CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Attacks on White-Box AES
Student:	Jean-Gaël Rigot
Supervisor:	Ing. Jiří Buček
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2016/17

Instructions

Create a white-box AES implementation according to Luo, Lai, and You [1] in C/C++.
Reproduce some attacks demonstrated by Bos, Hubain, Michiels, and Teuwen [2] using a suitable emulation tool for instrumentation.
Try to perform the Differential Computation Analysis attack on the white-box AES of [1].

References

[1] Luo, Rui, Xuejia Lai, and Rong You. "A new attempt of white-box AES implementation." *Security, Pattern Analysis, and Cybernetics (SPAC), 2014 International Conference on.* IEEE, 2014.
[2] Bos, Joppe W., et al. *Differential computation analysis: hiding your white-box designs is not enough.* Cryptology ePrint Archive, Report 2015/753, https://eprint. iacr. org/2015/753, 2015.

L.S.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 2, 2016

Czech Technical University in Prague

Faculty of Information Technology

Department of Computer Systems

Master's thesis

Attacks on White-Box AES

Jean-Gaël Rigot

Supervisor: Ing. Jiří Buček

17th May 2016

Acknowledgements

First I would like to thank my thesis advisor Ing. Jiří Buček who always provided me with very valuable advice both on the technical and organizational aspect of the project. He was always there to answer my questions and give me suggestion for improvement and interesting leads while allowing this thesis to be my own work.

I would also like to thank my family and friends for their unconditional support without which I could not have been able to realize this work.

Declaration

In Prague on 17th May 2016 ....

Citation of this thesis

Rigot, Jean-Gaël. *Attacks on White-Box AES.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Využití kryptografie (šifrování) je v zabránění přístupu k datem bez příslušného oprávnění. V některých případech jsou ale šifrovací algoritmy spuštěny na neznámém zařízení, například v případě přístupu k obsahu, chráněnému autorskými právy, na osobním počítači (hudba, filmy...). V tomto případě tradiční šifrovací modely nestačí zajistit bezpečnost dát. Ochranou dát v těchto podmínkách se zabývá white-box kryptografie (šifrování). Práce prezentuje různé návrhy ochranných prvků, jako i možnosti útoku na ně. Nedávný návrh dle Luo et al.[1] k ochraně AES (Advance Encryption Standard)[2] byl implementován v C a studován v souvislosti s typy útoků dle Bos et al.[3].

Klíčová slova white-box kryptografie, white-box šifrování, AES, white-box implementace šifry, diferenciační komputační analýza

Abstract

Cryptography is used to prevent people accessing data they are not authorized to access. However in some case the algorithm used for encrypting data are performed on an untrusted device for example in the case of access to copyrighted content on a personal computer (music, films...). In this case the traditional cryptographic model is not enough to assure the security of the data and the study of protection under this context is called white-box cryptography. This thesis presents different designs found in the llitterature as well as attacks to break those implementation. The recently proposed design by Luo et al.[1] to protect AES (Advance Encryption Standard) [2] has been implemented in C and studied in regards to the types of attacks introduced by Bos et al.[3]

Keywords white-box cryptography, AES, white-box cipher implementation, differential computation analysis

Contents

List of Figures

List of Tables

Introduction

In today's world, the security of information is a critical issue for public and private organizations as well as for individuals. In order to keep an important piece of information secret, cryptographic techniques have been developed to transform this piece of information in a form that will be understandable only by the person possessing another information, the key, that will allow him to transform back the information to its original, meaningful content. The area of cryptography saw major developments for more and more secure encryption methods and more performant types of attacks since the advent of computers and performant algorithms have been designed to encrypt the data such as today's standard Advance Encryption Standard (AES) [2] for symmetric cryptography and RSA named after the initials of its creators for asymmetric cryptography. And with its democratization and massive adoption by companies and consumers, these cryptographic methods are more and more performed on open devices (PC, smartphones) that are not always to be trusted.

One typical case of when the device should not be trusted concerns Digital Rights Management (DRM). In the modern digital economy (movies, music, video games) there are some case when someone can have access to a purchasable content but should not be able to use it because he does not have the key in order to access it; but this person might want to access it nevertheless and could try to attack the cryptographic software. Another possibility is that a machine performing encryption has been infected by a malware with the goal to recover the key to be able to decipher messages encoded by this software. This fact is not taken in account in traditional cryptography that concerns mostly about the impossibility of practically finding the key when presented with pairs of original input and encoded input. In order to protect the encrypted content against this type of threats another attack model needs to be developed and studied. This model is called white-box cryptography where the attacker has total control of the machine during the execution of the program performing the encryption.

It was introduced in 2002 by Chow et al. who proposed an implementation of the two most known block ciphers (AES and the Data Encryption Standard (DES)) supposed to resist attacks in this configuration. Both have been proved to be broken a few years later and since, different proposal of implementation and improvements have been made, and different attacks have been developed against it. It remains an open problem to find a proven secure implementation of AES in the white-box context and interesting proposals have been made in the recent past. Different attacks also have been developed recently. The original idea of this master thesis was to study the resistance of the design proposed by Luo et al. in 2014 [1] to the attack developed by Bos et al. in 2015 [3]. However, as I could not find an implementation of this design and did not received a reply to the mail I sent to the author, I decided to program this implementation and then to perform the analysis.

After a brief introduction to the goal of cryptography in general and a focus on the cipher AES in chapter 1, this thesis will present the different designs that have been proposed to construct an implementation that could resist attacks in the white-box context in chapter 2. The different attacks developed against these designs will be presented in chapter 3. The design given in [1] will be explained in chapter 4 along with the presentation of my program implementing it. Finally the chapter 5 will present the type of attacks introduced in [3] and demonstrate the results obtained against the previous implementation.

Chapter **1**

Cryptography and AES Cipher

This chapter will briefly describe the basic vocabulary used when speaking about cryptography and more specifically about block ciphers, and quickly describe the goal of an attacker when attempting to break a block cipher. It will then provide a description of the most widely used standard block cipher, the Advanced Encryption Standard.

1.1 Cryptography

From its origin, the goal of cryptography is for two persons (traditionally named Alice and Bob for example purposes) to communicate in a way that prevents someone intercepting the message (Oscar) to understand its content. Cryptography has been used and improved since about 4000 years, and played an important role in military and diplomatic communications and so has cryptanalysis, its counterpart that attempts to "break" the code and allow the attacker to read the message, as proven in World War 2 with the Enigma machine for example. In order to describe the methods used some vocabulary is necessary: A communication channel where a message can be intercepted is called insecure. The original content Alice wants to send to Bob is called *plaintext*, its encoded version not understandable by Oscar is called *ciphertext* and is obtained using an encryption algorithm with a predetermine *key*. Bob who possess information about the key can use the corresponding decryption algorithm to recover the plaintext. This mechanism is presented in Figure 1.1 and is called a *cryptosystem*. This term is given the following mathematical definition by Stinson [5]:

Definition 1 *A cryptosystem is a five-tuple* $\{\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D}\}$ *where the following conditions are satisfied:*

1. $\mathcal{P}$ *is a finite set of possible plaintexts;*

2. $\mathcal{C}$ *is a finite set of possible ciphertexts;*

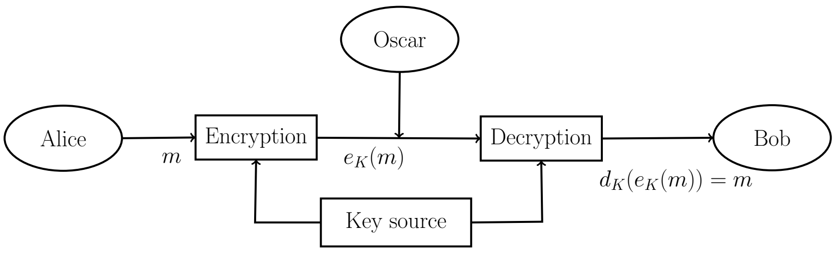

Figure 1.1: Exchange of an encrypted message m over an insecure channel

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

A broader definition of cryptography is given by Menezes, van Oorschot and Vanstone[6]:

Definition 2 *Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication.*

If confidentiality expresses the goal previously stated that only authorized people should be able to access the information, other aspects are also included in this definition. Data integrity means that it is possible to verify that the information has not been modified by unauthorized people, entity authentication means that the identity of said entity (a person, a computer terminal...) can be confirmed without any doubts and data origin authentication express the same notion about the source of the information.

There is a variety of mathematical tools called *primitive* that are used to achieve a better security of the information, they can be separated in three main categories depending on the type of key used:

- unkeyed primitives, that do not require a key
- symmetric-key primitives, that require a key shared by the sender and receiver, generally exchanged over a secure communication channel (or at least two keys e and d respectively for encryption and decryption that are computationally "easy" to determine knowing only the other one)
- public-key primitives, that requires a key in two parts, one is public (known to everyone), the other private (only known by the user)

The focus here will be put on symmetric cryptography and more specifically symmetric-key ciphers.

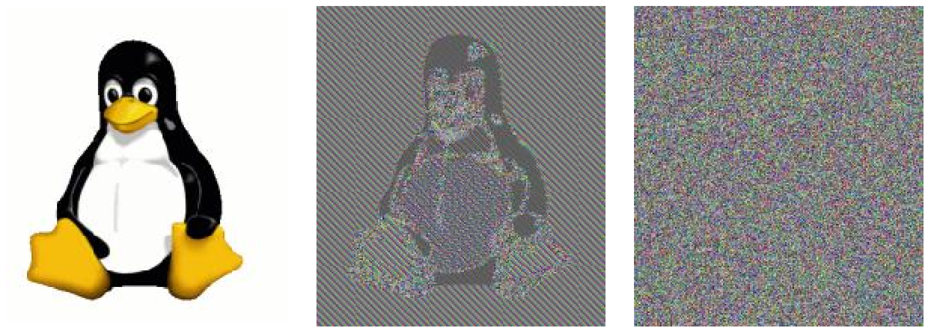

Figure 1.2: The original mascot, its encrypted version by Lunkwill using ECB mode and using another mode of operation on [9]

1.2 Block Cipher

There are two types of symmetric ciphers, stream ciphers and block ciphers. The former operates on individuals plaintext digits while the latter works on a bigger chunk of data of a fixed size.[7] Often the stream cipher is implemented using exclusive OR (XOR) operation with a pseudorandom generated digit stream called the keystream. We will now consider the second category in more details.

A block cipher can be defined this way:[8]

Definition 3 *A block cipher is a function* $E : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$

It means that a block cipher transforms n bits of plaintext and k bits of key into n bits of ciphertext, the numerical values of this number being specified by the algorithm used. It is important to consider that block ciphers are not directly usable by end-user and need to be integrated correctly into the system to protect in order to guarantee its security. In order to do so, several modes of operation have been defined. The most direct use of the cipher is called Electronic Codebook (ECB) and consists in splitting the plaintext into chunks of the required size n and encrypting each block separately. This mode of operation is not recommended to use for a message longer than one block if the same key is to be reused, since it conserves pattern in the plaintext as demonstrated on the Wikipedia article of block cipher encryption modes with the encrypted version of the Linux mascot designed by Larry Ewing with The GNU Image Manipulation Program (GIMP) in Figure 1.2. Therefore we can see that a secure cipher does not guarantee a secure implementation. A more recommended mode is Cipher Block Chaining (CBC) where the cipher text of one block (or the initialisation vector for the first block) is XORed to the plaintext of the following one therefore preventing two identical blocks to be encrypted the same way. There exist also other modes like Counter (CTR) or Cipher Feedback (CFB).

The traditional goal of cryptanalysis of block cipher is to recover the key that was used under different attacks hypotheses: mainly the known plaintext attack where the attacker has knowledge of pair of plaintext and ciphertext and the chosen plaintext attack where the attacker can choose the plaintext he wants to know the result of encryption and adapt his strategy. We will see later how the white-box context differ from these two attacks scenarios. The attack can be performed using brute-force (trying every possible key) or exploiting a weakness of the cipher to reduce the search space.

The most known and used block ciphers used to be the DES, standardized in 1977. It is no longer recommended because of its small keyspace and has been replaced by the cipher that superseded it, the AES.

1.3 Advanced Encryption Standard (AES)

AES was designated a standard by the U.S. National Institute of Standards and Technology (NIST) in 2001 following a competition to choose a replacement for DES. It is a part of the Rijndael family of block ciphers with different block and key lengths designed by Joan Daemen and Vincent Rijmen for this selection process. It is described in the Federal Information Processing Standard (FIPS) Publication 197[2]. Its blocks are 128 bits long and the key can be 128, 192 or 256 bits long. The 128 input bits are considered by the algorithm as a 2-dimensional array of 4×4 bytes which represent elements of the AES field: the finite field $GF(2^8)$ using the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ as a modulus. This means the addition is represented by a XOR function between 2 bytes that represent polynomials with coefficient in $GF(2) = \{0, 1\}$ and degree strictly less than 8, and the multiplication that does not have a simple equivalent to a byte operation is the result of polynomial multiplication (using addition modulo 2) reduced modulo $m(x)$ which is represented as $\{01\}\{1b\}$. The 4×4 array also called state of the algorithm can also be considered as an array of 32-bits words of length 4 and represented by its columns.

Several operations are defined on this state:

- **SubBytes()** is the non-linear operation of AES. It operates independently on each byte and substitutes a byte by the result of an affine transformation applied to the byte's inverse. As these values are fixed, they can be directly looked up in a table, the so-called S-box shown in Table 1.1.

- **ShiftRows()** is an operation where the different rows of the array are cyclically shifted from an index depending on the position of the row. The first row, $r = 0$ is unchanged, the second row $r = 1$ shifted once to the left and so on.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table 1.1: the AES S-box

- **MixColumns()** is an operation on the columns of the state. Each column is considered as a polynomial with coefficients in $GF(2^8)$ and is multiplied with a fix polynomial $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ modulo $x^4 + 1$. This operation is often represented as a matrix multiplication:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{1.1}$$

- **AddRoundKey()** is the only key-dependent operation applied to the state. It consists on applying a bitwise XOR operation between the columns of the state and a word of the round key generated from the key according to a key schedule.

The key schedule is defined using the S-box substitution of byte, cyclic rotations and XOR operation with a round constant Rcon[i] determined as a power of $x(\{02\})$ to the round. The algorithm consists of 10 rounds for AES-128, 12 rounds for AES-192 and 14 for AES-256. It is described in the NIST publication as the pseudo-code in Algorithm 1, where Nr describes the number of rounds, Nb is the number of columns of the state which is always 4, Nk the number of 32-bit words in the key (4, 6 or 8) and w[] contains the different round keys.

Since its introduction AES has been extensively studied to find a potential weakness that would allow to recover the key faster than a brute force search (i.e. with time complexity of 2^{128} for AES-128), and though some results have been achieved on reduced-round AES or exploiting encryption with two

Algorithm 1 AES Cipher

```
Cipher(byte in[4*Nb], byte out[4*Nb], byte key[4*Nk])
byte state[4,Nb]
word k[(Nr+1),Nb]
k[0,(Nr+1)*Nb-1] = ExpandRoundKey(key)
state = in
AddRoundKey(state, k[0, Nb-1])
for round = 1 step 1 to Nr–1 do
    SubBytes(state)
    ShiftRows(state)
    MixColumns(state)
    AddRoundKey(state, k[round*Nb, (round+1)*Nb-1])
end for
SubBytes(state)
ShiftRows(state)
AddRoundKey(state, k[Nr*Nb, (Nr+1)*Nb-1])
out = state
```

different key k and $k \oplus \delta$ with known difference δ in the related-key model[10], as today there is no threat to the practical use of AES.

Chapter 2

White-Box Cryptography and AES Implementations

This chapter will describe the white-box attack context and how it differs from the traditional model. Then the different design proposed for the implementation of AES will be presented.

2.1 Black-box and White-Box Attack Context

The traditional key-recovery attack model on block cipher assumes only knowledge of plaintext-ciphertext pairs, chosen or not. This is called the black-box model: it is like the cipher is a black-box from which we know only the input and output (the encryption algorithm is known too, but not the important parameter: the key that we want to recover). But sometimes the attacker has more power than that and can also perform measurements and even inject a fault during the execution of the cipher implementation. This is the case for example with power measurements or electromagnetic disturbances [11]. Therefore it is also important to study the security of ciphers and their implementation under different threat models than the classical one. Allowing the attacker to perform these side-channel attacks in the analysis can be referred to as the grey-box model.

The white-box model grants even more power to the attacker, because in these scenarios, he has full control over the environment that executes the program and can use as many tools he wants (like debugger, disassembler, instrumentation framework..) in order to recover the key. This changes of the model reflects on the Figure 2.1. This context was defined by Chow et al. in 2002 [12] together with a first proposal implementation for AES cipher. The same year they also proposed a DES implementation in an attempt to find solutions against these newly studied attacks. They give the following list of threats that are applicable in the white-box attack context:

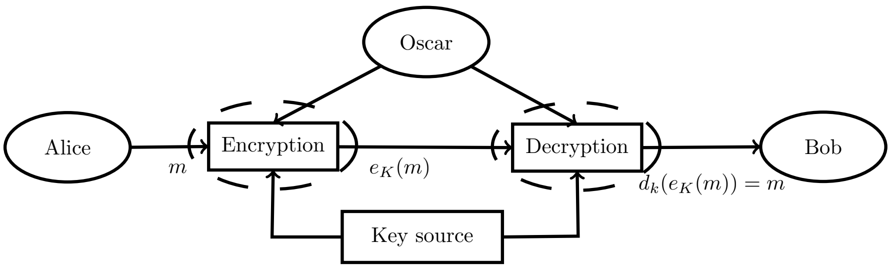

Figure 2.1: The possibilities of Oscar change in the white-box model. He has a total access to the encryption and/or decryption process (Oscar could even be Alice or Bob in the scenario of DRM and copyright violation for example). In the grey-box model, he would only have access to partial information about this process through side-channel attacks.

1. *fully-privileged attack software shares a host with cryptographic software, having complete access to the implementation of algorithms*
2. *dynamic execution (with instantiated cryptographic keys) can be observed*
3. *internal algorithm details are completely visible and alterable at will*

If some cryptographic software is run in on an untrusty host, we need to protect it against such threats. This is the case mainly with digital rights management (DRM) to prevent unauthorized access to music or films, but is also useful for client software running in the cloud. The attacker in this model could be either the user of one of the endpoints or a malware installed on the host. However other interesting applications are possible like transforming a secret-key encryption into a public-key encryption scheme or transforming a Message Authentication Code (MAC) into a digital signature under certain conditions. This is done by distributing the white-box implementation of the algorithm (encryption or MAC) as public while keeping the key secret.[13] Study of this field can also lead to the development of techniques for protection against software tampering which means modifying a software's program in order to modify its functionality. Some protection can be achieved by considering the executable code as the encryption key used to generate the white-box implementation.[14] However white-box cryptography implementations are not suitable for every use. They are generally orders of magnitude slower than hardware implementations, need more resources than traditional implementations and it is important to correctly evaluate the requirements of the system before deciding if a white-box implementation is suited.[13]

White-box cryptography, code obfuscation, and tamper-resistance are all related and complementary techniques to achieve different security goals. For white-box cryptography the goal is to prevent the extraction of the key of a cryptographic algorithm; for code obfuscation it is to protect against reverse engineering of the algorithm and for tamper-resistance it is as said before to prevent modification of some functionality of the program. Some theoretical results have been obtained in this field: It was shown in 2001 by Barack et al.[15] that it is impossible to construct a general obfuscator that could turn any existing program to a virtual black box that does not leak information not contained in its input and output. This is however not a reason to give up attempts to construct white-box implementation of common ciphers. Indeed, we do not know if they are part of the class of functions that cannot be obfuscated or not and more favorable results have been obtained under different definitions of obfuscation.

2.2 White-Box Secure AES Implementations Attempts

This section will present the publicly available description of white-box implementations. Other design have been made by companies like Irdeto but they have not been publicly disclosed[16].

2.2.1 General Considerations

If we managed to construct a program that looks up the value of the ciphertext in a table given the value of the plaintext, it will not give more information to the attacker than the cipher in the black-box model. Unfortunately, it is impossible to construct such a big table in practice as AES has 128 bits input. This observation still is the base of the different white-box implementations as they convert the algorithm into a succession of smaller lookup tables performing the different steps of the algorithm. As all the published white-box implementations of AES are using lookup tables, there are some terms and common techniques that we can define. The tables are protected with different operations [17]:

- Encodings: these are non-linear permutation in order to encipher intermediate states of the algorithm, therefore achieving confusion. They are chosen so they cancel with the next.
- Mixing bijections: these are linear bijections generally represented by matrices over GF(2) used to improve diffusion for tables containing information about round key bytes.
- External encodings: these are encodings applied on the external boundaries of the program in order to prevent plaintext and ciphertext to appear during the execution

Confusion and diffusion are terms introduced by Shannon [18] considering here a table containing bytes of the key as a small block cipher. The former means that the relation between ciphertext and encryption key should be complex and the latter states that a small change in the plaintext should result in change of 50% of the bits of the ciphertext in probability.

Chow et al. also defined the equivalent to keyspace in the white-box context when it is achieved with a network of lookup tables, i.e. the upper bound of the research space to recover the key with the concept of white-box diversity and white-box ambiguity. The white-box diversity of a type of table expresses how many distinct constructions there are for a table (it represents how many choices there are for each parameter of a table and not how many different tables, as different constructions can lead to the same table). The white-box ambiguity of a type of table expresses then how many distinct constructions will result in the exact same table and measure how many possible interpretation of one table are possible for an attacker who wants to find the true one.[12]

The implementation of Chow et al. will be described in more detail as it is the first one, most of the other are based on it proposing improvements or bigger modification, and it has been explained in great details with bytes flow scheme by Muir and his tutorial could be consulted as a reference[17].

2.2.2 Chow's Implementation

The implementation described here is for AES-128 as it is the main example used in papers, but the idea could easily be applied to AES-192 or AES-256. The first step in the construction of Chow et al. is to reorder the operations of AES after the following observations:

- The first AddRoundKey(state, k_0) can be brought in the loop while the last of the loop AddRoundKey(state,k_9) is moved outside.
- SubBytes and ShiftRows commute because ShiftRows only change the positions of the bytes and SubBytes their values, therefore we can exchange their position.
- ShiftRows is a linear operation, so we can apply it before AddRoundKey if we also apply it on AddRoundKey which give us a different key schedule, we note $\hat{k}$ the round keys that have been shifted according the rules of ShiftRows from the round 1 to 10.

This give us the equivalent algorithm to AES-128 presented as Algorithm 2.

Once this step is done we can construct the different lookup tables of the implementation. The goal is to prevent the key to appear in memory and in the register during computation and so we precompute the outputs for all possible inputs of different operations and protect all key-dependent variables. In order to do so, we combine the SubBytes() and AddRoundKey() operations to define what are called T-boxes that are all different for each byte of each round :

Algorithm 2 AES-128 Equivalent Algorithm used for white-box implementation

```
Cipher(byte in[4*4], byte out[4*4], byte key[4*4])
byte state[4,4]
word k[11,4], $\hat{k}$[10,4]
k[0,11*4-1] = ExpandRoundKey(key)
$\hat{k}$[0,10*4-1] = ShiftRoundKey(k[0,10*4-1])
state = in
for round = 1 step 1 to 9 do
  ShiftRows(state)
  AddRoundKey(state, $\hat{k}$[(round-1)*4, round*4-1])
  SubBytes(state)
  MixColumns(state)
end for
AddRoundKey(state, $\hat{k}$[9*4, 10*4-1])
SubBytes(state)
ShiftRows(state)
AddRoundKey(state, $k$[10*4, 11*4-1])
out = state
```

$$T_i^r(x) = S(x \oplus \hat{k}_{r-1}[i]), \text{ for } i = 0...15 \text{ and } r = 1...9,$$
$$T_i^{10}(x) = S(x \oplus \hat{k}_9[i]) \oplus k_{10}[i], \text{ for } i = 0...15$$

The MixColumns() operation, which is the matrix product presented in Equation 1.1 can be decomposed into a XOR of 32 bits values, that are all dependent of one single byte of the state in the following way :

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x0 \\ x1 \\ x2 \\ x3 \end{bmatrix} = x0 \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus x1 \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus x2 \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus x3 \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \quad (2.1)$$

For each of the 4 different multiplications, we compute a so called Ty table and store 4 copies for each of the 9 rounds where MixColumns is applied. We proceed to directly connect the T-boxes previously computed to these Ty tables when it is the case, forming one type of table taking 8 bits as input and exiting 32 bits output. The values will then be XORed using XOR tables taking two nibbles (4 bits) as input and resulting in one as an output. It will consist of two stages in order to reduce 32 bits to 8: first 16 XOR tables will reduce 32 bits to 16, then 8 XOR tables will reduce 16 bits to 8. The ShiftRows() operation is realized by controlling the bytes flow and carefully choosing which input should be used by the tables.

In order to protect the tables, mixing bijections are added to it: an 8-bit to 8-bit named L-1 in entrance and 32-bit to 32-bit named MB before the exit of the T-box/Ty table. However they need to be canceled out in order to perform the AES operations on the correct values, leading to the creation of another type of table. The same technique of decomposing the matrix product (recall from Section 2.2.1 that mixing bijections are linear) into XORs of 32-bit vectors is used, and 4 bijections L are concatenated paying attention to use the correct one considering that ShiftRows() operation will be applied in the next round. The bijections can be generated uniformly at random by generating an invertible matrix of the right size. These two operations are composed in order to form a table. Their outputs will be reduced back to 8 bits using the same technique as before. The bytes flow of one round can be seen in a schema presented in the tutorial of Muir in Figure 2.2 .

Then, each table has its input and output encoded. The input encoding cancelling the output one of the previous table which means that the XOR tables will all be different and it is not possible to use only one. Finally external encodings are applied to the entrance and exit tables. This raises the number of table's type to 5 also presented in the tutorial of Muir in Figure 2.3.

This implementation results in the following number of tables:

- 9*(16+16) = 288 tables with 8-bit input and 32-bit output (tables with MixColumns and Mixing bijections)
- 9*4*(16+8+16+8) = 1728 tables with 8-bit input and 4-bit output (XOR tables)
- 16 tables with 8-bit input and 8-bit output (for the last round)

The storage requirements for a n-bit to m-bit table being $2^n * m$, we can compute the minimum memory needed to store the tables: $288*2^8*32+1728*2^8*4+16*2^8*8 = 508\,\mathrm{kB}$. Regarding the two measurements of security, the table with the smaller white-box ambiguity is the one performing the XOR operation which has a white-box ambiguity of about 2^{48}. It has also the smallest white-box cryptography of about $2^{132.8}$. However, as we will see in the next chapter, some attacks have been found against this implementation reducing the meaning of those numbers.

2.2.3 Perturbated White-Box AES

A totally different approach was taken in 2006 by Bringer et al. following the work of Billet and Gilbert in 2003 who introduced a traceable block cipher.

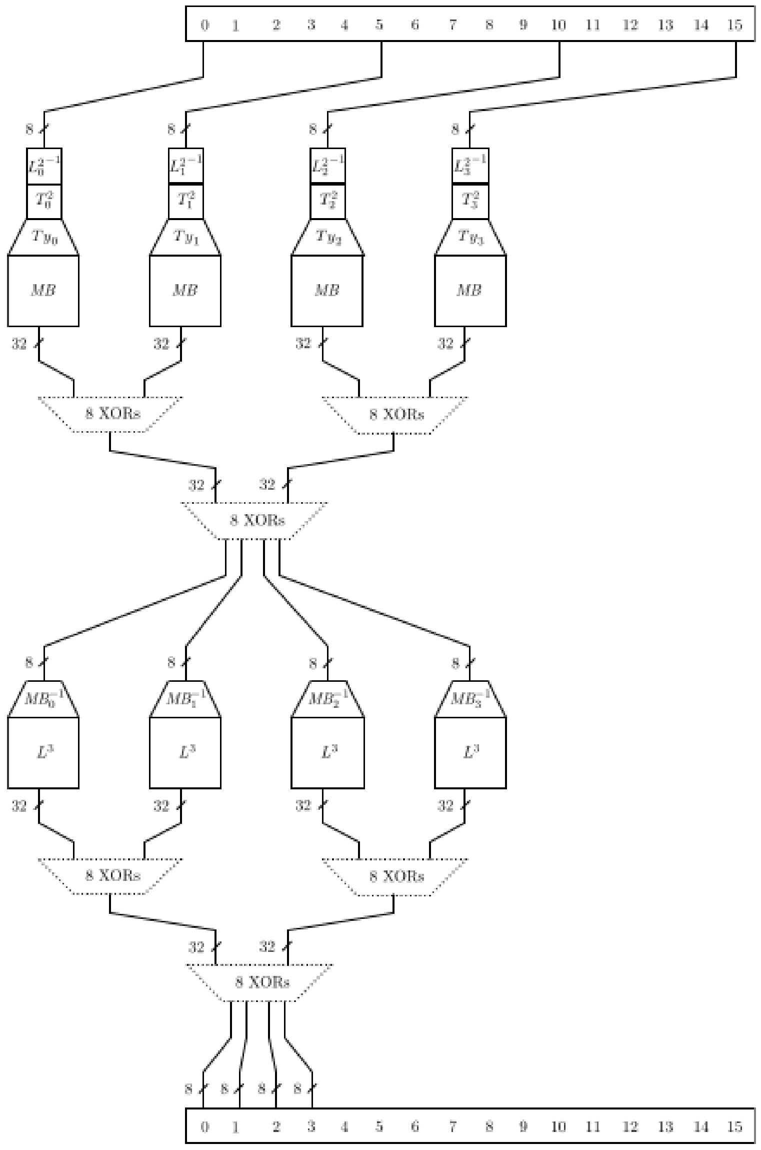

Figure 2.2: Round 2 of Chow's white-box AES implementation from [17]. It is similar to other rounds, round 1 and 10 are differents due to external encodings

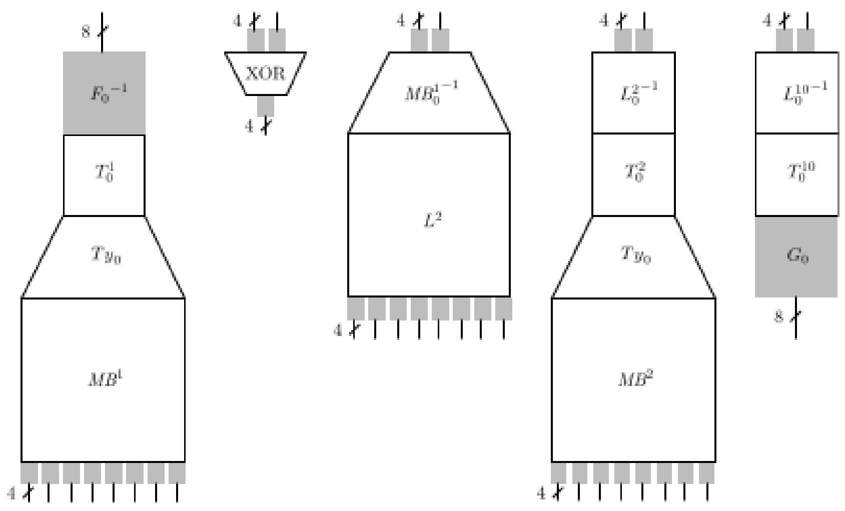

Figure 2.3: The different types of table in Chow's implementation from [17]

[1] In order to strengthen the security of these ciphers, Bringer et al. introduced perturbations to the original equations which lead them to apply the same ideas to an AES implementation in order to make it white-box resistant.[20] They modified the cipher in the following way:

- Random variables are introduced in every round but the last one to hide the interesting information.
- Specific terms taking often a predetermined value are added to the first round and carried away up to the last round.
- Polynomials that canceled when this value is reached are added to the last round.

The result of the evaluation of the polynomial will then be added to the result of the traditional cipher at the end, giving the right result only when the predetermined value is reached. Then a majority voting algorithm is used to determine the right result, the one the original cipher would have returned. For their AES implementation, they realize this by choosing correlated polynomials so that two of them will cancel, while the others will take different values to help attain the predetermined value frequently and so that the right input could always be determined.

[1]A traceable cipher can support a tracing scheme: many equivalent key can be generated from a 'meta-key' to be used for decryption, but it is computationally difficult to find the meta-key or generate another valid equivalent key only by knowing up to k keys, the 'meta-key' allowing to find at least one of the key used to generate the equivalent description. This type of cipher can be used for example to find a 'traitor' for example in the context of broadcasting encrypting content to multiple user like with paying television.[19]

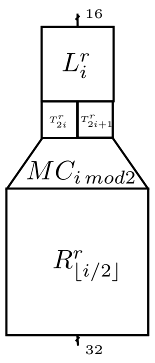

Figure 2.4: The table $Table_i^r$ in Xiao-Lai implementation

2.2.4 Xiao-Lai Implementation

In 2009, another implementation based on the concept of network of lookup tables is proposed by Xiao and Lai [21], but with a different arrangement of the operations into the tables. It is still based on the equivalent algorithm of AES presented in Algorithm 2 . The ShiftRows() operation is now performed as a matrix multiplication with the matrix M^r. It is embedded in the phase cancelling the mixing bijection, which is computed during the encryption and not stored as a lookup table. Another difference is that 2 T-boxes are embedded in one single table whereas in Chow's implementation there was only one by table. The MixColumns operation is again split but differently than in Equation 2.1. Here it is split by columns of 2 as specified in Equation 2.2 There is only one type of table shown in Figure 2.4 in this implementation which does not use non-linear encodings. The flow of the algorithm is shown in Figure 2.5. This implementation is the basis of the one designed by Luo, Lai and You which will be discussed in the chapter 4 as implementing this design was one of the requirements of this thesis. Storing the 11 128*128 matrices and the 8 tables for each of the 10 rounds requires $10 * 8 * 2^{16} * 32 + 11 * 128 * 128 = 20502\,\text{kB}$.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x0 \\ x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} 02 & 03 \\ 01 & 02 \\ 01 & 01 \\ 03 & 01 \end{bmatrix} \begin{bmatrix} x0 \\ x1 \end{bmatrix} \oplus \begin{bmatrix} 01 & 01 \\ 03 & 01 \\ 02 & 03 \\ 01 & 02 \end{bmatrix} \begin{bmatrix} x2 \\ x3 \end{bmatrix} \tag{2.2}$$

The white box diversity of the tables is 2^{1293} and its white-box ambiguity 2^{270}, these measurements being much higher for the matrices.

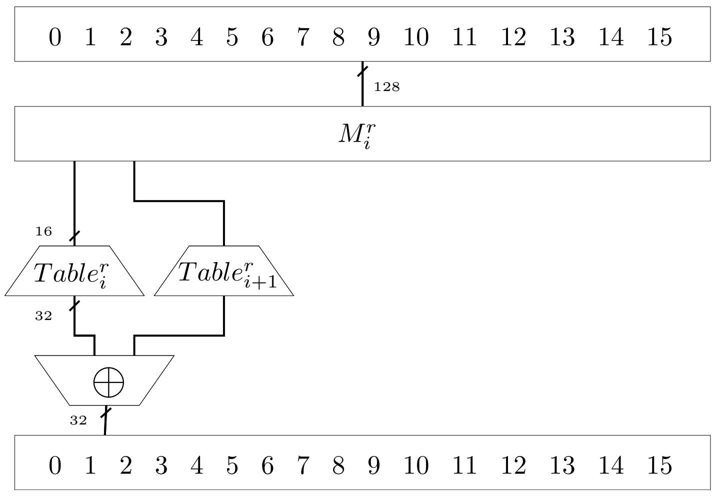

Figure 2.5: Example of one round of Xiao-Lai implementation. The multiplication by M_i^r and XOR operations are performed at run time and not as lookup tables

2.2.5 Karroumi Dual-cipher Implementation

Another white-box AES implementation based on the one designed by Chow et al. was proposed in 2010 by Karroumi [22] and has been implemented and studied in his master thesis by Klinec [4] in 2013. The improvement made to the original design are due to the observations that the operations performed during the AES encryption are simple algebraic operations in $GF(2^8)$ and changing the constants involved (i.e. the irreducible polynomial, the matrix coefficient or the affine transformation involved in the computation of the S-box) allow to construct a new family of ciphers that are tightly correlated to the original AES by linear mappings relating byte of the state of AES cipher to the one of the newly created cipher. These ciphers are called AES dual-ciphers[23, 24], and allow to have a different computation of AES operations, they are used to improve the security of the white-box implementation. For every round of AES, a different dual-cipher is chosen amongst the 61200 possible, the constants involved in SubBytes() and MixColumns() are changed to the one used in the dual cipher and these changes are taken into consideration when expanding the key. Then for each round, the tables performing these operations are modified to use the dual-cipher chosen. This requires encoding and decoding the bytes of the state with the linear mappings relating AES cipher with the dual cipher. The encodings are mixed together with the mixing bijections resulting in tables modified from the original implementation shown in Figure 2.6. This implementation has exactly the same number of tables as the original ones, thus requiring the same storage capacity and the bytes are flowing in the same way.

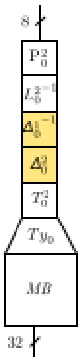

Figure 2.6: Example of a modified table with dual cipher from [4]

2.2.6 Conditional Re-encoding

A recent proposition in 2015 has been made from Lee et al.[25] to protect the AES implementation against the different cryptanalyses developed. It is also based on the original design of Chow et al. and modifies the encoding strategy of the bytes exiting the tables of type XOR in the last part of the round. After the traditional non-linear encoding applied they propose to conditionally encode this result again depending on some defined function deciding from the inputs of the table if the second encoding should be applied or not. They introduce a satellite variable γ in order to track if the second encoding has been applied or not and if it should be cancelled in the next table. They claim that this technique protects against three different cryptanalysis techniques which will be discuss in the next chapter and are the BGE attack, Michiels et al. attack generalizing it and the collision attack.

Chapter 3

Attacks on White-Box Implementations

This chapter will focus on the different attacks developed to break the designs presented in the previous chapter. The emphasis is on key-recovery attack that have been more studied. However it is important to mention that a possible way to circumvent the security of a specific program for plaintext-recovery is to practice code-lifting which means extracting the code or calling the library itself to perform the decryption without having to recover the key. This is the main reason for the external encodings[26].

3.1 Results against Perturbated White-Box AES

In the original paper describing the concept of perturbated white-box [20] , is already mentioned a possible attack against the AES implementation: If the attacker can fix all the bytes of the input of an intermediate round but one, the byte not fixed can be consider as a variable x. Then the computation performed in the round are involving 16 polynomials in x from which a linear combination can leads to the knowledge of SubBytes$(x + v)$. From there it is possible to obtain an overdefined system that can be solved by trying all the values of v, leading to the discovery of one byte of the round key. This attack apparently does not generalize well to white-box AEw/oS [2] implementations also presented in the paper, but other attacks were proved possible based on the fact that the last round does not contain the random equations which can lead to recovery of the different encodings which ultimately leads to the recovery of the key-dependent S-boxes [27].

[2] Advance Encryption without standard S-boxes (AEw/oS) is a generalized AES cipher where the S-boxes are part of the key.

3.2 Algebraic Attacks against Table Lookup Implementations

There have been some attacks developed against the table-lookup implementations as well. One class of these attacks is exploiting algebraic analysis of the tables in order to cancel the encodings and recover the round-key embedded in the tables.

3.2.1 The BGE Attack

The first kind of these attacks was developed in 2004 by Billet, Gilbert and Ech-Chatbi from whom the initials were taken to give a name to the attack: the BGE attack [28]. They noticed that the mixing bijections and input/output encodings were somehow redundant and could be considered as a whole as an 8-bit non-linear encoding, referred to as P and Q respectively for the input and output shown in Figure 3.1. Therefore if we rename T_i' the operation consisting of $T_i \circ P_i$, we get the following relationships for the MixColumns table :

$$\begin{aligned} y_0 &= Q_0(02 \cdot T_0'(x_0) \oplus 03 \cdot T_1' \oplus 01 \cdot T_2'(x_2) \oplus 01 \cdot T_3'(x_3)),\\ y_1 &= Q_1(01 \cdot T_0'(x_0) \oplus 02 \cdot T_1' \oplus 03 \cdot T_2'(x_2) \oplus 01 \cdot T_3'(x_3)),\\ y_2 &= Q_2(01 \cdot T_0'(x_0) \oplus 01 \cdot T_1' \oplus 02 \cdot T_2'(x_2) \oplus 03 \cdot T_3'(x_3)),\\ y_3 &= Q_3(03 \cdot T_0'(x_0) \oplus 01 \cdot T_1' \oplus 01 \cdot T_2'(x_2) \oplus 02 \cdot T_3'(x_3)); \end{aligned}$$

They found that it was possible to recover the output encodings up to an affine transformation using the relations obtained. By fixing x_2 and x_3 to 00 and constructing the lookup table for the two functions giving y_0 with x_1 set to 00 and 01 we can derive the following equations where β_{00} and β_{01} are unknown bytes:

$$\begin{aligned} f_{00}(x_0) &= Q_0(02 \cdot T_0'(x_0) \oplus \beta_{00}),\\ f_{01}(x_0) &= Q_0(02 \cdot T_0'(x_0) \oplus \beta_{01}); \end{aligned}$$

Then we can form another lookup table by composing f_{01} and f_{00}^{-1} which give us an interesting result:

$$\begin{aligned} f_{01} \circ f_{00}^{-1} &= (Q_0 \circ \oplus_{\beta_{01}} \circ 02 \cdot T_0') \circ ((02 \cdot T_0')^{-1} \circ \oplus_{\beta_{00}} \circ Q_0^{-1})\\ &= Q_0 \circ \oplus_{\beta} \circ Q_0^{-1} \end{aligned}$$

with $\beta = \beta_{01} \oplus \beta_{00}$

We can repeat this process to other values of x_1 to obtain elements of a group from which there is an isomorphism to $(GF(2^8), \oplus)$ that can be used

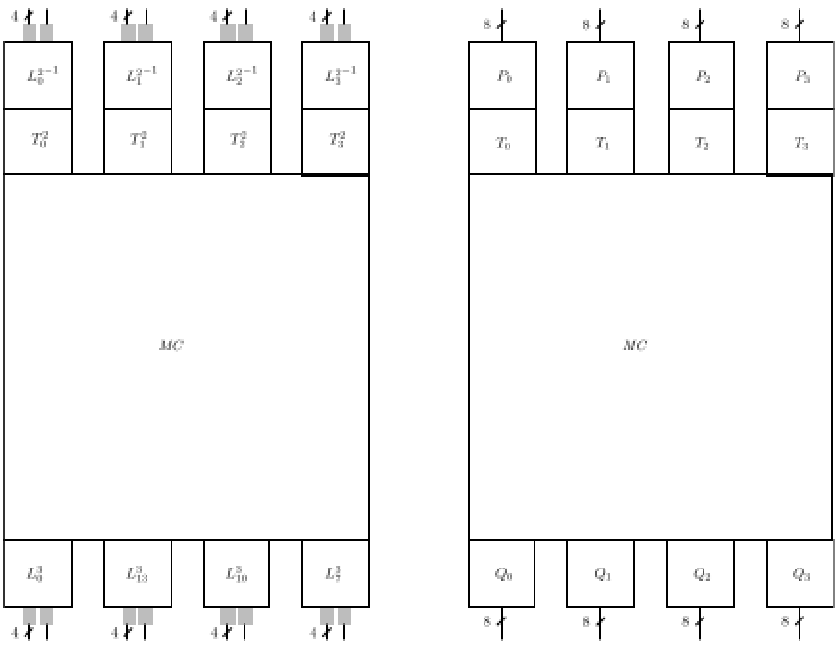

Figure 3.1: How the tables of Chow's implementation are considered for the BGE attack from [17]. The mixing bijection and encodings from the original description on the left are considered as general encodings P and Q on the right.

to form an approximation of each Q_i up to an affine transformation. We can then use the inverse of these new approximations in order to construct lookup tables with weaker (affine) encodings. From these tables it is possible to fully determine the encodings and so to compute the output of the T-box. If we also know the output encodings from the previous round, we can also have its input and so recover one byte of the round key. Performing this on two successive rounds (and so determining the output encodings of three consecutive rounds) allows to fully recovering the key, even on an improved version adding byte permutation to the design. The authors estimate the work factor being about 2^{30}, the most intensive part being to build the approximation of each output encodings (about 2^{24} operations being repeated 16*3 times).

Their attack has been improved in 2012 by Tolhuizen proposing a more efficient algorithm to recover the encoding up to an affine transformation [29] and further improved by Lepoint et al. in 2013, these two improvement reducing the work factor of the attack to 2^{22}. Lepoint et al. developed a method to recover the round key of the round $r+1$ more quickly once the one of the round r has been recovered and to reorder the bytes of the round key. They also introduced a faster way to test if a mapping from $GF(2^8)$ to itself is affine or not by performing an initial test to reduce the number of mappings tested by the original algorithm [30]. This is needed to fully determine the encoding. They and Klinec showed that the design of Karroumi with dual ciphers was also vulnerable to the BGE attack.

The attack was generalized in 2008 on the class of Substitution-Linear Transformation (SLT) [3] cipher with Maximum Distance Separation matrices [4] used for diffusion implemented in the same way as Chow's white-box AES, and is based on the same first phase of recovering the encodings up to an affine part. Then solving linear equivalence problems is needed to recover the key [31].

3.2.2 Collision Attack

Another attack against Chow's implementation was introduced in 2013 by Lepoint et al. exploiting collisions of internal variables on the output of the first round [32, 30]. We denote $S_i^{1,0}$ as the composition of decoding (in the same notation P as the BGE attack) a byte, adding the round key and substituting the byte thanks to the S-box:

$$S_i(\cdot) = S(k_i^{(1,0)} \oplus (P_i^{(1,0)})(\cdot)) \text{ for } 0 \leqslant i \leqslant 3$$

It is possible to inject into the tables of one column values like $(\alpha, 0, 0, 0)$ and $(0, \beta, 0, 0)$, then by inspecting the output of the first byte we can obtain relations about the functions S_i and as they are bijective we obtain exactly 256 pairs (α, β) including the trivial one $(0, 0)$ satisfying the relation:

$$02 \cdot S_0(\alpha) \oplus 03 \cdot S_1(0) = 02 \cdot S_0(0) \oplus 03 \cdot S_1(\beta)$$

Repeating these operations also for different positions of input bits, we can form a system of 4*255 equations with maximum rank 509 (and very probably with this rank according to the results obtained by Lepoint et al.) and 510 unknowns. It is possible to find the coefficients expressing each unknown in relation to one of them. The system can then be solved for the non-trivial solution exploiting some property of the function S_0 that are not satisfied if the value of the unknowns are replaced by a wrong guess. This fact makes the solving equivalent to test the functions with a guessed value of the unknowns to know if they are equals to the null function or not.

Once the function S_i has been recovered it is possible to recover the output encodings Q_i of the first round, that leads to discover the input encodings of

[3]Substitution-Linear Transformation cipher is a class of cipher composed of multiple rounds consisting in a bijection on $GF(2^n)$. This bijection being the composition of a XOR operation with a round key followed by a substitution by a non-linear S-box and then a multiplication of the result with an invertible matrix for diffusion. AES and Serpent are two examples of SLT ciphers.

[4]A Maximum Distance Separation matrix represents a function with some diffusion properties and is often used in cryptography

the second round. Exploiting the same property of a given function involving the key that is not verified with wrong value of the key, it is possible to recover the round key by testing the nullity of different functions.

The most work-intensive part of the attack is the total recovery of the function S_i which requires about 2^{20} operations. As it must be done for each column of the AES state, this leads to a total work factor of $4 * 2^{20} = 2^{22}$.

3.2.3 Cryptanalysis of Xiao-Lai White-Box Implementation

The design of Xiao and Lai (see Section 2.2.4) was made to resist the BGE attack by preventing the possibility to reduce encodings to small parts. This is the reason why their tables are working on 2 T-boxes in order to benefit from the effect of ShiftRows() [21]. However, a cryptanalysis of their implementation was demonstrated in 2012 by De Mulder et al. [33]. The first step of the attack is to recover the input linear encoding by solving a linear equivalence problem as explained below. The TMC tables are transformed in order to be key-independent tables noted $\overline{TMC}$ and related to the concatenation of two modified S-boxes $\bar{S}$ embedded in the table by two linear applications A and B in the way that:

$$\begin{aligned}
&\overline{TMC} = B \circ \bar{S} \parallel \bar{S} \circ A \\
&\text{ where } A = L_i, B = R_{\lfloor i/2 \rfloor} \circ MC_{imod2} \\
&\text{ and } \bar{S} \parallel \bar{S} \text{ is the concatenation of the modified S-boxes}
\end{aligned}$$

Solving the Linear Equivalence (LE) problem here means to recover A and B. De Mulder et al. provide an algorithm in order to do that, based on the one by Biryukov et al. [24]. And so can be recovered the input encodings of the first round.

It is then possible to recover bytes of the shifted round key for the first round by finding the value that give a null result x_0^i from the TMC table which respects the formula:

$$x_0^i = (L_i^1)^{-1}((\hat{k}_{2i}^1 \oplus 52) \parallel (\hat{k}_{2i+1}^1 \oplus 52))$$

Reversing the ShiftRows operation on the modified round key gives the original first round key which is the AES key.

It is even possible to recover the input and output encodings. Once we know the linear input encodings of the first round we can reverse the construction of the first matrix M^1 to find the input encoding via the formula:

$$\hat{k}_{2i}^1 \parallel \hat{k}_{2i+1}^1 = L_i^1(x_0^i) \oplus (52 \parallel 52)$$

And once the input encoding and the key have been found, we can recover the output encoding by decrypting all the base vector of 128 bits output by regular AES, then applying the input encoding to the result and providing this to the white-box AES in order to construct the matrix from the final results y_i:

$$y_i = OUT(AES_k(IN^{-1}(IN(AES_k^{-1}(e_i))))) = OUT(e_i)$$

The phase of the attack requiring the more operations is the algorithm solving the linear equivalence problem to find the input encoding of each tables of the first round. It is of the order of 2^{29} for a single execution. As there are 8 tables in one round, it means that the work factor of the attack is $8 * 2^{29} = 2^{32}$.

3.3 Differential Fault Analysis

Apart from the algebraic attacks exploiting the structure of the lookup tables and their relations in the implementation, it is also possible to use techniques developed in the grey-box model. One possibility is to adapt Differential Power Analysis (DPA). This will be discuss in more detail in the chapter 5. Another possibility is to use Differential Fault Analysis (DFA) as presented in [34]. This approach consists in injecting faults into the execution of the algorithm and analyzing the differences of the results with the expected result under normal conditions. A typical method to inject fault on hardware implementation would be to modify the voltage of reference over a short amount of time.

In order to realize this attack on a white-box software implementation, it is needed that the output of the cipher is not encoded because the "faulty" ciphertext must be compared with the regular one under known conditions. Then the attacker needs to determine the position of the code where to inject the error, in order to do so, he can perform a static and a dynamic analysis of the code or also inject error randomly and observe how it modifies the result. Once this is done, the attacker will want to inject the fault. To do so he can choose among several techniques depending on the software he wants to break, like directly modifying the code if he can lift it in high level language, or use instrumentation tools, a scriptable debugger or a whole system emulator as it is explained in [34].

Different attacks exploiting differential fault analysis on AES have been described in the literature [35, 36], the one presented in [34] that has been applied to white-box implementations and is described in [37] involves injecting a fault in one byte of a column before the last MixColumns operation. After being propagated to the whole column, the 10th round key is added, then the rows are shifted and the last round key is added. This results in expression of the form:

$S(2X \oplus 3B \oplus C \oplus D \oplus K_{10,0}) \oplus K_{11,0} = O'_0$

where B,C,D represent bytes of the state, X is a faulty byte replacing A and O'_0 the modified output

That can be compared to the original output by applying XOR between the two equations, which will lead to a set of relations, for each fault injected of the type:

$$S(2A \oplus 3B \oplus C \oplus D \oplus K_{10,0}) \oplus S(2X \oplus 3B \oplus C \oplus D \oplus K_{10,0}) = O_0 \oplus O'_0$$

One fault allows writing 4 different relations each about a different byte of the key. By repeating these operations multiple times the attacker can recover the bytes of the 10th round key in one column. He can then repeat the process for different columns of the state. The authors of [34] successfully attack different AES (and DES) challenges proposed notably one implementing Karroumi dual-cipher scheme, where the output was available in non-encoded form. They required less than 100 fault injection for each experiment to recover the key.

Chapter 4

AES Implementation of Luo-Lai-You

This chapter will introduce the implementation of Luo, Lai and You that I programmed in C as a part of this project. The programs are publicly available for further improvements or to be used as a test for attacks and can be found on my Github repository[5]. Then the program and its design is explained. Finally the performance and results of the program is examined.

4.1 Presentation of Luo-Lai-You White-Box AES

The design was proposed in 2014 and is largely based on the implementation of Xiao and Lai presented in section 2.2.4 [1]. As before, the reordered AES algorithm presented in Algorithm 2 is used. The tables containing two T-boxes (recall it is the composition of AddRoundKey and SubBytes) and the operation of MixColumns for two columns of the matrix is modified by adding non-linear encodings and is now called nTMC. There are 8 nTMC tables by round, the last round being slightly different as there are two round key in the T-boxes and there is no MixColumns operation. The structure of the tables can be seen in Figure 4.1.

In order to cancel the non-linear encodings a new type of table is introduced as it is no longer possible to simply merge the results. The table essentially performs the same operation as the M^r matrix in the previous implementation with non-linear encodings on top of it. They are called TSR (for table ShiftRows) and they consist of cancelling the mixing bijection, then applying the ShiftRows operation as a matrix form and applying a mixing bijection to cancel the one of the next nTMC table. If the table is at the boundary (i.e. first or last round) it performs the external input decoding or output encoding. As the output of each of these tables is 128-bit long, the

[5]`https://github.com/JeanGa128/AES-whitebox`

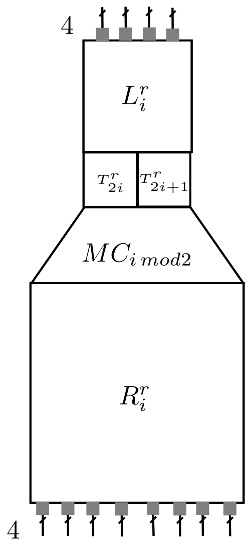

Figure 4.1: nTMC table for round 1 to 9. The last round will not have a part of MixColumn matrix but of the Identity matrix.

output encodings are a block diagonal matrix. The blocks are 16×16 bits long and they are the inverse function of the mixing bijection of the next nTMC table. The input encodings are a strip of the output mixing bijection of the previous nTMC table and are composed of two 32×4 matrices from which the output are added with XOR. That can be represented by concatenating the two 4 columns matrices into an 8 column matrix and the two input nibbles into one byte. One important thing to notice is how a byte of the output of a nTMC table is split between two TSR tables as it is necessary to add the corresponding nibbles that are result of different part of the multiplication with MixColumns. The structure of the TSR table is shown in Figure 4.2.

The outputs of the TSR tables are then reduced using XOR tables in order to form the input of the next nTMC tables which is 16 bytes long and corresponds to the encoded version of the AES state right after each ShiftRows operation. In order to reduce the number of lookup tables during the phase of reduction, a new type of table is introduced as TXOR3 which accepts 3 nibbles as input and exits the nibble corresponding to a XOR operation between the 3 inputs. The flow of bytes in the structure is schematized in Figure 4.3.

This implementation requires $8 * 10 * 32 * 2^{16} + (16 + 10 * 32) * 128 * 2^8 + (160 + 10 * 416) * 4 * 2^8 + (160 + 10 * 288) * 4 * 2^{12} = 28444\,\text{kB}$. Regarding the security of this implementation, the table having the smaller white-box diversity and ambiguity are the TXOR tables with respectively $2^{132.8}$ and $2^{48.2}$. Furthermore, the author claims that it is resistant against the BGE attack as well as the one proposed by De Mulder et al. on the previous Xiao-Lai implementation.

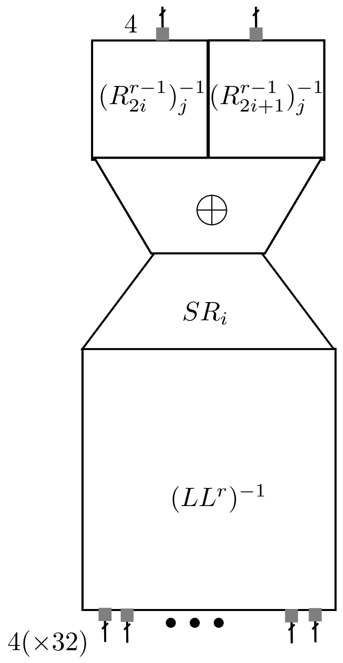

Figure 4.2: TSR table for round 2 to 9. The first round has external input encodings and the last round does not have ShiftRows but also has external encoding.

4.2 Implementation

I wrote the programs in C. Parts of the code implementing a standard AES have been taken on [38] for basic operations in AES field, the S-box and the standard key schedule. The global strategy was to construct one program generating all the tables into a file *table.h* that will be compiled with the program in charge of the encryption, the goal being that there is no information about the key leaking from this second program. The construction process of the final encryption program is shown in Figure 4.4.

4.2.1 Matrices Operations

In order to construct the tables in the program generating them, some matrix operations were needed. Apart from the matrix multiplication, selecting stripes of a matrix and concatenating two matrices, it was necessary to program the inversion of a matrix of $GF(2)$. In order to inverse a matrix I used Gaussian elimination which in $GF(2)$ only requires 2 operations, swapping rows of the matrix and adding a row to another, as there is no other coefficient than 0 and 1 the operation of multiplying by a constant is not needed here and similarly there is no necessity to determine the inverse of an element

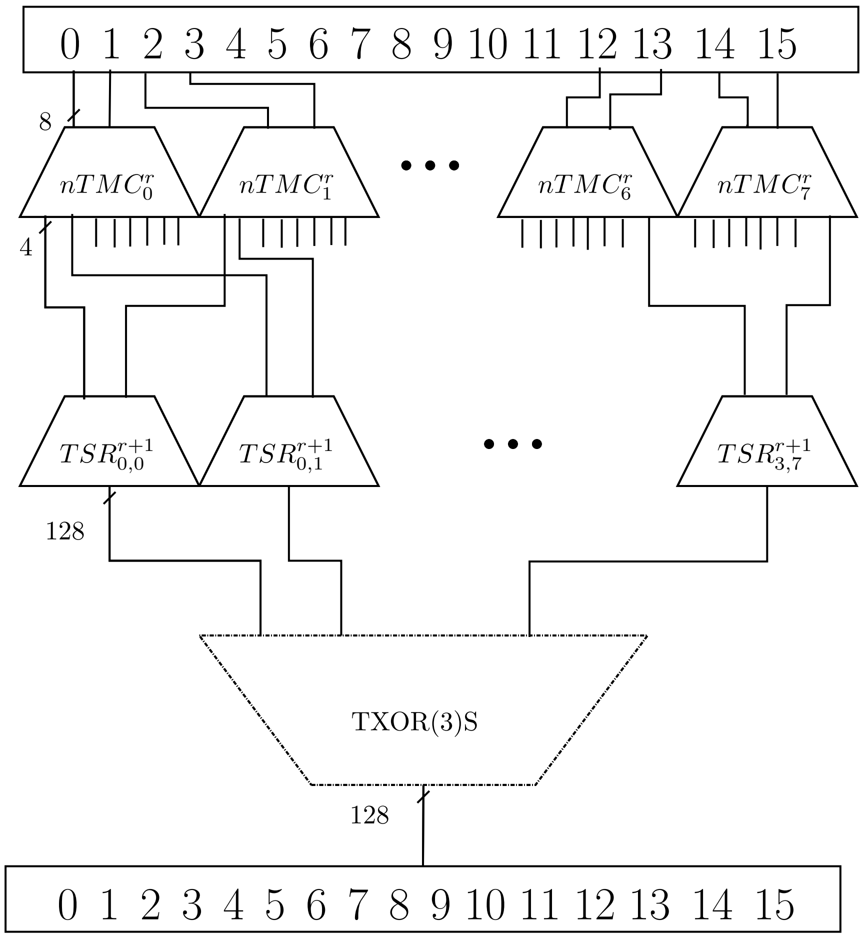

Figure 4.3: Example of bytes flow for one round of Luo et al. implementation. There are 8 $nTMC$ tables feeding to 4×8 TSR tables. TXOR(3)S represents a cascade of TXORs and TXOR3s tables to reduce the output of TSR tables.

in order to find the correct coefficient. The Gaussian elimination is performed on the matrix that need to be inversed while the same operations are performed on a matrix of the same size that started as the identity and will finish as the inverse of the original matrix. The need for the inversion motivated my decision to work considering bits as a prototype implementation as it is easier to write and understand the code (especially testing if a given coefficient is null or equal to one and going over a loop) that if it were more optimized working directly on bytes. However this would be a nice optimization to the code to perform matrix multiplication in the same way they are performed in the encryption program and could speed up the operation that is very used during the generation of the TSR and nTMC tables.

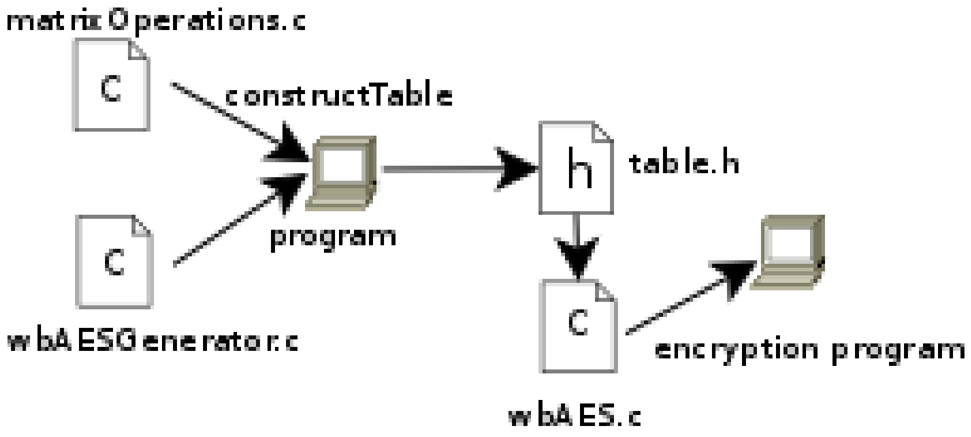

Figure 4.4: The construction of the encryption program

4.2.2 Generation of the Tables

This program realizes the precomputation of the tables that will be used during runtime by the encryption program. The strategy I used concerning the construction of the tables in the generation program was to define all of them as global variables, to construct them, then encode them and record them into the file table.h that will be used for compilation of the encryption program. As the number of TSR tables in the first round is different than in the rest of the algorithm they were given a special variable and so were the TXOR and TXOR3 tables for the same reason. It means there is one variable containing TSR tables of round 1, one containing the remainding TSR tables, one containing the nTMC tables, one containing the TXOR tables for round 1, one containing the other TXOR tables, one containing the TXOR3 tables for the round 1, and one containing the rest of TXOR3 tables. The different mixing bijections and encodings (internal and external) are also defined as global variables that are constructed before they are needed in the construction or modification of the tables. The random permutations were generated using Fisher-Yates shuffle algorithm presented in [39] and in its modern version in [40, 41]. The algorithm is slightly modified to work on an increasing loop and is shown in Algorithm 3. It is applied on an array containing an ordered sequence from 0 to 15. The inverse permutations were computed by directly looking at the value given by an index i and storing i it at the position of this value in the inverse table.

An crucial part of the work was to connect the number of the table (ordered by their indexes) with the correct input encodings once the output encodings of the previous tables have been determined. It was indeed an important part of the task as there are 18 784 different encodings if the external encodings are included. This fact led to several equation present in the function

Algorithm 3 Adaptation of the Fisher-Yates shuffle algorithm used to generate the encodings with an array initialized to 0...n-1

```
Shuffle(array[n])
int j
byte temp
for i = 0 step 1 to n-2 do
  j = rand() %(n-i)                # r andom number between 0 an n-i
  temp = array[i]
  array[i] = array[i+j]
  array[i+j] = temp
end for
```

encodeEverything(). The way encodings are affected to one specific nTMC tables is demonstrated as an example in algorithm 4. The table number is determined by the round and the index of the table inside the round *i*. The number of the encodings are determined with an equation with 3 terms added: the “offset” of the first tables and inside the round (offset is meant here as the number of encodings previously used), the number of tables per round times the number of the current round (starting numbering the rounds at 0) and the part that is dependent of the index. As said before there is choice in the last part when deciding output encodings, but the next input encodings has to be chosen carefully to cancel the correct encoding. It is possible to separately enable or disable the application of mixing bijections, encodings and external encodings by modifying the values of the *#define* on top of the file.

Algorithm 4 Affectation of encodings to nTMC tables

```
affectEncodingsnTMC()
int nTMCNumber
for round = 0 step 1 to 9 do
  for i = 0 step 1 to 7 do
    nTMCNumber = 8*round+i    # The number of the table concerned
    for k = 0 step 1 to 3 do
      nTMCInIndexes[k]=832+1792*round+4*i+k
                  # The indexes of input encodings, 832 is the “offset”,
      1792 the number of encodings per round and 4*i+k the specific part
      to the table
    end for
    for k = 0 step 1 to 7 do
      nTMCOutIndexes[k]=832+1792*round+8*i+k
    end for
    encodenTMC(nTMCNumber,nTMCInIndexes,nTMCOutIndexes)
  end for
end for
```

Operating system:	Debian GNU/Linux 8.3
Processor:	Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
Memory:	3590MB

Table 4.1: Configuration of the virtual machine

4.2.3 Encryption

The program that performs the encryption starts by processing the 16 bytes input, encode it, performs the encryption then decode the output so it would be the one given by the standard AES. Here the matrix multiplication needed for the input encoding and output decoding are performed on the bytes using bitwise AND for multiplication and bitwise XOR for addition (the operation are on elements of $GF(2)$), then the parity of the byte is computed which is in fact the result of XOR between all bits. Different variables were used to represent the input and output of the different phases of lookup tables depending on the required length to store the data. Here again it is important to follow the bytes flow to plug the correct input to a table as shown for example with the preparation of the input of the TSR table in the variable inputStage1_2 that takes one nibble from the output of one nTMC table and another one from a different table. An additional example is the reduction with TXOR tables where it is important that these are used with the same numbering used to encode them during the generation of the tables by the first program.

4.3 Results and Performance

The tests were conducted in a virtual machine with the configuration describe in Table 4.1. The first thing to see is that the program encrypts correctly the data the same way AES would do which is indeed the case.

The article in which the implementation was introduced specifies that the memory space required to store all the tables is 28444kB. The file table.h which contains the tables as well as the external encodings is much bigger than that with about 124 MB due to inefficiency when storing the data as text format, however once compiled the size of the program is nearly 36 MB. The difference with the 28 MB can be explained by the fact that in my program the values of TXOR and TXOR3 tables are stored as full bytes when they are only 4-bits long. If we compute again the storage requirements with this fact in mind, the result obtained is: $8*10*32*2^{16}+(16+10*32)*128*2^{8}+(160+10*416)*8*2^{8}+(160+10*288)*8*2^{12}=35064\,\text{kB}$ which is consistent with the size of the program.

The second compilation requires some time and resources, but it can be considered a precomputation step and its performance is not as crucial as the

Regular (Software) AES	0.68 ms
White-box AES of Luo et al.	1.5 ms

Table 4.2: Comparison of running time with software AES

encryption program one. To measure the performance of this one, I compared its execution time with the one of the regular AES found on [38] which is a simple classical software implementation of AES. Both program were run with random input 1000 times and the execution time was measured using the time built-in shell command and adding the user and system time. The results were in this way averaged over 1000 measurements. They can be seen in Table 4.2. The white-box program will obviously perform worse against more optimized version of AES or hardware implementation.

Chapter 5

Differential Computation Analysis

As said previously, one possibility to attack white-box implementation is to adapt the techniques that have been developed in the grey-box attack context. Another possibility than differential fault analysis presented in section 3.3 is to adapt Differential Power Analysis (DPA) to the white-box context with the so-called Differential Computation Analysis (DCA). This chapter will focus on explaining the principle of differential power analysis and how it can be applied to white-box implementations before looking at the first results obtained against the implementation presented in the previous chapter.

5.1 Differential Power Analysis

The differential power analysis is described in [42, 43]. It is based on the fact that most of the implementations of cryptographic systems are made on electronic devices and these devices may consume a different amount of power depending on the operation they are performing. This observation leads to the idea that if we record the power consumption during the execution of an algorithm in a so-called power trace, we can deduce information about the operations that were performed. This would be called Simple Power Analysis (SPA) and it is efficient if the flow of instruction is dependent of the key. But in the case there is no key-dependent branching in the algorithm, it is necessary to exploit how the power is depending on the data that are manipulated. However the change of data in the power consumption is often smaller and the analysis can be perturbed by the noise. That is the reason why it is necessary to apply statistical techniques in order to reduce the influence of the noise on the result.

In its form presented in [42] DPA needs a selection function $D(C, b, K_s)$ that models the system and says if the bits b should take the value 1 or 0 depending on the ciphertext C and the guessed key K_s. The difference of average power between when b should take the value 1 and when it should take the value 0 is computed. This value should tend to 0 when the guessed key is wrong and the sample of traces is big enough because the selection function will be uncorrelated to the power measurement.

A variant of DPA attack, originally called Correlation Power Analysis (CPA), requires forming a power consumption model depending on the data processed in order to compare it with the real measurement. The most common models are the Hamming weight that computes the number of “1” in a byte (it seems logical that 1 being represented by a higher voltage the more bits set to one, the more power the device operating on the byte will consume) and the Hamming distance that correspond to the bits that changed between two values (or the Hamming weight of the result of XOR on the two values).

Once this is done, it is necessary to find a point in the algorithm where data depending on a small part of the key and on a known quantity (plaintext or ciphertext that needs to be recorded at the same time during the attack) in an easily predictable way so we can make a guess what this value should be for every trace if we suppose a value for the part of the key. Then a correlation is performed between the power measurements recorded and the guessed value for the point previously mentioned for every possible value of the part of the key we want to determine. That should indicates which is the real value of the part of the key and also the moment in time when the value is computed or used. This approach is applied to attack DES and AES for example in [44]. In order for the attack to succeed it is important that the traces are sufficiently aligned (that the same operations are performed at the same time) but some techniques can be used to circumvent this problem, sometimes at the price of needing more traces.

5.2 Adaptation to Memory Trace

A methodology has been described by Bos et al.[3] in order to apply similar techniques in the white-box context, and Sanfelix et al. [34] also described how to perform such attacks. In this context, we can directly access the data that have been computed by recording the memory operations that are made during the execution. It is possible to extract information by looking at the values that are written or read to the stack, or looking at the lower bits of addresses that vary with indices when a lookup table operation is done. Such information can be obtained using many different tools like debugger scripting, binary instrumentation tools (PIN, Valgrind) or system emulation recording (QEMU, unicorn)

The method starts with disabling Address Space Layout Randomization (ASLR) which randomizes the address space of the executable, its data, heap, stack and libraries. This is done in order to have the traces aligned in memory.

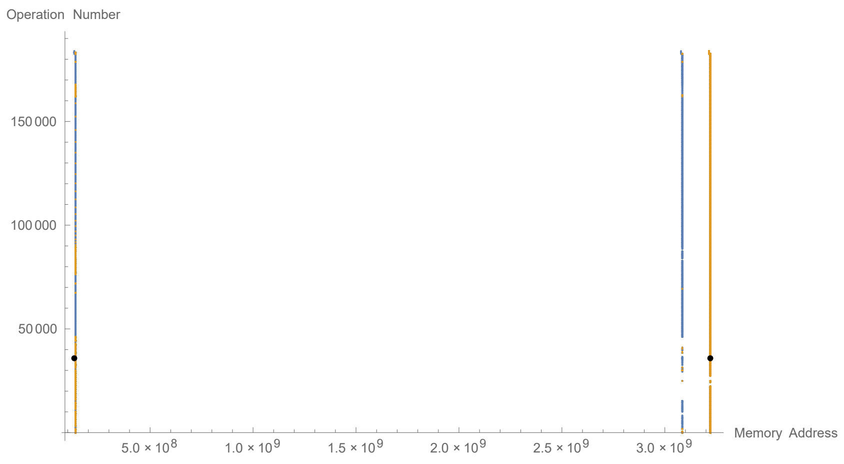

Figure 5.1: Visualization of a simple memory trace of an assembly program

Then a single trace is recorded in order to analyze when the block cipher is used and it can help determine which one is used (for example the number of round). Visualization can be useful for that if it is possible to examine correctly the memory space. Then it is needed to record multiple traces with random plaintexts, keeping track of the plaintext or ciphertext for each trace depending if the first or the last round is attacked (it allows to recover the key by small part to attack the border rounds). The end of the description of their method explained how to use classical DPA tools on the traces obtained by serializing the values recorded which means converting the bytes values into series of bits in order to exploit the results on classical DPA software.

I decided to use bochs[45] to record the traces, as it is possible to record the memory operations and to my knowledge it was not used to attempt this type of attacks before. Bochs is a system emulator written in C++. After multiple attempts on already existing images and creating images with different Linux distributions, I created an image of Debian Wheezy that could run well both in VirtualBox and bochs in order to set up correctly and run the different program I wanted to attack. The image needed to be light in order to boot in a reasonable time and run fast enough after, as the slowdown is the biggest problem I encountered with bochs. To ensure that it was possible to use bochs for the purpose I wanted, I traced a very simple assembly program that pushes a value from the program data to the stack then pops it. The results can be seen on figure 5.1 where the horizontal axis represents the memory addresses, the vertical axis represents time increasing upward and the black dots represent the operation when the interesting data were processed. In blue are the read operations and in orange the write (they have been printed above the read).

In order to train, I recorded the execution of the white-box DES program

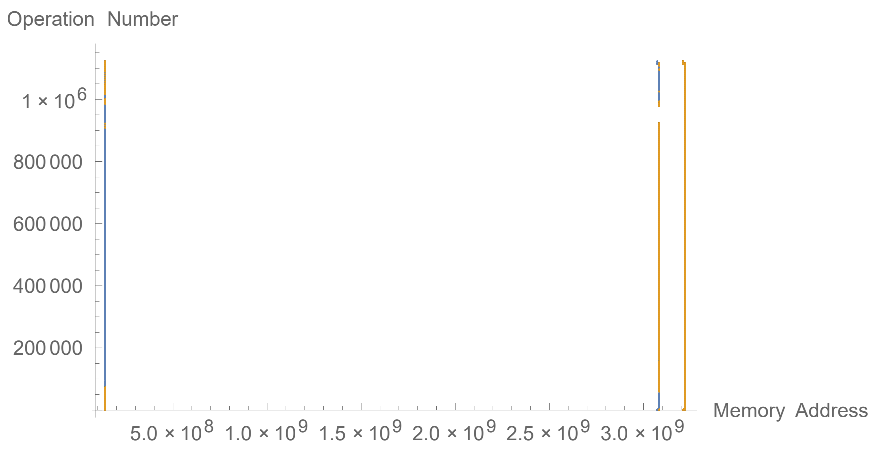

Figure 5.2: Visualization of a memory trace of execution of a white-box DES program

by Wyseur on [46] that was attack in both article mentioned earlier. A visualization of the traces can be seen in Figure 5.2. On the global picture it is possible to see the data and program instructions on the left part, then the heap and the stack on the right. On a zoom on the stack in Figure 5.3 it is possible to observe the 16 rounds of DES as in the article presenting the attack; it is also possible to find the input bytes represented as black dots. The traces were recorded using bochs debugger option "trace-mem on", filtered using the grep command in order to record only the operations performed in the user ring and not in kernel ring (the filter was pl=3) and redirected to a file for saving.

5.3 Demonstration of the Attack on a Non-Protected Implementation

In order to explain the attack on a simple example, a version of AES unprotected against white-box attack obtained on [38] and modified in order to accept command-line input was used. 30 traces were recorded using a script inside the bochs machine that generates and saves a plaintext, performs the encryption and saves the ciphertext, then prints "hell" (32-bit word beginning hello world, used in order to separate the different traces in a following phase) and starts again with a different plaintext. The same method as in the previous section was used to record the trace. The traces were separated in multiple files each time a sequence 0x6C6C6568 (representing "hell") was encountered; the small files were deleted as the sequence appears 5 times in memory for printing it one time. Then the program was inspected to determine the address in the stack in which the first byte of the input was written.

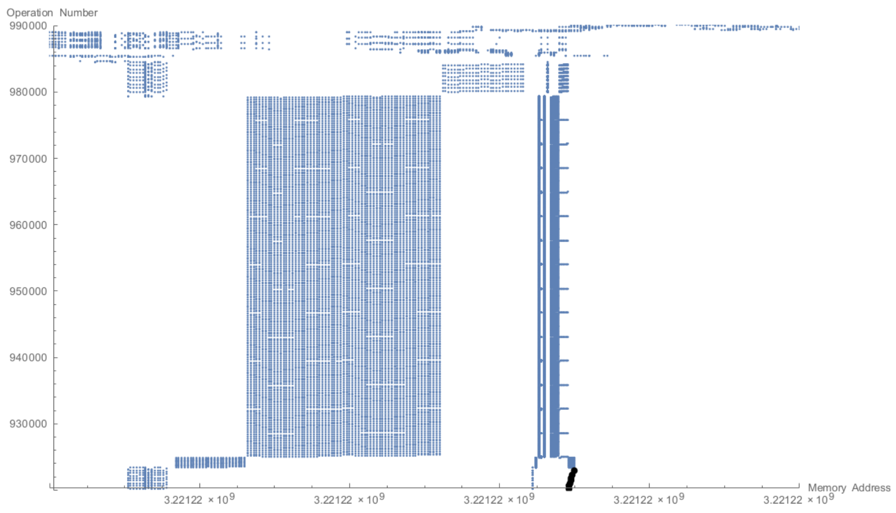

Figure 5.3: Zoom on the stack where the 16 rounds of DES are identifiable and the input can be found just before

This information was used to remove the beginning of the files to keep the interesting part only. Similarly finding the output values made it possible to find an approximate number of memory operations per round; it was then possible to keep only the first round, keeping a margin of security.

The point of the algorithm chosen for attack is classical for AES: the output of the S-Box of the first round, after adding the first round key (which is equal to the key) as showed in Figure 5.4. Mathematica was used to extract the valuable information from the traces and perform the correlation between the key guesses and the values written or read from the memory. As the measurements are exact and without noises, only 4 traces were necessary to find all the bytes of the key correctly. In Figure 5.5 is shown an example of the correlation coefficient for different key guesses on one byte computed with 4 traces and 30 traces. The value returned by the program is the one maximizing the absolute value of the correlation coefficient for one moment in the traces.

It is also possible to try to recover the key by exploiting the value right after adding the round key because it is used as an index to lookup the value from the S-box as shown in the Figure 5.4. The last bits of the memory can be observed as they will be related to this index. This method proved less successful although giving honorable results. With 30 traces, Mathematica returned 4 bytes of the key correctly without ambiguity, for 8 other bytes the right value was amongst the 2 possible values returned, for 2 other bytes it was amongst 4 and for the 2 last bytes the correlation coefficient was printed as 1 but for precision reason it was not returned by Mathematica in the set of possible values, but the exact negation which had a correlation coefficient of

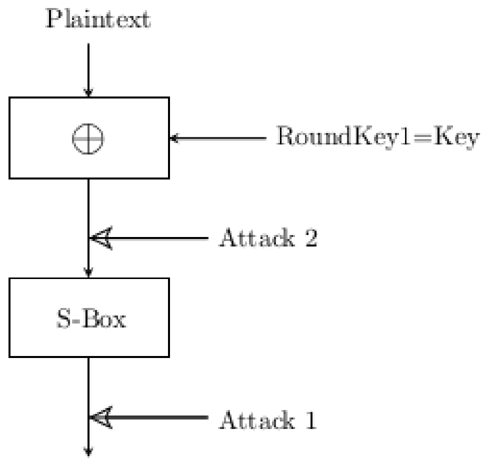

Figure 5.4: Points where the attacks are performed

-1 was returned. Interestingly for the values guessed correctly with only one value returned their negation also had a correlation coefficient of -1 but was not returned by Mathematica for the same precision reason.

5.4 Attempt on Luo-Lai-You Implementation

Unfortunately bochs was not suited to perform the attack on the two implementation of AES presented in the two articles. As said before, the major problem encountered with bochs was its lack of speed. One of the programs required a graphical environment to run and it was extremely long only to start the xserver inside bochs without enabling the tracing of memory which slow down bochs even more. The other program was the one written by Klinec in [4] and could be run perfectly in bochs. However when the recording of traces was turned on it proved to be extremely long to record only one execution; and even if another problem encountered with storage of the traces in the memory could have been avoided with a finer filtering, the duration was a serious problem to record 160 traces needed in [3] let alone the 2000 traces used in [34].

Fortunately, the C implementation described in chapter 4.2 was light enough to record a meaningful number of traces in order to attempt the attack on the implementation of Luo, Lai and You. As the implementation differs significantly from the other one from the perspective of DPA attack, a first phases was performed against an unprotected version of the program: the mixing bijections and encodings were set to identity using the *#define* at the beginning of the program generating the tables. In this implementation, the first appearance of a key-dependent value is after the first nTMC table where

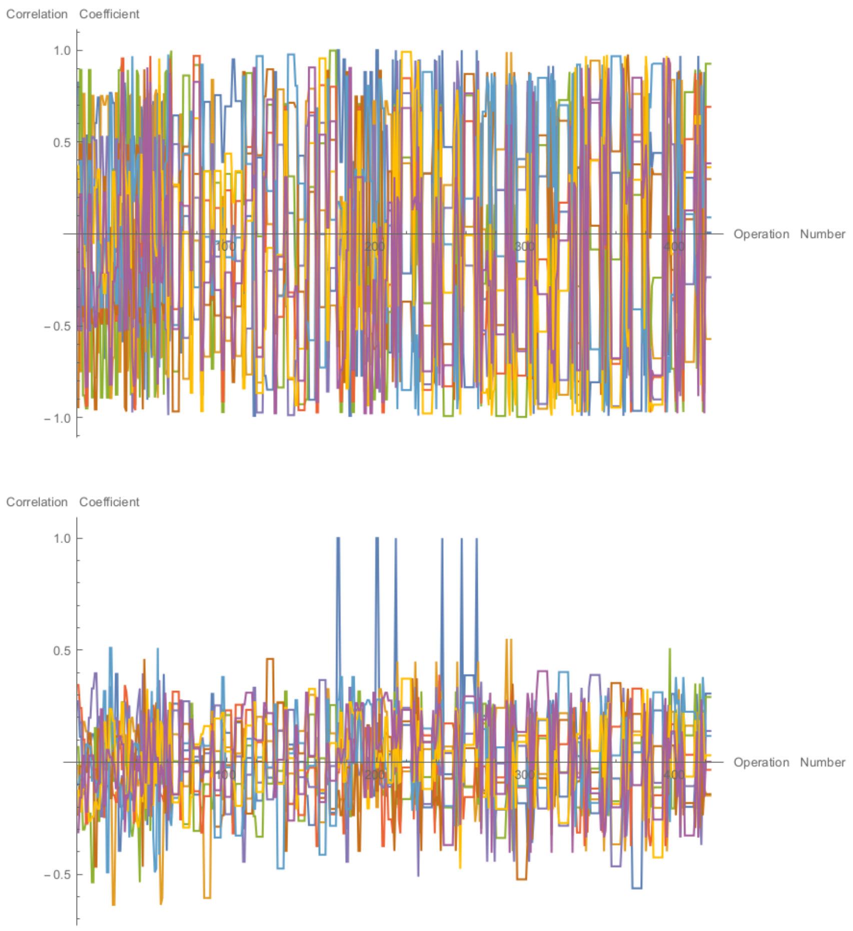

Figure 5.5: Examples of correlations for one byte of the key for different guesses. The top one has been computed with 4 traces, the bottom one with 30

the key is embedded inside the T-Boxes used during the construction of the table. However, here two bytes of the key are used in the same table and are already mixed together within two columns of the MixColumns operation. This makes the job of the attacker more difficult as this situation is not typical for studying AES and now the hypotheses on the value of the key have to be made on 2 bytes which means 2^{16} possible values instead of 2^8. The model that will be used for correlation also needs to be changed as here the targeted point is the output of the first nTMC table which would correspond to an intermediate computation in MixColumns process. I chose to attack the simplest line consisting of two 1 and present in both halves of the matrix. The value computed from the plaintext for each possible value for 2 bytes of the key will then be the XOR of the result of two AddRoundKey and SubBytes operation, the first ShiftRows being taken in account by carefully choosing the plaintext bytes involved.

Again 30 traces have been recorded and the same operations as in the previous section have been made to separate the traces in different files, and align them on the first operation copying a byte of the plaintext on the stack. But by the time the interesting operations are performed the traces were misaligned. In order to realign them and also to reduce the number of points examined during the correlation computation, a filter was applied on the addresses to keep only the lower ones corresponding to the data of the program where the lookup tables are saved. The attack was performed using the values read from the memory. Identically as in the previous section the measurements are exact and the guessed value really appears in the traces; so 6 traces were enough to determine all the values. Performing the attack on a more protected version proved to be harder as I was not able to recover the key with 30 traces and probably more traces would be needed to achieve a result, but it would require more time to capture a significant amount of data with bochs. It would be interesting to study if the fully protected version also leaks the same kind of information that would allow a recovery of the key or if the encodings and mixing bijections are sufficiently obfuscating the intermediate results to prevent the realization of this kind of attack. However I would not advice to use bochs for this purpose because although it is a really powerful tool from which I exploited only a very small part, the lack of speed can be a real issue if the program is too big or if the number of trace needed is high, which is the case with more protected program.

Conclusion

In this thesis several existing attempts of white-box attack resistant implementations have been presented along with different types of attacks against these implementations. Apart from the latest implementations that have therefore not been studied for long (Luo-Lai-You design and conditional re-encoding), all have been proven to be broken in the literature and the principle of the attacks have been explained here.

I wrote one program in C in order to generate the tables for the design of Luo et al. and one other also in C to use these tables to perform the encryption. These programs can be used for future analysis of the security of this implementation particularly concerning differential computation analyses and differential fault injection. Indeed if the algebraic analyses can be performed from the description of the implementation, it is necessary to have a software implementation of the design to test its resistance against these techniques adapted from side-channel attacks.

I made a first analysis of the program using bochs to record memory traces. It has been noticed that two bytes of the key are involved for the first time in the computation at the same moment, making the job of the attacker more difficult than in traditional cases for DPA on AES. Even if only the unprotected version without mixing bijections and encodings has been broken, this is not enough to say that the global scheme is secure against this type of attacks as I had issues recording a bigger number of traces that would be needed to attack it. I would recommend for this purpose using more suited tools than bochs, one that could record the traces faster as it was the issue for me. However a big number of traces would results in longer time to perform the correlation with the issue that 65536 (2^{16}) hypotheses for the key should be tested against the usual 256 (2^8).

Further work could consist on attempting to break the fully protected program with either DFA or DCA or study the global security of the implementation. The other recent scheme with conditional re-encoding could also be studied to test and compare its resistance against this type of attacks. Unlike the design of Luo et al. this proposal involves only one byte of the key at a time like the original design by Chow.

Bibliography

[1] Luo, R.; Lai, X.; You, R. A new attempt of white-box AES implementation. In *Security, pattern analysis, and cybernetics (SPAC), 2014 international conference on*, IEEE, 2014, pp. 423–429.

[2] NIST, A. Advanced encryption standard. *FIPS Publication*, volume 197, 2001.

[3] Bos, J. W.; Hubain, C.; Michiels, W.; et al. Differential computation analysis: Hiding your white-box designs is not enough. Technical report, Cryptology ePrint Archive, Report 2015/753, https://eprint. iacr. org/2015/753, 2015.

[4] Klinec, D.; et al. *White-box attack resistant cryptography.* Dissertation thesis, Master's thesis, Masaryk University, Brno, Czech Republic, 2013. https://is. muni. cz/th/325219/fi_m, 2013.

[5] Stinson, D. R. *Cryptography: theory and practice.* CRC press, 2005.

[6] Menezes, A. J.; Van Oorschot, P. C.; Vanstone, S. A. *Handbook of applied cryptography.* CRC press, 1996.

[7] Robshaw, M. J. Stream ciphers. 1995.

[8] Bellare, M.; Rogaway, P. Introduction to modern cryptography. *UCSD CSE*, volume 207, 2005: p. 207.

[9] Block cipher mode of operation. `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`, accessed: 2016-09-05.

[10] Jean, J. *Cryptanalyse de primitives symétriques basées sur le chiffrement AES.* Dissertation thesis, Citeseer, 2013.

[11] Barenghi, A.; Breveglieri, L.; Koren, I.; et al. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, volume 100, no. 11, 2012: pp. 3056–3076.

[12] Chow, S.; Eisen, P.; Johnson, H.; et al. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, Springer, 2002, pp. 250–270.

[13] Joye, M. On white-box cryptography. *Security of Information and Networks*, 2008: pp. 7–12.

[14] Michiels, W.; Gorissen, P. Mechanism for software tamper resistance: an application of white-box cryptography. In *Proceedings of the 2007 ACM workshop on Digital Rights Management*, ACM, 2007, pp. 82–89.

[15] Barak, B.; Goldreich, O.; Impagliazzo, R.; et al. On the (im) possibility of obfuscating programs. In *Advances in cryptology—CRYPTO 2001*, Springer, 2001, pp. 1–18.

[16] Wiener, M. J. Applying Software Protection to White-Box Cryptography. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, ACM, 2015, p. 1.

[17] Muir, J. A. A Tutorial on White-box AES. In *Advances in Network Analysis and its Applications*, Springer, 2012, pp. 209–229.

[18] Shannon, C. E. Communication theory of secrecy systems*. *Bell system technical journal*, volume 28, no. 4, 1949: pp. 656–715.

[19] Billet, O.; Gilbert, H. A traceable block cipher. In *Advances in Cryptology-ASIACRYPT 2003*, Springer, 2003, pp. 331–346.

[20] Bringer, J.; Chabanne, H.; Dottax, E. White Box Cryptography: Another Attempt.

[21] Xiao, Y.; Lai, X. A secure implementation of white-box AES. In *Computer Science and its Applications, 2009. CSA '09. 2nd International Conference on*, IEEE, 2009, pp. 1–6.

[22] Karroumi, M. Protecting white-box AES with dual ciphers. In *Information Security and Cryptology-ICISC 2010*, Springer, 2010, pp. 278–291.

[23] Barkan, E.; Biham, E. In how many ways can you write Rijndael? In *Advances in Cryptology—ASIACRYPT 2002*, Springer, 2002, pp. 160–175.

[24] Biryukov, A.; De Canniere, C.; Braeken, A.; et al. A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In *Advances in Cryptology—EUROCRYPT 2003*, Springer, 2003, pp. 33–50.

[25] Lee, S.; Choi, D.; Choi, Y.-J. Conditional Re-encoding Method for Cryptanalysis-Resistant White-Box AES. *ETRI Journal*, volume 37, no. 5, 2015: pp. 1012–1022.

[26] Brecht, W. White-box cryptography: hiding keys in software. *NAGRA Kudelski Group*, 2012.

[27] De Mulder, Y.; Wyseur, B.; Preneel, B. Cryptanalysis of a perturbated white-box AES implementation. In *Progress in Cryptology-INDOCRYPT 2010*, Springer, 2010, pp. 292–310.

[28] Billet, O.; Gilbert, H.; Ech-Chatbi, C. Cryptanalysis of a white box AES implementation. In *Selected Areas in Cryptography*, Springer, 2004, pp. 227–240.

[29] Tolhuizen, L. Improved cryptanalysis of an AES implementation. In *Proceedings of the 33rd WIC Symposium on Information Theory in the Benelux, Boekelo, The Netherlands, May 24–25, 2012*, WIC (Werkgemeenschap voor Inform.-en Communicatietheorie), 2012.

[30] Lepoint, T.; Rivain, M.; De Mulder, Y.; et al. Two attacks on a white-box AES implementation. In *Selected Areas in Cryptography–SAC 2013*, Springer, 2013, pp. 265–285.

[31] Michiels, W.; Gorissen, P.; Hollmann, H. D. Cryptanalysis of a generic class of white-box implementations. In *Selected Areas in Cryptography*, Springer, 2008, pp. 414–428.

[32] Lepoint, T.; Rivain, M. Another Nail in the Coffin of White-Box AES Implementations. *IACR Cryptology ePrint Archive*, volume 2013, 2013: p. 455.

[33] De Mulder, Y.; Roelse, P.; Preneel, B. Cryptanalysis of the Xiao–Lai white-box AES implementation. In *Selected Areas in Cryptography*, Springer, 2012, pp. 34–49.

[34] Sanfelix, E.; Mune, C.; de Haas, J. Unboxing the White-Box.

[35] Tunstall, M.; Mukhopadhyay, D.; Ali, S. Differential fault analysis of the advanced encryption standard using a single fault. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, Springer, 2011, pp. 224–233.

[36] Kim, C. H.; Quisquater, J.-J. New differential fault analysis on AES key schedule: two faults are enough. In *Smart Card Research and Advanced Applications*, Springer, 2008, pp. 48–60.

[37] Dusart, P.; Letourneux, G.; Vivolo, O. Differential fault analysis on AES. In *Applied Cryptography and Network Security*, Springer, 2003, pp. 293–306.

[38] kokke. Small portable AES128 in C. `https://github.com/kokke/tiny-AES128-C`, accessed: 2016-09-05.

[39] Fisher, R. A.; Yates, F.; et al. Statistical tables for biological, agricultural and medical research. *Statistical tables for biological, agricultural and medical research.*, , no. Ed. 3., 1949.

[40] Durstenfeld, R. Algorithm 235: random permutation. *Communications of the ACM*, volume 7, no. 7, 1964: p. 420.

[41] Knuth, D. E. The art of computer programming II. 1997.

[42] Kocher, P.; Jaffe, J.; Jun, B. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, Springer, 1999, pp. 388–397.

[43] Kocher, P.; Jaffe, J.; Jun, B.; et al. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, volume 1, no. 1, 2011: pp. 5–27.

[44] Hnath, W. *Differential power analysis side-channel attacks in cryptography.* Dissertation thesis, Worcester Polytechnic Institute, 2010.

[45] Bochs. `http://bochs.sourceforge.net/`, accessed: 2016-11-05.

[46] WhiteBoxCrypto. `http://www.whiteboxcrypto.com/challenges.php`, accessed: 2016-11-05.

Appendix **A**

Acronyms

AES Advance Encryption Standard

AEw/oS Advance Encryption without standard S-boxes

ASLR Address Space Layout Randomization

CBC Cipher Block Chaining

CFB Cipher Feedback

CPA Correlation Power Analysis

CTR Counter

DCA Differential Computation Analysis

DES Data Encryption Standard

DFA Differential Fault Analysis

DPA Differential Power Analysis

DRM Digital Rights Management

ECB Electronic Codebook

FIPS Federal Information Processing Standard

GF Galois Field

GIMP GNU Image Manipulation Program

LE Linear Equivalence

MAC Message Authentication Code

NIST National Institute of Standards and Technology

nTMC	Table MixColumns with non-linear encodings
SLT	Substitution-Linear Transformation
SPA	Simple Power Analysis
TSR	Table ShiftRows
TMC	Table MixColumns
XOR	exclusive OR

Appendix **B**

Contents of CD

- `readme.txt` ... the file with CD contents description
- `img` ... the directory with the image of the bochs virtual machine
  - `bochsrc.txt` ... the bochs configuration file
  - `debian_wheezy.img` ... the bochs image
  - `users.txt` ... users and passwords
- `src` ... the directory of source codes
  - `whiteboxAES` ... the directory of source code of the C programs
  - `DCA` ... the directory of data and notebooks of the attacks
    - `preprocessing.sh` ... example script for preprocessing traces' data
    - `StackAnalysis` ... the directory for the assembly program
    - `wbDESAnalysis` ... the directory for the DES program
    - `AESAnalysis` ... the directory for the regular AES
    - `unsafeAESAnalysis` . the directory for unprotected white-box AES
    - `mixingAESAnalysis` . the directory for white-box AES with mixing bijections
  - `thesis` ... the directory of LaTeX source codes of the thesis
    - `figures` ... the thesis figures directory
    - `*.tex` ... the LaTeX source code files of the thesis
- `text` ... the thesis text directory
  - `thesis.pdf` ... the thesis text in PDF format