-ing them. Each of the *AddRoundKey* steps at the end of the rounds serve to XOR the state with the corresponding round key.

For decryption purposes, the *AddRoundKey* steps work equally, only that the round key order is the opposite.

A.3 Key Schedule:

The key schedule generates each sub-key, needed for each round of the algorithm. The first round key is derived from the main secret key. Each of the following round keys are derived from the previous ones and the algorithm applied is always the same.

The first step is to take the previous 128 bit key and divide it into four words of 32 bits denoted as ω_i.

$$Key = b_{127}\, b_{126}\, b_{125} \ldots b_2\, b_1\, b_0 = B_{15}B_{14}B_{13}B_{12}B_{11}B_{10}B_9B_8B_7B_6B_5B_4B_3B_2B_1B_0 \quad (50)$$

$$\omega_0 = B_{15}B_{14}B_{13}B_{12}\ ;\ \omega_1 = B_{11}B_{10}B_9B_8\ ;\ \omega_2 = B_7B_6B_5B_4\ ;\ \omega_3 = B_3B_2B_1B_0 \quad (51)$$

Once we have our separate words, the least significant word ω_3 goes through a function composed of three transformations:

1) Circular byte shift to the left → $\omega_3' = B_2B_1B_0B_3$ (52)

2) Byte SBOX substitution → $\omega_3'' = SB(B_2)SB(B_1)SB(B_0)SB(B_3)$ (53)

3) Add (XOR) with round constant → $f(\omega_3) = \omega_3'' \oplus RC$ (54)

The round constant RC is a fixed 32-bit value for each round. These fixed values are gathered in the following table:

Round	RC
R1	01000000H
R2	02000000H
R3	04000000H
R4	08000000H
R5	10000000H
R6	20000000H
R7	40000000H
R8	80000000H
R9	1B000000H
R10	36000000H

Table 23. Round constant values

Then, each sub-key or round key SK_n is constructed word by word to be concatenated as follows:

$$SK_0 = f(\omega_3) \oplus \omega_0 \quad (55)$$

$$SK_1 = f(\omega_3) \oplus \omega_0 \oplus \omega_1 \quad (56)$$

$$SK_2 = f(\omega_3) \oplus \omega_0 \oplus \omega_1 \oplus \omega_2 \quad (57)$$

$$SK_3 = f(\omega_3) \oplus \omega_0 \oplus \omega_1 \oplus \omega_2 \oplus \omega_3 \quad (58)$$

$$\mathrm{SK_n} = \{\mathrm{SK_0}, \mathrm{SK_1}, \mathrm{SK_2}, \mathrm{SK_3}\} \quad , \quad \mathrm{n} \in [1,10] \quad (59)$$

Appendix B: AES LUTs

B.1 AES SBOX and reverse SBOX

AES SBOX

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1X	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2X	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3X	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4X	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5X	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6X	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8

7X	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8X	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9X	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
AX	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
BX	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
CX	BA	78	25	2E	AC	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
DX	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
EX	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
FX	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 24. AES SBOX LUT

AES reverse SBOX

	X0	**X1**	**X2**	**X3**	**X4**	**X5**	**X6**	**X7**	**X8**	**X9**	**XA**	**XB**	**XC**	**XD**	**XE**	**XF**
0X	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1X	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E7	CB
2X	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3X	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4X	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5X	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6X	90	D8	AB	00	8C	BV	D3	0A	F7	E4	58	05	B8	B3	45	06
7X	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8X	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9X	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	AC	75	DF	6E
AX	47	F1	1A	71	AD	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
BX	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
CX	AF	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
DX	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
EX	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
FX	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Table 25. AES reverse SBOX LUT

B.2 Galois multiply LUTs

Galois multiply by 2

	X0	**X1**	**X2**	**X3**	**X4**	**X5**	**X6**	**X7**	**X8**	**X9**	**XA**	**XB**	**XC**	**XD**	**XE**	**XF**
0X	00	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E

1X	20	22	24	26	28	2A	2C	2E	30	32	34	36	38	3A	3C	3E
2X	40	42	44	46	48	4A	4C	4E	50	52	54	56	58	5A	5C	5E
3X	60	62	64	66	68	6A	6C	6E	70	72	74	76	78	7A	7C	7E
4X	80	82	84	86	88	8A	8C	8E	90	92	94	96	98	9A	9C	9E
5X	A0	A2	A4	A6	A8	AA	AC	AE	B0	B2	B4	B6	B8	BA	BC	BE
6X	C0	C2	C4	C6	C8	CA	CC	CE	D0	D2	D4	D6	D8	DA	DB	DE
7X	E0	E2	E4	E6	E8	EA	EC	EE	F0	F2	F4	F6	F8	FA	FB	FE
8X	1B	19	1F	1D	13	11	17	15	0B	09	0F	0D	03	01	07	05
9X	3B	39	3F	3D	33	31	37	35	2B	29	2F	2D	23	21	27	25
AX	5B	59	5F	5D	53	51	57	55	4B	49	4F	4D	43	41	47	45
BX	7B	79	7F	7D	73	71	77	75	6B	69	6F	6D	63	61	67	65
CX	9B	99	9F	9D	93	91	97	95	8B	89	8F	8D	83	81	87	85
DX	BB	B9	BF	BD	B3	B1	B7	B5	AB	A9	AF	AD	A3	A1	A7	A5
EX	DB	D9	DF	DD	D3	D1	D7	D5	CB	C9	CF	CD	C3	C1	C7	C5
FX	FB	F9	FF	FD	F3	F1	F7	F5	EB	E9	EF	ED	E3	E1	E7	E5

Table 26. Galois LUT for multiply by 2

Galois multiply by 3

	X0	**X1**	**X2**	**X3**	**X4**	**X5**	**X6**	**X7**	**X8**	**X9**	**XA**	**XB**	**XC**	**XD**	**XE**	**XF**
0X	00	03	06	05	0C	0F	0A	09	18	1B	1E	1D	14	17	12	11
1X	30	33	36	35	3C	3F	3A	39	28	2B	2E	2D	24	27	22	21
2X	60	63	66	65	6C	6F	6A	69	78	7B	7E	7D	74	77	72	71
3X	50	53	56	55	5C	5F	5A	59	48	4B	4E	4D	44	47	42	41
4X	C0	C3	C6	C5	CC	CF	CA	C9	D8	DB	DE	DD	D4	D7	D2	D1
5X	F0	F3	F6	F5	FC	FF	FA	F9	E8	EB	EE	ED	E4	E7	E2	E1
6X	A0	A3	A6	A5	AC	AF	AA	A9	B8	BB	BE	BD	B4	B7	B2	B1
7X	90	93	96	95	9C	9F	9A	99	88	8B	8E	8D	84	87	82	81
8X	9B	98	9D	9E	97	94	91	92	83	80	85	86	8F	8C	89	8A
9X	AB	A8	AD	AE	A7	A4	A1	A2	B3	B0	B5	B6	BF	BC	B9	BA
AX	FB	F8	FD	FE	F7	F4	F1	F2	E3	E0	E5	E6	EF	EC	E9	EA
BX	CB	C8	CD	CE	C7	C4	C1	C2	D3	D0	D5	D6	DF	DC	D9	DA
CX	5B	58	5D	5E	57	54	51	52	43	40	45	46	4F	4C	49	4A
DX	6B	68	6D	6E	67	64	61	62	73	70	75	76	7F	7C	79	7A
EX	3B	38	3D	3E	37	34	31	32	23	20	25	26	2F	2C	29	2A
FX	0B	08	0D	0E	07	04	01	02	13	10	15	16	1F	1C	19	1A

Table 27. Galois LUT for multiply by 3

Galois multiply by 9

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X	00	09	12	1B	24	2D	36	3F	48	41	5A	53	6C	65	7E	77
1X	90	99	82	8B	B4	BD	A6	AF	D8	D1	CA	C3	FC	F5	EE	E7
2X	3B	32	29	20	1F	16	0D	04	73	7A	61	68	57	5E	45	4C
3X	AB	A2	B9	B0	8F	86	9D	94	E3	EA	F1	F8	C7	CE	D5	DC
4X	76	7F	64	6D	52	5B	40	49	3E	37	2C	25	1A	13	08	01
5X	E6	EF	F4	FD	C2	CB	D0	D9	AE	A7	BC	B5	8A	83	98	91
6X	4D	44	5F	56	69	60	7B	72	05	0C	17	1E	21	28	33	3A
7X	DD	D4	CF	C6	F9	F0	EB	E2	95	9C	87	8E	B1	B8	A3	AA
8X	EC	E5	FE	F7	C8	C1	DA	D3	A4	AD	B6	BF	80	89	92	9B
9X	7C	75	6E	67	58	51	4A	43	34	3D	26	2F	10	19	02	0B
AX	D7	DE	C5	CC	F3	FA	E1	E8	9F	96	8D	84	BB	B2	A9	A0
BX	47	4E	55	5C	63	3A	71	78	0F	06	AD	14	2B	22	39	30
CX	9A	93	88	82	BE	B7	AC	A5	D2	DB	C0	C9	F6	FF	E4	ED
DX	0A	03	18	11	2E	27	3C	35	42	4B	50	59	66	6F	74	7D
EX	A1	A8	B3	BA	85	8C	97	9E	E9	E0	FB	F2	CD	C4	DF	D6
FX	31	38	23	2A	15	1C	07	0E	79	70	6B	62	5D	54	4F	46

Table 28. Galois LUT for multiply by 9

Galois multiply by 11

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X	00	0B	16	1D	2C	27	3A	31	58	53	4E	45	74	7F	62	69
1X	B0	BB	A6	AD	9C	97	8A	81	E8	E3	FE	F5	C4	CF	D2	D9
2X	7B	70	6D	66	57	5C	41	4A	23	28	35	3E	0F	04	19	12
3X	CB	C0	DD	D6	E7	EC	F1	FA	93	98	85	8E	BF	B4	A9	A2
4X	F6	FD	E0	EB	DA	D1	CC	C7	AE	A5	B8	B3	82	89	94	9F
5X	46	4D	50	5B	6A	61	7C	77	1E	15	08	03	32	39	24	2F
6X	8D	86	9B	90	A1	AA	B7	BC	D5	DE	C3	C8	F9	F2	EF	E4
7X	3D	36	2B	20	11	1A	07	0C	65	6E	73	78	49	42	5F	54
8X	F7	FC	E1	EA	DB	D0	CD	C6	AF	A4	B9	B2	83	88	95	9E
9X	47	4C	51	5A	6B	60	7D	76	1F	14	09	02	33	38	25	2E
AX	8C	87	9A	91	A0	AB	B6	BD	D4	DF	C2	C9	F8	F3	EE	E5
BX	3C	37	2A	21	10	AB	06	0D	64	6F	72	79	48	43	5E	55

CX	01	0A	17	1C	2D	26	3B	30	59	52	4F	44	75	7E	63	68
DX	B1	BA	A7	AC	9D	96	8B	80	E9	E2	FF	F4	C5	CE	D3	D8
EX	7A	71	6C	67	56	5D	40	4B	22	29	34	3F	0E	05	18	13
FX	CA	C1	DC	D7	E6	ED	F0	FB	92	99	84	8F	BE	B5	A8	A3

Table 29. Galois LUT for multiply by 11

Galois multiply by 13

	X0	**X1**	**X2**	**X3**	**X4**	**X5**	**X6**	**X7**	**X8**	**X9**	**XA**	**XB**	**XC**	**XD**	**XE**	**XF**
0X	00	0D	1A	17	34	39	2E	23	68	65	72	7F	5C	51	46	4B
1X	D0	DD	CA	C7	E4	E9	FE	F3	B8	B5	A2	AF	8C	81	96	9B
2X	BB	B6	A1	AC	8F	82	95	98	D3	DE	C9	C4	E7	EA	FD	F0
3X	6B	66	71	7C	5F	52	45	48	03	0E	19	14	37	3A	2D	20
4X	6D	60	77	7A	59	54	43	4E	05	08	AF	12	31	3C	2B	26
5X	BD	B0	A7	AA	89	84	93	9E	D5	D8	CF	C2	E1	EC	FB	F6
6X	D6	DB	CC	C1	E2	EF	F8	F5	BE	B3	A4	A9	8A	87	90	9D
7X	06	0B	1C	11	32	3F	28	25	6E	63	74	79	5A	57	40	4D
8X	DA	D7	C0	CD	EE	E3	F4	F9	B2	BF	A8	A5	86	8B	9C	91
9X	0A	07	10	1D	3E	33	24	29	62	6F	78	75	56	5B	4C	41
AX	61	6C	7B	76	55	58	4F	42	09	04	13	1E	3D	30	27	2A
BX	B1	BC	AB	A6	85	88	9F	92	D9	D4	C3	CE	ED	E0	F7	FA
CX	B7	BA	AD	A0	83	8E	99	94	DF	D2	C5	C8	EB	E6	F1	FC
DX	67	6A	16	1B	38	35	22	2F	64	69	7E	73	50	5D	4A	47
EX	0C	01	16	1B	38	35	22	2F	64	69	7E	73	50	5D	4A	47
FX	DC	D1	C6	CB	E8	E5	F2	FF	B4	B9	AE	A3	80	8D	9A	97

Table 30. Galois LUT for multiply by 13

Galois multiply by 14

	X0	**X1**	**X2**	**X3**	**X4**	**X5**	**X6**	**X7**	**X8**	**X9**	**XA**	**XB**	**XC**	**XD**	**XE**	**XF**
0X	00	0E	1C	12	38	36	24	2A	70	7E	6C	62	48	46	54	5A
1X	E0	EE	FC	F2	D8	D6	C4	CA	90	9E	8C	82	A8	A6	B4	BA
2X	DB	D5	C7	C9	E3	ED	FF	F1	AB	A5	B7	B9	93	9D	8F	81
3X	3B	35	27	29	03	0D	1F	11	4B	45	57	59	73	7D	6F	61
4X	AD	A3	B1	BF	95	9B	89	87	DD	D3	C1	CF	E5	EB	F9	F7
5X	4D	43	51	5F	75	7B	69	67	3D	33	21	2F	05	0B	19	17
6X	76	78	6A	64	4E	40	52	5C	06	08	1A	14	3E	30	22	2C
7X	96	98	8A	84	AE	A0	B2	BC	E6	E8	FA	F4	DE	D0	C2	CC

8X	41	4F	5D	53	79	77	65	6B	31	3F	2D	23	09	07	15	1B
9X	A1	AF	BD	B3	99	97	85	8B	D1	DF	CD	C3	E9	E7	F5	FB
AX	9A	94	86	88	A2	AC	BE	B0	EA	E4	F6	F8	D2	DC	CE	C0
BX	7A	74	66	68	42	4C	5E	50	0A	04	16	18	32	3C	2E	20
CX	EC	E2	F0	FE	D4	DA	C8	C6	9C	92	80	8E	A4	AA	B8	B6
DX	0C	02	10	1E	34	3A	28	26	7C	72	60	6E	44	4A	58	56
EX	37	39	2B	25	0F	01	13	1D	47	49	5B	55	7F	71	63	6D
FX	D7	D9	CB	C5	EF	E1	F3	FD	A7	A9	BB	B5	9F	91	83	8D

Table 31. Galois LUT for multiply by 14

Appendix C: Implementation codes

C.1 AES128

```
/*****************************************************************************/
/* Includes:                                                                 */
/*****************************************************************************/
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include "aes.h"

/*****************************************************************************/
/* Defines: Fixed values of the AES128 implementation                        */
/*****************************************************************************/
// Number of rows and columns of the AES state
#define Nb 4
// The number of rounds in AES128 cipher
#define Nr 10
// Key length in bytes [128 bit]
#define KEYLEN 16

/*****************************************************************************/
/* Private variables:                                                        */
/*****************************************************************************/
// state matrix holding the intermediate values during encryption
typedef uint8_t state_t[4][4];
static state_t* state;

// The array that stores the round keys (11keys x 16bytes = 176bytes).
static uint8_t RoundKey[176];

// The Key input to the AES algorithm
static uint8_t* Key;

// SBOX LUT
static const uint8_t sbox[256] =   {
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
0xd7, 0xab, 0x76,
0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
0xa4, 0x72, 0xc0,
```

```
0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
0xd8, 0x31, 0x15,
0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
0x27, 0xb2, 0x75,
0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
0xe3, 0x2f, 0x84,
0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
0x4c, 0x58, 0xcf,
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
0x3c, 0x9f, 0xa8,
0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
0xff, 0xf3, 0xd2,
0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
0x5d, 0x19, 0x73,
0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
0x5e, 0x0b, 0xdb,
0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
0x95, 0xe4, 0x79,
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
0x7a, 0xae, 0x08,
0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
0xbd, 0x8b, 0x8a,
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
0xc1, 0x1d, 0x9e,
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
0x55, 0x28, 0xdf,
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
0x54, 0xbb, 0x16 };

// Reverse SBOX LUT
static const uint8_t rsbox[256] ={
0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81,
0xf3, 0xd7, 0xfb,
0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4,
0xde, 0xe9, 0xcb,
0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42,
0xfa, 0xc3, 0x4e,
0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d,
0x8b, 0xd1, 0x25,
0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d,
0x65, 0xb6, 0x92,
0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7,
0x8d, 0x9d, 0x84,
0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8,
0xb3, 0x45, 0x06,
0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01,
0x13, 0x8a, 0x6b,
0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0,
0xb4, 0xe6, 0x73,
0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c,
0x75, 0xdf, 0x6e,
0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa,
0x18, 0xbe, 0x1b,
0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78,
0xcd, 0x5a, 0xf4,
0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27,
0x80, 0xec, 0x5f,
0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93,
0xc9, 0x9c, 0xef,
0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83,
0x53, 0x99, 0x61,
```

```
0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55,
0x21, 0x0c, 0x7d };

// Round constant array
static const uint8_t Rcon[10] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
0x1b, 0x36 };

/*****************************************************************************/
/* Private functions:                                                        */
/*****************************************************************************/
// Returns the SBOX substitution of a byte
static uint8_t getSBoxValue(uint8_t num)
{
  return sbox[num];
}

// Returns the inverse SBOX substitution of a byte
static uint8_t getSBoxInvert(uint8_t num)
{
  return rsbox[num];
}

// Function used to multiply by x in MixColumns computation
static uint8_t xtime(uint8_t x)
{
  return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
}

// Function used for InvMixColumns multiplications
static uint8_t Multiply(uint8_t x, uint8_t y)
{
  return (((y & 1) * x) ^
       ((y>>1 & 1) * xtime(x)) ^
       ((y>>2 & 1) * xtime(xtime(x))) ^
       ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^
       ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))));
}

// This function implements the AES key schedule.
static void KeyExpansion(void)
{
  uint8_t i, j, k;
  uint8_t tempa[4]; // Used for the column/row operations

  // The first round key is derived from the input key.
  for(i = 0; i < Nb; ++i)
  {
    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
  }

  // All other round keys are found from the previous round key.
  for(; (i < (Nb * (Nr + 1))); ++i)
  {
    for(j = 0; j < 4; ++j)
    {
      // Store previous key in tempa
      tempa[j]=RoundKey[(i-1) * 4 + j];
    }
```

```
    if (i % Nb == 0)
    {
      // Function RotWord()
      k = tempa[0];
      tempa[0] = tempa[1];
      tempa[1] = tempa[2];
      tempa[2] = tempa[3];
      tempa[3] = k;

      // Function Subword()
      tempa[0] = getSBoxValue(tempa[0]);
      tempa[1] = getSBoxValue(tempa[1]);
      tempa[2] = getSBoxValue(tempa[2]);
      tempa[3] = getSBoxValue(tempa[3]);

      // XOR with round constant
      tempa[0] =  tempa[0] ^ Rcon[i/Nb - 1];
    }
    // Add the round key to the array
    RoundKey[i * 4 + 0] = RoundKey[(i - Nb) * 4 + 0] ^ tempa[0];
    RoundKey[i * 4 + 1] = RoundKey[(i - Nb) * 4 + 1] ^ tempa[1];
    RoundKey[i * 4 + 2] = RoundKey[(i - Nb) * 4 + 2] ^ tempa[2];
    RoundKey[i * 4 + 3] = RoundKey[(i - Nb) * 4 + 3] ^ tempa[3];
  }
}

// This function XORs the round key to state.
static void AddRoundKey(uint8_t round)
{
  uint8_t i,j;
  for(i=0;i<4;++i)
  {
    for(j = 0; j < 4; ++j)
    {
      (*state)[i][j] ^= RoundKey[round * Nb * 4 + i * Nb + j];
    }
  }
}

// The SubBytes function
static void SubBytes(void)
{
  uint8_t i, j;
  for(i = 0; i < 4; ++i)
  {
    for(j = 0; j < 4; ++j)
    {
      (*state)[i][j] = getSBoxValue((*state)[i][j]);
    }
  }
}

//Inverse SubBytes
static void InvSubBytes(void)
{
  uint8_t i,j;
  for(i=0;i<4;++i)
  {
    for(j=0;j<4;++j)
    {
      (*state)[i][j] = getSBoxInvert((*state)[i][j]);
```

```
    }
  }
}

// Shiftrows function. Offset = Row number (0,1,2,3).
static void ShiftRows(void)
{
  uint8_t temp;
  // Rotate first row 1 column to left
  temp           = (*state)[0][1];
  (*state)[0][1] = (*state)[1][1];
  (*state)[1][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[3][1];
  (*state)[3][1] = temp;

  // Rotate second row 2 columns to left
  temp           = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp       = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate third row 3 columns to left
  temp       = (*state)[0][3];
  (*state)[0][3] = (*state)[3][3];
  (*state)[3][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[1][3];
  (*state)[1][3] = temp;
}

//Inverse ShiftRows
static void InvShiftRows(void)
{
  uint8_t temp;
  // Rotate first row 1 column to right
  temp=(*state)[3][1];
  (*state)[3][1]=(*state)[2][1];
  (*state)[2][1]=(*state)[1][1];
  (*state)[1][1]=(*state)[0][1];
  (*state)[0][1]=temp;

  // Rotate second row 2 columns to right
  temp=(*state)[0][2];
  (*state)[0][2]=(*state)[2][2];
  (*state)[2][2]=temp;
  temp=(*state)[1][2];
  (*state)[1][2]=(*state)[3][2];
  (*state)[3][2]=temp;

  // Rotate third row 3 columns to right
  temp=(*state)[0][3];
  (*state)[0][3]=(*state)[1][3];
  (*state)[1][3]=(*state)[2][3];
  (*state)[2][3]=(*state)[3][3];
  (*state)[3][3]=temp;
}

// MixColumns function mixes the columns of the state matrix
static void MixColumns(void)
```

```
{
  uint8_t i;
  uint8_t Tmp,Tm,t;
  for(i = 0; i < 4; ++i)
  {
    t = (*state)[i][0];
    Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3];
    Tm =(*state)[i][0]^(*state)[i][1]; Tm=xtime(Tm); (*state)[i][0]^=Tm^Tmp;
    Tm =(*state)[i][1]^(*state)[i][2]; Tm=xtime(Tm); (*state)[i][1]^=Tm^Tmp;
    Tm =(*state)[i][2]^(*state)[i][3]; Tm=xtime(Tm); (*state)[i][2]^=Tm^Tmp;
    Tm  = (*state)[i][3] ^ t ;  Tm = xtime(Tm);  (*state)[i][3] ^= Tm ^ Tmp;
  }
}
// Inerse MixColumns
static void InvMixColumns(void)
{
  uint8_t i;
  uint8_t a,b,c,d;
  for(i=0;i<4;++i)
  {
    a = (*state)[i][0];
    b = (*state)[i][1];
    c = (*state)[i][2];
    d = (*state)[i][3];

    (*state)[i][0] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^
Multiply(d, 0x09);
    (*state)[i][1] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^
Multiply(d, 0x0d);
    (*state)[i][2] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^
Multiply(d, 0x0b);
    (*state)[i][3] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^
Multiply(d, 0x0e);
  }
}

static void Round(uint8_t round)
{
  SubBytes();
  ShiftRows();
  MixColumns();
  AddRoundKey(round);
}

static void LastRound()
{
  SubBytes();
  ShiftRows();
  AddRoundKey(Nr);
}

static void InvRound(uint8_t round)
{
  InvShiftRows();
  InvSubBytes();
  InvMixColumns();
  AddRoundKey(round);
}

static void InvLastRound()
{
```

```
  InvShiftRows();
  InvSubBytes();
  AddRoundKey(0);
}

// Encryption function
static void Cipher(void)
{
  uint8_t round;
  AddRoundKey(0);  // Initial AddRoundKey.

  for(round = 1; round < Nr; ++round)   //From round 1 to 9
    {Round(round);}

  LastRound();          //Last round
}

// Decryption function
static void InvCipher(void)
{
  uint8_t round;
  AddRoundKey(Nr);

  for(round=Nr-1; round>0; --round)
    {InvRound(round);}    //From round 10 to 2

  InvLastRound();
}

/*****************************************************************************/
/* Public functions:                                                         */
/*****************************************************************************/
void AES128_ECB_indp_setkey(uint8_t* key)
{
  Key = key;
  KeyExpansion();
}

void AES128_ECB_indp_crypto(uint8_t* input)
{
  state = (state_t*)input;
  Cipher();
}

void AES128_ECB_indp_inv_crypto(uint8_t* input)
{
  state = (state_t*)input;
  InvCipher();
}
```

C.2 AES128 with dummy round insertion (only encryption)

```
/*****************************************************************************/
/* Includes:                                                                 */
/*****************************************************************************/
#include "aes.h"
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

/*****************************************************************************/
```

```
/*         Defines: Fixed values of the AES128 implementation              */
/*****************************************************************************/
// Number of rows and columns
#define Nb 4
// The number of rounds in AES128 cipher.
#define Nr 10
// Key length in bytes [128 bit]
#define KEYLEN 16

/*****************************************************************************/
/*        COMMENT/UNCOMMENT THE COUNTERMEASURES YOU WANT TO ACTIVATE       */
/*****************************************************************************/
//     #define OneDummy
     #define TwoDummies

/*****************************************************************************/
/* Private variables:                                                        */
/*****************************************************************************/
// state matrix holding the intermediate results during encryption
typedef uint8_t state_t[4][4];
static state_t* state;

// The array that stores the round keys (11keys x 16bytes = 176bytes).
static uint8_t RoundKey[176];

// The Key input to the AES Program
static uint8_t* Key;

// Dummy state matrix1 for dummy round operations
static uint8_t dummy_state_t1[4][4];
static state_t* dummy_state1 = &dummy_state_t1;

#ifdef TwoDummies
// Dummy state matrix2 for dummy round operations
static uint8_t dummy_state_t2[4][4];
static state_t* dummy_state2 = &dummy_state_t2;
#endif

// This vector will store the needed random selection values for each round
static uint8_t RandomVector[10];

//10 dummy keys needed for single dummy
#ifdef OneDummy
#define DummyKeyNum 10
static uint8_t DummyKeys[DummyKeyNum*KEYLEN];
#endif

//20 dummy keys needed for double dummy
#ifdef TwoDummies
#define DummyKeyNum 20
static uint8_t DummyKeys[DummyKeyNum*KEYLEN];
#endif

//AES SBOX LUT
static const uint8_t sbox[256] =   {
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
0xd8, 0x31, 0x15,
```

```
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
0x27, 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
0xe3, 0x2f, 0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
0x4c, 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
0xff, 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
0x5d, 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
0x5e, 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
0x95, 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
0x7a, 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
0xbd, 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
0xc1, 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
0x54, 0xbb, 0x16 };

// The round constant array
static const uint8_t Rcon[11] =
{0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36};

/*****************************************************************************/
/* Private functions:                                                        */
/*****************************************************************************/
static uint8_t getSBoxValue(uint8_t num)
{
  return sbox[num];
}

static uint8_t xtime(uint8_t x)
{
  return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
}

static uint8_t getByte(void)
{
  return rand() % 256;
}

static void RandomGeneration(void)
{
  //Fill the dummy state 1 with random values
  for (uint8_t i = 0; i<Nb; i++)
  {
    for(uint8_t j = 0; j<Nb; j++)
    {
      (*dummy_state1)[i][j] = getByte();
    }
  }
  #ifdef TwoDummies
  //Fill the dummy state 2 with random values
```

```
  for (uint8_t i = 0; i<Nb; i++)
  {
    for(uint8_t j = 0; j<Nb; j++)
    {
      (*dummy_state2)[i][j] = getByte();
    }
  }
  #endif
  // Generate all needed dummy keys
  for (uint16_t i = 0; i<KEYLEN*DummyKeyNum; i++)
  {
    DummyKeys[i] = getByte();
  }

  #ifdef OneDummy
  for (uint8_t i = 0; i<10 ; i++)
  {
    RandomVector[i] = rand() % 2;
  }
  #endif
  #ifdef TwoDummies
  for (uint8_t i = 0; i<10 ; i++)
  {
    RandomVector[i] = rand() % 3;
  }
  #endif
}

static void KeyExpansion(void)
{
  uint8_t i, j, k;
  uint8_t tempa[4]; // Used for the column/row operations

  // The first round key is the key itself.
  for(i = 0; i < Nb; ++i)
  {
    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
  }

  // All other round keys are found from the previous round keys.
  for(; (i < (Nb * (Nr + 1))); ++i)
  {
    for(j = 0; j < 4; ++j)
    {
      // Store previous key in tempa
      tempa[j]=RoundKey[(i-1) * 4 + j];
    }
    if (i % Nb == 0)
    {
      // Function RotWord()
      k = tempa[0];
      tempa[0] = tempa[1];
      tempa[1] = tempa[2];
      tempa[2] = tempa[3];
      tempa[3] = k;

      // Function Subword()
      tempa[0] = getSBoxValue(tempa[0]);
```

```
      tempa[1] = getSBoxValue(tempa[1]);
      tempa[2] = getSBoxValue(tempa[2]);
      tempa[3] = getSBoxValue(tempa[3]);

      // XOR with round constant
      tempa[0] =  tempa[0] ^ Rcon[i/Nb];
    }
    // Add the round key the array
    RoundKey[i * 4 + 0] = RoundKey[(i - Nb) * 4 + 0] ^ tempa[0];
    RoundKey[i * 4 + 1] = RoundKey[(i - Nb) * 4 + 1] ^ tempa[1];
    RoundKey[i * 4 + 2] = RoundKey[(i - Nb) * 4 + 2] ^ tempa[2];
    RoundKey[i * 4 + 3] = RoundKey[(i - Nb) * 4 + 3] ^ tempa[3];
  }
}

// This function XORs the round key to state.
static void AddRoundKey(state_t *state, uint8_t round)
{
  uint8_t i,j;
  for(i=0;i<4;++i)
  {
    for(j = 0; j < 4; ++j)
    {
      (*state)[i][j] ^= RoundKey[round * KEYLEN + i * Nb + j];
    }
  }
}

// This function XORs the dummy key to the dummy state.
static void DummyAddRoundKey(state_t *state, uint8_t round)
{
  uint8_t i,j;
  for(i=0;i<4;++i)
  {
    for(j = 0; j < 4; ++j)
    {
      (*state)[i][j] ^= DummyKeys[round * KEYLEN + i * Nb + j];
    }
  }
}


// The SubBytes function
static void SubBytes(state_t *state)
{
  uint8_t i, j;
  for(i = 0; i < 4; ++i)
  {
    for(j = 0; j < 4; ++j)
    {
      (*state)[i][j] = getSBoxValue((*state)[i][j]);
    }
  }
}

// The ShiftRows function. Offset = Row number (0,1,2,3).
static void ShiftRows(state_t *state)
{
  uint8_t temp;

  // Rotate first row 1 columns to left
```

```
  temp           = (*state)[0][1];
  (*state)[0][1] = (*state)[1][1];
  (*state)[1][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[3][1];
  (*state)[3][1] = temp;

  // Rotate second row 2 columns to left
  temp           = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp       = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate third row 3 columns to left
  temp       = (*state)[0][3];
  (*state)[0][3] = (*state)[3][3];
  (*state)[3][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[1][3];
  (*state)[1][3] = temp;
}

// MixColumns function
static void MixColumns(state_t *state)
{
  uint8_t i;
  uint8_t Tmp,Tm,t;
  for(i = 0; i < 4; ++i)
  {
    t = (*state)[i][0];
    Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3];
    Tm =(*state)[i][0]^(*state)[i][1]; Tm=xtime(Tm); (*state)[i][0]^=Tm^Tmp;
    Tm =(*state)[i][1]^(*state)[i][2]; Tm=xtime(Tm); (*state)[i][1]^=Tm^Tmp;
    Tm =(*state)[i][2]^(*state)[i][3]; Tm=xtime(Tm); (*state)[i][2]^=Tm^Tmp;
    Tm  = (*state)[i][3] ^ t ;  Tm = xtime(Tm);  (*state)[i][3] ^= Tm ^ Tmp;
  }
}

static void Round(state_t *state, uint8_t round)
{
  SubBytes(state);
  ShiftRows(state);
  MixColumns(state);
  AddRoundKey(state, round);
}

static void LastRound(state_t *state, uint8_t round)
{
  SubBytes(state);
  ShiftRows(state);
  AddRoundKey(state, round);
}

static void DummyRound(state_t *state, uint8_t round)
{
  SubBytes(state);
  ShiftRows(state);
  MixColumns(state);
  DummyAddRoundKey(state, round);
}
```

```
static void DummyLastRound(state_t *state, uint8_t round)
{
  SubBytes(state);
  ShiftRows(state);
  DummyAddRoundKey(state, round);
}

#ifdef OneDummy
static void CipherWithDummies(void)
{
  uint8_t round;
  uint8_t DummyKeyCounter = 0;
  AddRoundKey(state,0);

  //From round 1 to 9 a dummy per round is added in random order
  for(round = 1; round < Nr; ++round)
  {
      if (RandomVector[round-1]==0)
    {
        Round(state, round);
        DummyRound(dummy_state1, DummyKeyCounter);DummyKeyCounter++;
    }
    else
    {
      DummyRound(dummy_state1, DummyKeyCounter);DummyKeyCounter++;
        Round(state, round);
    }
  }

  //Last round will also have a dummy added in random order
  if (RandomVector[round-1]==0)
  {
      LastRound(state, round);
      DummyLastRound(dummy_state1, DummyKeyCounter);
  }
  else
  {
      DummyLastRound(dummy_state1, DummyKeyCounter);
      LastRound(state, round);
  }
}
#endif

#ifdef TwoDummies
static void CipherWithDummies(void)
{
  uint8_t round;
  uint8_t DummyKeyCounter = 0;
  AddRoundKey(state, 0);

  //From round 1 to 9 two dummies per round are added in random order
  for(round=1; round < Nr; ++round)
    {
        if (RandomVector[round-1] == 0)
        {
            Round(state, round);
            DummyRound(dummy_state1,DummyKeyCounter);DummyKeyCounter++;
            DummyRound(dummy_state2,DummyKeyCounter);DummyKeyCounter++;
        }
        else if (RandomVector[round-1] == 1)
```

```
        {
            DummyRound(dummy_state1,DummyKeyCounter);DummyKeyCounter++;
            Round(state, round);
            DummyRound(dummy_state2,DummyKeyCounter);DummyKeyCounter++;
        }
        else
        {
            DummyRound(dummy_state1,DummyKeyCounter);DummyKeyCounter++;
            DummyRound(dummy_state2,DummyKeyCounter);DummyKeyCounter++;
            Round(state, round);
        }
    }

  //Last round will also have two dummies added in random order
  if (RandomVector[round-1] == 0)
  {
      LastRound(state, round);
      DummyLastRound(dummy_state1, DummyKeyCounter);DummyKeyCounter++;
      DummyLastRound(dummy_state2, DummyKeyCounter);
  }
  else if (RandomVector[round-1] == 1)
  {
      DummyLastRound(dummy_state1, DummyKeyCounter);DummyKeyCounter++;
      LastRound(state, round);
      DummyLastRound(dummy_state2, DummyKeyCounter);
  }
  else
  {
      DummyLastRound(dummy_state1, DummyKeyCounter);DummyKeyCounter++;
      DummyLastRound(dummy_state2, DummyKeyCounter);
      LastRound(state, round);
  }
}
#endif

/*****************************************************************************/
/* Public functions:                                                         */
/*****************************************************************************/
void AES128_ECB_indp_setkey(uint8_t* key)
{
  Key = key;
  KeyExpansion();
}

void AES128_ECB_indp_crypto(uint8_t* input)
{
  state = (state_t*)input;
  CipherWithDummies();
}

void AES128_ECB_indp_precompute_randoms(uint8_t* seed)
{
  uint32_t seed1 = seed[0];
  uint32_t seed2 = seed[1];
  uint32_t seed3 = seed[2];
  uint32_t seed4 = seed[3];
  uint32_t mySeed = (seed1 << 24) ^ (seed2 << 16) ^ (seed3 << 8) ^  seed4;
  srand(mySeed);
  RandomGeneration();
}
```

C.3 AES128 with randomization (only encryption)

```
/*****************************************************************************/
/* Includes:                                                                 */
/*****************************************************************************/
#include "aes.h"
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

/*****************************************************************************/
/*         Defines: Fixed values of the AES128 implementation                */
/*****************************************************************************/
// Number of rows and columns
#define Nb 4
// The number of rounds in AES128 cipher.
#define Nr 10
// Key length in bytes [128 bit]
#define KEYLEN 16

/*****************************************************************************/
/*        COMMENT/UNCOMMENT THE COUNTERMEASURES YOU WANT TO ACTIVATE         */
/*****************************************************************************/
#define Random_SBOX
#define Random_MixColumns
#define Random_AddRoundKey

/*****************************************************************************/
/* Private variables:                                                        */
/*****************************************************************************/
// state matrix holding the intermediate results during encryption
typedef uint8_t state_t[16];
static state_t* state;

// The array that stores the round keys (11keys x 16bytes = 176bytes).
static uint8_t RoundKey[176];

// The Key input to the AES Program
static uint8_t* Key;

// Depending on which randomization features were chosen, the corresponding
arrays will be generated
#if defined Random_SBOX || defined Random_AddRoundKey
static uint8_t random_sequence[160];
#endif
#ifdef Random_MixColumns
static uint8_t random_columns[40];
#endif

//AES SBOX
static const uint8_t sbox[256] =   {
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
0xd8, 0x31, 0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
0x27, 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
0xe3, 0x2f, 0x84,
```

```
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
0x4c, 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
0xff, 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
0x5d, 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
0x5e, 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
0x95, 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
0x7a, 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
0xbd, 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
0xc1, 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
0x54, 0xbb, 0x16};

// The round constant array
static const uint8_t Rcon[11] =
{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36};

/*****************************************************************************/
/* Private functions:                                                        */
/*****************************************************************************/
static uint8_t getSBoxValue(uint8_t num)
{
  return sbox[num];
}

// Function used for MixColumns computation
static uint8_t xtime(uint8_t x)
{
  return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
}

// Depending on which randomization features were chosen, the corresponding
functions will be generated
#if defined Random_SBOX || defined Random_AddRoundKey
void GenerateSBOXnAddRoundKeyRandomizationVector(void)
{
  for (int round=0; round<10; round++)
  {
       uint8_t random_vector[KEYLEN];
       uint8_t values[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
       uint8_t val;
       for (int i = 15; i >= 0 ;i--){
         val = rand() % (i+1);
         random_vector[i] = values[val];
         for( int j = val; j < i ; j++){
           values[j] = values[j+1];
         }
       }
       memcpy(random_sequence + round*16, random_vector, 16);
  }
}
```

```
#endif

#ifdef Random_MixColumns
void GenerateMixColumnRandomizationVector(void)
{
  for (int round=0; round<10; round++)
  {
       uint8_t random_vector[Nb];
       uint8_t values[4] = {0,1,2,3};
       uint8_t val;
       for (int i = 3; i >= 0 ;i--){
         val = rand() % (i+1);
         random_vector[i] = values[val];
         //shift all remaining values:
         for( int j = val; j < i ; j++){
           values[j] = values[j+1];
         }
       }
       memcpy(random_columns + round*4, random_vector, 4);
  }
}
#endif

//This function produces 11 round keys.
static void KeyExpansion(void)
{
  uint32_t i, j, k;
  uint8_t tempa[4]; // Used for the column/row operations

  // The first round key is the key itself.
  for(i = 0; i < Nb; ++i)
  {
    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
  }

  // All other round keys are found from the previous round keys.
  for(; (i < (Nb * (Nr + 1))); ++i)
  {
    for(j = 0; j < 4; ++j)
    {
      // Store previous key in tempa
      tempa[j]=RoundKey[(i-1) * 4 + j];
    }
    if (i % Nb == 0)
    {
      // Function RotWord()
      k = tempa[0];
      tempa[0] = tempa[1];
      tempa[1] = tempa[2];
      tempa[2] = tempa[3];
      tempa[3] = k;

      // Function Subword()
      tempa[0] = getSBoxValue(tempa[0]);
      tempa[1] = getSBoxValue(tempa[1]);
      tempa[2] = getSBoxValue(tempa[2]);
      tempa[3] = getSBoxValue(tempa[3]);
```

```
      // XOR with round constant
      tempa[0] =  tempa[0] ^ Rcon[i/Nb - 1];
    }
    // Add the round key to the array
    RoundKey[i * 4 + 0] = RoundKey[(i - Nb) * 4 + 0] ^ tempa[0];
    RoundKey[i * 4 + 1] = RoundKey[(i - Nb) * 4 + 1] ^ tempa[1];
    RoundKey[i * 4 + 2] = RoundKey[(i - Nb) * 4 + 2] ^ tempa[2];
    RoundKey[i * 4 + 3] = RoundKey[(i - Nb) * 4 + 3] ^ tempa[3];
  }
}


#ifndef Random_AddRoundKey
static void AddRoundKey(uint8_t round)
{
  for(uint8_t i=0; i<16; ++i)
  {
    (*state)[i] ^= RoundKey[round * KEYLEN + i];
  }
}
#endif

#ifdef Random_AddRoundKey
static void AddRoundKey(uint8_t round)
{
  for(uint8_t i=0; i<16; ++i)
  {
    (*state)[random_sequence[i]]^=RoundKey[round*KEYLEN+random_sequence[i]];
  }
}
#endif

#ifndef Random_SBOX
static void SubBytes(uint8_t round)
{
  uint8_t i;
  for(i=0;i<16;++i)
  {
    (*state)[i] = getSBoxValue((*state)[i]);
  }
}
#endif

#ifdef Random_SBOX
static void SubBytes(uint8_t round)
{
       for (uint8_t i=0; i<16; i++)
       {
         (*state)[random_sequence[(round-1)*KEYLEN + i]] =
         getSBoxValue((*state)[random_sequence[(round-1)*KEYLEN + i]]);
       }
}
#endif

static void ShiftRows(uint8_t round)
{
  uint8_t temp;

  temp           = (*state)[1];
  (*state)[1] = (*state)[5];
  (*state)[5] = (*state)[9];
```

```
  (*state)[9] = (*state)[13];
  (*state)[13] = temp;

  temp           = (*state)[2];
  (*state)[2] = (*state)[10];
  (*state)[10] = temp;

  temp       = (*state)[6];
  (*state)[6] = (*state)[14];
  (*state)[14] = temp;

  temp       = (*state)[3];
  (*state)[3] = (*state)[15];
  (*state)[15] = (*state)[11];
  (*state)[11] = (*state)[7];
  (*state)[7] = temp;
}


#ifndef Random_MixColumns
static void MixColumns(uint8_t round)
{
  uint8_t i;
  uint8_t Tmp,Tm,t;
  for(i = 0; i < 4; ++i)
  {
    t = (*state)[i][0];
    Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3];
    Tm =(*state)[i][0]^(*state)[i][1]; Tm=xtime(Tm); (*state)[i][0]^=Tm^Tmp;
    Tm =(*state)[i][1]^(*state)[i][2]; Tm=xtime(Tm); (*state)[i][1]^=Tm^Tmp;
    Tm =(*state)[i][2]^(*state)[i][3]; Tm=xtime(Tm); (*state)[i][2]^=Tm^Tmp;
    Tm  = (*state)[i][3] ^ t ;  Tm = xtime(Tm);  (*state)[i][3] ^= Tm ^ Tmp;
  }
}
#endif

#ifdef Random_MixColumns
static void MixColumns(uint8_t round)
{
  uint8_t i, j;
  uint8_t Tmp,Tm,t;
  for(i = 0; i < 4; ++i)
  {
    j = random_columns[(round-1)*4 + i];
    t = (*state)[0 + j*4];
    Tmp=(*state)[0+j*4]^(*state)[1 + j*4]^(*state)[2 + j*4]^(*state)[3+j*4];
    Tm=(*state)[0+j*4]^(*state)[1+j*4];Tm=xtime(Tm);(*state)[0+j*4]^=Tm^Tmp;
    Tm=(*state)[1+j*4]^(*state)[2+j*4];Tm=xtime(Tm);(*state)[1+j*4]^=Tm^Tmp;
    Tm=(*state)[2+j*4]^(*state)[3+j*4];Tm=xtime(Tm);(*state)[2+j*4]^=Tm^Tmp;
    Tm=(*state)[3+j*4]^t;              Tm=xtime(Tm);(*state)[3+j*4]^=Tm^Tmp;
  }
}
#endif

static void Round(uint8_t round)
{
    SubBytes(round);
    ShiftRows(round);
    MixColumns(round);
    AddRoundKey(round);
}
```

```
static void LastRound(void)
{
    SubBytes(Nr);
    ShiftRows(Nr);
    AddRoundKey(Nr);
}

// Normal encryption function
static void Cipher(void)
{
  uint8_t round = 0;
  AddRoundKey(round);

  for(round = 1; round < Nr; ++round)
    {Round(round);}
  LastRound();
}

/*****************************************************************************/
/* Public functions:                                                         */
/*****************************************************************************/

void AES128_ECB_indp_setkey(uint8_t* key)
{
  Key = key;
  KeyExpansion();
}

void AES128_ECB_indp_crypto(uint8_t* input)
{
  state = (state_t*)input;
  Cipher();
}

void AES128_ECB_indp_precompute_randoms(uint8_t* seed)
{
  #if defined Random_SBOX || defined Random_AddRoundKey || defined
Random_MixColumns
  uint32_t seed1 = seed[0];
  uint32_t seed2 = seed[1];
  uint32_t seed3 = seed[2];
  uint32_t seed4 = seed[3];
  uint32_t mySeed = (seed1 << 24) ^ (seed2 << 16) ^ (seed3 << 8) ^  seed4;
  srand(mySeed);
  #endif

  #if defined Random_SBOX || defined Random_AddRoundKey
  GenerateSBOXnAddRoundKeyRandomizationVector();
  #endif
  #ifdef Random_MixColumns
  GenerateMixColumnRandomizationVector();
  #endif
}
```

C.4 Random delay implementation

```
// This vector stores the random values used in the delay function
#define RandomVectorLength 10
static uint8_t RandomVector[RandomVectorLength];
```

```
// Generation of random values for the delay function. Max argument sets the top
limit of the uniform distribution
static void UniformRandom(uint8_t Max)
{
  for (uint8_t i = 0; i<RandomVectorLength ; i++)
       {
        RandomVector[i] = (rand() % Max)+1;
       }
}

// This is the optimized delay function (dummy loop)
static void Delay(uint8_t DelayLenght)
{
  asm volatile("MOV R16, %0" : "=r" (DelayLenght) : "0" (DelayLenght));
  asm volatile("LOOP:              \n"
               "DEC R16            \n"
               "BRNE LOOP            ");
}
```

C.5 AES128 with Boolean masking

```
/*****************************************************************************/
/* Includes:                                                                 */
/*****************************************************************************/
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "aes.h"

/*****************************************************************************/
/* Defines:                                                                  */
/*****************************************************************************/
// Number of rows and columns
#define Nb 4
// The number of rounds in AES128 cipher.
#define Nr 10
// Key length in bytes [128 bit]
#define KEYLEN 16

/*****************************************************************************/
/* Masked = 1 (apply masking) ; Masked = 0 (no masking applied)               */
/*****************************************************************************/
#define MASKED 1

/*****************************************************************************/
/* Private variables:                                                        */
/*****************************************************************************/
typedef uint8_t state_t[4][4];
static state_t* state;

//For debugging purposes uncomment
//static uint8_t state_debbug_t[4][4];
//static state_t* state_debbug = &state_debbug_t;

// The array that stores the round keys (11keys x 16bytes = 176bytes).
static uint8_t RoundKey[11*KEYLEN];

// The Key input to the AES Program
```

```
static uint8_t* Key;

// Vector to store the masked round keys
static uint8_t RoundKeyMasked[11*KEYLEN];

// Vector to store the masks
static uint8_t mask[10];

static uint8_t round;

// AES SBOX
static const uint8_t sbox[256] = {
    //0     1    2      3     4    5     6     7      8    9     A      B    C
D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
0x54, 0xbb, 0x16};

// AES reverse SBOX
static const uint8_t rsbox[256] = {
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81,
0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4,
0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42,
0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d,
0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d,
0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7,
0x8d, 0x9d, 0x84,
```

```
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8,
0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01,
0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0,
0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c,
0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa,
0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78,
0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27,
0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93,
0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83,
0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55,
0x21, 0x0c, 0x7d};

// Galois multiplication LUTs for MixColumns and InvMixColumns
static const uint8_t mul_02[256] = {
    0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e, 0x10, 0x12, 0x14, 0x16, 0x18,
0x1a, 0x1c, 0x1e,
    0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e, 0x30, 0x32, 0x34, 0x36, 0x38,
0x3a, 0x3c, 0x3e,
    0x40, 0x42, 0x44, 0x46, 0x48, 0x4a, 0x4c, 0x4e, 0x50, 0x52, 0x54, 0x56, 0x58,
0x5a, 0x5c, 0x5e,
    0x60, 0x62, 0x64, 0x66, 0x68, 0x6a, 0x6c, 0x6e, 0x70, 0x72, 0x74, 0x76, 0x78,
0x7a, 0x7c, 0x7e,
    0x80, 0x82, 0x84, 0x86, 0x88, 0x8a, 0x8c, 0x8e, 0x90, 0x92, 0x94, 0x96, 0x98,
0x9a, 0x9c, 0x9e,
    0xa0, 0xa2, 0xa4, 0xa6, 0xa8, 0xaa, 0xac, 0xae, 0xb0, 0xb2, 0xb4, 0xb6, 0xb8,
0xba, 0xbc, 0xbe,
    0xc0, 0xc2, 0xc4, 0xc6, 0xc8, 0xca, 0xcc, 0xce, 0xd0, 0xd2, 0xd4, 0xd6, 0xd8,
0xda, 0xdc, 0xde,
    0xe0, 0xe2, 0xe4, 0xe6, 0xe8, 0xea, 0xec, 0xee, 0xf0, 0xf2, 0xf4, 0xf6, 0xf8,
0xfa, 0xfc, 0xfe,
    0x1b, 0x19, 0x1f, 0x1d, 0x13, 0x11, 0x17, 0x15, 0x0b, 0x09, 0x0f, 0x0d, 0x03,
0x01, 0x07, 0x05,
    0x3b, 0x39, 0x3f, 0x3d, 0x33, 0x31, 0x37, 0x35, 0x2b, 0x29, 0x2f, 0x2d, 0x23,
0x21, 0x27, 0x25,
    0x5b, 0x59, 0x5f, 0x5d, 0x53, 0x51, 0x57, 0x55, 0x4b, 0x49, 0x4f, 0x4d, 0x43,
0x41, 0x47, 0x45,
    0x7b, 0x79, 0x7f, 0x7d, 0x73, 0x71, 0x77, 0x75, 0x6b, 0x69, 0x6f, 0x6d, 0x63,
0x61, 0x67, 0x65,
    0x9b, 0x99, 0x9f, 0x9d, 0x93, 0x91, 0x97, 0x95, 0x8b, 0x89, 0x8f, 0x8d, 0x83,
0x81, 0x87, 0x85,
    0xbb, 0xb9, 0xbf, 0xbd, 0xb3, 0xb1, 0xb7, 0xb5, 0xab, 0xa9, 0xaf, 0xad, 0xa3,
0xa1, 0xa7, 0xa5,
    0xdb, 0xd9, 0xdf, 0xdd, 0xd3, 0xd1, 0xd7, 0xd5, 0xcb, 0xc9, 0xcf, 0xcd, 0xc3,
0xc1, 0xc7, 0xc5,
    0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5, 0xeb, 0xe9, 0xef, 0xed, 0xe3,
0xe1, 0xe7, 0xe5};

static const uint8_t mul_03[256] = {
    0x00, 0x03, 0x06, 0x05, 0x0c, 0x0f, 0x0a, 0x09, 0x18, 0x1b, 0x1e, 0x1d, 0x14,
0x17, 0x12, 0x11,
    0x30, 0x33, 0x36, 0x35, 0x3c, 0x3f, 0x3a, 0x39, 0x28, 0x2b, 0x2e, 0x2d, 0x24,
0x27, 0x22, 0x21,
```

```
    0x60, 0x63, 0x66, 0x65, 0x6c, 0x6f, 0x6a, 0x69, 0x78, 0x7b, 0x7e, 0x7d, 0x74,
0x77, 0x72, 0x71,
    0x50, 0x53, 0x56, 0x55, 0x5c, 0x5f, 0x5a, 0x59, 0x48, 0x4b, 0x4e, 0x4d, 0x44,
0x47, 0x42, 0x41,
    0xc0, 0xc3, 0xc6, 0xc5, 0xcc, 0xcf, 0xca, 0xc9, 0xd8, 0xdb, 0xde, 0xdd, 0xd4,
0xd7, 0xd2, 0xd1,
    0xf0, 0xf3, 0xf6, 0xf5, 0xfc, 0xff, 0xfa, 0xf9, 0xe8, 0xeb, 0xee, 0xed, 0xe4,
0xe7, 0xe2, 0xe1,
    0xa0, 0xa3, 0xa6, 0xa5, 0xac, 0xaf, 0xaa, 0xa9, 0xb8, 0xbb, 0xbe, 0xbd, 0xb4,
0xb7, 0xb2, 0xb1,
    0x90, 0x93, 0x96, 0x95, 0x9c, 0x9f, 0x9a, 0x99, 0x88, 0x8b, 0x8e, 0x8d, 0x84,
0x87, 0x82, 0x81,
    0x9b, 0x98, 0x9d, 0x9e, 0x97, 0x94, 0x91, 0x92, 0x83, 0x80, 0x85, 0x86, 0x8f,
0x8c, 0x89, 0x8a,
    0xab, 0xa8, 0xad, 0xae, 0xa7, 0xa4, 0xa1, 0xa2, 0xb3, 0xb0, 0xb5, 0xb6, 0xbf,
0xbc, 0xb9, 0xba,
    0xfb, 0xf8, 0xfd, 0xfe, 0xf7, 0xf4, 0xf1, 0xf2, 0xe3, 0xe0, 0xe5, 0xe6, 0xef,
0xec, 0xe9, 0xea,
    0xcb, 0xc8, 0xcd, 0xce, 0xc7, 0xc4, 0xc1, 0xc2, 0xd3, 0xd0, 0xd5, 0xd6, 0xdf,
0xdc, 0xd9, 0xda,
    0x5b, 0x58, 0x5d, 0x5e, 0x57, 0x54, 0x51, 0x52, 0x43, 0x40, 0x45, 0x46, 0x4f,
0x4c, 0x49, 0x4a,
    0x6b, 0x68, 0x6d, 0x6e, 0x67, 0x64, 0x61, 0x62, 0x73, 0x70, 0x75, 0x76, 0x7f,
0x7c, 0x79, 0x7a,
    0x3b, 0x38, 0x3d, 0x3e, 0x37, 0x34, 0x31, 0x32, 0x23, 0x20, 0x25, 0x26, 0x2f,
0x2c, 0x29, 0x2a,
    0x0b, 0x08, 0x0d, 0x0e, 0x07, 0x04, 0x01, 0x02, 0x13, 0x10, 0x15, 0x16, 0x1f,
0x1c, 0x19, 0x1a};

static const uint8_t mul_09[256]={
      0x00,0x09,0x12,0x1b,0x24,0x2d,0x36,0x3f,0x48,0x41,0x5a,0x53,0x6c,0x65,0x7e
,0x77,
      0x90,0x99,0x82,0x8b,0xb4,0xbd,0xa6,0xaf,0xd8,0xd1,0xca,0xc3,0xfc,0xf5,0xee
,0xe7,
      0x3b,0x32,0x29,0x20,0x1f,0x16,0x0d,0x04,0x73,0x7a,0x61,0x68,0x57,0x5e,0x45
,0x4c,
      0xab,0xa2,0xb9,0xb0,0x8f,0x86,0x9d,0x94,0xe3,0xea,0xf1,0xf8,0xc7,0xce,0xd5
,0xdc,
      0x76,0x7f,0x64,0x6d,0x52,0x5b,0x40,0x49,0x3e,0x37,0x2c,0x25,0x1a,0x13,0x08
,0x01,
      0xe6,0xef,0xf4,0xfd,0xc2,0xcb,0xd0,0xd9,0xae,0xa7,0xbc,0xb5,0x8a,0x83,0x98
,0x91,
      0x4d,0x44,0x5f,0x56,0x69,0x60,0x7b,0x72,0x05,0x0c,0x17,0x1e,0x21,0x28,0x33
,0x3a,
      0xdd,0xd4,0xcf,0xc6,0xf9,0xf0,0xeb,0xe2,0x95,0x9c,0x87,0x8e,0xb1,0xb8,0xa3
,0xaa,
      0xec,0xe5,0xfe,0xf7,0xc8,0xc1,0xda,0xd3,0xa4,0xad,0xb6,0xbf,0x80,0x89,0x92
,0x9b,
      0x7c,0x75,0x6e,0x67,0x58,0x51,0x4a,0x43,0x34,0x3d,0x26,0x2f,0x10,0x19,0x02
,0x0b,
      0xd7,0xde,0xc5,0xcc,0xf3,0xfa,0xe1,0xe8,0x9f,0x96,0x8d,0x84,0xbb,0xb2,0xa9
,0xa0,
      0x47,0x4e,0x55,0x5c,0x63,0x6a,0x71,0x78,0x0f,0x06,0x1d,0x14,0x2b,0x22,0x39
,0x30,
      0x9a,0x93,0x88,0x81,0xbe,0xb7,0xac,0xa5,0xd2,0xdb,0xc0,0xc9,0xf6,0xff,0xe4
,0xed,
      0x0a,0x03,0x18,0x11,0x2e,0x27,0x3c,0x35,0x42,0x4b,0x50,0x59,0x66,0x6f,0x74
,0x7d,
      0xa1,0xa8,0xb3,0xba,0x85,0x8c,0x97,0x9e,0xe9,0xe0,0xfb,0xf2,0xcd,0xc4,0xdf
,0xd6,
```

```
        0x31,0x38,0x23,0x2a,0x15,0x1c,0x07,0x0e,0x79,0x70,0x6b,0x62,0x5d,0x54,0x4f
,0x46
};

static const uint8_t mul_11[256]={
        0x00,0x0b,0x16,0x1d,0x2c,0x27,0x3a,0x31,0x58,0x53,0x4e,0x45,0x74,0x7f,0x62
,0x69,
        0xb0,0xbb,0xa6,0xad,0x9c,0x97,0x8a,0x81,0xe8,0xe3,0xfe,0xf5,0xc4,0xcf,0xd2
,0xd9,
        0x7b,0x70,0x6d,0x66,0x57,0x5c,0x41,0x4a,0x23,0x28,0x35,0x3e,0x0f,0x04,0x19
,0x12,
        0xcb,0xc0,0xdd,0xd6,0xe7,0xec,0xf1,0xfa,0x93,0x98,0x85,0x8e,0xbf,0xb4,0xa9
,0xa2,
        0xf6,0xfd,0xe0,0xeb,0xda,0xd1,0xcc,0xc7,0xae,0xa5,0xb8,0xb3,0x82,0x89,0x94
,0x9f,
        0x46,0x4d,0x50,0x5b,0x6a,0x61,0x7c,0x77,0x1e,0x15,0x08,0x03,0x32,0x39,0x24
,0x2f,
        0x8d,0x86,0x9b,0x90,0xa1,0xaa,0xb7,0xbc,0xd5,0xde,0xc3,0xc8,0xf9,0xf2,0xef
,0xe4,
        0x3d,0x36,0x2b,0x20,0x11,0x1a,0x07,0x0c,0x65,0x6e,0x73,0x78,0x49,0x42,0x5f
,0x54,
        0xf7,0xfc,0xe1,0xea,0xdb,0xd0,0xcd,0xc6,0xaf,0xa4,0xb9,0xb2,0x83,0x88,0x95
,0x9e,
        0x47,0x4c,0x51,0x5a,0x6b,0x60,0x7d,0x76,0x1f,0x14,0x09,0x02,0x33,0x38,0x25
,0x2e,
        0x8c,0x87,0x9a,0x91,0xa0,0xab,0xb6,0xbd,0xd4,0xdf,0xc2,0xc9,0xf8,0xf3,0xee
,0xe5,
        0x3c,0x37,0x2a,0x21,0x10,0x1b,0x06,0x0d,0x64,0x6f,0x72,0x79,0x48,0x43,0x5e
,0x55,
        0x01,0x0a,0x17,0x1c,0x2d,0x26,0x3b,0x30,0x59,0x52,0x4f,0x44,0x75,0x7e,0x63
,0x68,
        0xb1,0xba,0xa7,0xac,0x9d,0x96,0x8b,0x80,0xe9,0xe2,0xff,0xf4,0xc5,0xce,0xd3
,0xd8,
        0x7a,0x71,0x6c,0x67,0x56,0x5d,0x40,0x4b,0x22,0x29,0x34,0x3f,0x0e,0x05,0x18
,0x13,
        0xca,0xc1,0xdc,0xd7,0xe6,0xed,0xf0,0xfb,0x92,0x99,0x84,0x8f,0xbe,0xb5,0xa8
,0xa3
};

static const uint8_t mul_13[256]={
        0x00,0x0d,0x1a,0x17,0x34,0x39,0x2e,0x23,0x68,0x65,0x72,0x7f,0x5c,0x51,0x46
,0x4b,
        0xd0,0xdd,0xca,0xc7,0xe4,0xe9,0xfe,0xf3,0xb8,0xb5,0xa2,0xaf,0x8c,0x81,0x96
,0x9b,
        0xbb,0xb6,0xa1,0xac,0x8f,0x82,0x95,0x98,0xd3,0xde,0xc9,0xc4,0xe7,0xea,0xfd
,0xf0,
        0x6b,0x66,0x71,0x7c,0x5f,0x52,0x45,0x48,0x03,0x0e,0x19,0x14,0x37,0x3a,0x2d
,0x20,
        0x6d,0x60,0x77,0x7a,0x59,0x54,0x43,0x4e,0x05,0x08,0x1f,0x12,0x31,0x3c,0x2b
,0x26,
        0xbd,0xb0,0xa7,0xaa,0x89,0x84,0x93,0x9e,0xd5,0xd8,0xcf,0xc2,0xe1,0xec,0xfb
,0xf6,
        0xd6,0xdb,0xcc,0xc1,0xe2,0xef,0xf8,0xf5,0xbe,0xb3,0xa4,0xa9,0x8a,0x87,0x90
,0x9d,
        0x06,0x0b,0x1c,0x11,0x32,0x3f,0x28,0x25,0x6e,0x63,0x74,0x79,0x5a,0x57,0x40
,0x4d,
        0xda,0xd7,0xc0,0xcd,0xee,0xe3,0xf4,0xf9,0xb2,0xbf,0xa8,0xa5,0x86,0x8b,0x9c
,0x91,
        0x0a,0x07,0x10,0x1d,0x3e,0x33,0x24,0x29,0x62,0x6f,0x78,0x75,0x56,0x5b,0x4c
,0x41,
```

```
        0x61,0x6c,0x7b,0x76,0x55,0x58,0x4f,0x42,0x09,0x04,0x13,0x1e,0x3d,0x30,0x27
,0x2a,
        0xb1,0xbc,0xab,0xa6,0x85,0x88,0x9f,0x92,0xd9,0xd4,0xc3,0xce,0xed,0xe0,0xf7
,0xfa,
        0xb7,0xba,0xad,0xa0,0x83,0x8e,0x99,0x94,0xdf,0xd2,0xc5,0xc8,0xeb,0xe6,0xf1
,0xfc,
        0x67,0x6a,0x7d,0x70,0x53,0x5e,0x49,0x44,0x0f,0x02,0x15,0x18,0x3b,0x36,0x21
,0x2c,
        0x0c,0x01,0x16,0x1b,0x38,0x35,0x22,0x2f,0x64,0x69,0x7e,0x73,0x50,0x5d,0x4a
,0x47,
        0xdc,0xd1,0xc6,0xcb,0xe8,0xe5,0xf2,0xff,0xb4,0xb9,0xae,0xa3,0x80,0x8d,0x9a
,0x97
};

static const uint8_t mul_14[256]={
        0x00,0x0e,0x1c,0x12,0x38,0x36,0x24,0x2a,0x70,0x7e,0x6c,0x62,0x48,0x46,0x54
,0x5a,
        0xe0,0xee,0xfc,0xf2,0xd8,0xd6,0xc4,0xca,0x90,0x9e,0x8c,0x82,0xa8,0xa6,0xb4
,0xba,
        0xdb,0xd5,0xc7,0xc9,0xe3,0xed,0xff,0xf1,0xab,0xa5,0xb7,0xb9,0x93,0x9d,0x8f
,0x81,
        0x3b,0x35,0x27,0x29,0x03,0x0d,0x1f,0x11,0x4b,0x45,0x57,0x59,0x73,0x7d,0x6f
,0x61,
        0xad,0xa3,0xb1,0xbf,0x95,0x9b,0x89,0x87,0xdd,0xd3,0xc1,0xcf,0xe5,0xeb,0xf9
,0xf7,
        0x4d,0x43,0x51,0x5f,0x75,0x7b,0x69,0x67,0x3d,0x33,0x21,0x2f,0x05,0x0b,0x19
,0x17,
        0x76,0x78,0x6a,0x64,0x4e,0x40,0x52,0x5c,0x06,0x08,0x1a,0x14,0x3e,0x30,0x22
,0x2c,
        0x96,0x98,0x8a,0x84,0xae,0xa0,0xb2,0xbc,0xe6,0xe8,0xfa,0xf4,0xde,0xd0,0xc2
,0xcc,
        0x41,0x4f,0x5d,0x53,0x79,0x77,0x65,0x6b,0x31,0x3f,0x2d,0x23,0x09,0x07,0x15
,0x1b,
        0xa1,0xaf,0xbd,0xb3,0x99,0x97,0x85,0x8b,0xd1,0xdf,0xcd,0xc3,0xe9,0xe7,0xf5
,0xfb,
        0x9a,0x94,0x86,0x88,0xa2,0xac,0xbe,0xb0,0xea,0xe4,0xf6,0xf8,0xd2,0xdc,0xce
,0xc0,
        0x7a,0x74,0x66,0x68,0x42,0x4c,0x5e,0x50,0x0a,0x04,0x16,0x18,0x32,0x3c,0x2e
,0x20,
        0xec,0xe2,0xf0,0xfe,0xd4,0xda,0xc8,0xc6,0x9c,0x92,0x80,0x8e,0xa4,0xaa,0xb8
,0xb6,
        0x0c,0x02,0x10,0x1e,0x34,0x3a,0x28,0x26,0x7c,0x72,0x60,0x6e,0x44,0x4a,0x58
,0x56,
        0x37,0x39,0x2b,0x25,0x0f,0x01,0x13,0x1d,0x47,0x49,0x5b,0x55,0x7f,0x71,0x63
,0x6d,
        0xd7,0xd9,0xcb,0xc5,0xef,0xe1,0xf3,0xfd,0xa7,0xa9,0xbb,0xb5,0x9f,0x91,0x83
,0x8d
};

// The round constant array
static const uint8_t Rcon[10] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
0x1b, 0x36};

// Arrays to store the computed masked AES SBOX and reverse SBOX LUTs
static uint8_t SboxMasked[256];
static uint8_t rSboxMasked[256];

/*****************************************************************************/
/* Private functions:                                                        */
/*****************************************************************************/
```

```
/* For debugging purposes
static void BlockCopy(uint8_t* output, const uint8_t* input)
{
  for (uint8_t i=0; i<AES_KEYLEN; ++i)
  {
    output[i] = input[i];
  }
}
*/

/* Normal Mix Columns:
* [w]      [2  3  1  1]    [a]
* [x]  =   [1  2  3  1]  * [b]
* [y]      [1  1  2  3]    [c]
* [z]      [3  1  1  2]    [d]
// Transform m0, m1, m2 and m3 into m6, m7, m8 and m9
static void calcMixColmask(uint8_t mask[10])
{
  mask[6] = mul_02[mask[0]] ^ mul_03[mask[1]] ^ mask[2]         ^ mask[3];
  mask[7] = mask[0]         ^ mul_02[mask[1]] ^ mul_03[mask[2]] ^ mask[3];
  mask[8] = mask[0]         ^ mask[1]     ^ mul_02[mask[2]] ^ mul_03[mask[3]];
  mask[9] = mul_03[mask[0]] ^ mask[1]     ^ mask[2]         ^ mul_02[mask[3]];
}

/* Normal Inverse Mix Columns:
* [w]      [0E  0B  0D  09]    [a]
* [x]  =   [09  0E  0B  0D]  * [b]
* [y]      [0D  09  0E  0B]    [c]
* [z]      [0B  0D  09  0E]    [d]
// Transform m0, m1, m2 and m3 into m6, m7, m8 and m9 for inverse cypher
static void calcInvMixColmask(uint8_t mask[10])
{
mask[6]=mul_14[mask[0]]^ mul_11[mask[1]]^ mul_13[mask[2]]^ mul_09[mask[3]];
mask[7]=mul_09[mask[0]]^ mul_14[mask[1]]^ mul_11[mask[2]]^ mul_13[mask[3]];
mask[8]=mul_13[mask[0]]^ mul_09[mask[1]]^ mul_14[mask[2]]^ mul_11[mask[3]];
mask[9]=mul_11[mask[0]]^ mul_13[mask[1]]^ mul_09[mask[2]]^ mul_14[mask[3]];
}

//XOR-ing function for mask manipulation
static void remask(state_t * s, uint8_t m1, uint8_t m2, uint8_t m3, uint8_t m4,
uint8_t m5, uint8_t m6, uint8_t m7, uint8_t m8)
{
  for (int i = 0; i < 4; i++)
  {
    (*s)[i][0] = (*s)[i][0] ^ (m1);
    (*s)[i][0] = (*s)[i][0] ^ (m5);
    (*s)[i][1] = (*s)[i][1] ^ (m2);
    (*s)[i][1] = (*s)[i][1] ^ (m6);
    (*s)[i][2] = (*s)[i][2] ^ (m3);
    (*s)[i][2] = (*s)[i][2] ^ (m7);
    (*s)[i][3] = (*s)[i][3] ^ (m4);
    (*s)[i][3] = (*s)[i][3] ^ (m8);
  }
}

//Calculate the Sbox to change from m4 to m5
static void calcSboxMasked(uint8_t mask[10])
{
  for (int i = 0; i < 256; i++){
    SboxMasked[i ^ mask[4]] = sbox[i] ^ mask[5];
  }
```

```
}

//Calculate the ReverseSbox to change from m4 to m5
static void calcrSboxMasked(uint8_t mask[10])
{
  for (int i = 0; i < 256; i++){
    rSboxMasked[i ^ mask[4]] = rsbox[i] ^ mask[5];
  }
}

//Precompute all the masks, masked round keys and masked SBOX
static void InitMaskingEncrypt(void)
{
  memcpy(RoundKeyMasked, RoundKey, AES_keyExpSize);

  //Randomly generate the masks: m0 m1 m2 m3 m4 and m5
  for (uint8_t i = 0; i < 6; i++){
    mask[i] = (rand() % 255)+1;
  }

  //Calculate m6, m7, m8 and m9
  calcMixColmask(mask);

  //Calculate the masked Sbox
  calcSboxMasked(mask);


  //Mask the last round key in order to unmake masking at the end of cypher
  remask((state_t *) &RoundKeyMasked[(Nr * Nb * 4)], 0, 0, 0, 0, mask[5],
mask[5], mask[5], mask[5]);

  //Mask the rest of round keys change from m6, m7, m8 and m9 to m4
  for (int i = 0; i < Nr; i++)
  {
    remask((state_t *) &RoundKeyMasked[(i * Nb * 4)], mask[6], mask[7], mask[8],
mask[9], mask[4], mask[4], mask[4], mask[4]);
  }
}

//Precompute all the masks, masked round keys and masked reverse SBOX
static void InitMaskingDecrypt(void)
{
  memcpy(RoundKeyMasked, RoundKey, AES_keyExpSize);

  //Randomly generate the masks: m0 m1 m2 m3 m4 and m5
  for (uint8_t i = 0; i < 6; i++){
    mask[i] = rand() % 0xFF;
  }

  //Calculate m6, m7, m8 and m9
  calcInvMixColmask(mask);

  //Calculate the masked reverse Sbox
  calcrSboxMasked(mask);

  //Mask the last round key in order to unmake masking at the end of decypher
  remask((state_t *) &RoundKeyMasked[(Nr * Nb * 4)], 0, 0, 0, 0, mask[4],
mask[4], mask[4], mask[4]);

  //Mask the rest of round keys change from m6, m7, m8 and m9 to m4
  for (int i = 0; i < Nr; i++)
```

```
  {
    remask((state_t *) &RoundKeyMasked[(i * Nb * 4)], mask[0], mask[1], mask[2],
mask[3], mask[5], mask[5], mask[5], mask[5]);
  }
}


#define getSBoxValue(num) (sbox[(num)])
#define getSBoxInvert(num) (rsbox[(num)])

// The SubBytesMasked Function Substitutes the values in the state matrix with
values in the masked S-box.
static void SubBytesMasked(state_t *state)
{
  uint8_t i, j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[i][j] = SboxMasked[(*state)[i][j]];
    }
  }
}

// This function adds the masked round key to state.
static void AddRoundKeyMasked(uint8_t round, state_t *state, const uint8_t *
RoundKeyMasked) //, const uint8_t* RoundKey)
{
  uint8_t i, j;
  for(i = 0; i < 4; i++){
    for (j = 0; j < 4; ++j)
    {
      (*state)[i][j] ^= RoundKeyMasked[(round * Nb * Nb) + (i * Nb) + j];
    }
  }
}

//Key schedule
static void KeyExpansion()
{
  unsigned i, j, k;
  uint8_t tempa[4]; // Used for the column/row operations

  // The first round key is the key itself.
  for (i = 0; i < Nk; ++i)
  {
    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
  }

  // All other round keys are found from the previous round keys.
  for (i = Nk; i < Nb * (Nr + 1); ++i)
  {
    {
      k = (i - 1) * 4;
      tempa[0] = RoundKey[k + 0];
      tempa[1] = RoundKey[k + 1];
      tempa[2] = RoundKey[k + 2];
      tempa[3] = RoundKey[k + 3];
```

```
    }

    if (i % Nk == 0)
    {
      // This function shifts the 4 bytes in a word to the left once.
      // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]

      // Function RotWord()
      {
        const uint8_t u8tmp = tempa[0];
        tempa[0] = tempa[1];
        tempa[1] = tempa[2];
        tempa[2] = tempa[3];
        tempa[3] = u8tmp;
      }

      // SubWord() is a function that takes a four-byte input word and
      // applies the S-box to each of the four bytes to produce an output word.

      // Function Subword()
      {
        tempa[0] = getSBoxValue(tempa[0]);
        tempa[1] = getSBoxValue(tempa[1]);
        tempa[2] = getSBoxValue(tempa[2]);
        tempa[3] = getSBoxValue(tempa[3]);
      }

      tempa[0] = tempa[0] ^ Rcon[i / Nk - 1];
    }

    j = i * 4;
    k = (i - Nk) * 4;
    RoundKey[j + 0] = RoundKey[k + 0] ^ tempa[0];
    RoundKey[j + 1] = RoundKey[k + 1] ^ tempa[1];
    RoundKey[j + 2] = RoundKey[k + 2] ^ tempa[2];
    RoundKey[j + 3] = RoundKey[k + 3] ^ tempa[3];
  }
}


// This function adds the round key to state.
static void AddRoundKey(uint8_t round, state_t *state, const uint8_t *RoundKey)
{
  uint8_t i, j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[i][j] ^= RoundKey[(round * Nb * 4) + (i * Nb) + j];
    }
  }
}

// The SubBytes function
static void SubBytes(state_t *state)
{
  uint8_t i, j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
```

```
      (*state)[j][i] = getSBoxValue((*state)[j][i]);
    }
  }
}

// The ShiftRows function
static void ShiftRows(state_t *state)
{
  uint8_t temp;

  // Rotate first row 1 columns to left
  temp = (*state)[0][1];
  (*state)[0][1] = (*state)[1][1];
  (*state)[1][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[3][1];
  (*state)[3][1] = temp;

  // Rotate second row 2 columns to left
  temp = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate third row 3 columns to left
  temp = (*state)[0][3];
  (*state)[0][3] = (*state)[3][3];
  (*state)[3][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[1][3];
  (*state)[1][3] = temp;
}

static uint8_t xtime(uint8_t x)
{
  return ((x << 1) ^ (((x >> 7) & 1) * 0x1b));
}

// MixColumns function
static void MixColumns(state_t *state)
{
  uint8_t i;
  uint8_t Tmp,Tm,t;
  for(i = 0; i < 4; ++i)
  {
    t = (*state)[i][0];
    Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3];
    Tm =(*state)[i][0]^(*state)[i][1]; Tm=xtime(Tm); (*state)[i][0]^=Tm^Tmp;
    Tm =(*state)[i][1]^(*state)[i][2]; Tm=xtime(Tm); (*state)[i][1]^=Tm^Tmp;
    Tm =(*state)[i][2]^(*state)[i][3]; Tm=xtime(Tm); (*state)[i][2]^=Tm^Tmp;
    Tm  = (*state)[i][3] ^ t ;  Tm = xtime(Tm);  (*state)[i][3] ^= Tm ^ Tmp;
  }
}

#define Multiply(x, y)                          \
  (((y & 1) * x) ^                              \
   ((y >> 1 & 1) * xtime(x)) ^                  \
   ((y >> 2 & 1) * xtime(xtime(x))) ^           \
   ((y >> 3 & 1) * xtime(xtime(xtime(x)))) ^ \
   ((y >> 4 & 1) * xtime(xtime(xtime(xtime(x))))))
```

```
#endif

// InvMixColumns function
static void InvMixColumns()
{
  int i;
  uint8_t a, b, c, d;
  for (i = 0; i < 4; ++i)
  {
    a = (*state)[i][0];
    b = (*state)[i][1];
    c = (*state)[i][2];
    d = (*state)[i][3];

    (*state)[i][0] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^
Multiply(d, 0x09);
    (*state)[i][1] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^
Multiply(d, 0x0d);
    (*state)[i][2] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^
Multiply(d, 0x0b);
    (*state)[i][3] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^
Multiply(d, 0x0e);
  }
}

// The InvSubBytes function
static void InvSubBytes()
{
  uint8_t i, j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[j][i] = getSBoxInvert((*state)[j][i]);
    }
  }
}

// The masked InvSubBytes
static void InvSubBytesMasked()
{
  uint8_t i, j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[j][i] = rSboxMasked[(*state)[j][i]];
    }
  }
}

// InvShiftRows function
static void InvShiftRows()
{
  uint8_t temp;

  // Rotate first row 1 columns to right
  temp = (*state)[3][1];
  (*state)[3][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[1][1];
```

```
  (*state)[1][1] = (*state)[0][1];
  (*state)[0][1] = temp;

  // Rotate second row 2 columns to right
  temp = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate third row 3 columns to right
  temp = (*state)[0][3];
  (*state)[0][3] = (*state)[1][3];
  (*state)[1][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[3][3];
  (*state)[3][3] = temp;
}

// Masked cyher function
static void CipherMasked()
{
  //Plain text masked with m6,m7,m8,m9
  remask(state, mask[6], mask[7], mask[8], mask[9], 0, 0, 0, 0);

  // Masks change from m6,m7,m8,m9 to m4
  AddRoundKeyMasked(0, state, RoundKeyMasked);
//  AddRoundKey(0, state_debbug, RoundKey);

  // All rounds, but last one without MixColumns()
  for (round = 1;; round++)
  {
    // Mask changes from m4 to m5
    SubBytesMasked(state);
 //   SubBytes(state_debbug);

    //No impact on mask
    ShiftRows(state);
//    ShiftRows(state_debbug);
    if (round == Nr)
    {
      break;
    }
    //Change mask from m5 to m0,m1,m2,m3
    remask(state, mask[0], mask[1], mask[2], mask[3], mask[5], mask[5], mask[5],
mask[5]);

    // Masks change from m0,m1,m2,m3 to m6,m7,m8,m9
    MixColumns(state);
 //   MixColumns(state_debbug);


    // Masks change from m6,m7,m8,m9 to m4
    AddRoundKeyMasked(round, state, RoundKeyMasked);
//    AddRoundKey(round, state_debbug, RoundKey);
  }

  // Mask are removed by the last addroundkey
  // From m6 to 0
  AddRoundKeyMasked(Nr, state, RoundKeyMasked);
```

```
//  AddRoundKeyMasked(Nr, state_debbug, RoundKey);
}

// Normal cypher function.
static void Cipher()
{
  uint8_t round = 0;

  AddRoundKey(0, state, RoundKey);

  for (round = 1;; ++round)
  {
    SubBytes(state);
    ShiftRows(state);
    if (round == Nr)
    {
      break;
    }
    MixColumns(state);
    AddRoundKey(round, state, RoundKey);
  }
  // Add round key to last round
  AddRoundKey(Nr, state, RoundKey);
}

// Masked deciphering function
static void InvCipherMasked()
{
  AddRoundKeyMasked(Nr, state, RoundKeyMasked);

  for (round = (Nr - 1);; --round)
  {
    InvShiftRows(state);

    InvSubBytesMasked(state);

    AddRoundKeyMasked(round, state, RoundKeyMasked);

    if (round == 0)
    {
      break;
    }

    InvMixColumns(state);

    remask(state, mask[6], mask[7], mask[8], mask[9], mask[4], mask[4], mask[4],
mask[4]);
  }

  remask(state, mask[0], mask[1], mask[2], mask[3], 0, 0, 0, 0);
}

// Normal decipher function
static void InvCipher()
{
  uint8_t round = 0;

  AddRoundKey(Nr, state, RoundKey);

  for (round = (Nr - 1);; --round)
  {
```

```
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(round, state, RoundKey);
    if (round == 0)
    {
      break;
    }
    InvMixColumns(state);
  }
}


/*****************************************************************************/
/* Public functions:                                                         */
/*****************************************************************************/
void AES128_ECB_indp_setkey(uint8_t* key)
{
  Key = key;
  KeyExpansion();
}

void AES128_ECB_indp_crypto(uint8_t* input)
{
  state = (state_t*)input;
//  uint8_t debbug[16];
//  BlockCopy(debbug, input);
//  state_debbug = (state_t*)debbug;

#if defined(MASKED) && (MASKED == 1)
  CipherMasked();
#else
  Cipher();
#endif
}

void AES128_ECB_indp_inv_crypto(uint8_t* input)
{
  state = (state_t*)input;

#if defined(MASKED) && (MASKED == 1)
  InvCipherMasked();
#else
  InvCipher();
#endif
}


void AES128_ECB_indp_precompute_randoms(uint8_t* seed)
{
  uint32_t seed1 = seed[0];
  uint32_t seed2 = seed[1];
  uint32_t seed3 = seed[2];
  uint32_t seed4 = seed[3];
  uint32_t mySeed = (seed1 << 24) ^ (seed2 << 16) ^ (seed3 << 8) ^  seed4;
  srand(mySeed);
  InitMaskingEncrypt();
}

void AES128_ECB_indp_precompute_inv_randoms(uint8_t* seed)
{
```

```
  uint32_t seed1 = seed[0];
  uint32_t seed2 = seed[1];
  uint32_t seed3 = seed[2];
  uint32_t seed4 = seed[3];
  uint32_t mySeed = (seed1 << 24) ^ (seed2 << 16) ^ (seed3 << 8) ^  seed4;
  srand(mySeed);
  InitMaskingDecrypt();
}
```

Glossary

AES: Advanced Encryption Standard

ATM: Automated Teller Machine

ATMEL: Advanced Technology for Memory and Logic

API: Aplication Programming Interface

ARM: Advanced RISC Machines

AVR: Alf-Egil Bogen Vegard Wollan RISC microcontroller

CMOS: Complementary Metal-Oxide Semiconductor

CPA: Correlation Power Analysis

CPU: Central Processing Unit

DES: Data Encryption Standard

DPA: Differential Power Analysis

DRM: Digital Right Management

DSS: Digital Signature Standard

DUT: Device Under Test

EM: Electro Magnetic

EMA: Electro Magnetic Analysis

EMV: Europay, Mastercard, and Visa

FI: Fault Injection

FIB: Focused Ion Beam

FPGA: Field Programmable Gate Array

GF: Galois Field

GND: Ground (Electronics)

Hd: Hamming distance

HODPA: High Order Differential Power Analysis

HOCPA: High Order Correlation Power Analysis

Hw: Hamming weight

IEC: International Electrotechnical Commission

IPsec: Internet Protocol security

ISO: International Organization for Standardization

IT: Information Technology

MCU: MicroController Unit

NIST: National Institute of Standards and Technology

OS: Operating System

PIN: Personal identification Number

RISC: Reduced Instruction Set Computer

RSA: Rivest, Shamir and Adleman (Cryptographic Algorithm)

ROM: Read Only Memory

SBOX: Substitution Box

SCA: Side Channel Analysis

SEM: Scanning Electron Microscope

SNR: Signal to Noise Ratio

SPA: Simple Power Analysis

SSD: Solid State Drive

SSL: Secure Sockets Layer

TA: Template Attacks

USB: Universal Serial Bus

WWW: World Wide Web