Algebraic side-channel attacks:
Use of SAT solvers with non-profiled attacks.
Florian Delporte
Acknowledgement

I would like to express my respectful gratitude to my thesis director Prof. Olivier Markowitch at Université Libre de Bruxelles for the useful comments, remarks and engagement through the learning process of this master thesis.

I would also like to thank my supervisor Nikita Veshchikov for introducing me to the topics as well as for his availability, encouragements and support on the way. His guidance helped me in all the time of research and writing this thesis. His office door was always open whenever I had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it.

My sincere thanks also goes to Frédéric Lafitte who provided the CryptoSAT tool used in this research and without whom I never could go so far in the findings of this study. I would also like to thank him for being the second reader of this thesis, and I am gratefully indebted to him for his very valuable comments on this thesis.

I would like to acknowledge my aunt Dominique Boland for the review she made of this master thesis and for her insightful advices. And last but not the least, I would like to thank my family and loved ones, who have supported me throughout the entire process of this work.
Abstract

In 2009, Mathieu Renauld and François-Xavier Standaert proposed new kinds of powerful attacks combining Algebraic Attacks with Side-channel Attack (SCA). As example of such attacks, they used a Bayesian template attack against Present. The results of this attack were very good since they managed to recover the correct key with only one SCA.

Based on these results, one could wonder if any SCA could be used for these attacks and if some additional steps would be required to be compatible with an Algebraic Attack. To answer this question, we will perform an Algebraic Side-channel Attack (ASCA) using a very simple yet powerful SCA: the Correlation Power Analysis.

This attack analyzes the power traces recorded from a cryptographic device in order to compute the correlation between these traces and the hypotheses. These n best hypotheses can then be inserted into the Algebraic Attack to find the correct hypothesis. The problem of a Correlation Power Analysis is that its performance degrades when the traces are too noisy or when there is not enough data. These poor performances introduce impossibilities in the Algebraic Attack by rejecting the correct hypothesis key.

This means that when the performances of the SCA degrade, the Algebraic Attack is unable to correct these errors and the performance of the constructed ASCA cannot be better than the Correlation Power Analysis.

Based on this ascertainment, we proposed several properties that a SCA needs to respect in order to be a good candidate for an ASCA:

These attacks should never introduce impossibilities in the Algebraic Attack by being very efficient even when the side-channel traces are noisy or when there are not enough power traces. Or the SCA could use an additional step to detect the impossibilities before sending its results to the Algebraic Attack.

The candidate should be able to extract information about the cryptographic device during all the encryption process and not only at a specific moment such as in a typical Correlation Power Analysis.

Finally, we stated that a good ASCA should be able to perform a tradeoff between the amount of side-channel traces and the computation time of the Algebraic Attack in order to find the correct solution every time with sometimes a long computation time when the traces are not good enough.

An additional goal of this master thesis is the presentation of an automated tool called CryptoSAT allowing to describe a cryptographic algorithm by a Satisfiability Problem (SAT) problem. This tool will be used in order to construct the SAT problem defining the Present block cipher. In this work we will show how easy it is to use this tool.
Contents

1 Introduction .................................................. 1
   1.1 Motivations .................................................. 2
   1.2 Organization ................................................. 3

I State of the art .............................................. 5

2 Cryptography ................................................... 7
   2.1 Classical ciphers ......................................... 8
      2.1.1 Substitution ............................................. 8
      2.1.2 Permutation ............................................. 8
   2.2 Modern cipher ............................................. 9
      2.2.1 Symmetric encryption .................................. 9

3 Cryptanalysis of encryption algorithms ..................... 15
   3.1 Brute-force attacks ...................................... 16
   3.2 Frequency attacks ........................................ 17
   3.3 Attacks on modern ciphers ................................. 18
      3.3.1 Attacks on the last round ............................. 19
      3.3.2 Differential cryptanalysis ......................... 19
      3.3.3 Linear cryptanalysis .................................. 22
      3.3.4 Algebraic Cryptanalysis ............................. 22
      3.3.5 Side-Channel attacks ................................. 23

4 Side-channel attacks ......................................... 25
   4.1 Targets of a side-channel attack ........................ 26
   4.2 Attacks on cryptographic devices ......................... 26
   4.3 Power analysis ............................................. 28
      4.3.1 Simple Power Analysis ............................... 28
      4.3.2 Differential Power Analysis ....................... 30
      4.3.3 Attacks based on the Correlation Coefficient .... 31
      4.3.4 Template Attack ...................................... 33
   4.4 Key Recovery .............................................. 34

5 Algebraic side-channel attacks .............................. 35
   5.1 Algorithm complexity ..................................... 36
   5.2 Offline phase: algebraic attack ......................... 38
      5.2.1 Deriving the system of equations .................. 38
      5.2.2 Conversion to a SAT problem ....................... 38
5.2.3 CryptoSAT .................................................. 39
5.3 Online phase: side-channel attacks .......................... 40
5.4 Computing the solution .................................... 40
  5.4.1 SAT Solvers .............................................. 41
  5.4.2 CryptoMiniSat .......................................... 42

II Implementation of the attack .................................. 43

6 Implementation of Present ..................................... 45

7 Offline phase: CryptoSAT ........................................ 47
  7.1 Installing CryptoSAT ....................................... 47
  7.2 Generate the SAT problem ................................. 47
    7.2.1 Target ................................................. 48
    7.2.2 Instance ............................................... 50
    7.2.3 Solution ............................................... 51
    7.2.4 Check the generated instance ......................... 52
    7.2.5 Extending the CryptoSAT package ................. 52
    7.2.6 Summary .............................................. 53

8 Online phase: Correlation Power Analysis .................. 55
  8.1 Step 1: Choosing an intermediate result of the executed algorithm .... 55
  8.2 Step 2: Measuring the Power Consumption .................. 56
  8.3 Step 3: Calculating Hypothetical Intermediate Values ............ 57
  8.4 Step 4: Mapping Intermediate Values to Power Consumption Values ... 59
  8.5 Step 5: Comparing the Hypothetical Power Consumption Values with the Power Traces ................................................. 59
    8.5.1 Acceleration of the computation ....................... 60
  8.6 Computation for all the bytes of the key .................. 60
  8.7 Assessing the results of the Correlation Power Analysis .......... 61
  8.8 Discussion of the results of the CPA ...................... 62

9 Algebraic side-channel attack on Present ................... 67

10 Conclusions .................................................. 73

III Appendix ..................................................... 83

Programs ......................................................... 85
  10.1 Implementation of Present ................................ 85
  10.2 Present Optimized Version [59] .......................... 87
  10.3 Computation of the pairs XOR Distribution table of Present ........ 93
  10.4 CryptoSAT code .......................................... 93
  10.5 Present Arduino Version .................................. 101
  10.6 Trace reader .............................................. 102
  10.7 Correlation Power Analysis attack on Present ............. 108
Chapter 1

Introduction

Since the appearance of writing humans had to transmit written messages to communicate with each other despite the distance between them. If the paper (or any other support) used to communicate contained some sensitive information then their authors wanted to keep them secret from anyone other than the recipient. To ensure that this information would remain secret they had to find a way to hide this information and one way to do it is to apply some transformation to the data.

An encryption algorithm is composed of a transformation function of the data which can’t be inverted without knowing the key. The key is a secret information which is required to retrieve the plaintext starting from the ciphertext.

As shown in Figure 1.1, the main objective of cryptography is to allow two (or more) persons to communicate together over an insecure channel in such a way that an opponent cannot understand the exchanged information. These persons are usually called Alice and Bob while the opponent is Oscar. The channel could be a telephone line or a computer network.

In this figure, Alice has a message (plaintext) and applies a transformation (encryption) on it in order to hide the original content. This process uses a secret information (key). The result of the encryption is called a ciphertext and the plaintext cannot be recovered from the ciphertext without the secret key. The decryption is the transformation converting a ciphertext into a plaintext using the key.

![Figure 1.1: Graphical representation of a secure communication over an insecure channel](image)
Historically, the goal of Cryptography was only to allow confidentiality between persons but nowadays, it has several additional functions. Modern cryptography allows confidentiality using encryption but also authenticity and non-repudiation of data using signature mechanism and integrity using hash functions. One way to find the weaknesses of a cryptographic algorithm is by trying to break its security in order to retrieve the secret key or the plaintext. Such attacks belong to the field of Cryptanalysis.

1.1 Motivations

Nowadays, smart cards are frequently used as cryptographic devices for strongly authenticate users. Moreover, the secret information needs for the authentication of this user are stored on these smart cards. So, these smart cards need to be secure. With the attack discovered in [33], these smart cards were found very weak against SCA. Indeed, the Differential Power Analysis (DPA) is very powerful against unprotected devices but to perform such attack, one needs to have access to the cryptographic device. These attacks are very powerful when based on a lot of power traces coming from the device under attack. Moreover, they are also sensitive to the noise present in the power traces. Since the correlation coefficient is the most common way to analyze the relation between data, the Correlation Power Analysis (CPA) is a very good choice to establish the relation between the power traces and the corresponding hypothesis of the key. As you will see in Section 8, these attacks are easy to implement.

There are several attacks on cryptographic algorithms and one of them is the algebraic cryptanalysis. This attack describes the cryptographic algorithm into a system of polynomial equations [3]. Once described in such a way, some flaws in the design of the cryptographic algorithm can be found by solving the system.

A combination of these two attacks is very powerful since the SCA who is unable to exploit the flaws of the cryptographic algorithm can rely on the algebraic attack to use it while the algebraic attack is strengthen by adding the information leaked through the SCA to the system of equations.

Such a combination was performed on Present [9] in [46]. In this attack, they managed to successfully recover the secret key of the cryptographic device under attack with the observation of a single power trace obtained by the SCA. This attack used a Bayesian Template method to recover the information leaked by the device.

The question at the origin of this master thesis is: is every Side-Channel Attack a good candidate to be used in an Algebraic Side-Channel Attack?

In order to answer this question, we will study the performances of an ASCA using a CPA. Beside answering this question, this master thesis will also test the utilization of an automated tool called CryptoSAT [23] to describe the Present cryptographic algorithm. This tool uses the C++ code of a cryptographic algorithm to establish the corresponding system of equations. The algebraic attack used in this work will be developed using CryptoSAT in order to determine the easiness of this tool.
1.2 Organization

This work will be divided in several parts. The first part will present the knowledge required to understand the concepts related to a ASCA while the second part will present the implementation of an ASCA on Present using a Power Correlation Analysis as side-channel source of information. This part will also discuss the results of this attack.

In the first part, we will describe first the field of cryptography and make a small introduction about the common encryption techniques. This chapter will also describe the targeted cryptographic algorithm Present.

Then we will present some attacks on the encryption techniques previously presented. The Present block cipher will be used as example of these attacks.

The next chapter will describe the SCA and how they are performed. We will then describe the CPA and how they can be performed.

We will end this part by presenting the basis of the ASCAs. This will present the Algebraic Attacks and the tool CryptoSAT used to describe Present in a system of polynomial equations. This chapter will then describe how to use the information from the SCA in order to solve the system of equations of the Algebraic Attack.

The second part will describe in detail how a CPA can be used in an ASCA. We will first present an implementation of the Present block cipher.

The next chapter will describe the basis of the CryptoSAT tool and how it can be extended to integrate our implementation of Present.

Then, we will present how a CPA can be executed on Present. The results of this attack will then be discussed to evaluate the strengths and weaknesses of this attack.

We will end this work by presenting how the results of this attack can be injected into CryptoSAT to solve the system of equations. Based on the results of the ASCA, we will try to answer the question at the starting point of this work.
Part I

State of the art
Chapter 2

Cryptography

As presented in the introduction, the main objective of cryptography is to allow two people to communicate together over an insecure channel in such a way that an opponent cannot understand the exchanged information. These people are usually called Alice and Bob while the opponent is Oscar. The channel could be a telephone line or a computer network.

This system allowing such secure communications is called a cryptosystem and is defined by the Definition 1 [55].

Definition 1. A cryptosystem is a five-tuple $(P, C, K, E, D)$, where the following conditions are satisfied:

1. $P$ is a finite set of possible plaintexts;
2. $C$ is a finite set of possible ciphertexts;
3. $K$, the keyspace, is a finite set of possible keys;
4. For each $K \in K$ there is an encryption rule $e_K \in E$ and a corresponding decryption rule $d_K \in D$. Each $e_K : P \to C$ and $d_K : C \to P$ are functions such that $d_K(e_K(x)) = x$ for every plaintext element $x \in P$.

Cryptography is defined as the “Practice of the enciphering and deciphering of messages in secret code in order to render them unintelligible to all but the intended receiver.” [19, Cryptography] “Coding (...) takes place using a key that ideally is known only by the sender and intended recipient of the message” [26].

One way to improve cryptographic algorithms is to try to break them. Cryptanalysis is “the branch of cryptography concerned with decoding encrypted messages when you’re not supposed to be able to” [26].

In the field of cryptanalysis, the classical approach is to try to break an algorithm by studying how the cryptographic algorithm works and to inverse its transformation function without knowing the secret key.

Cryptology is the science and study of cryptanalysis and cryptography.
2.1 Classical ciphers

To describe Cryptography and its uses, the best way is to follow an historical approach. At the beginning of the cryptography, the manipulated information was mainly textual [54] but with the centuries, its scope widened to transform any kind of information in order to ensure confidentiality, authenticity and non-repudiation.

2.1.1 Substitution

This operation consists in the replacement of a letter of the plaintext by another letter of the alphabet.

This can be made in two ways:

- **Monoalphabetic cipher**: This cipher simply transform the plaintext by applying a transformation on a letter at a time. For example, the Caesar’s cipher replaces each letter by another one in the alphabet. The key of this cipher is the difference between the position in the alphabet of the plaintext’s letter and the one of the ciphertext’s.

- **Polyalphabetic cipher**: This cipher divides the text into blocks of a fixed size and encrypts each block by a monoalphabetic cipher with a different key. An example of a polyalphabetic cipher is the Vigenere cipher. This cipher uses a word as cipher key and each letter of the word is used to encode the corresponding letter in the plaintext. The order of this letter in the alphabet defines the substituted letter. If the plaintext is greater than the cipher key then the key is repeated as much as needed.

2.1.2 Permutation

This operation consists in the permutation of all the letters of the plaintext. So all the letters stay the same in the ciphertext but in a different order.

These transformations on the data are very weak because they don’t transform enough the data to have a ciphertext completely independent of the plaintext then some correlation can be found by statistical means. Shannon observed [52] that any good encryption algorithm must disguise the redundancies of the plaintext. To do so, Shannon proposed to diffuse and confuse the redundancy by mixing the transformations. The confusion is “to make the relation between the simple statistics of the ciphertext and the simple description of the key a very complex and involved one”, while in the method of diffusion “the statistical structure of the plaintext which leads to its redundancy is dissipated into long range statistics in the cryptogram” [52].

These concepts have been reinterpreted and an explicit description is given by Massey [37]:

- **Confusion**: The ciphertext statistics should depend on the plaintext statistics in a manner too complicated to be exploited by the cryptanalyst.

- **Diffusion**: Each digit of the plaintext and each digit of the secret key should influence many digits of the ciphertext.

It is the task of the designer [32] to correctly mix the operations on the data to give a secure encryption algorithm with enough diffusion and confusion.
One famous example of a combination of substitutions is the Enigma machine [41] which uses rotors to set the secret key and when a key is pressed, the rotors apply a certain number of substitution on that letter. This encryption machine worked pretty well until Alan Turing and his colleagues at Bletchley park broke it by developing the “Bombe” [28] used to perform a brute-force attack on an Enigma machine.

2.2 Modern cipher

In modern cryptography, there are two main families of encryption algorithms:

**Asymmetric encryption** Each user has a pair of key. One key is public and the other one is private. These keys have different uses in an encryption scheme. The public key is used to encrypt data for a receiver and the private key is used to decrypt the data. RSA [48] is an example of an asymmetric encryption algorithm.

**Symmetric encryption** One secret key is shared between the users and everyone in possession of this key can recover the encrypted information. AES and DES are some examples of symmetric encryption algorithms [32]. Notice that DES block cipher is obsolete and should not be used anymore even if it has a big historical interest.

An important note has to be made about cryptographic algorithms. Modern cryptography respect Kerckhoffs’s assumption which means that the robustness of an algorithm can’t be ensured by hiding the used algorithm. The key must be a sufficient condition to ensure the security of the encrypted data.

The scope of this work focuses on the symmetric encryption algorithm called Present.

2.2.1 Symmetric encryption

Symmetric algorithms are divided in two classes based on their goal an the structure of the plaintext :

**Stream cipher** In this class of cipher algorithms, the goal is to encrypt a (possibly large) flow of data. When this flow is a video or audio feed these algorithms have to be quick. Generally, a stream cipher creates a pseudo-random generator used to produce a sequence of bits that will be XORed with the plaintext \( p \in \mathcal{P} \). The output of this XOR operation is the corresponding ciphertext \( c \in \mathcal{C} \) and the pseudo-random generator produces the key \( k \in \mathcal{K} \). The \( p \) can be retrieved from \( c \) by XORing it with \( k \).

**Block cipher** Block cipher encryption algorithms divide the plaintext in blocks and then apply the encryption algorithm on each block separately. These blocks can be encrypted separately with or without chaining the blocks. Chaining two blocks together means that the second block is mixed with some information from the first block. The different chaining methods are called “Modes of operation”. The mode of operation “Electronic Codebook (ECB)” does not chain the data together. Some more advanced modes exist such as Cipher Block Chaining (see [39, Section 7.2.2]). When using ECB mode, the patterns in the content of the plaintext remain after the encryption. For example if an image is encrypted, the data will be locally modified but the general shapes of the initial image will remain. Since this work focuses on the cryptanalysis of a single
Figure 2.1: CBC mode of operation [30]

block, we can assume the use of the ECB mode without encountering the problems of this property or that the attacked block is the first block of the chained blocks. Indeed, as shown in Figure 2.1, the chaining of the blocks takes place only from the second block of data:

This class of algorithms operates on “blocks of data” of a fixed size. These blocks will be transformed by a function \( f \) using the secret key such that \( f(p, k) = c \). The function \( f \) has to be defined by the design of the block cipher. A block cipher has two important parameters [32]:

1. the block size, which is denoted as \( b \)
2. the key size, which is denoted as \( K \)

As already seen, a secure block cipher must not leak any exploitable information about the secret key during the encryption process. The block size \( b \) determines the space of all the possible permutations that can transform a block [32]. The key size \( K \) determines the number of permutations actually generated without some eventual equivalent keys. Two keys are equivalent when the result of the encryption of a block with the two keys has the same result. A block cipher has to be designed in such a way that each block of the ciphertext seems to be chosen at random amongst all the possible permutations.

Generally, block ciphers are organized into several steps referred as rounds where the data are transformed by a round function using a round key. In addition, a block cipher can follow a structure determined by a scheme. The most common schemes are the Feistel Network, the Quasi-Feistel Network and the Substitution-Permutation Network.

**Feistel Network**  Some block ciphers, such as DES [43], follow a particular structure (Figure 2.2a). These ciphers are called Feistel ciphers. A \( b \)-bit Feistel cipher consists of the repetition of \( r \) rounds of an identical structure. Each block (of size \( b \)) is divided in two parts and at each round, one half of the block (\( b_1 \)) will be transformed by a round function (called Feistel function) which uses a sub-key and the result will be XORed with the other half of the block (\( b_2 \)). The result of this XOR will replace the content of \( b_2 \). At the next step, the two blocks, \( b_1 \) and \( b_2 \), will be inverted such that \( b_2 \) will be
transformed by the Feistel function \([32]\). This operation is represented in the Figure 2.2a.

The designer of a Feistel cipher has to define three elements:

1. The transformation performed by the Feistel function
2. The generation of the sub-keys used by the Feistel function at each round
3. The number of round applied to the input block

**Lai-Massey Scheme**  Another scheme used by some block ciphers (such as IDEA \([58]\)) is the Lai-Massey scheme. This scheme is an instance of the Quasi-Feistel Networks \([58]\) which is a generalization of the Feistel Network based on finite quasigroups. As for the Feistel networks, the Lai-Massey scheme uses a function \(F\) but it also defines another one \(H\) called the “Half-round function” (see Figure 2.2b). This function is executed on the two halves of the data and the result is merged by using a “combiner” \([58]\). The result is then used as input of the function \(F\) with the round key.

The result of \(F\) is then XORed with the two halves of the result of the function \(H\).

**Substitution-Permutation Network**  A simpler scheme is the Substitution-Permutation (SP) Network. It is a cipher composed of several rounds where the data are transformed using the substitution and permutation operations. An example of two rounds of a SP-Network is shown in Figure 2.2c.

**Present: An ultra-lightweight block cipher**

In this part, we introduce the block cipher studied and attacked in this document. It is a lightweight algorithm suitable for extremely constrained environment such as RFID \([9]\). This block cipher has been designed not to compete with AES or other standard block ciphers but to be competitive with the leading compact stream ciphers.
There are two advantages for a block cipher over a stream cipher [9]. First, a block cipher is a versatile primitive and by chaining the blocks together (in counter mode for example) we obtain a stream cipher. Secondly, until 2007, the design of block ciphers was more studied than the one of stream ciphers [9]. But from 2004 to 2009, the “eStream” [20] competition took place in order to construct an efficient and compact stream cipher suitable for widespread adoption. Seven stream ciphers were proposed in the final portfolio of the competition.

Present is a Substitution-Permutation network cipher [39] with 31 rounds. The scheme of a round is presented in Figure 2.3. It uses a 64-bit block and two possible lengths for the key size which can be 80 or 128 bits long. The recommended key size is 80 bits. This key size is considered a more than adequate security for low-security applications such as in RFID tag but it also matches the design goals of hardware-oriented stream ciphers in the eSTREAM project [9]. These requirements were taken as comparison with stream ciphers.

Each of the 31 rounds are divided in several steps [9]:

- **addRoundKey**: is a XOR operation between the block and the 64-bit round key. The round key is composed of the 64 leftmost bits of the key at this round (see Key Schedule.)

- **sBoxLayer**: is a non-linear substitution layer where the block is transformed by a 4-bits to 4-bits S-box. See Table 2.1 for the details of this S-box.

- **pLayer**: is a linear bitwise permutation presented in Table 2.2

**The Key Schedule.** In the 80-bit version of Present, the key is stored in a key register $K$ represented as $k_{79}k_{78}...k_0$. Each round key is produced by following these steps (Figure 2.4):

1. **Key extraction**: the round key is extracted by selecting the leftmost 64 bits of the key.

2. **Rotation**: the key is rotated by 61 positions to the left so that $k_{79}k_{78}...k_1k_0$ become $k_{18}k_{17}...k_{20}k_{19}$. 

Figure 2.3: Graphical representation of a round of the Substitution-Permutation network cipher Present [30]
Figure 2.4: Key schedule of Present for the round $i$

Table 2.1: Action of the Present S-box in hexadecimal notation

<table>
<thead>
<tr>
<th>x</th>
<th>C 5 6 B 9 A D 3 E F</th>
</tr>
</thead>
<tbody>
<tr>
<td>S[x]</td>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E F</td>
</tr>
</tbody>
</table>

3. **Substitution**: The 4 leftmost bits are transformed by the S-box presented in 2.1.

4. **XOR with the round counter**: a XOR operation is performed between the roundcounter which is the index of the round and the bits $k_{19}k_{18}k_{17}k_{16}k_{15}$ of the key. Notice that the round counter is initialized at 1.

Table 2.2: Permutation of Present. Bit $i$ is moved to bit in position $P(i)$

<table>
<thead>
<tr>
<th>i</th>
<th>P(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>P(i)</td>
</tr>
<tr>
<td>i</td>
<td>P(i)</td>
</tr>
<tr>
<td>i</td>
<td>P(i)</td>
</tr>
</tbody>
</table>

Table 2.2: Permutation of Present. Bit $i$ is moved to bit in position $P(i)$
Chapter 3
Cryptanalysis of encryption algorithms

After presenting in Section 2 some encryption algorithms, we present some attacks on these algorithms. As observed by Shannon [52], there is almost no correlation between the plaintext and the ciphertext in a good encryption algorithm. Nonetheless, if the attacker has a possibility to correctly identify that a plaintext corresponds to a ciphertext, an “unbreakable” encryption algorithm does not exist because an attacker can always find the secret key by trying all the possible keys. In standard cryptanalysis, there are some assumption made to perform an attack. Generally, the cryptanalysis follows the Kerckhoff’s assumption which says that the attacker knows the encryption algorithm used. Cryptanalysis divides the attacks in different classes depending on the type of data available [32, Section 1.2]:

- **Known ciphertext**: The attacker has only the ciphertext without knowing anything about the plaintext. Here the attacker will rely on some knowledge of the plaintext to perform the attack.

- **Known plaintext**: The attacker knows the plaintext corresponding to each of the ciphertext. This situation is the one used in the attacks describes in this document.

- **Chosen plaintext**: The attacker chooses the plaintext and can obtain the corresponding ciphertext. This attack is helpful when the attacker wants to encrypt a specific sequence to reveal more information about the key. This might appear unlikely but some situations allow the attacker to “choose” the content of the message. As example [28], during the World War II, the British RAF would drop mines at specific locations to exploit the messages sent by their opponent when discovering the mines.

- **Chosen ciphertext**: The attacker chooses the ciphertext and can decrypt it thanks to a decryption device. This attack is helpful when the attacker wants to decrypt a specific sequence to reveal more information about the key.

- **Adaptive chosen plain/ciphertext**: This is an interactive form of chosen plain/ciphertext. The attacker obtains the encryption of new chosen messages in an interactive way, perhaps after seeing an original pool of chosen plaintext, ciphertext pairs.
• **Related key attacks**: The attacker can observe the operations of an encryption algorithm with different keys without knowing the keys. For example, an attacker might know that 80 bits of the key are the same without knowing the value of the bits.

The success of a cryptanalysis will be measured according to the consumed resources. These resources are [32]:

- **Time**: The time, or work effort, is generally the first point of comparison used by the analyst to evaluate the performance of a cryptanalysis. Sometimes it is the only resources considered.

- **Memory**: The amount of memory is very important. Sometimes it is so great that it creates an insurmountable bottleneck and the attack remains impractical.

- **Number of Queries**: The number of queries can also be used to evaluate the performances of an algorithm. It is the number of executions of the algorithm with some changes in parameters such as the plaintext. As already seen, the source of information determines the class of the attack and some classes are not always achievable. Moreover, the amount of queries is also important to evaluate the performance of a cryptanalysis. Sometimes the number of queries required to perform an attack is so great that the time needed to perform these queries makes this attack impractical.

### 3.1 Brute-force attacks

These attacks can always be performed against any encryption algorithm. It searches for the correct key in the entire key space. There is no design which can prevent this attack. The designer’s aim is to ensure that there is no better option for the attacker [32]. So, the security of the algorithm is determined by the size of the key space. Nevertheless, the protection offered by a key of a fixed size will decrease because of the improvement of the computer hardware. The designer can try to take this improvement into account by using the Moore’s Law [40]. One interpretation of the Moore’s Law is that the amount of computational power at a fixed cost will double roughly every 18 months.

The exhaustive search can be easily parallelized, so, by adding extra resources at the problem, it can be solved quicker. As example, in 1999, DES key search was performed simultaneously across the Internet in order to test around $92 \times 10^9$ DES keys each second [32].

In the Table 3.1, are presented the resistance against a brute-force attack of an encryption algorithm depending on the size of the key. As we can see, the computation time of these attacks increase exponentially with the key size. For this reason, this attack can not be effectively applied to a modern encryption algorithm if it has a large enough key size.
Table 3.1: A broad headline assessment of the security offered by different key length against a brute-force attack. [32]

<table>
<thead>
<tr>
<th>K (bits)</th>
<th>search time (operations)</th>
<th>Status (2010)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>$2^{40}$</td>
<td>easy to break no security</td>
</tr>
<tr>
<td>64</td>
<td>$2^{64}$</td>
<td>practical to break poor security</td>
</tr>
<tr>
<td>80</td>
<td>$2^{80}$</td>
<td>not currently feasible reasonable security</td>
</tr>
<tr>
<td>128</td>
<td>$2^{128}$</td>
<td>very strong excellent security</td>
</tr>
<tr>
<td>256</td>
<td>$2^{256}$</td>
<td>exceptionally strong astronomical levels of security</td>
</tr>
</tbody>
</table>

3.2 Frequency attacks

The first class of algorithms presented in Section 2 was the one of the classical ciphers. To break these algorithms, an attacker can study the frequency of each letter in the ciphertext and on the basis of these frequencies, it is possible to retrieve enough information to reveal the key. This attack is possible because of the non-uniform distribution of the frequencies in the different languages. As we can see in Figure 3.1 the distribution of the frequency of each letter in the English language does not follow a uniform distribution which means that if a cipher algorithm does not break the properties of the language, the output can be analyzed to retrieve the plaintext and even the key used.

![Figure 3.1: An estimation of the distribution of letters in English language text [5].](image)

On the basis of the probabilities in Figure 3.1, the alphabet can be partitioned in five groups [5]:

1. E: having the highest probability (about 0.120)
2. T, A, O, I, N, S, H, R: having a probability between 0.06 and 0.09
3. $D, L$: having a probability around 0.04

4. $C, U, M, W, F, G, Y, P, B$: having a probability between 0.015 and 0.028

5. $V, K, J, X, Q, Z$: having a probability less than 0.01

These attacks are very easy to implement. Indeed, in Python, a frequency attack of a monoalphabetic cipher defined in the function `encrypt` is shown in the function `perform_frequency_attack`:

```python
def encrypt(plaintext, key):
ciphertext = 
for char in plaintext:
    char_index = string.ascii_lowercase.index(char)
    char_index = (char_index + key) % 26
    ciphertext += chr((ord('a') + char_index))
return ciphertext

def perform_frequency_attack(ciphertext):
for letter in ciphertext:
frequencies[string.ascii_lowercase.index(letter)] += 1
key = frequencies.index(max(frequencies)) - string.ascii_lowercase.index('e')
return key
```

It is also useful to consider sequences of two or three consecutive letters (digrams and trigrams). Based on these probabilities, an attacker can rank the possible keys by decreasing probabilities.

The frequency attacks are very powerful when used on a long ciphertext thanks to the frequency of the letters in the ciphertext which will converge to the distributions of the frequencies of the English language.

The classical ciphers are very weak against these attacks and can be easily broken. The frequency attacks are very powerful when used on a long ciphertext thanks to the frequency of the letters in the ciphertext which will converge to the distributions of the frequencies of the language.

These attack are efficient only when each letter is encrypted individually. This attack becomes impracticable when the data are encrypted by block such as it is the case with the block ciphers.

### 3.3 Attacks on modern ciphers

These attacks on the design of the encryption algorithm tries to find a weakness in the design of the cryptographic algorithm by studying its mathematical properties. For block ciphers, these attacks can be separated in three categories [12] which are the attacks on the last round, the differential cryptanalysis and the linear cryptanalysis.
\[ c = F_2( F_1(p, k_1), k_2) \]

Figure 3.2: 2 steps of an encryption process

\[ p = F_1^{-1}( F_2^{-1}(c, k_2), k_1) \]

Figure 3.3: 2 steps of a decryption process

3.3.1 Attacks on the last round

As presented in Section 2.2.1, block ciphers are divided in several rounds combining some linear and non-linear transformations. At each round, the block of data is transformed by a function \( F \) which takes the round key as its argument. The block cipher defines the number of rounds \( r \) performed.

In this category of attacks, the attacker wants to retrieve the \( r \)-th round key. This can be done in a known plaintext attack when the attacker has a reduced cipher detector allowing to distinguish the output of the function \( F \) at round \( r \) from a value after a random permutation. By using this detector, the attacker can find the correct round key by computing \( F^{-1} \) which is the inverse of \( F \). As shown in Equation 3.2 As already explained in Chapter 2, an encryption process successively applies a round function on the plaintext \( p \) using round keys \( k_i \) to compute the ciphertext \( c \) (see Figure 3.2.) The decryption reverse the process using the inverse of the round function \( F^{-1} \) (see Figure 3.3.)

The difficulty of this attack is to find an efficient reduced cipher detector which is fast and requires the minimum of plain/ciphertext pairs.

3.3.2 Differential cryptanalysis

This attack originally presented on a reduced variant of DES by Eli Biham and Adi Shamir [7] studies how the differences in the input of the encryption algorithm of a round influences its output. This attack is a chosen plaintext attack. As example, they attacked DES which is a block cipher using the Feistel structure.

The differences between two words are most often defined by the XOR of these two words.

In an encryption algorithm, the differences (and their statistical properties) between two plaintexts and two ciphertexts are functions of the complexity of the encryption algorithm. In our example with Present, the statistical properties of the differences will depend on the S-box used. When Eli Biham and Adi Shamir studied this attack, they found out that the differences between the input and the output of an S-box are not uniformly distributed. The frequency of the differences for the S-box are shown in the Table 3.2. In this Figure, the first column indicates the differences between two inputs of the S-box and the columns are the differences between the two outputs corresponding to these inputs. A cell indicates the number of observations where the inputs generate a given difference between the outputs.

In the Table 3.2, we can see that some input differences have a higher probability to generate some specific differences in the output and some of them have a null prob-
ability. This observation is the starting point of the differential cryptanalysis. Thanks to this, some specific inputs can be encrypted in such a way that their differences will be propagated until the last round with a high probability. If the resulting difference is not the one expected, the attacker knows that the most probable key is not the good one so he can try the second most probable one.

<table>
<thead>
<tr>
<th>Input XOR</th>
<th>Output XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>16</td>
</tr>
<tr>
<td>0x1</td>
<td>0</td>
</tr>
<tr>
<td>0x2</td>
<td>0</td>
</tr>
<tr>
<td>0x3</td>
<td>0</td>
</tr>
<tr>
<td>0x4</td>
<td>0</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
</tr>
<tr>
<td>0x6</td>
<td>0</td>
</tr>
<tr>
<td>0x7</td>
<td>0</td>
</tr>
<tr>
<td>0x8</td>
<td>0</td>
</tr>
<tr>
<td>0x9</td>
<td>0</td>
</tr>
<tr>
<td>0xA</td>
<td>0</td>
</tr>
<tr>
<td>0xB</td>
<td>0</td>
</tr>
<tr>
<td>0xC</td>
<td>0</td>
</tr>
<tr>
<td>0xD</td>
<td>0</td>
</tr>
<tr>
<td>0xE</td>
<td>0</td>
</tr>
<tr>
<td>0xF</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.2: Pairs XOR distribution table of the Present S-box.

Example with Present

This section shows how a particular difference between two inputs can be propagated through several rounds of Present. To find such a propagation, the attacker can find an \(n\)-round iterative differential of Present for some \(n\). For Present, there is a 4-round iterative differential \([57]\) when the initial difference between the inputs is 0x0000 0000 0000 4004. With this value, we can see in the Figure 3.4 that the difference is computed by the S-box and the result is 0x0000 0000 0000 5005 with a probability of 1/4 which will be transformed into 0x0000 0009 0000 0009 by the permutation.

![Figure 3.4: Propagation of the input difference 0x0000 0000 0000 4004 through one round of Present.](image)

This observation shows that the differences propagated through a round of Present are independent of the key.

The Figure 3.5 shows the 4-round iterative differential of Present. After the propagation of the differences shown in Figure 3.4, the next round will transform the differences into 0x0000 0004 0000 0004 also with a probability of 1/4. The output differences of the
second round will then be 0x0000 0101 0000 0000.
For the third round, the output differences of the S-box will be 0x0000 0909 0000 0000 (with probability 1/4) and the permutation will transform it into 0x0500 0000 0000 0500.
The last round of the iterative differential of Present will transform the output differences into 0x0100 0000 0000 0100 with a probability of 1/8. This will be transformed into the initial 0x0000 0000 0000 4004 by the permutation.

![Diagram](image.png)

Figure 3.5: Propagation of the input difference through 4 round of the Present encryption algorithm [12]

By using this input difference, it may remain until the end of the four rounds with a certain probability:

$$\frac{1}{4} \times 1 \times \frac{1}{4} \times 1 \times \frac{1}{4} \times 1 \times \frac{1}{8} \times 1 = 0.00195$$

where $\frac{1}{4}$ and $\frac{1}{8}$ are the probabilities of having these differences at the output of the S-box and 1 is the probability for the output of the permutation.

With this construction, the attacker has a structure which will allow him to attack the Present encryption algorithm by identifying the pairs (plaintext, ciphertext) encrypted using Present. Indeed, for a reduced Present of 4 rounds, if we take 1 million pairs (plaintext, ciphertext) with an initial difference of 0x0000 0000 0000 4004, then the number of output differences equals to 0x0000 0000 0000 4004 after 4 rounds should be approximately 1950. If this is not true, the pairs are probably obtained through a random permutation.
By using this method [57], Present can be attacked in order to gather some information about the key. Meiqin Wand originally claimed that he was able to recover 32 subkey bits but this affirmation was reduced to 30 bits by Manoj Kumar and Pratibha Yadav and Meena Kumari [35].

### 3.3.3 Linear Cryptanalysis

This known plaintext attack, proposed by Matsui in 1993, is achievable when a block cipher has the following weakness: there are two sets of positions, one for the input ($U$) and one for the output ($V$), such that the XOR of bits at these positions are the same for most of the secret keys [12]. This property reveals a high probability occurrence of linear expressions between the plaintext, ciphertext and the key. In other words, an attacker wants to find an expression of the form described in the Equation 3.1 which has a high or low probability of occurrence [27]:

$$X_{i_1} \oplus X_{i_2} \oplus \ldots \oplus X_{i_u} \oplus Y_{j_1} \oplus Y_{j_2} \oplus \ldots \oplus Y_{j_v} = 0 \quad (3.1)$$

where $X_i$ represents the $i^{th}$ bit of the input and $Y_j$ the $j^{th}$ bit of the output and $\oplus$ is a bitwise XOR operation.

If a cipher satisfies this expression with a high probability, it has a poor randomization ability [27]. For a good cipher, if we randomly select the sets $U$ and $V$, the probability of satisfying this equation approaches $1/2$. A linear cryptanalysis exploits the deviation of this probability. To have more information about linear cryptanalysis, see [38, 27].

A linear cryptanalysis of Present was performed by Cho Joo Yeon [14]. During this attack, they managed to recover the 80 bits of the key from a reduced-round Present with 25 rounds out of 31. These good results are mainly based on the observation [14] that Present has a lot of linear approximation with roughly the same correlations. This weakness is caused by the lack of diffusion in the pLayer of Present. This permutation is performed using a lookup table which is very efficient when implemented in hardware but it potentially has the weakness that its input and output bits have a one-to-one correspondence.

### 3.3.4 Algebraic Cryptanalysis

Some other attacks exploit the flaws in the design of a cryptographic algorithm by converting the cipher and optionally some supplemental information such as file formats into a system of polynomial equations [3]. Solving such system of equations allows an attacker to retrieve the secret key but this can be hard because of the huge number of equations and variables. The attacker will then try to reduce a maximum of equations by applying some algebraic transformations on the system.

These attack are very powerful to detect any flaw in the design of the cryptographic algorithm so these attacks could be used during the design as a tool to detect its weaknesses.
3.3.5 Side-Channel Attacks

Whilst the previous attacks on modern ciphering attack the design of a cryptographic algorithm, the SCA exploits any information unforeseen during its design. To do so, the attacker will target a physical implementation of a cryptographic algorithm and he will try to gather any information leaked by this device. These attacks require for an attacker to have a physical access to the device to gather the side-channel leakages.

In this work, we will show how these two cryptanalysis techniques can be combined together in order to use the advantages of these two attacks.
Chapter 4

Side-channel attacks

During World War II, a new field of cryptanalysis was found at Bell Labs which were specialized in the conception of the Bell-telephone mixing device (a cryptographic device used by the US army.) One day, they discovered by accident that each time a letter was encrypted by the Bell 131-B2 a spike appeared on an oscilloscope somewhere else in the lab. After this discovery, they studied the phenomenon and during a demonstration to the authorities they managed to recover 75% of the encrypted data transmitted during 3 hours. And that demonstration was made from a building across the street without having access to the encryption devices. [1]

This attack on a cryptographic device was the first known SCA ever made. A SCA is an attack which exploits the physical characteristics of a cryptographic device or any other source of information unforeseen during the design of an cryptographic algorithm. Each cryptographic algorithm has to be implemented on a device to encrypt and decrypt some data. This implementation on a physical device is the target of the SCA.

SCA focus on the physical characteristics of a cryptographic device in order to gather some information about the key. A SCA targets an implementation of an encryption algorithm while the attacks presented in the Chapter 3 target all the implementations of this algorithm. The goal of these attacks is to find the secret key while the attacker has access to the targeted encryption device.

The main advantage of a SCA is that the attack is much easier to perform than the “classical” cryptanalysis because it does not require a perfect knowledge of the cryptographic algorithm and does not require a lot of computing power to be executed.

In a SCA, there are two main steps (which can be divided in smaller steps): an interactive phase where an attacker gathers all information leaked through the device’s physical properties and an exploitation phase where the data are processed to recover the secret in the gathered information. The SCA are of divide-and-conquer nature which means that the key will be divided into chunks (generally into bytes) which can be more efficiently attacked.
4.1 Targets of a side-channel attack

The target of a side channel attack is a cryptographic device which embeds a secret key and can use it to perform cryptographic operations. A cryptographic key is a very sensitive information which ensures the secrecy of the data so it must be protected very carefully. Because of that, the keys are generally stored on an off-line device. This device is programmed to execute only one program which is the encryption/decryption and therefore, it is not sensitive to viruses or flaws in the computer’s operating system.

The typical cryptographic device in use nowadays is a smart card. A smart card contains a chip with no battery so the chip can be used only when a power source is connected to it. This characteristic makes these devices very easy to attack because the supply voltage and the clock have to be provided (and so it can be easily measured) by the card reader.

![Smart card chip and its connection points: supply voltage (V_{DD}), reset signal (RST), clock (CLK), ground connection (GND), Input/Output (I/O), and an external voltage for programming (V_{PP})](image)

In a smart card, there are two main components used in SCA:

- Power supply
- Clock

Smart cards are simple devices and so those two elements have to be provided by an external source to work properly. So the total power consumption of the algorithm (and the total computation time) can be calculated with ease.

When an attacker wants to attack a smart card, all he needs is a smart card reader, a computer to analyze the results and an oscilloscope connected to the smart card through a probe which will measure the voltage drop across a resistance inserted on the GND line of the power supply of the device.

4.2 Attacks on cryptographic devices

An attacker, while performing a SCA, can take two approaches [36, Section 1.2]. Either he chooses to perform a simple observation of the devices characteristics. This is called a “Passive attack”.

Or he acts on the device by inducing an error into it to exploit the information revealed when an error occurs. This is called an “Active attack”.

The information revealed while doing a SCA can come from different sources [36, Section 1.2]. The data source of a SCA can be obtained by inducing a fault to the device...
to produce an error in the workflow of the cryptographic device. When a fault occurs on
the device, it may skip some instructions which can be very interesting for the iterative
encryption algorithm since it may reveal the intermediate steps of the algorithm. The
fault can be caused by manipulating the clock or the power supply of the device. This
attack is called a “Fault inducing attack”.

A SCA is an attack using any information from a source which were not take into
account during the design of this algorithm. It exists a lot of different sources of infor-
mation, hereunder are some examples.

Some attacks can retrieve information about the device by analyzing the photons
emitted by the device during the encryption or decryption of the plaintext (“Photon
emission analysis” [21]) or by analyzing the electromagnetic field around the device dur-
ding its execution (“Electromagnetic analysis attack” [2].)

An attacker can also exploit the computation time of an algorithm to reveal some
information about the key (“Timing attack”.) Indeed, in some algorithms, an instruction
could be avoided depending on the value of a certain bit of the key.
For the RSA encryption algorithm, a specific computation technique uses a conditional
instruction [34]. It means that if a bit of the key is set to one, an extra calculation will
be made. This specific implementation allows an attacker to have information from the
computation time. To reveal the exact value of a bit when the previous ones are known,
the attacker needs to execute the loop once again to find the value of the bit. An attack
on the entire key can be performed by starting with an initial value of 0.
The main information that an attacker can get from this construction is the Hamming
weight of the key. The Hamming weight of a block is the amount of bits at 1 in this
block. It can be retrieved by determining how long takes a computation of an iteration
of the loop when the value is 1 and when the value is 0. This can be computed by
executing the algorithm on a key with all 0 or 1 and dividing the total time by the
number of steps of the loop. By using this method an attacker can execute a brute-force
on the encryption algorithm in only

\[
\omega \left( \frac{\omega}{HW(key)} \right)
\]

instead of

\[2^\omega\]

where \(\omega\) is the size of the key and \(HW(key)\) represents the Hamming weight of the
key. For example, with a 20-bits key, there are 1048576 possible keys. If the key has a
Hamming weight of 12, the brute-force attacks performed after the timing attack has to
test only 125970 values which is 8 times smaller.

An attacker can also choose to analyze the power consumption of a cryptographic
device while performing an encryption or decryption (“Power analysis attack”.) In a
similar way than the timing attack, the power analysis attack uses the power consump-
tion to recover the operations and the operands performed during the algorithm.

To gather the information from these sources, an attacker can perform three kinds
of attacks based on the manipulation of the device under attack.
In “Invasive attacks”, the attacker breaks into the internal components of the chip to access some specific point in the system. This is called depackaging a chip. It can be very expensive to do due to the protections against it. One example of a protection is a sensor which reacts to light and erases the key.

Another class of attacks depackages the device but no contact to the buses is made so the dissipation layers stay untouched. Generally the chip is depackaged to induce a fault thanks to X-rays or a laser. These attacks are called “Semi-invasive attacks.”

The last category of attacks is the “Non-invasive attacks” where the attacker only uses the external information provided by the cryptographic device.

This work will focus on the passive and non-invasive power analysis attacks. Indeed, the passive attacks can be used without leaving any traces on the device and at the smallest possible cost and these attacks do not have to act on the device to gather the data so they are easier to perform. Moreover, the non-invasive attacks are very useful because the attacker may want to keep the device untouched to remain undetected.

The power analysis attacks are interesting because they don’t require to depackgage the cryptographic device and they are not very expansive. The oscilloscope is the only costly device. These two characteristics make these attacks very interesting to any attacker.

4.3 Power analysis

This attack measures the power consumption of a cryptographic device and uses it to gather some information on the execution of the algorithm. The recorder power consumption of a device under attack is called “power trace”.

Measuring the power consumption of a smart card is an easy task thanks to the fact that the power source has to be provided from the smart card reader. So an attacker can connect the card reader to an oscilloscope and add a small resistor in series to the source of energy of the smart card like showed in Figure 4.2.

In power analysis, it is very important to remove the noise which can be produced by the measuring instruments and some of the physical protections of the cryptographic device. To do so, an attacker has to perform several computations of the same operation of the cryptographic device and take the mean of the power consumption. It can be automatized on some oscilloscopes.

Based on the obtained and cleaned power traces an attacker can perform two kinds of attacks.

4.3.1 Simple Power Analysis

This attack studies the power traces and tries to find some patterns corresponding to the instructions and based on this information an attacker can retrieve some information about the key.

This attack was discovered for an unprotected implementation of RSA [48, 49] which used a condition depending on the key or part of the key.* As example if we look at a power trace of a conditional algorithm, we can obtain a power trace like the one showed in Figure 4.3. In this figure, there are clearly two different patterns that repeat themselves. As we can see in this graph and in the algorithm, some operation is a conditional
instruction which is performed only when the value of the key bit is one. With this information an attacker is able to find the key only by looking at the power traces of the algorithm.

Figure 4.3: Power trace of an unprotected implementation of RSA [49]

When the algorithm is modified to compute at each iteration all the instructions even when it is not necessary, the Simple Power Analysis (SPA) is not effective enough to attack it.
4.3.2 Differential Power Analysis

The DPA is a more popular attack than the SPA because it does not require to have a perfect comprehension of the execution of the algorithm and the algorithms can be attacked even when the power traces are extremely noisy. Unlike the SPA, the DPA requires a lot of power traces so for these attacks it is generally necessary to possess the cryptographic device for a certain amount of time.

The main difference between SPA and DPA lies in the fact that the SPA analyses the power consumption along the time axis and the attacker tries to find some patterns in the data. But the DPA analyses the power consumption of the different executions at a fixed point. In other words, the DPA exploits the dependencies between the data of the algorithm and its power consumption.

In DPA, there is a general method to attack a cryptographic device. This attack is composed of five steps [36]:

1. **Choosing an intermediate result**: The attacker has to select a result obtained when the computation of the algorithm is not completed. This result has to be a function of a subset of the key and a variable (which is in most scenarios the plaintext or the ciphertext.)

2. **Measuring the power consumption**: The attacker will use the cryptographic device to compute some input data and record the corresponding power consumption. The attacker has now information about the power consumption on a part of the algorithm depending on the input data. The attacker writes the known data values in a vector \(d = (d_1, d_2, \ldots, d_D)\) where \(d_i\) is the data value in the \(i^{th}\) encryption run and \(D\) is the number of different pairs of data blocks available to the attacker. The power trace corresponding to the data block \(d_i\) is denoted as \(t_i = (t_{i,1}, t_{i,2}, \ldots, t_{i,T})\) where \(T\) is the power trace’s length. The power traces are stored in a matrix \(T\) of size \(D \times T\).

   The power traces need to be aligned for a DPA attack such as two power traces at time \(t_i\) correspond to the same operation. This can be done using a trigger signal for the oscilloscope or using some alignment techniques [36].

3. **Calculating hypothetical intermediate values**: For each execution of the algorithm on the input data, the attacker will compute the intermediate results for each hypothetical keys. The results are stored in a matrix \(V\) of size \(D \times K\) where \(K\) is the size of the key hypotheses. This computation is represented by:

   \[v_{i,j} = f(d_i, k_j)\] where \(i = 1, \ldots, D\) and \(j = 1, \ldots, K\)

   Each column contains the intermediate results for the execution of the algorithm with the corresponding hypothesis key \(k_j\). The goal of a DPA is to find the column of \(V\) which has been processed during the \(D\) encryptions. When this goal is met, the correct key \(k_{ck}\) is known.

4. **Mapping intermediate values to power consumption values**: The attacker will build a power model corresponding to the attacked device. On the basis of that power model, the attacker will compute the hypothetical power consumption values. The result of this step (stored in a matrix \(H\) of size \(D \times K\)) is determined by the knowledge of the attacker about the attacked device. An efficient DPA
attack will need a power model that matches precisely the power consumption of the device. The most commonly used power models are the Hamming-distance and Hamming-weight model [36].

5. Comparing the hypothetical power consumption values with the power traces: The final step is a statistical analysis of the hypothetical power consumption to find the power traces of the hypothesis key which match the measured power traces. This step tries to predict the key used by computing the distance between the recorded power traces and the simulated ones. This prediction has to take into account a certain “noise” in the data. This “noise” can be produced during the acquisition process or by a small misconception of the model.

The main advantage of the DPA is that it does not require a good knowledge of the encryption algorithm but it requires a very good and accurate comprehension of the encryption device to build a correct power model. Another way to build a power model is to make some assumptions based on a lot of power traces. The DPA is very powerful if the power model is accurate thanks to the statistical analysis which allows the attacker to remove the noise in the data.

4.3.3 Attacks based on the Correlation Coefficient

The correlation coefficient is the most common way to determine linear relationship between two random variables \( X, Y \) [36]. Therefore, in this work, it will be used to determine the linear relationship between the correct key and the power traces. Usually, the electric noise does not change a lot between two points of a power trace when device under attack performs the same operation. Hence, there is a relation between the noise at one point and the one present in neighboring points [36]. This relation can be seen in the scatter plot in Figure 4.5. In these plots, we can see that the dots are placed on a diagonal indicating a positive correlation. By contrast, there is no correlation when the points of the power trace are not neighboring.

In statistics, the correlation and the covariance express the linear relationship between \( X \) and \( Y \). The most commonly used is the correlation coefficient \( \rho(X, Y) \) [36]. The correlation coefficient is based on the covariance (see the Definition 2). It is a dimensionless quantity bounded between plus and minus one [36]. The correlation is typically unknown but it can be estimated by \( r \) as shown in Definition 3.

**Definition 2.** \( \rho(X, Y) = \frac{Cov(X, Y)}{\sqrt{Var(X) \times Var(Y)}} \)

**Definition 3.** \( r = \frac{\sum_{i=1}^{n}(x_i - \bar{x}) \times (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \times \sum_{i=1}^{n}(y_i - \bar{y})^2}} \)

In Definition 2 and 3, \( x_i, y_i \) are two sets of random variables and \( \bar{x}, \bar{y} \) are the mean values of these sets.
Figure 4.4: Step 3 to 5 of the DPA attack. [36]
In DPA, the correlation coefficient is used to determine the linear relationship between the column $h_i$ and $t_j$ where $h_i$ is the column containing the hypothetical power consumption and $t_j$ is the recorded power traces. The estimations $r_{i,j}$ of the correlation coefficients are stored in a matrix $R$. The computation of this estimation is shown in Figure 4.1.

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \bar{h}_i) \times (t_{d,j} - \bar{t}_j)}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h}_i)^2 \times \sum_{d=1}^{D}(t_{d,j} - \bar{t}_j)^2}}$$ (4.1)

### 4.3.4 Template Attack

This attack, referred as the “strongest form of side-channel attack possible in an information theoretic sense” [13], can break implementations of a cryptographic algorithm using only one or a limited number of side-channel samples.

A template attack requires that the attacker has access to an identical device than the one under attack which can be programmed as needed for this attack. In the previously presented attacks, the noise in the input data have to be reduced but for the Template Attacks (TA), the attacker will model such noise and use it to fully extract information present in the samples. The modeling of the noise is done using a precise multivariate characterization [13]. This model is called a template or a profile.

The template is computed using Signal Detection and Estimation Theory and uses likelihood ratios for hypothesis testing. Essentially, the device under attack can perform up to $K$ possible operation sequences ($O_1, ... O_K$) and the attacker wants to identify which operation is being executed. In signal processing, the noise is defined [13] by the combination of the intrinsic and ambient noise. The goal of the TA is to distinguish theses noises. Indeed, for an operation $O_i$, the intrinsic noise is constant. The opera-
tion can be found using a maximum likelihood technique which allows to identify the operation which maximize the probability to observe a certain noise. The attacker will use the experimental device to identify a small section of the sample which depends only on a few bits. Then a template for each possible value of the hypothesis is build through experimentation. The template is composed of the mean signal and noise probability distribution. This templates can then be used in order to select some hypothesis for the few bits attacked. This process is repeated with a larger amount of bits of the key.

TA uses an extend-and-prune iterative strategy and the success of these attacks depends on the effectiveness of the pruning strategy to reduce the combinatorial explosion caused by the extension process.

4.4 Key Recovery

As we saw, most SCA are of divide-and-conquer nature which means that their results are a set of chunks of ranked hypotheses for the secret key. The difficulty is then to find the correct combination of hypotheses of these chunks.

In 2013, N. Veyrat-Charvillon, B. Gérard, M. Renaud, and F-X. Standaert proposed an optimal key enumeration algorithm [42] using the ranked key chunks and their probabilities. This key enumeration returns the full combined keys in the optimal order from the most probable ones to the less ones. Unfortunately, this algorithm uses plenty of memory which limits its practical use especially for noisy side-channel traces.

In 2015, A. Bogdanov, I. Kizhvatov, K. Manzoor, E. Tischhauser and M. Witteman proposed a suboptimal but high-performance Score-based Key Enumeration Algorithm (SKEA) [10] for the conquer stage of SCA. SKEA uses scores assigned to each key part candidate based on its likelihood to be the correct key part. This likelihood is defined by the information leaked by SCA, for example, it can be derived from a correlation value. Based on these scores, SKEA computes the cumulated score of a complete hypothesis key by summing the scores of each key part composing this hypothesis.

Finally, SKEA enumerates the keys ordered by decreasing cumulated score in order to enumerate in priority good key candidates.
Chapter 5

Algebraic side-channel attacks

After studying in more detail the algebraic attacks and the SCA, it is possible to build an hybrid attack by using the advantages of both attacks. One example of a powerful combination of the two attacks is the attack that follows the Hamming weight leakage model [44]. A classical key recovering attack will compute the value of some of the bits of the key and operate a brute-force attack on the rest of the key. With the Hamming weight leakage model, we saw that the brute-force will try at most

\[
\omega \left( \text{HW}(\text{key}) \right)
\]

instead of

\[2^\omega\]

which can be a great improvement.

If we try to apply this attack on reduced round Present, we can reduce the brute-force search. Knowing that the best differential cryptanalysis on DES achieved to recover 30 of 80 bits of the key, the brute-force has to be executed at most on

\[2^{50}\]

possible keys. With a SCA, we are able to recover the Hamming weight of the key. For example, with a 80-bits key with an Hamming weight of 32, if the Present differential cryptanalysis has recovered a subset of the key containing 20 '0' and 10 '1' then there are

\[32 - 10 = 22\]

positions of bits set to one to find. So the brute-force has to be performed on:

\[
\binom{50}{22} = 8,875 \times 10^{13} \approx 2^{46}
\]

This value is much easier to compute than the original \(2^{80}\) from the beginning. Nevertheless, a better SCA leaks more information than only the Hamming weight of the key so better attack can be built.

In 2002, algebraic attacks using overdefined systems of equations have been proposed as a new very powerful cryptanalysis technique [17]. These attacks are based on the hypothesis that the targeted ciphers can be described by an overdefined system of algebraic equations.
Using such attack, they showed [17] that 256-bits Serpent (The second best algorithm in the AES competition) can be (marginally) broken by this attack. But for Rijndael (the winner of the competition), this attack has worse results than exhaustive search. Based on these results, they suggested a new criterion for the design of the s-boxes: “they should not be describable by a system of polynomial equations that is too small or too overdefined.” [17] This attack was performed on a 5-rounds Present in [31] and they managed to recover half of the key within 3 minutes on a desktop PC.

This new kind of attacks can be improved when combined with side-channel attacks. This attack called “Algebraic side-channel attack” has several interesting features [46]:

1. They exploit the information leakages of all the rounds.
2. They can retrieve the key after a single encryption.
3. They can succeed in a unknown plaintext/ciphertext attack.
4. They can defeat countermeasures as boolean masking.

In 2009, M. Renauld and F-X. Standaert proposed a way to associate the information leaked through the side-channel with an algebraic attack by expressing this information on the key in a set of binary variables and adding it to the system created during the algebraic attack [44, 47].

These attacks are composed of two parts (1) the algebraic attack where the targeted cipher is described into a system of polynomial equations involving the key bits as variables and (2) an online phase where the system is attacked through a SCA to gather some addition information added into the system of equations.

In these attacks, we transform the system of equations containing the description of the cryptographic algorithm and the results of the SCA into a SAT problem. SAT is fundamental to solve many problems in automated reasoning so it has been widely studied and many optimization methods, parallel algorithms and practical techniques have been developed to solve it [25]. Hereunder is a small introduction about the complexity of algorithms and the SAT problem.

5.1 Algorithm complexity

The worst-case complexity of an algorithm defines its worst-case running time. Some problems can be solved by a polynomial-time algorithm (the worst-case running time is \(O(n^k)\) for some constant \(k\)) but this is not true for all problems. Indeed, there are some problems such as Turing’s “Halting Problem” which are unsolvable no matter how much time is allowed [56].

Generally, the solvable problems are categorized into several classes [16]:

**Polynomial-time problems** The problems that can be solved by a polynomial-time algorithm are tractable or “easy” and belong to the class called \(P\). These problems can be decided in polynomial time.
Superpolynomial-time problems  By contrast, the ones solvable by a superpolynomial-time algorithm are intractable or hard and belong to the class NP. These problems can be verified in polynomial time.

NP-Complete  Lastly, a problem belongs to the NP-Complete class if it is in NP and is as “hard” as any problem in NP. A problem A is as “hard” as another problem B if there is a polynomial time algorithm which transforms A into B [15]. By using a reduction algorithm, we can build a polynomial-time algorithm (described in Figure 5.1) to decide A if there is a polynomial-time algorithm that decides B. Such a reduction is divided into three steps [16]:

1. The first step is to use a polynomial-time reduction algorithm to transform an instance α of problem A into an instance β of problem B.

2. Then to compute the result of the decision problem B by executing the polynomial-time decision algorithm on β.

3. And finally use the answer for β as the answer for α

![Figure 5.1: Graphical representation of a polynomial-time reduction algorithm used to solve a decision problem A in polynomial time using a polynomial-time decision algorithm for the problem B [16].](image)

The status of NP-Complete problems is unknown. So far, there is no polynomial-time algorithm to resolve such a problem and the question $P = NP$ is one of the foremost open question in computer sciences since it was first posed in 1971 by Stephen A. Cook [15].

In 1971, Stephen A. Cook proved that the SAT problem belongs to the NP-Complete class [15]. Since then, a lot of problems are proven NP-Complete using a polynomial-time reduction to a SAT Problem [24].

In the SAT or “Propositional Satisfiability” problem the goal is to find an assignment for some variables which satisfies a propositional formula composed of the element of the language SAT [16]:

- $n$ boolean variables $x_1, x_2, \ldots, x_n$
- $m$ boolean connectives. These are some boolean functions mapping one or two inputs to one output. These functions are $\land$ (AND), $\lor$ (OR), $\neg$ (NOT), $\rightarrow$ (implication) and $\leftrightarrow$ (if and only if).
- boolean constants $\in \{0, 1\}$.
- parentheses.
5.2 Offline phase: algebraic attack

5.2.1 Deriving the system of equations

The first step of the algebraic attack is to describe the targeted cipher into a system of equations. A trivial definition would be a system of the form:

\[
C_1 = f_1(P_1, \ldots, P_{64}, K_1, \ldots, K_{80}) \\
C_2 = f_2(P_1, \ldots, P_{64}, K_1, \ldots, K_{80}) \\
C_{64} = f_{64}(P_1, \ldots, P_{64}, K_1, \ldots, K_{80})
\]

where \(P_i, C_i\) and \(K_i\) are respectively the i-th bit of the plaintext, ciphertext and the key.

However, this representation is useless in this unreduced form due to the nature of the functions \(f_1, \ldots, f_{64}\) which are built to involve every bit of the plaintext and the key. This makes the equations of the system too big to resolve it directly. Moreover, this representation would include a lot of high degree monomials caused by the 31 successive rounds [46].

The complexity of this system of equations can be reduced using several algebraic transformations. For example, we could introduce new internal variables [17] to work with a huge number of low degree polynomial equations instead of small number of huge, high degree equations. For each round \(i\) of Present, three groups of variables can be used [46]:

- one variable \(x_i\) for each input bit of the s-box
- one variable \(y_i\) for each output bit of the s-box
- one variable \(k_i\) for each bit of each sub-key

The most interesting parts of the cipher are the non-linear elements which are the substitutions. They are the hardest parts to reduce to low degree equations. But for small s-boxes, it has been shown that such equations can be built in a systematic manner [8].

5.2.2 Conversion to a SAT problem

The next step is to reduce the system of equations to a SAT problem expressed in a Conjunctive Normal Form (CNF) because most of the SAT solvers take their input as a CNF formula. A SAT problem is in CNF if it is a conjunction (AND) of clauses which are a disjunction (OR) of literals [46]. A literal can be a variable \((x)\) or its negation \((\neg x)\).

The conversion into a SAT problem from a system of algebraic equations is divided in two steps: the translation of every monomial of degree higher than one and the translation of exclusive disjunctions (XOR) into conjunctions and disjunctions [46].

To remove the monomials of degree higher than one [4], we can introduce a dummy variable a set of clauses. For example, to remove the monomial \(x_1x_2x_3x_4\) we replace it by a variable \(a\) and the set of clauses:

\[
(x_1 \lor \neg a) \land (x_2 \lor \neg a) \land (x_3 \lor \neg a) \land (x_4 \lor \neg a) \land (a \lor \neg x_1 \lor \neg x_2 \lor \neg x_3 \lor \neg x_4)
\]
For each monomial of degree \(d > 1\), we replace it by a variable and \(d + 1\) clauses.

To translate the XOR-equations [4], we need to introduce dummy variables to bound the number of terms in the equation otherwise the number of new clauses could grow exponentially [4, 18] with long XOR-equations. Each equation \(x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_n = 0\) is transformed into:

\[
\begin{align*}
x_1 & \oplus x_2 \oplus x_3 \oplus b_1 = 0 \\
b_1 & \oplus x_4 \oplus x_5 \oplus b_2 = 0 \\
\vdots \\
b_m & \oplus x_{n-1} \oplus x_n = 0
\end{align*}
\]

Using this method, the long XOR-equations of size \(n\) are reduced to an equivalent set of \(m = \lceil \frac{n}{2} \rceil - 1\) (for \(n > 2\)) 4-term equations and \(m\) dummy variables.

A 4-term equation of the form \(a \oplus b \oplus c \oplus d\) can be transformed in a set of 8 clauses:

\[
\begin{align*}
(\neg a \lor b \lor c \lor d) & \land (a \lor \neg b \lor c \lor d) \land (a \lor b \lor \neg c \lor d) \land (a \lor b \lor c \lor \neg d) \\
(\neg a \lor \neg b \lor c \lor \neg d) & \land (\neg a \lor \neg b \lor \neg c \lor d) \land (a \lor \neg b \lor \neg c \lor \neg d)
\end{align*}
\]

By combining these two steps, we can build a CNF formula for the system of equations describing the targeted cipher.

5.2.3 CryptoSAT

As already mentioned, to validate an algorithm, an evaluator needs to check its algebraic properties to see if it can be described in a too simple or too over-defined system of polynomial equations. This evaluation can be time consuming as seen in Section 5.2. So, in 2014, F. Lafitte proposed an extensible package for SAT-based cryptanalysis [23]. This tool allows to automatically verify the properties of symmetric key primitives and their C/C++ implementations. It has been built to have a certain amount of objectives [23]:

- **Transparency**: The user is not required to be familiar with the verifications methods.
- **Language**: The target is specified in a well-known language.
- **Integration**: The user can stay on one platform and use some scripts to automate the entirety of the analysis.
- **Flexibility**: The tool is able to verify some arbitrary propositional properties.
- **Verifiability**: The tool must guarantee that the SAT encoding are reliable by including test vectors and unit testing.

This tool uses the following process to generate SAT instances based on the implementation of the targeted cipher [23]. An algorithm can always be decomposed into a sequence of instructions and encoded into a SAT problem by taking the conjunction of these instructions. With this top-down approach, if the instructions can be expressed in a CNF the construction of the SAT instances is trivial. To transform an instruction into a CNF, CryptoSAT has selected the common operators found in symmetric key primitives such as the bitwise logical operator NOT(\(\neg\)), XOR(\(\oplus\)), AND(\(\land\)), OR(\(\lor\)) as well as addition modulo \(2^l\) (\(\oplus\)) and left rotation of amplitude \(s\) (\(\ll s\)) with \(s\) a constant integer. The s-boxes are also frequent component of primitives [23]. These operators are overloaded by CryptoSAT in order to convert the C/C++ instructions into a SAT formula.

An overview of the CryptoSAT package is provided in Figure 5.2
5.3 Online phase: side-channel attacks

In 2009, M. Renaud and F-X. Standaert performed a successful attack [46] on an implementation of PRESENT with an 80-bit key. This attack used 8-bit RISC-based micro-controller which leaked a power consumption strongly correlated with the Hamming weight of the manipulated data. In their attack they used a Bayesian template attack to perform a partial key recovery. The goal behind this attack was to recover the maximum information about the intermediate rounds of the encryption algorithm. The Bayesian template attack [13] uses a profiling step to recover these data. The profiling step requires to have a device with a known key prior to the attack but once this step done, the profile can be re-used for as many attacks as possible.

5.4 Computing the solution

The results of the SCA are introduced in a SAT solver with the formula describing the encryption algorithm. The weakness of this attack lies in the fact that the SAT solvers can hardly deal with errors of input such as impossibilities. This can be solved using the following techniques [46]:

- Detection of impossibilities: rejects the leakages data which leads to incoherent input and output of the S-boxes.

- Selection of most likely Hamming weights: uses the solutions with highest probability first.

Once the additional clauses are added to the SAT problem describing the targeted cryptographic algorithm, the last step is to solve this problem using a SAT Solver.
5.4.1 SAT Solvers

As mentioned in Section 5.1, the SAT problem belongs to the NP-Complete class. Solve a superpolynomial-time algorithm such as the SAT problem is far from trivial. But since this problem belongs to the class NP-Complete and since a lot of problems [24] can be reduced to a SAT problem, the researchers focused their attention to find an efficient solution to the SAT problem.

The area of SAT Solving has seen enormous progress over the last years thanks to a tradition of yearly SAT competitions [50] and SAT races/challenges. Each year these competitions develop new algorithms, better heuristics and refined implementations techniques [51] to solve these problems. A decade ago, many problems were out of reach of the solvers of that time but now, these same problems can be handled routinely.

The DIMACS-CNF format

The first step to allow these competitions to take place was to develop a standardized way of describing the SAT problems. The DIMACS-CNF [29] format was proposed as a solution. A DIMACS-CNF file contains ASCII characters and a set of lines where the fields are separated by at least one whitespace. The file begins with a preamble where two types of lines can appear:

- **Comments**: which gives some human-readable information about the CNF formula or the SAT problem of this formula. These lines are ignored by the programs using the DIMACS-CNF file. A comment line begins with a lowercase “c”.

- **Problem line**: there is only one problem line per input file which has to appear before the clauses. The problem line begins with a lowercase “p” and has the following format:

  \[ p \text{ FORMAT VARIABLES CLAUSES} \]

  where the **FORMAT** allows the programs to determine the expected format. It should contain the word “cnf”. The **VARIABLES** contains an integer value defining the number of variable in the instance whilst the **CLAUSES** contains its number of clauses.

  Originally, this line had to be the last one of the preamble [29] but nowadays, the comments lines are accepted anywhere in the DIMACS-CNF file.

The remaining of the document contains the clauses. They must appear immediately after the problem line. The DIMACS-CNF format assumes that the variables are numbered from 1 up to \( n \). Each clause is represented by a sequence of numbers separated by either a space, a tabulation or a newline character. The end of a clause is represented by the value 0. A variable could appear in these clauses in two versions: the non-negated version of the variable \( x_i \) is represented by the value \( i \) whilst its negated version is represented by the value \( -i \).

By following the DIMACS-CNF guidelines [29], the example:

\[
(x_1 \lor x_3 \lor \neg x_4) \land (x_4) \land (x_2 \lor \neg x_3)
\]

can be expressed in the following DIMACS-CNF format:
The last competition was the “SAT-Race 2015” which took place during the “18th International Conference on Theory and Applications of Satisfiability Testing”.

This competition consisted of three tracks:

- **Main Track**: where the solver are executed on the “StarExec-cluster” which is a cross community logic solving service. It was developed at the university of Iowa with the main goal to facilitate the experimental evaluation of logic solvers. The solvers were allowed to run during one hour and the number of instances solved (from a set of benchmark instances) will be evaluated across 5 runs.

- **Parallel Track**: where the solvers have access to more than one core. For SAT-Race 2015, the competitors were evaluated on 8 core computers and on 64 core computers.

- **Incremental Library Track**: Many SAT problems try to solve a sequence of incrementally generated instances. To solve these kinds of problems, the SAT solver is invoked multiple times with different assumptions and with the addition of new clauses and variables. An Incremental SAT solvers has an advantage against a traditional SAT solver for these kinds of problems since it can reuse the knowledge acquired during the previous invocations.

This work will focus only on the Main Track since the SAT problem describing a cryptographic algorithm is not incremental and since no highly parallel computer was available for this work. Notice that a cryptographic algorithm could be described in an incremental way by describing each round by a different SAT problem. There were 28 solvers competing in the Main Track during the SAT-Race 2015. The competitor was benchmarked over 300 “applications”. The results are the following:

- **1st Prize**: abcdSAT by Jingchao Chen (261 solved)
- **2nd Prize**: MiniSatBCD by Jingchao Chen (256.4 solved)
- **3rd Prize**: Riss v5.05 (blackbox) by Lucas Kahlert, Franziska Krüger, Norbert Manthey and Aaron Stephan (249.4 solved)

**5.4.2 CryptoMiniSat**

Unfortunately, the code source or executable of these SAT solvers are not published. For this work, the SAT Solver used is CryptoMiniSat 4. CryptoMiniSat (Máté Soós) ranked 8th in the SAT-Race 2015 and is licensed under LGPLv2. CryptoMiniSat (available for download at http://www.msoos.org/cryptominisat4/) is a modern, multi-threaded symplifying SAT solver. It requires the UNIX packages “cmake” “libboost-program-options-dev” “zlib1g-dev” to be installed.
Part II

Implementation of the attack
Chapter 6

Implementation of Present

For this work, an implementation of the Present algorithm was realized. Each step of the encryption algorithm was implemented into a set of functions.

AddRoundKey. As shown in Section 2.2.1, the first step of present is a XOR operation between the block and the round key. This operation is executed by using the C binary XOR operator \( ^{\oplus} \). Each time a round key is used by the encryption algorithm, it is transformed into the next round key as specified by the key schedule.

sBoxLayer. The second step of the algorithm uses the Present S-box defined in Table 2.1 to transform the block under encryption. This can be done using a lookup table for the transformations of a nibble (half a byte). The nibbles are extracted from the original byte by masking the other half. If the nibble is the leftmost one, it needs to be shifted by 4 positions to the right before the transformation by the S-box and the results will be shifted by four positions to the left. This process is done for every byte of the block.

pLayer. The third step of an encryption round of present is a permutation of the bits of the blocks using the lookup table presented in Table 2.2. This permutation can be performed using a combination of masks and shifts.

Key schedule. After extracting the 64 leftmost bits of the key, the key is modified through the key schedule composed of the following three steps (notice that the key is stored in an array where the index 79 contains the leftmost bit and the last index, the rightmost one):

- The bits are rotated by 61 position to the left using some specifics masks and shifts.
- The 4 leftmost bits are transformed by the S-box using, again, a shift of 4 positions to the right, a lookup table and a shift of 4 positions to the left.
- The round counter is XORed with the bits \( k_{19}...k_{15} \)
Full round encryption. If each of the previous transformations are done in a separate function, a full round encryption can be executed by executing 31 times the sequence of operations:

- **addRoundKey** which executes the key schedule after the XOR of the round key with the block of plaintext.
- **sBoxLayer** which applies the S-box to the block of plaintext.
- **pLayer** which permutes the content of the block.

Optimized version. This implementation is not the most optimized one. Indeed, Bo Zhu and Zheng Gong proposed a new implementation [59] where the substitutions and permutations are executed at the same time using a lookup table. Using such lookup table optimized this algorithm for speed at the cost of an increased memory usage. Indeed, this allows for each byte of the output to be computed in one memory access instead of one memory access and 4 operations for the masking and shifting of the nibbles for the S-box and 24 operations for the permutation. Such lookup table can be constructed by computing for each input of the S-box the corresponding output of the permutation.

Since this implementation is more optimized it is more susceptible to be used in a real-life cryptographic device. For this reason and because a large number of encryption is required for a DPA attack, the optimized version was used as the target of the ASCA of this work. The code of this implementation of Present can be found in the annexes (Section 10.2 while the less optimized one can be found in Section 10.1.)
Chapter 7

Offline phase: CryptoSAT

This chapter describes the technical details of CryptoSAT [23, 22] and how it can be used to extract the description of a cryptographic algorithm.

7.1 Installing CryptoSAT

CryptoSAT is an R package which can be installed by using the following command in an R session:

```r
install.packages(cryptoSAT.tar.gz, repos = NULL, type="source")
```

Note that to install CryptoSAT the user needs to have the package R.oo installed. It allows oriented programming in R. Two functions from that package will be used by CryptoSAT:

- `setConstructorS3(name, function(initializer)...)` which creates an object `name`.
- `setMethodS3(name, object, function(this,..)...)` which creates the function `object$name`.

7.2 Generate the SAT problem

CryptoSAT describes 3 objects allowing the user to create and solve the SAT problem describing a cryptographic algorithm [23]:

- **Target**: Allows the user to define the SAT problem corresponding to the targeted cryptographic algorithm. It allows the user to define the parameters of the cryptographic algorithm and to compute the corresponding Instance object.

- **Instance**: Allows the user to add to the SAT problem some properties on bits. This object also allows to compute the Solution which is the result of a SAT solver choosen by the user.

- **Solution**: It contains the solution of the SAT solver executed on the SAT problem describing the targeted cryptographic algorithm.
7.2.1 Target

Generally, the SAT instance corresponds to variants of a primitive. These variants are parametrized by the number of rounds, the sboxes or some additional constants used such as the initialization vector.

The Target object allows the user to define these variants and to describe the cryptographic algorithm based on its C++ code. Once the Target correctly defined, the user can produce the SAT instance corresponding to the targeted cryptographic algorithm.

Creation of the Target

This object is created using a constructor defined using the method `setConstructorS3` from the `R.oo` library. The following R code defines the extension of Target for Present:

```r
setConstructorS3("Present", function() {
    extend(Target("Present"), "Present", .parameters = data.frame(
        rounds = 31
    ))
})
```

Once this constructor defined, the user can construct an instance of Target with the following code:

```r
target <- Present()
```

Now, `target` defines a Present object for which the parameters can be accessed or modified through the inherited methods `target$getParameter(name)` and `target$setParameter(name,value)`.

Definition of the save method

To complete the definition of the target, the user needs to define a method `save(filename)` that produces the parametrized C++ code defining the targeted cryptographic algorithm.

The original C++ code needs to be updated to use the types defined in the class `U*`. This class redefines the types `uint8_t, uint16_t, uint32_t` and `uint64_t` into `U8, U16, U32` and `U64`. The operators of `U*` are overloaded so that they output clauses.

Using the `U` types, the user has access to a `print` function which will write into two files (“variables.txt” and “values.txt”). These two files will be used when executing the C++ defined in the `save` method to contain the information needed to build the SAT problem of Present.

The definition of the `save` method can be found in the Appendix (Section 10.4). For this implementation, the `U.h` class was modified to be automatically casted to integer. This was done by adding the following line to the `U*` class:

```cpp
class U { 
  public:
    ...
    operator int() { return value;}
  ...
}
```
In some block ciphers, the data are modified by a non-linear substitution using a S-box. As seen for Present in Table 2.1, a S-box is a lookup table applied on a part of the data to encrypt. Present uses a $4 \times 4$ S-box.

There are different C++ implementations of a same S-box depending of the size of the input and output. For example a $4 \times 4$ S-box is implemented differently if applied to a byte [23]:

```c++
const unsigned char sBox4[] = {12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2};

byte = ((sBox4[byte >> 4] << 4) | sBox4[byte & 0xF]);
```

or to a word [23]:

```c++
const unsigned uint16_t sBox4[] = {12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2};

sBoxValue = state & 0xF; // get lowest nibble
state &= 0xFFFFFFFFFFFFFF0; // kill lowest nibble
state |= sBox4[sBoxValue]; // put new value to lowest nibble (sBox)
```

Finally, in some implementation it is the leftmost nibble of the look-up table that is used instead of the leftmost one, hence an additional boolean parameter is defined by CryptoSAT.

CryptoSAT defines an object Sbox in order to define all the S-boxes in the same way. First the Sbox object needs to be parametrized with the values of the lookup table by concatenation of the string of the hexadecimal values of the S-box. Then, it can be initialized by specifying the size of the S-box ($n \times m$) and the input and output word length:

```r
values <- c("c","5","6","b","9","0","a","d","3","e","f","8","4","7","1","2")
S <- Sbox(values, n=4, m=4, iwl=8, owl=8, name="presentSbox", msb=FALSE)
```

An object Sbox defines three methods producing the C++ code corresponding to the S-box [23]:

- **LUT.cpp()** which returns the C++ lookup table for this S-box.
- **CNF.cpp()** which returns a function that produces the clauses corresponding to a call to the Sbox object.
- **SIG.cpp()** which returns the signature of the function obtained with **CNF.cpp()** while the sbox is compiled into a separate binary.
- **FUN.cpp()** which uses the results of **LUT.cpp()** and **CNF.cpp()** to produce the return value and the clauses of a call to the Sbox object.
Since the compilation of the code returned by `CNF.cpp()` requires some time to compile (i.e. \(m^2 n\) instructions [23] for a \(n \times m\) S-box) CryptoSAT proposes two solutions to avoid the compilation at each call of the function. First, within a same R session, the method `target$generateInstance()` will automatically detect that an object Sbox has already been compiled. CryptoSAT also provides the user with a function `compileSboxes` which puts the output of the function `CNF.cpp()` into a separate file, compiles it and return the name of the resulting binary.

Once created, the Sbox object can be used in the `target$save()` method:

```r
setMethodS3("save", "Present", function(this, filename, ...) {
  f <- this$.init( filename )
  values <- c("c","5","6","b","9","0","a","d","3","e","f","8","4",
             "7","1","2")
  sbox <- Sbox(values, n=4, m=4, iwl=8, owl=8, name="presentSbox")
  this$.write(f, paste( sbox$LUT.cpp(), sbox$SIG.cpp(), sbox$FUN.cpp(), sep="\n" ))
  this$.write(f,"int main()"
    ...
  this$.write(f,"  round_key[0] = (round_key[0] & (U8)0x0F) | presentSbox(round_key[0] >> 4);")
  ...
})
```

The object Sbox can also be used for any lookup table used in the targeted cryptographic algorithm. This is interesting for the case of Present when implemented in an optimized way. Indeed, Bo Zhu and Zheng Gong proposed an optimized version [59] of the software implementation of Present. This version takes 1.83 ms on MICAz wireless sensor node to perform a full-round (31 rounds) encryption of Present. The main optimization proposed was to perform the computation of the substitution and permutation layer in one step by using several lookup tables. These lookup tables can be declared and used in cryptoSAT as if they were \(8 \times 8\) S-boxes.

### 7.2.2 Instance

Once the Target object is completely created and parametrized, the method `target$generateInstance()` can be used to obtain the instance of the target.

An Instance has 3 private members corresponding to the C++ output:

- **.clauses**: is a vector of the clauses of the SAT problem describing the targeted cryptographic algorithm.

- **.variables**: is a matrix containing the index of the propositions for the interesting words. These words are the one defined in the `target$save` method through the `U::print` method.

- **.values**: is a similar matrix containing the values taken by the propositions during the execution of the C++ program define in `target$save`. 

Since these fields are private, they are not meant to be directly accessed but CryptoSAT allows to act upon them through the methods \texttt{instance\$setEqual}(X,Y) and \texttt{instance\$setNotEq}(X,Y).

These methods transform the SAT problem by encoding [23] the equality (and inequality) into CNF as shown in Equation 7.1 and 7.2. In these equations, $A$ and $B$ are the two words set to equals or unequals and $l$ is the length of the two words.

\begin{align}
\Phi_{A=B} &= \bigwedge_{i=1}^{l} \Phi_{a_i=b_i} \\
\Phi_{A \neq B} &= \bigvee_{i=1}^{l} \Phi_{a_i \neq b_i} \\
&= \left( \bigvee_{i=1}^{l} t_i \right) \land \bigwedge_{i=1}^{l} \Phi_{t_i=\Phi_{a_i \neq b_i}} 
\end{align}

In the case of an ASCA, these functions can be used to encode the properties gathered by the SCA. This will be detailed in the Chapter 9.

### 7.2.3 Solution

A Solution is obtained from an Instance object by executing the method \texttt{instance\$solveWith(path_to_solver)}. This method can be used with any SAT solver if it respects the input-output format used in SAT competitions.

The Solution object has two purposes [23]:

- It provides the access to the SAT solver metrics such as the solving time or the memory usage. For example, the following code provides the user with the solving time:

  ```r
  solution <- instance\$solveWith("/bin/cryptominisat")
  if( solution\$isSAT() )
  cat("solved in", solution\$getReportedCPU(), " seconds\n")
  ```

- It provides access to the results of the solver. This is done using the method \texttt{solution\$getValueOf(variableName)}. For example, the following code prints the key found by the SAT solver:

  ```r
  solution\$getValueOf("K")
  ```
7.2.4 Check the generated instance

When extending CryptoSAT with a new Target, it is important to ensure that a generated instance respects the targeted cryptographic algorithm. This can be done using the “Known Answer Test” also called “Test Vector” of the cryptographic algorithm. The test vectors of Present are described in Figure 7.1.

<table>
<thead>
<tr>
<th>plaintext</th>
<th>key</th>
<th>ciphertext</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>5579C138</td>
</tr>
<tr>
<td>00000000</td>
<td>FFFFFFFF</td>
<td>E72C46C0</td>
</tr>
<tr>
<td>FFFFFFFF</td>
<td>FFFFFFFF</td>
<td>A112FFC7</td>
</tr>
<tr>
<td>FFFFFFFF</td>
<td>FFFFFFFF</td>
<td>3333CD3</td>
</tr>
</tbody>
</table>

Figure 7.1: Known Answer Test of Present [9]

For example, the following code tests the correct extension of CryptoSAT with Present for the first test vector of Figure 7.1.

```r
instance <- target$generateInstance()
for(i in 1:8){
  instance$setEqual(paste("X\[",i,"\]", sep = ""), "00")
  instance$setEqual(paste("K\[",i,"\]", sep = ""), "00")
}
for(i in 9:10){
  instance$setEqual(paste("K\[",i,"\]", sep = ""), "00")
}
instance <- solveWith(solver)
expected <- c("45", "84", "22", "7B", "38", "C1", "79", "55")
for(i in 1:10){
  checkEquals(solution$getValueOf(paste("K\[",i,"\]", sep = ""),
  expected[i])
}
```

7.2.5 Extending the CryptoSAT package

If the targeted cryptographic algorithm is an important, widely used and studied algorithm, it could be interesting for other researchers to share its CryptoSAT implementation.

To make the extension of the Target $T$ available for other researchers, two additional steps [23] are needed:

1. Write a small documentation file $T.Rd$ used to cite the source of the C++ implementation used for the target and the source of the test-vectors.

2. Write a file $T.runit.R$ implementing the unit-tests based on the test-vectors cited in $T.Rd$.

For this work, the CryptoSAT was extended to support the Present algorithm and the corresponding files Present.Rd and Present.runit.R were created.
7.2.6 Summary

To extend the CryptoSAT package with a new targeted cryptographic algorithm, three steps are required:

1. Extend the Target object and defines the relevant parameters such as the rounds, the constants, ...

2. If the algorithm uses an S-box or any lookup table, override the function `writeS-boxes()`.

3. Override the method `save(filename)` that will generate the parametrized C++ code corresponding the cryptographic algorithm.

Once these steps are performed, the Target object can be used to generate an instance and this instance can be manipulated and solved within CryptoSAT. Once integrated, the target is available for the community to reuse it.

As seen in this Chapter, it requires minimal effort to extend CryptoSAT with a new targeted cryptographic algorithm. The transformation from the C++ code to a SAT problem is pretty straightforward and does not need any familiarity with propositional reasoning. In addition, the automation offered by CryptoSAT is useful for the design and analysis of cryptographic algorithms specially under time constraints, as it can be the case in competitions.

Since the Target object can be parametrized, it is easy to script experiments by varying some parameters, properties or SAT solvers.
Chapter 8

Online phase: Correlation Power Analysis

As presented in the Section 4.3.2, a DPA focuses exclusively on the data dependency of a large number of power traces. These attacks generally follow a five-steps strategy.

8.1 Step 1: Choosing an intermediate result of the executed algorithm

For this step, the attacker chooses the interesting step of the algorithm which will be attacked. For this, he chooses some intermediate results which are a function $f(d,k)$ where $d$ is a known non-constant data (generally the plaintext eventually modified by some known transformation) and $k$ is a small part of the key (generally a byte from the key or from the round key.)

For Present, since it is a block cipher using a S-box, an interesting intermediate result for this attack would be after this S-box. The S-box is interesting to be attacked since the lifetime (time when it is unchanged) of a bit lasts until the end of the round [11]. During a DPA attack, a peak in the correlation will appear each time this bit and its 3 peers (since they are part of the same machine word) are transformed by the permutation.

The attack described here targets the output of the S-box of the first two round (see Section 8.6 for more explanation about the second round) but another CPA could target the last two rounds with the same idea.

To be able to compute this intermediate result, the Present.h\(^1\) file was extended with an additional method:

```
1 uint8_t sBoxOneByte(uint8_t plain, uint8_t keyBlock){
2   uint8_t xoredResult = plain ^ keyBlock;
3   uint8_t result = sbox[(xoredResult & 0x0F) >> 4]; // shift car sbox[] is << 4
4   result += sbox[(xoredResult & 0xF0) >> 4];
5   return result;
6 }
```

\(^1\)Notice that Present.h is the optimized code [59].
8.2 Step 2: Measuring the Power Consumption

To measure the power consumption of an encryption of a plaintext by the Present cryptographic algorithm, the optimized code [59] was transformed into an Arduino version. This is done by commenting the declaration of the types \_UINT\_T\_ and by adding the functions \_setup\_() which initialize the Arduino and \_loop\_() which will be indefinitely executed during the execution of the Arduino. The complete code can be found Section 10.5.

The \_setup\_() function initializes a communication channel over the Serial bus, sets a seed for the random computation, setups the trigger pin and initializes the plaintext at 0:

```c
void setup()
{
  Serial.begin(9600); //setup communication
  randomSeed(1); //setup random numbers
  pinMode(pinTrigger, OUTPUT);
  digitalWrite(pinTrigger, LOW);
  initPlaintext();
}
void initPlaintext()
{
  unsigned char i;
  for(i=0;i<8;++i){
    plainText[i]=0x00;
  }
}
```

The \_loop\_() function will read the value of the byte on the serial channel and execute the code corresponding to this byte:

```c
void loop () {
  if (Serial.available () > 0){
    byteIn = Serial.read (); //read command byte
    Serial.println(byteIn);
    if (byteIn == 'c'){
      digitalWrite(pinTrigger, HIGH);
      present(plainText, saveKey, workingBlock);
      digitalWrite(pinTrigger, LOW);
    } else if (byteIn == 'v'){
      changeValue();
    } else if(byteIn == 'p'){
      printBlock();
    } else if(byteIn == 'k'){
      printKey();
    } 
  }
}
```

This command byte can take several values:
• **c**: This command encrypts the plaintext using the key. It also set the trigger to *HIGH* during this encryption such that only the power consumption of the encryption is recorded by the oscilloscope. The trigger is also used to align the traces of the encryptions.

• **v**: Sets a random value for the content of the plaintext.

• **p**: Prints the content of the plaintext on the Serial channel.

• **k**: Prints the content of the key on the Serial channel.

When the Present implementation is compatible with Arduino, the code can be transferred to the device under attack.

The device under attack for this work is an ATmega 328P microchip. The detailed scheme of this acquisition setup can be found in Figure 8.1. This device is connected to an “Infinium 9000 Series” oscilloscope (model MSO9254A) used with a sampling rate of 200 millions per second.

To be able to measure the power consumption of the encryption of a large number of data, some Processing [45] code was used. Processing is an automation framework allowing to communicate with Arduino in order to automate some computation. This code was kindly provided by Nikita Veschikov and it allows to automate the computation of $D$ power traces. This code also allows to perform the average of several traces for each data $d_i \in [d_1, ..., d_D]$.

Once the power traces computed, the code in Section 10.6 allows to retrieve a matrix $T$ of size $D \times T$ where $D$ is the number of different plaintext encrypted and $T$ is the size of the recorded traces. One can retrieve the values for $d_i$ by executing the function `getValues(foldername)` of the code 10.6, whilst the function `readTracesFrom(folderName, numberOfValuesToRead)` allows to retrieve the matrix $T$ containing a certain number of power traces.

### 8.3 Step 3: Calculating Hypothetical Intermediate Values

The third step of a DPA is the computation of the hypothetical intermediate values. For each $d_i \in [d_1, ..., d_D]$, $k_i \in [k_1, ..., k_K]$ where $d_i$ is the plaintext values and $k_i$ is the hypothesis key bytes, the hypothetical intermediate value is computed by applying the function $f(d, k)$ on it. The results, obtained by the following calculation, are then stored in the matrix $V$ of size $D \times K$.

$$v_{i,j} = f(d_i, k_j) \quad \text{where } i = 1, ..., D \text{ and } j = 1, ..., K$$

This calculation is done by executing the following C++ instruction:

```cpp
V[i][j] = sBoxOneByte(values[i][blockIndex], hypothesis[j]);
```
(a) Scheme of acquisition setup using the ATMega 328P microchip

(b) Picture of the device under attack with the connection to the oscilloscope

Figure 8.1: Description of the device under attack
8.4 Step 4: Mapping Intermediate Values to Power Consumption Values

The fourth step is to use a simulation technique to map the hypothetical intermediate values $V$ to a matrix $H$ of hypothetical power consumption values. To do so, the attacker needs to use a simulation technique [36, Section 3.3]. Since the device under attack for this work leaks pretty well the Hamming Weight of the key [6, Section 6.1], this simulation technique is a good choice for this mapping.

The computation of the Hamming Weight of a block can be easily computed by using a mask in order to retrieve only the rightmost bit and by shifting the block $n - 1$ (where $n$ is the size of block) times to the right. The successive rightmost bits are then summed to compute the Hamming weight. With the function $getHammingWeight()$ defined, the mapping of $v_{i,j}$ to $h_{i,j}$ is pretty straightforward:

$$H[i][j] = getHammingWeight(V[i][j]);$$

Notice that the steps 3 and 4 can be computed in one operation since the matrix $V$ is not used in the computation of the results of this attack.

8.5 Step 5: Comparing the Hypothetical Power Consumption Values with the Power Traces

Once the matrices $H$ and $T$ are computed, the last step is to compute, for each value, hypothesis, and instant of the traces, the correlation between the value in $H$ and the one in $T$. The computation of the correlation in a SCA is recalled in Equation 8.1 where $i = 1, ..., K$ and $j = 1, ..., T$.

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \bar{h}_i) \times (t_{d,j} - \bar{t}_j)}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h}_i)^2 \times \sum_{d=1}^{D}(t_{d,j} - \bar{t}_j)^2}}$$  (8.1)

To compute the correlation for the hypothesis key $i$ at instant $j$, two additional values are required:

- $\bar{h}_i$: which is the mean value of the column $h_i$.
- $\bar{t}_j$: which is the mean value of the column $t_j$.

Once these two values are computed, the computation of the correlation is pretty straightforward. It can be computed by the following function:

```c
double computeMaxCorrelation(uint16_t hypIdx) {
    double maxCorrelation = 0;
    bool init = false;
    for (unsigned long time = 0; time < waves[0].size(); ++time) {
        double numerateur = 0, denomHyp = 0, denomTrace = 0;
        for (unsigned valIdx = 0; valIdx < valuesToRead; ++valIdx) {
            numerateur += (H[valIdx][hypIdx] - meanHypothesisHW[hypIdx]) * (waves[valIdx][time] - meanValue[time]);
            denomHyp += (H[valIdx][hypIdx] - meanHypothesisHW[hypIdx])^2;
            denomTrace += (waves[valIdx][time] - meanValue[time])^2;
        }
        double correlation = numerateur / sqrt(denomHyp * denomTrace);
        if ( correlation > maxCorrelation ) {
            maxCorrelation = correlation;
            init = true;
        }
    }
    return maxCorrelation;
}
```
\[
\text{denomHyp} += \text{pow}((H[\text{valIdx}][\text{hypIdx}] - \text{meanHypothesisHW}[\text{hypIdx}]), 2);
\]
\[
\text{denomTrace} += \text{pow}((\text{waves}[\text{valIdx}][\text{time}] - \text{meanValue}[\text{time}]), 2);
\]

\[
\text{double currentCorrelation} = \frac{\text{numerateur}}{\sqrt{\text{denomHyp} \times \text{denomTrace}}};
\]

\[
\text{if} (!\text{init} || \text{currentCorrelation} > \text{maxCorrelation}) \{
\text{maxCorrelation} = \text{currentCorrelation};
\text{init} = \text{true};
\}
\]

\[
\text{return maxCorrelation;}
\]

8.5.1 Acceleration of the computation

For this experiment, the encryption key is known which allows this experiment to be greatly accelerated. Indeed, it is possible to compute the correlation for a reduced set of instants \(t_i\) instead of the complete set of size \(T\). To find the instant where the correlation for the correct hypothesis is maximal, we can modify the function \text{computeMaxCorrelation} to return the index of the maximal correlation. This function can then be executed for the correct hypothesis key.

Once the value of \(j\) maximizing the correlation for the key is found, the time interval can be computed by taking the instants in a window centered at the index of the maximum correlation.

This method has the disadvantage of being sensitive to errors when the data are noisy. Indeed, if there is a high correlation at a wrong position of the algorithm, the computation of the entire attack will be executed on the wrong data. To avoid this problem, the research for the index of the highest correlation is computed by taking the maximum correlation over a window of instants. The returned index of the maximum correlation can be the index of the highest instant of the window and this value can be used to compute the \text{lowerBound} and the \text{upperBound} by taking a multiple of the window used to find the maximal correlation. For example, the lower bound could be \(\text{maxCorrelationIndex} - (\text{windowSize} \times 10)\) and the upper bound could be \(\text{maxCorrelationIndex} + (\text{windowSize} \times 9)\). The complete code of this research can be found in the function \text{findMaxCorrelationIndex()} in Section 10.7.

8.6 Computation for all the bytes of the key

In the previous sections, the method to perform a CPA on a single byte of the key was explained. This process can easily be extended to the 7 next bytes of the key by using a loop. Since the first round of Present is performed by using the 64 leftmost bits of the key, these 8 bytes of the key can be attacked after the S-box of the first round.

But for the last two bytes, there is no way to compute the correlation for these bits by observing only the first round of Present. The solution is then to compute the most probable key after the attack on the first
round, to compose the key and use it to perform one round of the encryption on the values. As we can see in Figure 8.2, at the second round, the two unknown bytes have an impact on the 3 leftmost bytes of the data being encrypted. Then, an attacker can perform a CPA on these bytes and recover the 3 leftmost bytes of the second round key.

Once we have the correlation for the hypothesis for the second round key, we can recover the last two bytes of the key by inversing the key schedule. The Present.h code was extended in order to be able to reverse the key schedule. The first step to invert the key schedule is to XOR the correct bytes with the round counter of the previous round. Then the leftmost nibble is transformed using a lookup table inversing the S-box. The final step of the inversion is a shift of the key by 61 positions to the right.

If we study in more detail (see Figure 8.2) the propagation of the key during the key schedule of the first round, we can see two interesting properties:

- When constructing the key from the results of the first round, the two last bytes of the key which are still unknown can be set to any values without affecting the remaining computations. Indeed, they will have an effect only on the values after the second round of the bytes 5, 6 and 7. Based on that observation, the byte 8 and 9 of the key after the first round were set to 0 during this attack.

- By following the same reasoning, we can see that the two last bytes of the round one influence only the results of the three first bytes of the second round. This means that the next round don’t need to be performed on all the blocks of the second round but only on the first three rounds.

### 8.7 Assessing the results of the Correlation Power Analysis

Once this attack is done, for each byte of the key, each hypothesis is associated with the one correlation. This correlation can be seen as the “score” of the hypothesis. The probability of that hypothesis being the correct one is high when the correlation is very high compared to other hypotheses.
Based on that observation, the results of the attack can be assessed by checking the position of each byte of the correct key in the sets of correlations. To compute the score in percents of an attack, the following function can be used:

```cpp
double computeScore(vector<vector<double>> hypothesisScore) {
    double score = 0;
    for (uint8_t blockIdx = 0; blockIdx < 10; ++blockIdx) {
        vector<size_t> resultIndexes = sort_indexes(hypothesisScore[blockIdx]);
        bool found = false;
        for (uint16_t i = 0; i < 256 && !found; ++i) {
            if (resultIndexes[i] == key[blockIdx]) {
                score += 255 - i;
                found = true;
            }
        }
    }
    return score / 25.5;
}
```

This function only takes the rank of the key into account, it does not use the distance from the other hypotheses.

For the ASCA, these results will be used to be injected into the SAT problem describing the cryptographic algorithm. As explained in Chapter 9, there is no standard way to indicate to a SAT solver that a solution should be evaluated in priority. Based on that assumption, only some of the best hypotheses will be proposed to the SAT solver. The computation of the score of the CPA should then be updated to match this behavior in order to have a score which can be compared with the score of the ASCA.

The function `computeScore()` can then be updated to set the score of a byte to 0 when the correct hypothesis key is not in the $n$ highest ranked hypotheses. This is done by executing the for-loop at line 6 only on the $n$ best hypotheses. The score will then be computed by adding $n - 1 - i$ to the score when the correct hypothesis is in the $n$ best or 0 otherwise. The total score will then be computed by dividing it by $(n-1)/10$.

### 8.8 Discussion of the results of the CPA

Once this attack is implemented, we need to compute the results for several parameters in order to be able to evaluate this attack. The performance of this attack will be based on the score of the attack (i.e. based on the ranking of the solution, see 8.7) depending of the noise in the data and the quantity of queries to the cryptographic device.

The amount of noise in the data is determined by the number of queries to the cryptographic device of the same data used to compute the average power trace. For this experiment, the averages are performed by the oscilloscope. For this experiment, different levels of average were used: no average, 2, 4, 8, 16 and 32.

The quantity of queries is the number of request to the cryptographic algorithm of different data.

To be able to evaluate the score of this attack, for each average level, a thousand
different plaintexts are used and the corresponding power traces are recorded. Based on these thousand queries, the CPA attack will be used several times for each evaluated queries number. This is performed by the following code:

```cpp
double score = 0;
uint8_t count = 0;
uint8_t outsideTop = 0;
for (uint16_t i = 0; i < (1000 - traceNbr); i += 25) {
    score += computeCPA(folderName.c_str(), mean, traceNbr, i);
    if (!::result) {
        outsideTop++;
    }
    count ++;
}
::outsideTop = (double) outsideTop * 100 / count;
return score / count;
```

As we can see, this code evaluate a CPA based on two parameters. The first one is the score as presented in Section 8.7 while the second one evaluate the number of attacks where at least one byte of the correct key was outside the first $n$ hypotheses. This evaluation is useful because the SAT solvers cannot detect any errors in the input so if this wrong result is used for the ASCA, the SAT problem will become unsatisfiable.

The first evaluation of this attack is to verify that the correlation between the correct hypothesis key and the power traces has some higher peaks than the correlation with incorrect hypotheses.

As we can see in the Figure 8.3, there is a big difference in the correlation graphs depending on the hypothesis value. In this graph, the the key has her value 0xAC. This value is easily confirmed by the graph since in Figure 8.3c, there are several peaks in the data and these peaks are much higher than the ones present in the other graphs. Notice that for these graphs, an average was performed using 32 computation and this attack used 200 different values.

As we will see hereunder, this graph was performed in a situation where the score was almost always 100%. By observing the Graphs 8.3a and 8.3b, we can see that even when the hypothesis is not the correct one, we have a good correlation when the hypothesis is “near” the correct key. A value is “near” another when the Hamming Weight of the XOR between the two values is almost zero. For example, The XOR between 0xAA and 0x0 has an Hamming Weight of 4 and the XOR between 0xAA and 0xAC has an Hamming Weight of 2. Indeed, if we execute this attack for this byte of the key, we see that 0xA2, 0xA6 and 0xAB are placed in the top 5.

The previous method observes the results of an attack to verify that the correct key has a high correlation with the recorded power traces. But to be able to evaluate the performance of this attack for a specific parameter, several sets of data and corresponding power traces are used. The results of these attacks are then averaged. To compute the final results of this attack, the average was computed at least on 20 to 40 attacks. Once this computation is done, we obtain the graphs presented in Figure 8.4 and Figure 8.5.
Figure 8.3: Graphical representation of the correlations between the power traces and an hypothesis byte.
On this graph, we can see several information about the performances of this attack. The first observation is that a Correlation Power analysis has very low score when the data are too noisy or when there is not enough power traces. This graph shows that the CPA require the power traces to be cleaned in order to extract any information from the cryptographic algorithm. A CPA has acceptable result with an average on two or more queries to the cryptographic device.

By studying the graph 8.4, we can also see for the cleaned data that the differences between the total score of a CPA with a reasonable noise and the one with almost no noise can be important since it can go up to 30%.

In the graph 8.5, we can see the percentage of the attacks where at least one byte of the correct key was rejected from the selected hypothesis. As we will see in the Chapter 9, this situation is really problematic when combined with an Algebraic attack since it will render the SAT problem unsatisfiable.

In this graph, we can see that the CPA is very sensitive to the noise in the power traces since the difference between the power traces with a reasonable noise and the clean ones can go up to 90% with 50 traces used.

The second observation is that this attack needs roughly 50 traces to perform well with cleaned power traces. With 50 traces, the results of this attack are high enough to consider doing a research of the correct key even using a simple brute-force method without too much computation time. But when the correct key has a too low correlation, the key searching algorithm can take too much time. For example, if the correct hypothesis is in the 10 highest correlation for each byte, it can take up to:

\[10^{10} \approx 2^{33}\]

we can see that this is considered “easy to break” as shown in Table 3.1. If the key is in the 25 highest correlation the computation goes to:

\[25^{10} \approx 2^{46}\]

which is still practically solvable but needs a lot of resources. Moreover, if a cryptographic device allows such a large number of wrong key it can be considered as a security breach.

For these reasons, a brute-force attack is not the right way to compute the correct key. A better algorithm would use the information from the SCA as heuristic for the search. An example of such algorithm was shown in Section 4.4: the SKEA which order the key parts hypotheses by decreasing correlation and produces complete hypotheses and orders them by maximum likelihood computed on the cumulated score of this key hypothesis.

In conclusion, we can see that a CPA retrieves the key with a high confidence only when it can analyze enough cleaned power traces. This attack performs correctly from 100 power traces but to reach 100% accuracy all the time, this attack needs about 350 power traces.
Figure 8.4: Results of a CPA depending on the power traces and the noise of the power traces.

Figure 8.5: Percentage of the attacks where at least a byte of the key was rejected by the CPA depending on the power traces and the noise of the power traces.
Chapter 9

Algebraic side-channel attack on Present

Once the SCA is performed and once the cryptographic algorithm is described by a SAT Problem, the ASCA on Present can be executed.

The goal of this attack is to combine the results of the SCA (i.e. for each byte of the key, a list of hypothesis with their corresponding correlation) and to inject the information extracted from this list into the Instance object of CryptoSAT.

This is done by expressing the results of the CPA into a SAT problem. For example, if the CPA selects the 10 best hypothesis, we can define these ten values by the following variables: $v_1, v_2, ..., v_{10}$ where $v_i \in [0, 256]$. We can define the set $S$ containing the results of the CPA.

Since CryptoSAT allows to encode the equalities, we can express the hypotheses into the following expression for the $i^{th}$ byte of the key:

$$K[i] = v_1 \lor K[i] = v_2 \lor ... \lor K[i] = v_{10}$$

But the function setEquals() of CryptoSAT is unable to encode the $\lor$. It only joins the equalities with the rest of the SAT problem by using $\land$. A solution is to negate this propositional formula. The negation of a $\lor$ is $\land$ and CryptoSAT allows to encode the inequality. To negate the set $S$, we can take its complement such that each value $w_i$ of $\bar{S}$ is absent of $S$. Then, $\bar{S}$ can be expressed by:

$$K[i] \neq w_1 \land K[i] \neq w_2 \land ... \land K[i] \neq w_{j}$$

where $w_1, ..., w_j \in \bar{S}$.

For example, if we want to exclude the value $CA_x$ from the possible hypothesis for the byte 1 of the key of the cryptographic algorithm, the corresponding CryptoSAT instruction would be:

```c
instance$setNotEq("K1[0]", "ca")
```

Notice that for this implementation of Present, the variables are named by three fields: the name, the byte index and the round where this variable is used. In our previous
example where we excluded an hypothesis from the possible keys, the name is $K$, the byte index is 1 and the round counter is 0 since the key is set before the execution of the cryptographic algorithm.

Because a SAT solver is not able to deal with errors in its input [46], when the correct hypothesis is rejected by the SCA, the SAT Problem will become unsatisfiable. Because of this weakness in the SAT solvers, the weaknesses of a CPA will have a strong impact on the results of the SCA. Indeed, when a correct key is rejected by the CPA, the ASCA will fail.

As presented in the Section 8.8, CPA is very sensitive to the noise in the data and the lack of power traces. For this reason, the results of an ASCA using a CPA are strongly correlated with the Graph 8.5. Indeed, when the CPA provides enough information to CryptoSAT. The correct key is found by the SAT solver but when the CPA fails to provide coherent information, the SAT solver is unable to retrieve the correct key.

As shown in Figure 9.1, the results of the CryptoSAT are very bad when the data are too noisy or with not enough power traces. This behavior is caused by the poor results of the CPA in these conditions.
The results of this ASCA shows that any SCA cannot be associated with an Algebraic Attack without respecting several constraints:

- Among the information retrieved by the SCA, none should introduce any error in the SAT problem. This can be done by an additional step at the end of the SCA in order to detect the impossibilities before injecting them into the CryptoSAT. In 2009, M. Renaud and F-X. Standaert [46] proposed an example of such verification where they rejected the leakages samples that rise incoherent inputs and outputs for the S-boxes. For the CPA used for this attack, we don’t have access to the information about the intermediate steps of the computation of this algorithm since it is based only on the output of the first S-box. Moreover, if we want to gather some additional information about the following rounds, the reliability of the results goes down each time a computed round key is mixed with the data.

- Since the SAT problem of the algebraic attack describes the entirety of the encryption process, a good candidate for an ASCA should gather some information about the entirety of the cryptographic algorithm. Indeed if the SAT solver has access to information about the intermediate results, it can use it as a parallel with the description of these intermediate steps in the SAT problem describing the cryptographic algorithm.

- An additional property which would be interesting for an ASCA, is the possibility to establish a tradeoff between the amount of information gathered from the SCA and the amount of time for the SAT solving. Indeed, for some attacks, the device under attack is available only for a certain amount of times where the attacker can query the cryptographic algorithm. In these cases, the data could be too noisy and unreliable for some parts of the algorithm. In this case, the SCA should reject the information leaked by such data in order to avoid the introduction of errors. This would reduce the amount of information introduced into the SAT problem. Even in this situation, a good ASCA should be able to find the correct solution with, optionally, a longer computation.

Since the CPA does not fulfill these constraints because of its inability to produce some reliable results when there is not enough power traces and since the correct keys can be rejected, the CPA is not a good candidate for an ASCA.

As we can see, the results of this attack are worse than the attack proposed by M. Renaud and F-X. Standaert [46]. Indeed, they claimed to be able to recover enough information about the encryption algorithm to always recover the correct key even with up to one power trace.

This Bayesian template attack allows the description of the entirety of the cryptographic algorithm both by the Algebraic Attack and the SCA. Since it is able to gather so much information about the cryptographic algorithm, a detection mechanism can be added to this attack to remove the impossibilities in order to introduce only the correct data into the SAT problem. The removal of these information does not prejudice the computation of the ASCA thanks to the amount of information gathered by the Bayesian Template Attack.

The main difference between an attack with a CPA and a Bayesian template attack is that the template attack profiles the implementation of the cryptographic algorithm
in order to gather information about its internal successive states. For a CPA, the result of this algorithm is a complete information which is either correct or wrong whilst a template attack produces partial information about several rounds of the algorithm.

With a CPA, the SAT solver is mainly used as an intelligent key enumeration algorithm but with a Bayesian Template attack, the SAT solver disposes of a profile of the implemented algorithm which can be efficiently matched with the description of the theoretic algorithm described by the SAT problem which makes a better use of the algebraic attack of the ASCA.

Even if the CPA is not a good solution, the results obtained by this SCA could be used in a more efficient way in order to avoid the introduction of impossibilities into the SAT problem. Indeed, the CPA produces a list of correlation for each hypothesis. The impossibilities introduced in the SAT problem is caused by keeping only the $n$ best hypothesis and rejecting the other hypothesis. The only parameter which can be modified to increase the performances of the ASCA is $n$. But still, the results of a CPA on a single trace or on traces with a lot of noise, will sometimes introduce an impossibility in the SAT problem. Each time that an impossibility is found, this one can be avoided by increasing $n$.

Setting $n$ correctly depending on the power traces can be a bit challenging for an attacker since there is no way to know if the correct hypothesis key belongs to the $n$ best hypotheses or not except by solving the SAT problem of the ASCA. But when an incorrect value is introduced into the SAT problem, there is no guarantee that the problem will render unsatisfiable. Sometimes, another solution could satisfy the SAT problem. To find the correct value, one can be tempted to assign to $n$ an arbitrary large value but this reduce a lot the information provided to the SAT solver. The Figure 9.2 can help to set the correct value of $n$. As we can see in this Figure, when there is not enough power traces or too much noise, the $n$ can go up to 150 which leaves only 106 rejected hypothesis to solve the SAT problem. Notice that these values are presented as example, these results can be a lot different depending on the power traces. Moreover, these values do not guarantee that the correct hypothesis is contained in the $n$ hypotheses it indicates only the average observed positions of the correct key during the CPA of this attack.

To completely avoid these impossibilities, a new kind of SAT solver needs to be developed to accept an ordered list of possibilities. Indeed, the correlations resulting from the CPA can be transformed into probabilities which could be used as heuristic in a custom made SAT solver. This method is pretty similar to the SKEA but the SAT solver has additional information about the cryptographic algorithm. In this case, all the hypothesis could be used in the SAT problem. If the correct hypotheses are ranked in the best proposals, the SAT solver would solve quickly this problem but the computation time would increase when the correct hypothesis is poorly ranked. A possible candidate for such problem is the optimizer proposed by Laurent Simon during ECRYPT II 2012 [53].
Figure 9.2: Average position of the correct hypotheses in the ordered list containing the hypothesis sorted by decreasing correlation.
Chapter 10

Conclusions

The main goal of this work was to assess the performance of a widely used and easy to implement SCA such as the CPA or any other non-profiled attacks when used in an ASCA. To be able to perform such attack, the first step is to implement the cryptographic algorithm and to encode it onto a physical device. This implementation was detailed in Chapter 6 and the source code can be found in 10.1. Since a more optimized version of the code is proposed by Bo Zhu and Zheng Gong [59], this version of Present will be used as targeted cryptographic algorithm.

Once the cryptographic algorithm is correctly implemented, it can be described by a SAT problem. To perform this description, the tool CryptoSAT [23] was used. The corresponding code can be found in Chapter 10.4. As we can see in this implementation and in Chapter 7, this tool can be easily extended in order to support the Present encryption algorithm. Since this cryptographic algorithm is important and used by the scientific community, its inclusion into CryptoSAT was performed during this work in order to make it publicly accessible.

The next step of this attack is to compute the power traces of several encryptions using this cryptographic device. The code contained in the device under attack is presented in Chapter 10.5.

Once the power traces recorded, the SCA can be implemented. For this work, the attack studied is a CPA with Hamming Weight leakage model. The complete code of this attack can be found in Chapter 10.7. This program will compute for each byte of the key and for each hypothesis, the correlation between power traces and the Hamming Weight of the each byte of the key.

Using the results of this CPA, an ASCA can be created by using the function $\text{instance}$\$\text{setNotEq}()$ of CryptoSAT for each hypothesis key rejected by the CPA.

As we saw in Chapter 9, a CPA attack is not a good solution to extract information leaked by the cryptographic device since it is too sensitive to the noise and it is not able to extract any information when there is not enough power traces available. In addition, the CPA can be executed only on the first or last two rounds because the probability of errors increases with successive XORs between the computed round key and the data. Moreover, when there is an error in the first computed round key, this error will spread across all the bytes of the key after a key schedule.
Despite its poor performances when the power traces are noisy or when there is not enough power traces available, the CPA is very easy to implement and can be efficient when provided with enough power traces but this study showed that this SCA is not a good ASCA candidate to extract leakage information from the power traces of a cryptographic device compared to the Bayesian Template Attack [46].

By studying the result of the CPA and the one of the ASCA, we proposed some properties needed for a SCA to be a good candidate in an ASCA. First of all, a candidate needs to be able to produce only valid information in order to avoid the introduction of impossibilities in the SAT problem. This can be achieved in two ways: either the candidate produces no wrong information at all, or an additional step is performed in order to distinguish the wrong information and to remove it from the results of the SCA.

The second property for a good candidate is the ability to gather information about the entirety of the encryption process and not only the information from a specific instant such for the CPA where only the first two rounds are attacked. Another property could be a plus for a candidate if it provides the possibility to establish a tradeoff between the amount of information resulting from the SCA and the amount of time for the SAT solving. Indeed, when the attacker does not have access to enough information or if the power traces are too noisy, the SCA should still produce enough information in order to solve the problem with an optionally longer computation.

Even if the CPA fails to satisfy these properties, we propose a way to implement a better ASCA using a CPA as SCA. This proposition would need a new kind of SAT solver in order to be able to specify the probability for a certain hypothesis to be the correct one. Indeed, the results of the CPA is an ordered list of hypothesis. If the SAT solver is able to use these information as heuristic, it could be resistant to errors introduced because of the noise or the lack of power traces.

In this work, only the outcome of the attack was evaluated but an interesting additional work could focus on the differences between the key enumeration algorithm and SAT solving. Indeed, the execution time of a key enumeration algorithm grows when the quality of the results of a SCA lowers. In this case an ASCA using these results could be quicker.

A further work could also find interesting to study the results of an ASCA with CPA targeting the last two rounds instead of the first ones.

An additional goal for this work was to test the extension of CryptoSAT with another encryption algorithm such as Present. As we saw in the Chapter 7, to extend CryptoSAT with a new cryptographic algorithm, a user needs its C++ implementation where the variables are of type U*. This type overloads the C++ operators and produces the corresponding propositional clauses. CryptoSAT also defines the $Sbox$ object which aims to define the S-boxes and lookup tables in a standardized way. Once CryptoSAT extended as needed, the user can modify the content of the SAT problem in order to add some additional information and use an user-defined SAT solver to resolve the problem. Any SAT solver can be used provided that it respects the conditions defined during the SAT competitions [51, 50]. During this work, we found out that the use of CryptoSAT is pretty straightforward and has good performances.
After the request from Frédéric Lafitte, the creator of CryptoSAT, we provided him with our CryptoSAT implementation of Present in order to make it accessible to the scientific community. The first step to validate an implementation is to test it using the official test vectors. The file `present.runit.R` (described in Section 10.4) defines four tests based on the four triplets plaintext, key and ciphertext defined by the authors of Present [9]. These tests can be executed by any user of CryptoSAT and will raise an error if the results are not the expected ones.

Once the implementation of Present is proven valid, it needs to be documented in order to define some basic information (a short description, its name, parameters, methods and author.) This description is then placed in the file `Present.Rd` described in Section 10.4.

These two files were sent to Frédéric Lafitte for inclusion in a future release.

For the writing of this master thesis, some of the graphs used as illustrations are from `tikz For Cryptographers` [30] but for Present, some graphs were not available so we created them. These are presented in Figures 2.2a, 2.2b, 2.2c and 2.4. These graphs and the corresponding code source was submitted to `tikzForCryptographers` to be accessible by the scientific community and will be available from the next release.
Bibliography


Glossary

**ASCA** Algebraic Side-channel Attack. v, 2, 3, 46, 51, 62, 63, 67–70, 73, 74

**CNF** Conjunctive Normal Form. 38, 39, 51

**CPA** Correlation Power Analysis. 2, 3, 55, 60–63, 65–70, 73, 74

**DPA** Differential Power Analysis. 2, 30–33, 46, 55, 57

**SAT** Satisfiability Problem. v, 36–42, 62, 63, 65, 67–70, 73, 74

**SCA** Side-channel Attack. v, 2, 3, 23, 25–27, 34–36, 40, 51, 59, 65, 67–70, 73, 74

**SKEA** Score-based Key Enumeration Algorithm. 34, 65, 70

**SPA** Simple Power Analysis. 29, 30

**TA** Template Attacks. 33, 34
Part III

Appendix
10.1 Implementation of Present

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include "present.h"

unsigned int blockSizeInByte = 8;
unsigned char sBox[16] = {0xC, 0x5, 0x6, 0xB, 0x9, 0x0, 0xA, 0xD, 0x3, 0xE, 0xF,
                          0x8, 0x4, 0x7, 0x1, 0x2};

unsigned char key[10];
unsigned char saveKey[10];
unsigned char roundCounter = 1;

void shift61BitsLeft();

size_t getKeySizeInByte(){
    return sizeof(key);
}

void initKey(unsigned char keyValue[]){
    unsigned char i;
    roundCounter = 0;
    for(i = 0; i < getKeySizeInByte(); ++i){
        key[i] = keyValue[i];
        saveKey[i] = keyValue[i];
    }
}

unsigned char getKeyByte(unsigned int index){
    if(index >= getKeySizeInByte()){
        fprintf(stderr, "getKeyByte: ArrayIndexOutOfBound: %u", index);
        exit(-1);
    }
    return key[index];
}

unsigned char sBoxOneByte(unsigned char toSubstitute) {
    unsigned char result = sBox[(int) toSubstitute >> 4]; // Compute the value of the 4 leftmost bits.
    result <<= 4;
    result += sBox[(int) toSubstitute & 0x0f]; // Compute the value of the 4 rightmost bits.
    return result;
}

unsigned long long sBoxOneBloc(unsigned long long block) {
```
```
unsigned i;
unsigned long long result = 0, value;
for (i = 0; i < blockSizeInByte; ++i){
    value = sBoxOneByte((unsigned char)(block & 0xff));
    result += value << i * 8;
    block >>= 8;
}
return result;
}
unsigned long long permuteBloc(unsigned long long blockMessage) {
unsigned char bitIndex, shortIndex, index = 0;
unsigned short mask = 1;
unsigned long long result = 0;
unsigned long long value = 0;
unsigned short block[4];
for (shortIndex = 0; shortIndex < 4; ++shortIndex){
    block[shortIndex] = (unsigned short)blockMessage;
    blockMessage >>= 16;
}
for (bitIndex = 0; bitIndex < 16; ++bitIndex){
    for (shortIndex = 0; shortIndex < 4; ++shortIndex){
        value = block[shortIndex] & mask;
        result += value << (index - bitIndex);
        ++index;
    }
    mask <<= 1;
}
return result;
}
unsigned long long addRoundKey(unsigned long long block){
return block ^ getRoundKey();
}
void shift61BitsLeft() {
unsigned char saveKey[getKeySizeInByte()];
unsigned char i;
for (i = 0; i < getKeySizeInByte(); ++i){
    saveKey[i] = key[i];
    key[i] = 0;
}
for (i = 0; i < getKeySizeInByte(); ++i){
    unsigned char newIndexLow = (unsigned char)((i + 2) % 10);
    unsigned char newIndexHigh = (unsigned char)((i + 3) % 10);
    key[newIndexHigh] += (saveKey[i] & 0xf) << 5;
    key[newIndexLow] += (saveKey[i] & 0xf8) >> 3;
}
}
void sBoxFourBitsLeft() {
unsigned char byte = getKeyByte(0);
byte &= 0xf0;
key[0] &= 0xf;
byte = (unsigned char)(sBoxOneByte(byte) & 0xf0);
key[0] ^= byte;
}
void xorKeyWithRoundCounter() {
    key[7] ^= roundCounter >> 1;
    key[8] ^= roundCounter << 7;
}
unsigned long long getRoundKey() {
    unsigned i = 0;
unsigned long long roundKey = 0;
for (i = 0; i < blockSizeInByte - 1; i++) {
    roundKey += key[i];
    roundKey <<= 8;
    roundKey += key[i];
}
return roundKey;

void updateKey80bits() {
    shift61BitsLeft();
    sBoxFourBitsLeft();
    xorKeyWithRoundCounter();
    updateRoundCounter();
}

void updateRoundCounter() {
    ++roundCounter;
}

unsigned long long encryptOneRound(unsigned long long block) {
    block = addRoundKey(block);
    block = sBoxOneBloc(block);
    block = permuteBloc(block);
    return block;
}

unsigned long long encryptSeveralRound(unsigned long long block, unsigned char rounds) {
    unsigned char i;
    for (i = 0; i < rounds; i++) {
        block = encryptOneRound(block);
    }
    return block;
}

int main() {
    return run_all_tests();
}

10.2 Present Optimized Version [59]
* "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
* LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
* FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
* COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
* INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
* BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
* CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
* ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*/

/*
* The core function of PRESENT block cipher.
* 1kB look-up tables are used for performance tuning.
* @author Bo Zhu, http://cis.sjtu.edu.cn/index.php/Bo_Zhu
* @author Zheng Gong, DIES Group, University of Twente
* @date July 21, 2009
*/

#ifndef __PRESENT_H__
#define __PRESENT_H__

// comment this out if this is used on PC
#define __UINT_T__

#ifndef __UINT_T__
#define __UINT_T__

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;
#endif /* __UINT_T__ */

#define present (plain, key, cipher) present_rounds ((plain), (key), 31, (cipher))

// actually is (sbox[] << 4)
static const uint8_t sbox [16] = {
  0xC0, 0x50, 0xB0, 0x90, 0x00, 0x5A0, 0x0D0, 0x330, 0x0E0, 0x0F0, 0x080, 0x40, 0
};

// look-up tables for speeding up permutation layer
static const uint8_t sbox_pmt_3 [256] = {
  0x50, 0x11, 0x14, 0x45, 0x41, 0x00, 0x44, 0x51, 0x05, 0x54, 0x55, 0x4A, 0x1A, 0
};

// comment this out if this is used on PC
#define __UINT_T__

#ifndef __UINT_T__
#define __UINT_T__

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;
#endif /* __UINT_T__ */

#define present (plain, key, cipher) present_rounds ((plain), (key), 31, (cipher))

// actually is (sbox[] << 4)
static const uint8_t sbox [16] = {
  0xC0, 0x50, 0xB0, 0x90, 0x00, 0x5A0, 0x0D0, 0x330, 0x0E0, 0x0F0, 0x080, 0x40, 0
};

// look-up tables for speeding up permutation layer
static const uint8_t sbox_pmt_3 [256] = {
  0x50, 0x11, 0x14, 0x45, 0x41, 0x00, 0x44, 0x51, 0x05, 0x54, 0x55, 0x4A, 0x1A, 0
};
static const uint8_t sbox_pmt_2[256] = {
    0x3C, 0x6C, 0x2D, 0x79, 0x78, 0x28, 0x39, 0x7C, 0x69, 0x3D, 0x7D, 0x38, 0x2C, 0x6D, 0x68, 0x29,
    0x9C, 0xCC, 0x8D, 0xD9, 0xD8, 0x88, 0x99, 0xC9, 0xD0, 0xDD, 0x98, 0x8C, 0xCD, 0xC8, 0x89,
    0x1E, 0x4E, 0x0F, 0x5B, 0x5A, 0x0A, 0x1B, 0x5E, 0x4B, 0x1F, 0x5F, 0x1A, 0x0E, 0x4F, 0x4A, 0x0B,
    0xB6, 0xE6, 0xA7, 0xF3, 0xF2, 0xA2, 0xB3, 0xF6, 0xE3, 0xB7, 0xF7, 0xB2, 0xA6, 0xEE, 0xE2, 0xA3,
    0xB4, 0xE4, 0xA5, 0xF1, 0xF0, 0xA0, 0xB1, 0xF4, 0xE1, 0xB5, 0xF5, 0xB0, 0xA4, 0xEE, 0xE0, 0xA1,
    0x14, 0x44, 0x05, 0x51, 0x50, 0x00, 0x11, 0x54, 0x41, 0x15, 0x55, 0x10, 0x04, 0x45, 0x40, 0x03,
    0x36, 0x66, 0x27, 0x33, 0x63, 0x76, 0x36, 0x72, 0x33, 0x76, 0x63, 0x37, 0x77, 0x32, 0x26, 0x77,
    0x67, 0x23, 0x92, 0xEC, 0xAD, 0xF9, 0xF8, 0xA8, 0xB9, 0xEC, 0xA9, 0xF9, 0xB8, 0xAC, 0xED, 0xED, 0xA9,
    0x96, 0xC6, 0x87, 0xD3, 0xD2, 0x80, 0x95, 0xD6, 0xC3, 0x97, 0xD7, 0x92, 0x86, 0xCE, 0xC2, 0x83,
    0x3E, 0x6E, 0x2F, 0x7B, 0x7A, 0x2A, 0x3B, 0x7E, 0x6B, 0x3F, 0x7F, 0x3A, 0x2E, 0x6F, 0x6A, 0x2B,
    0x9E, 0xBE, 0xAF, 0xFB, 0xEA, 0x99, 0xCE, 0x8F, 0xDB, 0xDA, 0x8A, 0xC9, 0xD6, 0xC3, 0x8B, 0xDB,
    0x9D, 0xC4, 0x85, 0xD1, 0xD0, 0x80, 0x91, 0xD4, 0xC1, 0x95, 0xD5, 0x90, 0x84, 0xC5, 0xC0, 0x81,
    0x16, 0x46, 0x07, 0x53, 0x52, 0x02, 0x13, 0x56, 0x43, 0x17, 0x57, 0x12, 0x06, 0x47, 0x42, 0x03,
};

static const uint8_t sbox_pmt_1[256] = {
    0x0F, 0x1B, 0x4B, 0x5E, 0x1E, 0x0A, 0x4E, 0x1F, 0x5A, 0x4F, 0x5F, 0x0E, 0x0B, 0x5B, 0x1A, 0x4A,
    0x27, 0x33, 0x63, 0x76, 0x36, 0x22, 0x66, 0x37, 0x72, 0x67, 0x77, 0x26, 0x23, 0x73, 0x32, 0x62,
    0x87, 0x93, 0xC3, 0xD6, 0x96, 0x82, 0xC6, 0x97, 0xD2, 0xC7, 0xD7, 0x86, 0x83, 0xD3, 0x92, 0xC2,
    0x92, 0xEC, 0xAD, 0xF9, 0xF8, 0xA8, 0xB9, 0xEC, 0xA9, 0xF9, 0xB8, 0xAC, 0xED, 0xED, 0xA9,
    0xB6, 0xE6, 0xA7, 0xF3, 0xF2, 0xA2, 0xB3, 0xF6, 0xE3, 0xB7, 0xF7, 0xB2, 0xA6, 0xEE, 0xE2, 0xA3,
    0xB4, 0xE4, 0xA5, 0xF1, 0xF0, 0xA0, 0xB1, 0xF4, 0xE1, 0xB5, 0xF5, 0xB0, 0xA4, 0xEE, 0xE0, 0xA1,
    0x14, 0x44, 0x05, 0x51, 0x50, 0x00, 0x11, 0x54, 0x41, 0x15, 0x55, 0x10, 0x04, 0x45, 0x40, 0x03,
    0x36, 0x66, 0x27, 0x33, 0x63, 0x76, 0x36, 0x72, 0x33, 0x76, 0x63, 0x37, 0x77, 0x32, 0x26, 0x77,
    0x67, 0x23, 0x92, 0xEC, 0xAD, 0xF9, 0xF8, 0xA8, 0xB9, 0xEC, 0xA9, 0xF9, 0xB8, 0xAC, 0xED, 0xED, 0xA9,
    0x96, 0xC6, 0x87, 0xD3, 0xD2, 0x80, 0x95, 0xD6, 0xC3, 0x97, 0xD7, 0x92, 0x86, 0xCE, 0xC2, 0x83,
    0x3E, 0x6E, 0x2F, 0x7B, 0x7A, 0x2A, 0x3B, 0x7E, 0x6B, 0x3F, 0x7F, 0x3A, 0x2E, 0x6F, 0x6A, 0x2B,
    0x9E, 0xBE, 0xAF, 0xFB, 0xEA, 0x99, 0xCE, 0x8F, 0xDB, 0xDA, 0x8A, 0xC9, 0xD6, 0xC3, 0x8B, 0xDB,
    0x9D, 0xC4, 0x85, 0xD1, 0xD0, 0x80, 0x91, 0xD4, 0xC1, 0x95, 0xD5, 0x90, 0x84, 0xC5, 0xC0, 0x81,
    0x16, 0x46, 0x07, 0x53, 0x52, 0x02, 0x13, 0x56, 0x43, 0x17, 0x57, 0x12, 0x06, 0x47, 0x42, 0x03,
};

static const uint8_t sbox_pmt_0[256] = {
    0xD0, 0x91, 0xC5, 0xC1, 0xB0, 0xC4, 0xD1, 0xB5, 0xD4, 0xD5, 0xC0, 0xB0, 0x90, 0xB4, 0x95, 0x81,
    0x95, 0x81, 0xD4, 0xB0, 0xC4, 0x91, 0xD5, 0x95, 0x80, 0xD1, 0xB5, 0xC0, 0xB0, 0x90, 0xD0, 0x91,
    0x94, 0xC4, 0x85, 0xD1, 0xD0, 0x80, 0x91, 0xD4, 0xC1, 0x95, 0xD5, 0x90, 0x84, 0xC5, 0xC0, 0x81,
    0x16, 0x46, 0x07, 0x53, 0x52, 0x02, 0x13, 0x56, 0x43, 0x17, 0x57, 0x12, 0x06, 0x47, 0x42, 0x03,
};
114 0xAF, 0xBE, 0xEE, 0xBE, 0xAA, 0xEE, 0xBF, 0xFA, 0xFF, 0xAE, 0x8, 0xFB, 0xA, 0xEA,
115 0xFD, 0x19, 0x49, 0x5C, 0x1C, 0x8, 0x4C, 0x1D, 0x85, 0x4D, 0x15, 0x8C, 0x19, 0
116 0xBE, 0xA3, 0x86, 0x16, 0x02, 0x46, 0x17, 0x02, 0x47, 0x15, 0x06, 0x13, 0
117 0x3, 0x12, 0x42,
118 0x7, 0x31, 0x61, 0x74, 0x34, 0x20, 0x64, 0x35, 0x70, 0x65, 0x75, 0x24, 0x21, 0
119 0x71, 0x30, 0x60,
120 0x85, 0x91, 0xC1, 0xD4, 0x94, 0x80, 0xC4, 0x95, 0xD0, 0xC5, 0xD5, 0x84, 0xB1, 0
121 0xD6, 0x86, 0x92,
122 0xC9, 0xCC, 0xD8, 0x9D, 0x8D, 0x88, 0x99, 0xCD, 0x8C, 0x98, 0
123 0xE1, 0xE4, 0xF0, 0xB5, 0xA5, 0xA0, 0xB1, 0xE5, 0xB4, 0xF1, 0xF5, 0xA1, 0xE0,
124 0xC8, 0xDC, 0x8C, 0x98, 0
125 0x4B, 0x4E, 0x5A, 0x1F, 0x0F, 0x0A, 0x1B, 0x4F, 0x1E, 0x5B, 0x5F, 0x0B, 0x4A,
126 0xED, 0xF8, 0xBD, 0xAD, 0xA8, 0xB9, 0xED, 0xBC, 0xF9, 0xFD, 0xA9, 0xEC,
127 0x49, 0x4C, 0x58, 0x1D, 0x8D, 0x8B, 0x89, 0xED, 0xBC, 0xF9, 0xFD, 0xA9, 0xEB,
128 0x85, 0x91, 0xC1, 0xD4, 0x94, 0x80, 0xC4, 0x95, 0xD0, 0xC5, 0xD5, 0x84, 0xB1, 0
129 0xF4, 0xA4, 0x80,
130 0x75, 0x2E, 0x3A,
131 0x69, 0x6C, 0x78, 0x3D, 0x2D, 0x38,
132 0xCB, 0xCE, 0xDA, 0x9F, 0x8F, 0x8A, 0x9B, 0xCF, 0x9E, 0xDB, 0xF8, 0xBE,
133 0xE3, 0xE6, 0xF2, 0xB7, 0xA7, 0xA2, 0xB3, 0xE7, 0xB6, 0xF3, 0xA8, 0xBE,
134 0xEB, 0xEE, 0xFA, 0xFF, 0xFA, 0xFB, 0x8A, 0xEA, 0
135 0x4D, 0x1C, 0x59, 0x5D, 0x09, 0x48, 0x5C, 0x0C, 0x18,
136 0x41, 0x44, 0x52, 0x17, 0x07, 0x02, 0x13, 0x47, 0x16, 0x53, 0x57, 0x03, 0x42,
137 0x61, 0x64, 0x70, 0x35, 0x25, 0x28, 0x39, 0x63, 0x75, 0x21, 0x60, 0x74,
138 0x43, 0x46, 0x58, 0x11, 0x01, 0x0B, 0x19, 0x4D, 0x1C, 0x59, 0x5D, 0x09, 0x48,
139 0x74, 0x24, 0x30,
140 0
141 0
142 0
143 0
144 0
145 0
146 0
147 0
148 0
149 0
150 0
151 0
152 0
153 0
154 0
155 0
156 0
157 0
158 0
159 0

// update key
round_key[3] = key[0] << 5 | key[1] >> 3;
round_key[2] = key[9] << 5 | key[0] >> 3;
round_key[0] = key[7] << 5 | key[8] >> 3;

round_key[0] = (round_key[0] & 0x0F) | sbox[round_key[0] >> 4];
round_key[7] ^= round_counter >> 1;
round_key[8] ^= round_counter << 7;

// substitution and permutation
cipher[0] =
    (sbox_pmt_3[state[0]] & 0xC0) |
    (sbox_pmt_2[state[1]] & 0x30) |
    (sbox_pmt_1[state[2]] & 0x0C) |
    (sbox_pmt_0[state[3]] & 0x03);
cipher[1] =
    (sbox_pmt_3[state[4]] & 0xC0) |
    (sbox_pmt_2[state[5]] & 0x30) |
    (sbox_pmt_1[state[6]] & 0x0C) |
    (sbox_pmt_0[state[7]] & 0x03);
cipher[2] =
    (sbox_pmt_0[state[0]] & 0xC0) |
    (sbox_pmt_3[state[1]] & 0x30) |
    (sbox_pmt_2[state[2]] & 0x0C) |
    (sbox_pmt_1[state[3]] & 0x03);
cipher[3] =
    (sbox_pmt_0[state[4]] & 0xC0) |
    (sbox_pmt_3[state[5]] & 0x30) |
    (sbox_pmt_2[state[6]] & 0x0C) |
    (sbox_pmt_1[state[7]] & 0x03);
cipher[4] =
    (sbox_pmt_1[state[0]] & 0xC0) |
    (sbox_pmt_0[state[1]] & 0x30) |
    (sbox_pmt_3[state[2]] & 0x0C) |
    (sbox_pmt_2[state[3]] & 0x03);
cipher[5] =
    (sbox_pmt_1[state[4]] & 0xC0) |
    (sbox_pmt_0[state[5]] & 0x30) |
    (sbox_pmt_3[state[6]] & 0x0C) |
    (sbox_pmt_2[state[7]] & 0x03);
cipher[6] =
    (sbox_pmt_2[state[0]] & 0xC0) |
    (sbox_pmt_1[state[1]] & 0x30) |
    (sbox_pmt_0[state[2]] & 0x0C) |
    (sbox_pmt_3[state[3]] & 0x03);
cipher[7] =
    (sbox_pmt_2[state[4]] & 0xC0) |
    (sbox_pmt_1[state[5]] & 0x30) |
    (sbox_pmt_0[state[6]] & 0x0C) |
    (sbox_pmt_3[state[7]] & 0x03);

for (round_counter = 2; round_counter <= rounds; round_counter++) {
    state[0] = cipher[0] ^ round_key[0];
    state[1] = cipher[1] ^ round_key[1];
cipher[0] =
(sbox_pmt_3[state[0]] & 0xC0) |
(sbox_pmt_2[state[1]] & 0x30) |
(sbox_pmt_1[state[2]] & 0x0C) |
(sbox_pmt_0[state[3]] & 0x03);
cipher[1] =
(sbox_pmt_3[state[4]] & 0xC0) |
(sbox_pmt_2[state[5]] & 0x30) |
(sbox_pmt_1[state[6]] & 0x0C) |
(sbox_pmt_0[state[7]] & 0x03);
cipher[2] =
(sbox_pmt_0[state[0]] & 0xC0) |
(sbox_pmt_3[state[1]] & 0x30) |
(sbox_pmt_2[state[2]] & 0x0C) |
(sbox_pmt_1[state[3]] & 0x03);
cipher[3] =
(sbox_pmt_0[state[4]] & 0xC0) |
(sbox_pmt_3[state[5]] & 0x30) |
(sbox_pmt_2[state[6]] & 0x0C) |
(sbox_pmt_1[state[7]] & 0x03);
cipher[4] =
(sbox_pmt_1[state[0]] & 0xC0) |
(sbox_pmt_0[state[1]] & 0x30) |
(sbox_pmt_3[state[2]] & 0x0C) |
(sbox_pmt_2[state[3]] & 0x03);
cipher[5] =
(sbox_pmt_1[state[4]] & 0xC0) |
(sbox_pmt_0[state[5]] & 0x30) |
(sbox_pmt_3[state[6]] & 0x0C) |
(sbox_pmt_2[state[7]] & 0x03);
cipher[6] =
(sbox_pmt_2[state[0]] & 0xC0) |
(sbox_pmt_1[state[1]] & 0x30) |
(sbox_pmt_0[state[2]] & 0x0C) |
(sbox_pmt_3[state[3]] & 0x03);
cipher[7] =
(sbox_pmt_2[state[4]] & 0xC0) |
(sbox_pmt_1[state[5]] & 0x30) |
(sbox_pmt_0[state[6]] & 0x0C) |
(sbox_pmt_3[state[7]] & 0x03);
round_key[5] ^= round_counter << 2; // do this first, which may be faster
// use state[] for temporary storage
state[2] = round_key[9];
state[1] = round_key[8];
state[0] = round_key[7];
round_key[3] = round_key[0] << 5 | round_key[1] >> 3;
round_key[0] = state[0] << 5 | state[1] >> 3;
round_key[0] = (round_key[0] & 0x0F) | sbox[round_key[0] >> 4];
10.3 Computation of the pairs XOR Distribution table of Present

```python
if __name__ == "__main__":
    sbox = [0xC, 0x5, 0x6, 0xB, 0x9, 0x0, 0xA, 0xD, 0x3, 0xE, 0xF, 0x8, 0x4, 0x7, 0x1, 0x2]
    frequencies = []
    for i in range(0, 16):
        frequencies.append([0] * 16)
    for i in range(0, 16):
        for j in range(0, 16):
            inputXOR = i ^ j
            outputXOR = sbox[i] ^ sbox[j]
            frequencies[inputXOR][outputXOR] +=1
    print("\hline")
    print(" Input & \multicolumn{16}{| c |}{ Output XOR}\\")
    print(" XOR & ", end="")
    for i in range(0, 15):
        print("\ multicolumn {1}{ r}{\$", format(i, 'x').capitalize () , " _x \$", sep="", end="")
    print(\$", format(15 , 'x'), " _x \$", sep="", end="")
    print(" \hline ")
    for line in range(0, 16):
        print("\$", format(line , 'x').capitalize () , " & ", sep="", end="")
        for column in range(0, 15):
            print(str(frequencies[line][column]) , " & ", sep="", end="")
        print(str(frequencies[line][15]), " " , sep="")
    print("\hline")
```

10.4 CryptoSAT code

Extension of the Target

```python
setConstructorS3("Present", function () {
    extend(Target("Present"), "Present", .parameters = data.frame( rounds = 31,
        sbox = paste("C", "5", "6", "B", "9", "0", "A", "D", "3", "E", "F", "8", "4", "7", "1", "2" ),
```

sbox_pmt_3 = paste(
  "F0", "B1", "B4", "E5", "E1", "A0", "E4", "F1", "A5", "F4", "F5", "E0", "B0",
  "97", "B3", "86", "50", "11", "14", "45", "41", "00", "44", "51", "05", "54", "55", "40", "10",
  "95", "B1", "84", "70", "31", "34", "66", "61", "20", "64", "71", "25", "74", "75", "60", "30",
  "17", "03", "06", "58", "19", "1C", "4D", "49", "08", "4C", "59", "0B", "5C", "5D", "48", "18",
  "1D", "09", "0C"
),

setMethodS3("compileBoxes", "Present", function(this, compiler, bin =NA, ...) {
this@hasSboxes(names=c("sbox","sbox_pmt_0","sbox_pmt_1","sbox_pmt_2","sbox_pmt_3"),
    sizes=c("4x4","8x8","8x8","8x8","8x8"),
    iowl=c("8x8","8x8","8x8","8x8","8x8"),
    compiler="/"/git/Memoire/cryptoSAT/sboxes/sbox.o")

getMethodS3("save", "Present", function(this, filename, ...) {
  R <- this$getParameter("rounds")
  f <- this$init(filename
  sbox <- Sbox(values=this$getParameter("sbox"), n=4, m=4, iwl=8, owl=8, name="sbox")
  sbox_pmt_0 <- Sbox(values=this$getParameter("sbox_pmt_0"), n=8, m=8, iwl=8, owl=8, name="sbox_pmt_0")
  sbox_pmt_1 <- Sbox(values=this$getParameter("sbox_pmt_1"), n=8, m=8, iwl=8, owl=8, name="sbox_pmt_1")
  sbox_pmt_2 <- Sbox(values=this$getParameter("sbox_pmt_2"), n=8, m=8, iwl=8, owl=8, name="sbox_pmt_2")
  sbox_pmt_3 <- Sbox(values=this$getParameter("sbox_pmt_3"), n=8, m=8, iwl=8, owl=8, name="sbox_pmt_3")
  this$.write(f, paste(sbox$LUT.cpp(), sbox$SIG.cpp(), sbox$FUN.cpp(), sep="\n"))

  this$.write(f, paste(sbox_pmt_0$LUT.cpp(), sbox_pmt_0$SIG.cpp(), sbox_pmt_0$FUN.cpp(), sep="\n"))
  this$.write(f, paste(sbox_pmt_1$LUT.cpp(), sbox_pmt_1$SIG.cpp(), sbox_pmt_1$FUN.cpp(), sep="\n"))
  this$.write(f, paste(sbox_pmt_2$LUT.cpp(), sbox_pmt_2$SIG.cpp(), sbox_pmt_2$FUN.cpp(), sep="\n"))
  this$.write(f, paste(sbox_pmt_3$LUT.cpp(), sbox_pmt_3$SIG.cpp(), sbox_pmt_3$FUN.cpp(), sep="\n"))

  this$.write(f, "int main()" )
  this$.write(f, "{"
  this$.write(f, "// full-round should be 31, i.e. rounds = 31"
  this$.write(f, "// plain and cipher can overlap, so do key and cipher"
  this$.write(f, "int round_counter = 1;"
  this$.write(f, paste(" int rounds = ", R, ",", sep=""))
  this$.write(f, "}
  this$.write(f, " ofstream vars("variables.txt", ios::out);"
  this$.write(f, " ofstream vals("values.txt", ios::out);"
  this$.write(f, " cipher[0].print(vars,vals,""X0",");
  this$.write(f, " cipher[1].print(vars,vals,""X1",");
  this$.write(f, " cipher[2].print(vars,vals,""X2",");
  this$.write(f, " cipher[3].print(vars,vals,""X3",");
  this$.write(f, " cipher[4].print(vars,vals,""X4",");
  this$.write(f, " cipher[5].print(vars,vals,""X5",");
  this$.write(f, " cipher[6].print(vars,vals,""X6",");
  this$.write(f, " cipher[7].print(vars,vals,""X7",");
  this$.write(f, " round_key[0].print(vars,vals,""K0",");
  this$.write(f, " round_key[1].print(vars,vals,""K1",");
  this$.write(f, " round_key[2].print(vars,vals,""K2",");
  this$.write(f, " round_key[3].print(vars,vals,""K3",");
  this$.write(f, " round_key[4].print(vars,vals,""K4",");
  this$.write(f, " round_key[5].print(vars,vals,""K5",");
  this$.write(f, " round_key[6].print(vars,vals,""K6",");
  this$.write(f, " round_key[7].print(vars,vals,""K7",");
  this$.write(f, " round_key[8].print(vars,vals,""K8",");
  this$.write(f, " round_key[9].print(vars,vals,""K9",");
  this$.write(f, " for (round_counter = 1; round_counter <= rounds; round_counter ++)
    {"
  this$.write(f, " state[0] = cipher[0] - round_key[0];"
  this$.write(f, " state[1] = cipher[1] - round_key[1];")

which may be faster

})

state[0] = round_key[7];
state[1] = round_key[8];
state[2] = round_key[9];
state[3] = round_key[10];
state[5] = round_key[12];
state[6] = round_key[13];

state[8] = cipher[8] ^ round_key[8];

97
Testing the Target

```r
# Test 1:
# input:
# Key: 00 00 00 00 00 00 00 00 00 00
# Plaintext: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
# output:
# Ciphertext: 55 79 C1 38 7B 22 84 45

present.test.1 <- function () {
  expected <- c("55", "79", "C1", "38", "7B", "22", "84", "45")
  target <- Present()
  instance <- target$generateInstance(binaries="sboxes/sbox.o")
  for (i in 0:7) {
    instance$setEqual(paste("X",i,"[0]",sep=""), "00")
  }
  for (i in 0:9) {
    instance$setEqual(paste("K",i,"[0]",sep=""), "00")
  }
  solution <- instance$solveWith(solver)
  for (i in 0:7) {
    checkEquals(toupper(solution$getValueOf(paste("X",i,"[32]",sep=""))), expected[i+1])
  }
  return 0;
}
```
# Test 2:
# input:
# Key : FF FF FF FF FF FF FF FF FF FF
# Plaintext : 00 00 00 00 00 00 00 00
# output:
# Ciphertext : E7 2C 46 C0 F5 94 50 49

```r
present_test.2 <- function () {
  expected = c("E7", "2C", "46", "C0", "F5", "94", "50", "49")
  target <- Present()
  instance <- target$generateInstance(binaries="sboxes/sbox.o")
  for( i in 0:7 ) {
    instance$setEqual(paste("X",i,"[0]",sep=""), "00")
  }
  for( i in 0:9 ) {
    instance$setEqual(paste("K",i,"[0]",sep=""), "FF")
  }
  solution <- instance$solveWith(solver)
  for ( i in 0:7 ) {
    checkEquals(toupper(solution$getValueOf(paste("X",i,"[32]",sep=""))), expected[i+1])
  }
}
```

# Test 3:
# input:
# Key : 00 00 00 00 00 00 00 00 00 00
# Plaintext : FF FF FF FF FF FF FF FF
# output:
# Ciphertext : A1 12 FF C7 2F 68 41 7B

```r
present_test.3 <- function () {
  expected = c("A1", "12", "FF", "C7", "2F", "68", "41", "7B")
  target <- Present()
  instance <- target$generateInstance(binaries="sboxes/sbox.o")
  for( i in 0:7 ) {
    instance$setEqual(paste("X",i,"[0]",sep=""), "FF")
  }
  for( i in 0:9 ) {
    instance$setEqual(paste("K",i,"[0]",sep=""), "00")
  }
  solution <- instance$solveWith(solver)
  for ( i in 0:7 ) {
    checkEquals(toupper(solution$getValueOf(paste("X",i,"[32]",sep=""))), expected[i+1])
  }
}
```

# Test 4:
# input:
# Key : FF FF FF FF FF FF FF FF FF FF
# Plaintext : FF FF FF FF FF FF FF FF
# output:
# Ciphertext : 33 33 DC D3 21 32 10 D2

```r
present_test.4 <- function () {
  expected = c("33", "33", "DC", "D3", "21", "32", "10", "D2")
  target <- Present()
  instance <- target$generateInstance(binaries="sboxes/sbox.o")
  for( i in 0:7 ) {
    instance$setEqual(paste("X",i,"[0]",sep=""), "FF")
  }
  for( i in 0:9 ) {
    instance$setEqual(paste("K",i,"[0]",sep=""), "FF")
  }
  solution <- instance$solveWith(solver)
  for ( i in 0:7 ) {
    checkEquals(toupper(solution$getValueOf(paste("X",i,"[32]",sep=""))), expected[i+1])
  }
}
```
Documenting the Target

**encoding** {UTF-8}

**name** {Present}

**alias** {Present}

**alias** {save.Present}

**title** {Target algorithm: PRESENT block cipher.}

**description**{

Present (described in "PRESENT: An Ultra-Lightweight Block Cipher" by A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, and C. Vikkelsoe) is an ultra-lightweight block cipher for which both the security and hardware efficiency have been equally optimized during the design of this cipher. The test vector used for Present was defined in the article presenting this block cipher. This implementation is based on the optimized version of PRESENT proposed by Bo Zhu and Zheng Gong accessible at the address: http://cis.sjtu.edu.cn/index.php/Software_Implementation_of_Block_Cipher_PRESENT_for_8-Bit_Platforms.

The returned value is an Present object that inherits from \code{Target}.

Parameters for this target include:

- \code{ rounds } \tab The number of rounds.
- \code{ sbox } \tab The S-box of Present.
- \code{ sbox_pmt_0, sbox_pmt_1, sbox_pmt_2, sbox_pmt_3 } \tab The lookup table used to perform the substitution and permutation in one step.

Checks that parameter values are in the appropriate range.

Gives access to the generated C++ file.

Florian Delporte

\seealso{ \code{\link{Target}}, \code{\link{\oo:Object}}}

\examples{
  target <- Present()
  target
}

\keyword{misc}
10.5 Present Arduino Version

Based on the code in Section 10.2

```c
/* The core function of PRESENT block cipher.
1kB look-up tables are used for performance tuning.
* @author Bo Zhu, http://cis.sjtu.edu.cn/index.php/Bo_Zhu
* @author Zheng Gong, DIES Group, University of Twente
* @date July 21, 2009
*/

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <limits.h>

#ifndef __PRESENT_H__
#define __PRESENT_H__

#ifndef __UINT_T__
#define __UINT_T__
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;
#endif /* __UINT_T__ */

char byteIn; // tmp byte read from serial

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <limits.h>
#endif /* __PRESENT_H__ */

void changeValue()
{
    unsigned char i = 0;
    for (i = 0; i < 8; ++i)
    {
        plainText[i] = (uint8_t) random(255);
    }
}

void printBlock()
{
    char i;
    Serial.print("0x");
    for (i = 7; i >= 0; --i)
    {
        if (saveKey[i] == 0)
        {
            Serial.print("00");
        }
        else
        {
            if (plainText[i] < 16)
            {
                Serial.print(plainText[i], HEX);
            }
            else
            {
                Serial.print(plainText[i] < 16) ? Serial.print("0") : Serial.print(plainText[i], HEX);
            }
        }
    }
    Serial.println();
}

void printKey()
{
    char tmp;
    Serial.print("0x");
    for (i = 9; i >= 0; --i)
    {
        if (saveKey[i] == 0)
        {
            Serial.print("00");
        }
        else
        {
            if (saveKey[i] < 16)
            {
                Serial.print("0") ;
            }
            else
            {
                Serial.print(saveKey[i], HEX);
            }
        }
    }
}
```
```cpp
void setup(){
  Serial.begin(9600); //setup communication
  randomSeed(1); //setup random numbers
  pinMode(pinTrigger, OUTPUT);
  digitalWrite(pinTrigger, LOW);
  initPlaintext();
}

void initPlaintext(){
  unsigned char i;
  for (i =0;i<8;++i){
    Plaintext[i]=0x00;
  }
}

void loop () {
  if (Serial.available() > 0){
    byteIn = Serial.read(); //read command byte
    Serial.println(byteIn);
    if (byteIn == 'c'){
      digitalWrite(pinTrigger, HIGH);
      present(plainText, saveKey, workingBlock);
      digitalWrite(pinTrigger, LOW);
    }else if (byteIn == 'v'){
      changeValue();
    }else if (byteIn == 'p'){
      printBlock();
    }else if (byteIn == 'k'){
      printKey();
    }
  }
}
```

### 10.6 Trace reader

Provided by Nikita Veshchikov and modified as needed.

```cpp
#ifndef __INF_TRACE_READER_CPP__
#define __INF_TRACE_READER_CPP__

#include <vector>
#include <string>
#include <fstream> // files
#include <iostream> // offset in a file
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>

//#define VERBOSE_TRACE // verbose load of a single trace
//#define VERBOSE_TRACES // verbose load of multiple traces

unsigned readTraceHeader (std::ifstream &inputFile){
  // for reading unknown parameters
  This function reads the header of a trace file and returns the number of points in the power trace
  the file descriptor (pointer) IS MODIFIED at the end:
  one would continue reading the file from the last position (after the header)
  */
  unsigned readTraceHeader(std::ifstream &inputFile){
    // for reading unknown parameters
```
unsigned tmpUns;
char tmpChar;
// parameters
unsigned pointsNbr, avgCount;
double xDispOrg, xIncr, xOrigin;
float xDispRange;
#endif VERBOSE_TRACE
printf("VERBOSE MODE ACTIVE\n");
printf("Reading file's header\n");
#endif
for(uint8_t i = 0; i<6; i++){
    inputFile.read((char *)&tmpUns, sizeof(unsigned));
    #ifdef VERBOSE_TRACE
    printf("%d ", tmpUns);
    #endif
}
#endif VERBOSE_TRACE
printf("\n");
#endif
inputFile.read((char *)&pointsNbr, sizeof(unsigned)); // points in a trace
inputFile.read((char *)&avgCount, sizeof(unsigned)); // number of averages
#endif VERBOSE_TRACE
printf("Points: %d\n", pointsNbr);
printf("AvgCount: %d\n", avgCount);
#endif
inputFile.read((char *)&xDispRange, sizeof(float));
inputFile.read((char *)&xDispOrg, sizeof(double));
inputFile.read((char *)&xIncr, sizeof(double));
inputFile.read((char *)&xOrigin, sizeof(double));
#endif VERBOSE_TRACE
printf("xDisp range: %e\n", xDispRange);
printf("xDisp org: %e\n", xDispOrg);
printf("xInc: %e\n", xIncr);
printf("xOrg: %e\n", xOrigin);
#endif
for(uint8_t i = 0; i<6; i++) { // unknown part of the header
    do {
        inputFile.read((char *)&tmpChar, sizeof(char));
        #ifdef VERBOSE_TRACE
        printf("0x%02X ", tmpChar);
        #endif
    } while (tmpChar != '\0');
    // printf( " ");
}
#endif VERBOSE_TRACE
printf("\n");
#endif
for(uint8_t i = 0; i<21; i++) { // scope and trace info (date, time, scopeID, channel)
    do {
        inputFile.read((char *)&tmpChar, sizeof(char));
        #ifdef VERBOSE_TRACE
        printf("%c", tmpChar);
        #endif
    } while (tmpChar != '\0');
    // printf( " ");
}
#endif VERBOSE_TRACE
printf("\n");
#endif
inputFile.read(&tmpChar, sizeof(char));
while(tmpChar != 0x0C){ // first special symbol
    #ifdef VERBOSE_TRACE
    printf("0x%02X", tmpChar);
    #endif
    inputFile.read(&tmpChar, sizeof(char));
}
    #ifdef VERBOSE_TRACE
    printf("0x%02X\n", tmpChar);
    #endif
    inputFile.read(&tmpChar, sizeof(char));
while(tmpChar != 0x01){ // second special symbol
    #ifdef VERBOSE_TRACE
    printf("0x%02X", tmpChar);
    #endif
    inputFile.read(&tmpChar, sizeof(char));
    #ifdef VERBOSE_TRACE
    printf("0x%02X\n", tmpChar);
    #endif
    inputFile.read(&tmpChar, sizeof(char));
    #ifdef VERBOSE_TRACE
    printf("0x%02X", tmpChar);
    #endif
    inputFile.read((char *)&tmpChar, sizeof(char)); // read zero
    #ifdef VERBOSE_TRACE
    printf(" %02X", tmpChar);
    #endif
    inputFile.read((char *)&tmpUns, sizeof(unsigned)); // read whatever this is
    #ifdef VERBOSE_TRACE
    printf("0x%02X", tmpUns);
    #endif
    return pointsNbr;
}

std::vector<float> readTraceFrom(const char* filename, unsigned offset = 0,
                                  unsigned pointsToRead = 0){
    float* traceBuffer;
    unsigned pointsNbr;
    std::ifstream inputFile;
    inputFile.open(filename, std::fstream::in | std::fstream::binary);
    if ( ! inputFile.is_open() ){
        perror("Error: could not open the trace file\n");
        exit(-1);
    }
    #ifdef VERBOSE_TRACE
    printf("Reading %s\n", filename_c_str());
    #endif
pointsNbr = readTraceHeader(inputFile);
if (offset >= pointsNbr){
    perror("Error: offset is bigger than the size of a trace.
    
    
exit(-1);
}

#ifdef VERBOSE_TRACE
printf("\nReading data...");
#endif

if ((pointsNbr > pointsToRead) and (pointsToRead !=0)){
    pointsNbr = pointsToRead;
} else if (pointsToRead !=0){
    perror("Warning: offset+pointsToRead > pointsNbr in the trace!\nReading till
the end of a trace.
    
    
}

traceBuffer = new float[pointsNbr];
// skip "offset" points
inputFile.seekg(sizeof(float)*offset, std::ios_base::cur);
inputFile.read((char*)traceBuffer, sizeof(float)*pointsNbr); // read trace
    
    
std::vector<float> trace(traceBuffer, traceBuffer + pointsNbr); // fill a vector
from the buffer

delete[] traceBuffer; // delete buffer
inputFile.close();

#ifdef VERBOSE_TRACE
printf(" done 
");
#endif

return trace;


filename full path to read from
points vector of interesting points

Allows to read only selected points from a power trace

std::vector<float> readTraceFrom(const char* filename, std::vector<unsigned>
    
    
points){
    float tracePoint;
    unsigned pointsNbr;
    std::ifstream inputFile;
    inputFile.open(filename, std::fstream::in | std::fstream::binary);
    if (!inputFile.is_open()){
        perror("Error: could not open the trace file\n");
        exit(-1);
    }

#ifdef VERBOSE_TRACE
printf("Reading %s", filename.c_str());
#endif

pointsNbr = readTraceHeader(inputFile);
if (points.size() >= pointsNbr){
    perror("Error: number of requested points is bigger than the size of a trace
.\n");
    exit(-1);
}

#ifdef VERBOSE_TRACE
printf("\nReading data...");
#endif
```cpp
/* */

std::vector<float> trace;
unsigned previousPointIdx = 0;

for(unsigned i=0; i<points.size(); ++i){
    ifndef VERBOSE_TRACE
    printf("%lu, ", points[i]);
    ifndef
    if (pointsNbr<(((int)points[i])-((int)previousPointIdx))){
        perror("Error: you want to read a point index that is bigger than the size of the trace\n");
        exit(-1);
    }
    inputFile.seekg(sizeof(float)*(((int)points[i])-((int)previousPointIdx)), std::ios_base::cur);
    inputFile.read((char*)&tracePoint, sizeof(float)); // read trace point from file
    trace.push_back(tracePoint);
    previousPointIdx = points[i]+1;
    }
    inputFile.close();
    ifndef VERBOSE_TRACE
    printf(" done\n");
    endif
    return trace;
}

/*
Next functions allow to read data from a folder that contains result of
experiments
done at DPA Lab ULB.
Folder "experiment_name" contains a directory "traces" with files wave<ID>.bin;
file values.csv with plaintexts and a file key.csv with the key
Each time the parameter 'path' would be the path to the "experiment_name" folder
*/

std::vector<std::vector<float>> readTracesFrom(const char * path, unsigned tracesCount, unsigned offset =0, unsigned length =0){
    std::string traceFile = std::string(path) + std::string("/traces/wave");
    std::string filename;
    std::vector<float> traces;
    ifndef VERBOSE_TRACES
    printf(" Loading traces..\n");
    endif
    for (unsigned i=0; i<tracesCount; ++i){
        filename = traceFile+std::to_string(i)+std::string(".bin");
        traces.push_back(readTraceFrom(filename.c_str(), offset, length));
    }
    ifndef VERBOSE_TRACES
    printf(" done!\n");
    endif
    return traces;
}

/*
Next functions allow to read data from a folder that contains result of
experiments
done at DPA Lab ULB.
Folder "experiment_name" contains a directory "traces" with files wave<ID>.bin;
file values.csv with plaintexts and a file key.csv with the key
Each time the parameter 'path' would be the path to the "experiment_name" folder
*/
```
std::vector<std::vector<float>> readTracesWithOffsetFile(const char* path, unsigned tracesCount, unsigned offsetFile = 0, unsigned offset = 0, unsigned length = 0) {
    std::string traceFile = std::string(path) + std::string("/traces/wave");
    std::string filename;
    std::vector<float> traces;

    #ifdef VERBOSE_TRACES
    printf("Loading traces ..");
    #endif
    for (unsigned i = offsetFile; i < offsetFile + tracesCount; ++i) {
        filename = traceFile + std::to_string(i) + std::string(".bin");
        traces.push_back(readTraceFrom(filename.c_str(), offset, length));
    }

    #ifdef VERBOSE_TRACES
    printf(" done !\n");
    #endif

    return traces;
}

std::vector<std::vector<float>> readTracesFrom(const char* path, unsigned tracesCount, std::vector<unsigned> points) {
    std::string traceFile = std::string(path) + std::string("/traces/wave");
    std::string filename;
    std::vector<float> traces;
    std::vector<float> traces;

    #ifdef VERBOSE_TRACES
    printf("Loading traces ..");
    #endif
    for (unsigned i = 0; i < tracesCount; ++i) {
        filename = traceFile + std::to_string(i) + std::string(".bin");
        traces.push_back(readTraceFrom(filename.c_str(), points));
    }

    #ifdef VERBOSE_TRACES
    printf(" done !\n");
    #endif

    return traces;
}

std::vector<unsigned char> getKey(const char* folderName) {
    std::string filename = std::string(folderName) + std::string("/key.csv");
    std::ifstream inputFile;
    inputFile.open(filename.c_str());

    // read data
    std::string line;
    std::vector<unsigned char> key;

    getline(inputFile, line);
    std::string tmp = "";
    unsigned i = 0;
    if(line[0] == '0')
        if(line[1] == 'x')
            i = 2;
    for(; i < line.size(); ++i) {
        if (line[i] != ',' && line[i] != 'n' && line[i] != '\' && line[i] != '') {
            tmp += line[i];
        } else if (tmp != "") {
            key.push_back(stoul(tmp, 0, 16)); // 0 is the default 2nd parameter, base
            tmp = "";
        }
    }
    return key;
}
std::vector<std::vector<unsigned char>> getValues(const char *folderName) {
    std::string filenameValues = std::string(folderName) + std::string("/values.csv");
    std::ifstream inputFile(filenameValues);
    if (!inputFile) {
        throw std::invalid_argument("Could not open file.");
    }
    std::string tmp = "";
    std::vector<std::vector<unsigned char>> values;
    unsigned currIndex = 0;
    getline(inputFile, line);
    std::string tmp = "";
    for (unsigned i = 0; i < line.size(); ++i) {
        if (line[i] != '\r' && line[i] != ';') {
            tmp += line[i];
        } else {
            values.push_back(std::vector<unsigned char>());
            unsigned long long val = stoull(tmp, 0, 16);
            for (uint8_t j = 0; j < 8; ++j) {
                values[currIndex].push_back((unsigned char) val);
                val >>= 8;
            }
            // 0 is the default 2nd parameter
            tmp = "";
            currIndex++;
        }
    }
    if (tmp != "") {
        currIndex++;
        values.push_back(std::vector<unsigned char>());
        values[currIndex].push_back(stoul(tmp, 0, 16));
    }
    inputFile.close();
    return values;
}

10.7 Correlation Power Analysis attack on Present

#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <math.h>
#include <numeric>
#include <sys/stat.h>
#include "infTraceReader.cpp"
#include "present.h"
using namespace std;

#define VERBOSE
#define VERBOSE_RESULTS

void extractDataScatterPlot(const char *folderName);
vector<
uint8_t> getHypothesis();
uint8_t getHammingWeight(uint8_t block);
void computeHMatrix(uint8_t blockIdx);
void computeMeanValuesForMatrices();
double computeMaxCorrelationAndSaveForGraph(uint16_t hypIdx, unsigned long maxCorrelationIndex);
unsigned long findMaxCorrelationIndex(uint8_t blockIdx, uint8_t key[10]);
vector<unsigned long> fetchMaxCorrelationIndexes();
void computeMaxCorrelationIndexes();
void extractWavesMeanValue();
void extractHypothesisHWMeanValue();
void computeHypothesisScoreForBlock(vector<double> & hypothesisScore, const vector<unsigned long> maxCorrelationIndexes, uint8_t blockIdx, uint8_t keyIdx);
double checkKey(vector<vector<double>>);
bool printResultRFile(vector<vector<double>> hypothesisScore);
void extractBytes8_9FromResultsOf2ndRound(vector<vector<double>> & hypothesisScore, vector<vector<double>> & hypothesisCorrelationForBlocks);
void extractDataCorrelation(uint8_t keyByte);
double computeCPA(const char * folderName, uint8_t mean, uint16_t valuesToRead, uint16_t offset);
double computeScore(uint8_t mean, uint16_t traceNbr);

// initialize original index locations
vector<size_t> idz(v.size());
 iota(idz.begin(), idz.end(), 0);
// sort indexes based on comparing values in v
sort(idz.begin(), idz.end(), [&v](size_t ii, size_t i2) {
    return v[ii] > v[i2];
});
return idz;

// initialize original index locations
vector<size_t> idz(v.size());
 iota(idz.begin(), idz.end(), 0);
// sort indexes based on comparing values in v
sort(idz.begin(), idz.end(), [&v](size_t ii, size_t i2) {
    return v[ii] < v[i2];
});

return idx;

const char * folderName;
uint8_t mean;
uint16_t valuesToRead;
uint16_t offset;
const uint16_t windowSize = 100;
uint16_t topSize = 25;
uint8_t key[10];
uint8_t computedKey[10];
uint8_t round_key[10];
vector<uint8_t> tmpKey;

vector<vector<uint8_t>> values;
vector<uint8_t> hypothesis;
vector<vector<float>> waves;
vector<vector<uint8_t>> H;
vector<float> meanHypothesisHW;
vector<float> meanValue;
vector<vector<double>> correlation; // For the graphs

bool result;

double outsideTop = 0;
double position;

int main ()
{
    ofstream res;
    for (uint8_t mean=2; mean<=4; mean*=2) {
        cout << " ================= Mean " + to_string(mean) + " ============ " << endl;
        res.open(" result " + to_string(mean) + ".txt ");
        res << " trace ; score ; excluded ; avgPosition " << endl;
        uint16_t quantity[24] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20, 25, 30, 40, 50, 75, 100, 125, 150, 175, 200, 250};
        for (uint16_t i = 0; i < 24; i++) {
            cout << " ________Quantity " + to_string(quantity[i]) + " _________ " << endl;
            res << to_string(quantity[i]) << "," << computeScore(mean, quantity[i]) << endl;
        }
        res.close();
    }
    return 0;
}

double computeScore(uint8_t mean, uint16_t traceNbr)
{
    double score = 0;
    uint8_t count = 0;
    uint8_t outsideTop = 0;
    double averagePosition = 0;
    for (uint16_t i=0; i < (1000-traceNbr); i+=25){
        :position = 0;
        cout << " Computing for offset " << to_string(i) << endl;
        string fileName = "../traces/present"+to_string(mean)+".txt";
        score += computeCPA(fileName.c_str(), mean, traceNbr, i);
        if (!::result) {
            outsideTop++;
        }
    }
    averagePosition += :position;
    count++;
}

::outsideTop = (double) outsideTop*100 / count;
::position = averagePosition / count;
return score / count;
double computeCPA(const char* folderName, uint8_t mean, uint16_t valuesToRead, uint16_t offset) {
    ::valuesToRead = valuesToRead;
    ::offset = offset;
    ::folderName = folderName;
    ::mean = mean;
    tmpKey = getKey(folderName);
    if(values.size() == 0){
        values = getValues(folderName);
    }
    hypothesis = getHypothesis();
    vector<vector<double>> hypothesisScore;
    hypothesisScore.resize(10);
    H.resize(valuesToRead);
    meanHypothesisHW.resize(hypothesis.size());
    correlation.resize(hypothesis.size());
    copy(tmpKey.begin(), tmpKey.end(), key);
    for (uint8_t i = 0; i < 10; ++i) {
        round_key[i] = 0;
    }
    for (uint8_t i = 0; i < 10; ++i) {
        vector<double> score(256, 0);
        hypothesisScore[i] = score;
    }
    waves = readTracesWithOffsetFile(folderName, valuesToRead, offset);
    vector<unsigned long> maxCorrelationIndexes = fetchMaxCorrelationIndexes();
    for (uint8_t blockIdx = 0; blockIdx < 8; ++blockIdx) {
        vector<double> hypothesisCorrelationForBlock(hypothesis.size());
        computeHypothesisScoreForBlock(hypothesisCorrelationForBlock, maxCorrelationIndexes, blockIdx, (uint8_t) 8 + blockIdx);
        hypothesisScore[blockIdx] = hypothesisCorrelationForBlock;
    }
    for (uint8_t blockIdx = 0; blockIdx < 8; ++blockIdx) {
        computedKey[blockIdx] = round_key[blockIdx];
    }
    for (uint8_t blockIdx = 0; blockIdx < 3; ++blockIdx) {
        vector<double> vec(hypothesis.size());
        computeHypothesisScoreForBlock(vec, maxCorrelationIndexes, blockIdx, (uint8_t) 8 + blockIdx);
        hypothesisCorrelationForBlocks[blockIdx] = vec;
    }
    extractBytes8_9FromResultsOf2ndRound(hypothesisScore, hypothesisCorrelationForBlocks);
    ::result = printResultRFile(hypothesisScore);
    return checkKey(hypothesisScore);
}

void extractBytes8_9FromResultsOf2ndRound(
    vector<vector<double>>& hypothesisScore,
    vector<vector<double>>& hypothesisCorrelationForBlocks) {
    for (uint16_t hypIdx = 0; hypIdx < hypothesis.size(); ++hypIdx) {
        double maxCorrelation8 = 0;
        double maxCorrelation9 = 0;
        for (uint8_t blockIdx = 0; blockIdx < 8; ++blockIdx) {
            double maxCorrelation8 = 0;
            double maxCorrelation9 = 0;
            for (uint8_t i = 0; i < 10; ++i) {
                round_key[i] = 0;
            }
            for (uint8_t i = 0; i < 10; ++i) {
                vector<double> score(256, 0);
                hypothesisScore[i] = score;
            }
            waves = readTracesWithOffsetFile(folderName, valuesToRead, offset);
            vector<unsigned long> maxCorrelationIndexes = fetchMaxCorrelationIndexes();
            for (uint8_t blockIdx = 0; blockIdx < 8; ++blockIdx) {
                vector<double> hypothesisCorrelationForBlock(hypothesis.size());
                computeHypothesisScoreForBlock(hypothesisCorrelationForBlock, maxCorrelationIndexes, blockIdx, (uint8_t) 8 + blockIdx);
                hypothesisScore[blockIdx] = hypothesisCorrelationForBlock;
            }
            for (uint8_t blockIdx = 0; blockIdx < 8; ++blockIdx) {
                computedKey[blockIdx] = round_key[blockIdx];
            }
            for (uint8_t blockIdx = 0; blockIdx < 3; ++blockIdx) {
                vector<double> vec(hypothesis.size());
                computeHypothesisScoreForBlock(vec, maxCorrelationIndexes, blockIdx, (uint8_t) 8 + blockIdx);
                hypothesisCorrelationForBlocks[blockIdx] = vec;
            }
            extractBytes8_9FromResultsOf2ndRound(hypothesisScore, hypothesisCorrelationForBlocks);
            ::result = printResultRFile(hypothesisScore);
            return checkKey(hypothesisScore);
        }
    }
}
for (uint8_t i = 0; i < 8; ++i) {
    // Iteration over the 3 bits from left keyBlock
    uint8_t composedKey8 = (uint8_t) (i << 5 | hypIdx >> 3);
    uint8_t composedKey9 = (uint8_t) (i << 5 | hypIdx >> 3);
    double correlation8 = hypothesisCorrelationForBlocks[0][composedKey8];
    double correlation9 = hypothesisCorrelationForBlocks[1][composedKey9];
    if (i == 0 || maxCorrelation8 < correlation8) {
        maxCorrelation8 = correlation8;
    }
    if (i == 0 || maxCorrelation9 < correlation9) {
        maxCorrelation9 = correlation9;
    }
}

hypothesisScore[8][hypIdx ^ 0x80] = maxCorrelation8; // xor hypothesis with round counter
for (uint8_t i = 0; i < 32; ++i) {
    // Iteration over the 5 bits from right keyBlock
    uint8_t composedKey8 = (uint8_t) (hypIdx << 5 | i);
    uint8_t composedKey9 = (uint8_t) (hypIdx << 5 | i);
    double correlation8 = hypothesisCorrelationForBlocks[1][composedKey8];
    double correlation9 = hypothesisCorrelationForBlocks[2][composedKey9];
    if (i == 0 || maxCorrelation8 < correlation8) {
        maxCorrelation8 = correlation8;
    }
    if (i == 0 || maxCorrelation9 < correlation9) {
        maxCorrelation9 = correlation9;
    }
}

hypothesisScore[8][hypIdx ^ 0x80] = (hypothesisScore[8][hypIdx ^ 0x80] + maxCorrelation8) / 2;

hypothesisScore[9][hypIdx] = (hypothesisScore[9][hypIdx] + maxCorrelation9) / 2;
}

double checkKey(vector<vector<double>> hypothesisScore) {
    double score = 0;
    double averagePosition = 0;
    for (uint8_t blockIdx = 0; blockIdx < 10; ++blockIdx) {
        vector<size_t> resultIndexes = decreasing_sort_indexes(hypothesisScore[9 - blockIdx]);
        char message[3];
        sprintf(message, "%02 x", (unsigned int) resultIndexes[0]);
        bool found = false;
        for (uint16_t i = 0; i < resultIndexes.size() && !found; ++i) {
            if (resultIndexes[i] == key[9 - blockIdx]) {
                if (i < ::topSize){
                    score += ::topSize - i;
                }
                averagePosition += i;
                found = true;
            }
        }
    }

    ::position = averagePosition / 10;
    score /= (double)::topSize/10;
    return score;
}

void computeHypothesisScoreForBlock(vector<double> &hypothesisScore, const vector<unsigned long> &maxCorrelationIndexes,
    uint8_t blockIdx, uint8_t keyIdx) {
    computeHMatrix(blockIdx);
computeMeanValuesForMatrices();
#endif
for (uint16_t hypIdx = 0; hypIdx < hypothesis.size(); ++hypIdx) {
  vector<double> vec(waves[0].size());
  correlation[hypIdx] = vec;
  hypothesisScore[hypIdx] = computeMaxCorrelationAndSaveForGraph(hypIdx, maxCorrelationIndexes[keyIdx]);
#ifdef VERBOSE
  if (hypIdx % 50 == 0)
  cout << endl;
  cout << "X";
  cout.flush();
#endif
}
vector<size_t> resultIndexes = decreasing_sort_indexes(hypothesisScore);
#ifdef VERBOSE_RESULTS
  for (auto i: resultIndexes) {
    cout << " hypothesis : " << to_string(i) << " = " << to_string(hypothesisScore[i]) << endl;
  }
#endif
round_key[blockIdx] = (uint8_t) resultIndexes[0];
}
H[valIdx][hypIdx] = getHammingWeight(sBoxOneByte(values[offset + valIdx], blockIdx), hypothesis[hypIdx]);

if (H[valIdx][hypIdx] >= 256 || H[valIdx][hypIdx] < 0) {
  cerr << "Incorrect HW result" << endl;
  exit(-1);
}

void computeMeanValuesForMatrices() {
  extractHypothesisHWMeanValue();
  extractWavesMeanValue();
}

void extractHypothesisHWMeanValue() {
  for (uint16_t hypIdx = 0; hypIdx < hypothesis.size(); ++hypIdx) {
    for (unsigned valIdx = 0; valIdx < valuesToRead; ++valIdx) {
      meanHypothesisHW[hypIdx] += H[valIdx][hypIdx];
    }
    meanHypothesisHW[hypIdx] /= valuesToRead;
  }
}

void extractWavesMeanValue() {
  for (unsigned long time = 0; time < waves[0].size(); ++time) {
    for (unsigned valIdx = 0; valIdx < valuesToRead; ++valIdx) {
      meanValue[time] += waves[valIdx][time];
    }
    meanValue[time] /= valuesToRead;
  }
}

unsigned long findMaxCorrelationIndex(uint8_t blockIdx, uint8_t key[10]) {
  double currentMaxCorrelation = 0;
  unsigned long maxCorrelationIdx = 0;
  vector<double> correlationWindow(windowSize);
  for (unsigned long time = 0; time < waves[0].size(); ++time) {
    double numerator = 0, denomHyp = 0, denomTrace = 0;
    for (unsigned valIdx = 0; valIdx < valuesToRead; ++valIdx) {
      numerator += (H[valIdx][key[blockIdx]] - meanHypothesisHW[key[blockIdx]]) *
        (waves[valIdx][time] - meanValue[time]);
      denomHyp += pow(H[valIdx][key[blockIdx]] - meanHypothesisHW[key[blockIdx]], 2);
      denomTrace += pow(waves[valIdx][time] - meanValue[time], 2);
    }
    double currentCorrelation = numerator / sqrt(denomHyp * denomTrace);
    correlationWindow += currentCorrelation;
    if (time < windowSize) {
      window[time] = currentCorrelation;
    } else {
      correlationWindow -= window[0];
      rotate(window.begin(), window.begin() + 1, window.end());
      window[windowSize - 1] = currentCorrelation;
    }
  }
  if (time == windowSize || correlationWindow > currentMaxCorrelation) {
    currentMaxCorrelation = correlationWindow;
    maxCorrelationIdx = time;
  }
}
return maxCorrelationIdx;
}

vector<uint8_t> getHypothesis() {
    vector<uint8_t> hypothesis;
    for (uint16_t i = 0; i < 256; ++i) {
        hypothesis.push_back((uint8_t)i);
    }
    return hypothesis;
}

vector<uint8_t> getHammingWeight(uint8_t block) {
    uint8_t HW = 0;
    for (uint8_t i = 0; i < 8; ++i) {
        HW += block & 0x1;
        block >>= 1;
    }
    return HW;
}

bool fileExists(const std::string &filename) {
    struct stat buf;
    return stat(filename.c_str(), &buf) != -1;
}

template vector<unsigned long> fetchMaxCorrelationIndexes()
{
    string filename = "maxCorrelationIndexes.csv";
    if (!fileExists(filename)) {
        computeMaxCorrelationIndexes();
    }
    ifstream indexes;
    indexes.open(filename);
    string content;
    getline(indexes, content);
    string buf;
    stringstream ss(content);
    vector<string> tokens;
    while (ss >> buf)
        tokens.push_back(buf);
    vector<unsigned long> maxCorrelationIndexes(tokens.size());
    for (uint8_t i = 0; i < tokens.size(); ++i) {
        stringstream(tokens[i]) >> maxCorrelationIndexes[i];
    }
    indexes.close();
    return maxCorrelationIndexes;
}

void computeMaxCorrelationIndexes() {
    ofstream maxCorrelationIndexes;
    maxCorrelationIndexes.open("maxCorrelationIndexes.csv");
    waves = readTracesFrom(folderName, valuesToRead, offset);
    meanValue.resize(waves[0].size());
    extractWavesMeanValue();
    for (uint8_t blockIdx = 0; blockIdx < 8; ++blockIdx) {
        cout << "bloc " << to_string(blockIdx) << endl;
        computeHMatrix(blockIdx);
        extractHypothesisHWMeanValue();
        maxCorrelationIndexes << to_string(findMaxCorrelationIndex(blockIdx, key)) << ";
    }
    for (uint16_t i = offset; i < offset + valuesToRead; ++i) {
uint8_t round_key[10];
copy(tmpKey.begin(), tmpKey.end(), round_key);

for (uint8_t blockIdx = 0; blockIdx < 3; ++blockIdx) {
    cout << " bloc " << to_string(blockIdx) << endl;
    computeHMatrix(blockIdx);
    extractHypothesisHWMeanValue();
    maxCorrelationIndexes << to_string(findMaxCorrelationIndex(blockIdx, round_key)) << ",";
}
maxCorrelationIndexes.close();

void extractDataScatterPlot(const char * folderName) {
    vector<vector<float>> waves;
    vector<unsigned> points;
    points.push_back(300);
    points.push_back(301);
    points.push_back(302);
    points.push_back(1307);
    waves = readTracesFrom(folderName, 1000, points);

    ofstream csvScatterPlot;
    csvScatterPlot.open("scatterPlotData.csv");
    for (unsigned i = 0; i < points.size(); ++i) {
        csvScatterPlot << " volts " << to_string(points[i]) << ";";
    }
    csvScatterPlot << endl;
    for (unsigned i = 0; i < waves.size(); ++i) {
        for (unsigned j = 0; j < waves[i].size(); ++j) {
            csvScatterPlot << waves[i][j] << ";";
        }
    }
    csvScatterPlot << endl;
    csvScatterPlot.close();
}

void extractDataCorrelation(uint8_t keyByte) {
    ofstream csvCorrelation;
    csvCorrelation.open("correlationData+to_string(keyByte)+".csv");
    csvCorrelation << " time;correlation " << endl;
    for (unsigned i = 0; i < 40000; ++i) {
        csvCorrelation << to_string(i) << " ;";
    }
    csvCorrelation << to_string(correlation[keyByte][i]) << endl;
}

csvCorrelation.close();

bool printResultRFile(vector<vector<double>> hypothesisScore) {
    bool result = true;
    ofstream output;
    // output.open("result/hypothesisResult"+to_string(offset)+"+"+to_string(valuesToRead)+"_"+to_string(mean)+
    // "_.R");
    output.open("resultVariationTopSize/result*to_string(::topSize)*".R");
}
for (uint8_t blockIdx = 0; blockIdx < 10; ++blockIdx) {
    vector<size_t> resultIndexes = increasing_sort_indexes(hypothesisScore[9 - blockIdx]);
    bool found = false;
    for (uint16_t i = 0; i < 256 - ::topSize; ++i) {
        if (resultIndexes[i] != key[9 - blockIdx]) {
            char message[2];
            sprintf(message, "%02x", (unsigned int) resultIndexes[i]);
            output << "instance setNotEq(\"K" + to_string(blockIdx) + \"[0\") \n" + message + \"\") \n" << endl;
        } else {
            result = false;
        }
    }
}
output.close();
return result;