Chapter 19

Faking Countermeasure Against Side-Channel Attacks

Rubén Lumbiarres-López, Mariano López-García and Enrique Cantó-Navarro

19.1. Introduction

Any cryptographic device is susceptible to being fraudulently attacked for the purpose of obtaining the cryptographic key. Such a key is used to cipher or decipher confidential data that must be protected. Side-channel attacks (SCAs) were proposed by Paul Kocher et al. [1] at the end of the 1990s (they were known as differential analysis of power consumption). These attacks have become a powerful tool that is suitable for obtaining a cryptographic key in a short period of time using low-cost equipment. This equipment basically consists of a data-acquisition set, needed for capturing the current traces that represent the consumed power, and a desk computer that is used for processing the information contained in such traces. These attacks exploit the existing relationship between the energy consumed by an electronic device and the data that are being processed at a specific instant of time. As data are related with the cryptographic key, this analysis could potentially reveal the secret key.

So far, techniques used by engineers to conceal the cryptographic key, usually known as countermeasures, are focused in two different ways: hiding and masking. The aim of hiding is to design systems whose power consumption is constant and independent of data. Yet, masking is intended for devices in which the consumed power depends not only on data, but also on a random mask that is unknown to the attacker. In both cases, the consumed power is not directly linked with the processed data, hence the secret key is safe. Although hiding and masking make a successful attack more difficult, the use of sophisticated algorithms, or the fact of including a high number of traces in the analysis, allows the attacker to breach the security of the system.

Rubén Lumbiarres-López
Electronic Engineering, Universidad Politècnica de Cataluña,
Avda. Victor Balaguer, 08800, Vilanova i la Geltrú, Spain

The countermeasure based on faking represents a paradigm shift towards the protection of cryptographic devices: the aim is not to design a non-vulnerable system, but to profit from the vulnerability of cryptographic devices in revealing a fake key when SCA analysis is performed.

This chapter shows the theoretical basis for applying the faking countermeasure to the AES 128-bit cryptographic algorithm [2]. Initially, such an algorithm is analysed and its weak points for performing a successful SCA analysis are noted. Afterwards, several modifications are proposed and applied for implementing the proposed faking countermeasure. Experimental results were obtained for a software implementation using a Sasebo-GII [3] development board and the set of measures proposed in [4] by E. Oswald.

19.2. Fundamentals

The correlation and the differences-in-means are two of the main techniques used to reveal cryptographic keys by means of SCA analysis [4]. These kinds of attacks do not need any previous knowledge on the internal structure of the system; only the power consumption and the plain-text that is being encrypted need be known. Therefore, it is assumed that a number of T current traces, and their respective T plain-texts that are chosen by the attacker, are available to perform the analysis. A simple expression to know the minimum number of necessary traces can be found in [4].

The consumed power is measured when the device is executing a set of instructions related to a specific part of the cryptographic algorithm. Data related with such a selected set of instructions should have the property that only depends on a few bits of the cryptographic key. Hence, only a subset of the bits that form the key need to be analysed, which makes the processing easier. The attack is based on comparing the actual power consumption of the electronic device with a theoretical model. Such a model is used to predict the expected power that is consumed by the device. As this consumption depends on the key, the model is evaluated for every possible key whose bits may affect the power consumption. This is the reason why subsets of 8 bits are chosen, which reduces the number of guessed hypotheses to 256. The hypothesis whose theoretical model has the higher similarity with the actual power consumed will be the real key [5, 6]. The Hamming weight (HW) and the Hamming distance (HD) are the most used power models [4]. The Hamming weight corresponds to the number of bits set to 1 (or zero) in the data that are being analysed. The Hamming distance is the difference between the number of bits set to 1 (or zero) at two consecutive instants of time t_1 and t_2, respectively. For instance HW(11010001) = 4, whereas HD(11010001, 01011001) = 2.

These attacks are known as first order attacks, because they are focused on a particular point related to a specific instant of time. Second order attacks are more sophisticated, because they are based on processing the power consumption of two points that may take place in two different instants of time. Additionally, second order attacks are able to eliminate the protection provided by masking using a simple logical operation. Indeed, let u and v be two sets of bits that should be masked by using mask m. Then, this operation could be performed as follows:

$$u_m = u \oplus m and v_m = v \oplus m, \quad (19.1)$$

where u_m and v_m are the masked values associated with the intermediate points u and v, and $\oplus$ represents the bit-wise logical exclusive-OR operator. Assuming that the Hamming weight model is accurate for predicting the real power consumption, from (19.1) the following expression could be obtained as:

$$HW(u_m \oplus v_m) = HW(u \oplus m \oplus v \oplus m) = HW(u \oplus v). \quad (19.2)$$

As can be seen, the effect of the mask is removed, since the HW for both masking and non-masking data are identical. Now, the system is potentially vulnerable, because the attacker knows the values of u and v (such values depend on the plain-text) and therefore the attacker is capable of predicting the theoretical power consumed by the device.

The following step is to find a suitable function for processing the power consumed by the pair values (u_m,v_m) produced at the instants of time (C_{ti},C_{tk}), respectively. Such a function should have a significant degree of correlation with the model of power consumption described in (19.1). In [7], a function based on the absolute value of both points is proposed, represented by Eq. (19.3), but other authorspropose the use of products, the square of the sum or even the square of the absolute value [8, 4].

$$F_{pre-processing} = abs(C_{ti} - C_{tk}). \quad (19.3)$$

The success of this process depends on the detailed knowledge held by the attacker of the instants of time in which points (C_{ti},C_{tk}) are produced. When such points are unknown, the process should be applied to all the pairs of values that arise from the combination of all possible points that form the current trace. Therefore, each processed trace consists in $n!$ values, where n represents the number of points included in the original captured traces.

19.2.1. AES Algorithm

The AES 128-bit algorithm [2] consists of 4 basic functions (*AddRoundKey*, *ShiftRows*, *SubBytes* and *MixColumns*) that are executed sequentially during several rounds following the diagram shown in Fig. 19.1. The number of rounds depends on the key-length, being 11 rounds (0 to 10) for the case of AES128-bit. The input and output of each function is a matrix of 4×4 bytes known as state. A basic description of these functions is as follows:

AddRoundKey: It performs the exclusive-OR bit-wise operation between each byte of the state and each byte of the key (its output is St_A).

ShiftRows: This function makes a shift between the bytes that form the rows of the state (its output is St_R).

SubBytes: It makes a SBOX substitution, which is defined in the AES 128-bit algorithm. The most usual way for performing these operations is by using a look-up table that is stored in memory (its output is St_B).

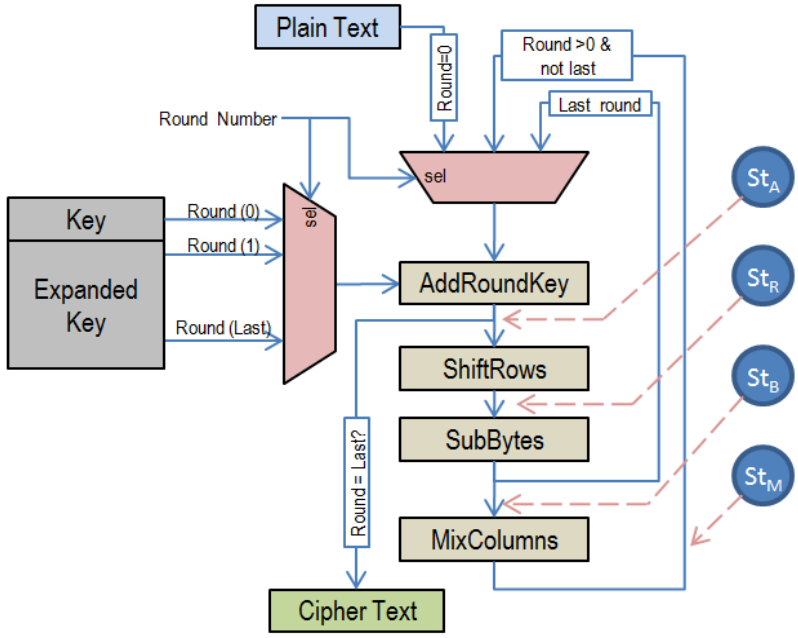

Fig. 19.1. AES algorithm structure.

MixColumns: Each column of state is treated as a polynomial in a Galois finite field, GF(2^8). Each column is multiplied (modulus x^4+1) with the fixed polynomial:

$$C(x) = 3x^3 + x^2 + x + 2, \tag{19.4}$$

or the polynomial:

$$C^{-1}(x) = 11x^3 + 13x^2 + 9x + 14, \tag{19.5}$$

when the cipher text is decrypted (the output of *MixColumns* is St_M).

KeyExpand: is another important function in the algorithm that only operates on the key. The key is subjected to an expansion process by which, starting with the original key, several keys of identical length are obtained. Each of these expanded keys are used in each of the 11 rounds performed during the execution of the AES128-bit algorithm.

19.2.2. AES Vulnerabilities

The fortress of the algorithm increases when the key is scattered through the matrix state. The *Mixcolumns* function performs efficiently the diffusion of the key, since after the first round each byte of the state at the output of *MixColumns* depends on both the 4 bytes of the key and the 4 bytes of the plain-text. This key dependence increases for each round.

This effect can be appreciated in Table 19.1, which represents the variation of the values of the matrix state (bytes B0 to B15) during two rounds (R0 to R2) and in two different

cases. In the first case, a random plain-text was chosen. In the second case, the same text was used, but changing only the less significant bit of byte 15 in the key. The shadowed values represent changes in the state matrix. As can be seen, all bytes of state at the output of function *MixColumns* in round 2 have changed, so that from this point any value of state depends on the complete value of the key.

Table 19.1. AES128-bit algorithm key dispersion.

State bytes																	
B0	B1	B2	B3	B4	B5	B6	B7	B7	B8	B10	B11	B12	B13	B14	**B15**		
15	4	83	250	3	140	142	46	132	46	7	109	99	162	137	**103**	AdKey	R0
118	242	237	45	123	100	25	49	95	49	197	60	251	58	167	**133**	SBytes	R1
118	100	197	**133**	123	49	167	45	95	58	237	49	251	242	25	60	SRow	
0	**111**	**23**	**42**	47	198	104	65	44	54	247	84	197	19	127	133	MCol	
220	**255**	**32**	**154**	180	143	**183**	168	187	200	**133**	107	253	146	**106**	**34**	AdKey	
134	**22**	**183**	**184**	141	115	**169**	194	234	232	**151**	127	84	79	**2**	**147**	SBytes	R2
134	115	**151**	**147**	141	232	**2**	**184**	234	79	**183**	194	84	**22**	**169**	127	SRow	
134	**81**	**110**	**72**	**152**	**248**	**178**	**13**	**107**	**116**	**141**	**66**	**68**	**231**	**138**	**189**	MCol	
84	**152**	**5**	**255**	**209**	**120**	**6**	**83**	**181**	**10**	**75**	**35**	**162**	**24**	**89**	**123**	AdKey	

The conclusion is that the main vulnerability of the algorithm is found in rounds 0 and 1 (it can be demonstrated that the same vulnerability arises in rounds 9 and 10) [5]. In these rounds the key is not scattered through the state and then it is more susceptible to attack.

In the first round, the state at the output of *AddRoundkey* corresponds to the combination of the plain-text with the non-expanded original key. The operation performed is:

$$St_{A(i,j)} = T_{(i,j)} \oplus K_{(i,j)}, \tag{19.6}$$

where $St_{A(i,j)}$ ($i = 1..4, j = 1..4$) represents one of the 16 bytes that form the state, and $T_{(i,j)}$ and $K_{(i,j)}$ refer to the plain-text and the key, respectively. As can be observed in (19.6), the state only depends on one byte of the plain-text and one byte of the key.

Although an attack on such a function can be addressed, it is difficult to discriminate between the true hypothesis and the false hypothesis, since the bit-wise exclusive-OR operator is linear. Consequently, it is expected that the correlations obtained for those hypotheses that have a similar Hamming weight will also be quite similar. Table 19.2 shows the result for the correlation when an attack over byte 1 of the key is performed. The maximum correlation is obtained for the real key (211) and corresponds to $\rho = 0.6692$. However, it is also observed that such a value is very close to other correlations obtained for key hypotheses that have identical or similar HWs.

The function *ShiftRows* only moves the bytes of state, but without performing any operation between them. Thus, any attack on this function is conceptually identical to an attack on *AddRoundKey*.

Table 19.2. Maximum correlations obtained when attacking function *AddRoundKey*.

Key hypothesis and Hamming weight (HW)		Max correlation
209	HW(11010001) = 4	0.6282
210	HW(11010010) = 4	0.6002
211	HW(11010011) = 5	0.6692
196	HW(11000011) = 4	0.6545
148	HW(10010011) = 4	0.6236

The output of function *SubBytes* is widely used as a target in SCAs analysis, because the non-linearity of such a function makes its output very different for key hypotheses having similar HWs. Its definition can be found in [2], and it will be represented by *SBOX*, as follows:

$$St_{B_{(i,j)}} = SBOX(St_{R_{(i,j)}}), \tag{19.7}$$

where $St_{R(i,j)}$ and $St_{B(i,j)}$ refer to the value of matrix state at the output of *ShiftRows* and *SubBytes,* respectively. It can be demonstrated that the elements of matrix state $St_{B(i,j)}$ only depend on one byte of the key, and therefore only a reduced number of hypotheses must be guessed. Table 19.3 represents the value of the correlation for several keys. As can be seen, the maximum correlation is given for the true key (ρ = 0.8143) and is clearly identifiable (one order of magnitude) even against those keys whose HWs are quite similar.

Table 19.3. Maximum correlations obtained when attacking the output of function *SubBytes*.

Key hypothesis		Max correlation
209	HW(11010001) = 4	0.0632
210	HW(11010010) = 4	0.0747
211	HW(11010011) = 5	0.8143
196	HW(11000011) = 4	0.0591
148	HW(10010011) = 4	0.0729

After the calculation of function *MixColumns* in the first round, each value of the matrix state depends on 4 bytes of the key. This means that the number of possible keys to be guessed for predicting the theoretical model is 2^{32}. The *Mixcolumns* operation is applied on the four columns of the matrix state and it is defined as:

$$\begin{pmatrix} St_{M_{(i,1)}} \\ St_{M_{(i,2)}} \\ St_{M_{(i,3)}} \\ St_{M_{(i,4)}} \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} St_{B_{(i,1)}} \\ St_{B_{(i,2)}} \\ St_{B_{(i,3)}} \\ St_{B_{(i,4)}} \end{pmatrix}, \tag{19.8}$$

where $St_{B(i,j)}$ and $St_{M(i,j)}$ represent the bytes of state at the output of *SubBytes* and *MixColumns*, respectively. From (19.8), it can be concluded that the first byte of state can be evaluated as:

$$St_{M(1,1)} = 2 \cdot St_{B(1,1)} \oplus 3 \cdot St_{B(1,2)} \oplus St_{B(1,3)} \oplus St_{B(1,4)}. \quad (19.9)$$

Taking into account that the attacker has full control over the plain-text *L.*, *Pan et al*. in [5] proposed the use of a plain-text in which all of its bytes are identical, except the byte that is the target of the attack. Assuming that in Eq.(19.9) such a byte is $St_{B(1,4)}$, then it is satisfied that:

$$St_{M(1,1)} = Constant \oplus St_{B(1,4)}. \quad (19.10)$$

Note that in (19.10), the expression of $St_{M(1,1)}$ is equivalent to masking the input byte $St_{M(1,1)}$ using a non-variable constant mask. Masking is only an effective technique if the mask changes its value randomly. Thus, although the value of the correlation would be affected, the output of function *MixColumns* will be vulnerable in an identical way as it is function *SubBytes*.

In round 2, the key is completely scattered in such a way that any byte of the matrix state depends on the 128 bits of the key. Thus, any attack in this round or the following rounds is equivalent to performing a brute force attack.

19.3. The FAKING Countermeasure

The basic idea of the proposed faking countermeasure is to carry out the encryption process using a false key. Such a false key is obtained by operating with an exclusive-OR operator, the real key and a mask-key that is randomly chosen:

$$Key_{FAKE(i,j)} = Key_{REAL(i,j)} \oplus Key_{MASK(i,j)}. \quad (19.11)$$

In (19.11), Key_{FAKE} refers to the false key, Key_{REAL} is the true key and Key_{MASK} is the key-mask. All of the keys are organized as a matrix of 4×4 bytes in such a way that they can be properly operated with the state. As the operations of the AES 128-bit algorithm are performed using the false key, and no additional countermeasures are taken, then any SCA analysis will lead to revealing the false key [9]. Fig. 19.2 shows the block diagram for implementing the faking countermeasure.

Note that, the real and the expanded keys are both masked with Key_{MASK}, so that in any round the false key is used for processing function *AddRoundKey*. Afterwards, the encryption follows the usual steps for processing the rest of the functions included in the standard AES128-bit algorithm. Obviously, although the system is effectively protected, if no additional actions are taken then the plain-text will be incorrectly encrypted with the false key. Then, to obtain the expected results the process should be reverted at some point. Functions described in Fig. 19.2 as *SboxTrans* and *MixColumns* are included to remove the faking countermeasure before a new round is started.

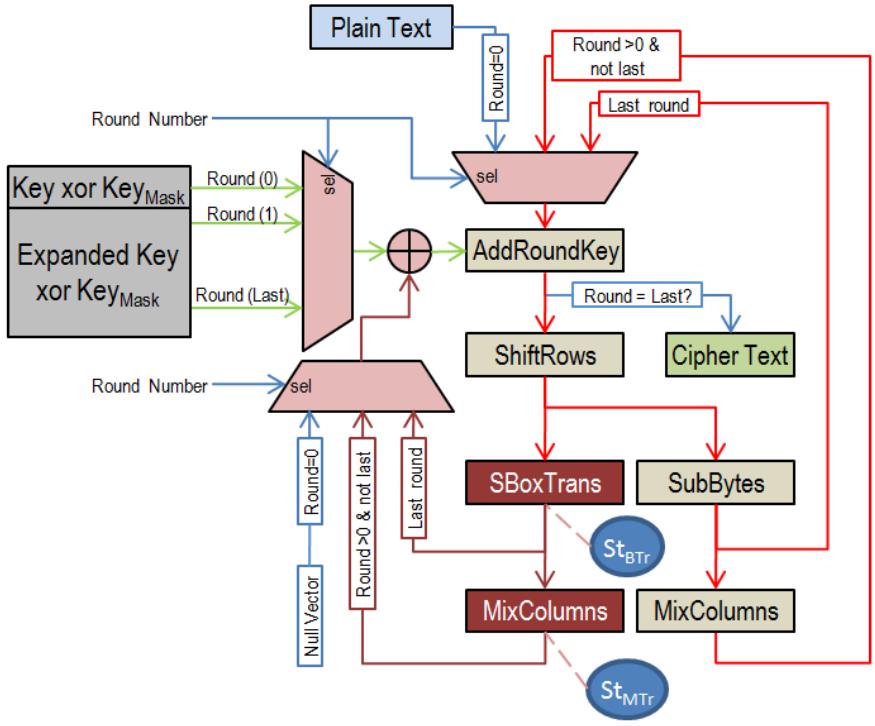

Fig. 19.2. Implementation of the faking countermeasure applied on the AES128-bit encryption algorithm.

Let *State(i,j)* be the output of function *ShiftRows*, then the effect of the non-linear function *SubBytes* over the matrix state will be:

$$St_{b_{(i,j)}} = SBOX\left(St_{r_{(i,j)}}\right) = SBOX\left(Key_{FAKE(i,j)} \oplus State_{(i,j)}\right). \quad (19.12)$$

By substituting (19.11) in (19.12), and developing the result in terms of the output that would be obtained if the real key was used, (19.12) can be expressed as:

$$\begin{aligned} St_{b_{(i,j)}} &= SBOX\left(Key_{REAL(i,j)} \oplus Key_{MASK(i,j)} \oplus State_{(i,j)}\right) \\ St_{b_{(i,j)}} &= SBOX\left(Key_{REAL(i,j)} \oplus State_{(i,j)}\right) \oplus St_{b_{TRANS(i,j)}}. \\ St_{b_{(i,j)}} &= St_{b_{REAL\,(i,j)}} \oplus St_{b_{TRANS(i,j)}} \end{aligned} \quad (19.13)$$

Note that, the output of *SubBytes* can be obtained by operating with an exclusive-OR, the matrix state encrypted with $Key_{REAL(i,j)}$ and a new matrix denoted as $St_{b_{TRANS(i,j)}}$, which represents the output of *SboxTrans* (see Fig. 19.2).

Finally, the result of (19.13) is processed by using function *MixColumns*. This function operates in the Galois finite field GF(2^8) and it only includes additions as products. An interesting property of such an algebraic structure is that it satisfies the distributive property of multiplication with respect to the addition, and thus:

$$A \cdot (B + C) = A \cdot B + A \cdot C. \quad (19.14$$

Using (19.13) and (19.14) it can be found that:

$$\begin{pmatrix} St_{M(i,1)} \\ St_{M(i,2)} \\ St_{M(i,3)} \\ St_{M(i,4)} \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} St_{b\,REAL\,(i,j)} + St_{b\,TRANS(i,j)} \\ St_{b\,REAL\,(i,j)} + St_{b\,TRANS(i,j)} \\ St_{b\,REAL\,(i,j)} + St_{b\,TRANS(i,j)} \\ St_{b\,REAL\,(i,j)} + St_{b\,TRANS(i,j)} \end{pmatrix}. \quad (19.15)$$

Afterward, the distributive propertyis developed:

$$\begin{pmatrix} St_{M(i,1)} \\ St_{M(i,2)} \\ St_{M(i,3)} \\ St_{M(i,4)} \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} St_{b\,REAL(i,j)} \\ St_{b\,REAL(i,j)} \\ St_{b\,REAL(i,j)} \\ St_{b\,REAL(i,j)} \end{pmatrix} + \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} St_{b\,TRANS(i,j)} \\ St_{b\,TRANS(i,j)} \\ St_{b\,TRANS(i,j)} \\ St_{b\,TRANS(i,j)} \end{pmatrix}. \quad (19.16)$$

The conclusion is that:

$$\begin{aligned} St_M &= MixColumns\,(St_{b\,REAL}) \oplus MixColumns\,(St_{b\,TRANS}) \\ St_M &= St_{M\,REAL} \oplus St_{M\,TRANS} \end{aligned}, \quad (19.17)$$

Where St_M is the output of function *MixColumns*, $St_{M\ REAL}$ is the output of function *MixColumns* if it was processed with the true key, and $St_{M\ TRANS}$ is a masking matrix that will be used at the end of each round.

Additionally, as the AES algorithm is executed in several rounds, the effects of the faking countermeasure should be reverted for each round. Thus, matrix $ST_{M\ TRANS}$, defined in (19.14), should be calculated in order to neutralize such an effect, since

$$St_M \oplus St_{M\,TRANS} = St_{M\,REAL} \oplus St_{M\,TRANS} \oplus St_{M\,TRANS} = St_{M\,REAL}. \quad (19.18)$$

On the other hand, matrix $ST_{M\ TRANS}$ could be obtained by applying function *MixColumns* over matrix $St_{b\ TRANS}$, which can by deduced by using (19.13) as:

$$\begin{aligned} St_{b\,TRANS(i,j)} &= St_{b\,REAL(i,j)} \oplus St_{b\,(i,j)} \\ St_{b\,TRANS(i,j)} &= SBOX\big(Key_{REAL(i,j)} \oplus State_{(i,j)}\big) \oplus Sb_{(i,j)} \end{aligned}. \quad (19.19)$$

by substituting (19.12) in (19.19):

$$St_{b\,TRANS(i,j)} = SBOX\big(Key_{REAL(i,j)} \oplus State_{(i,j)}\big) \oplus$$

$$\oplus SBOX\left(Key_{FAKE(i,j)} \oplus State_{(i,j)}\right). \quad (19.20)$$

Since elements of state are bytes, the 256 possible values for Sb_{TRANS} can be pre-calculated forming a table of 256 elements named $SBOX_{TRANS}$. For calculating such a table, the following substitution is proposed:

$$val = Key_{FAKE} \oplus State. \quad (19.21)$$

By using (19.11), Eq. (19.20) could be generalized for any value of the elements of state, as follows:

$$SBOX_{TRANS}(val) = SBOX(val \oplus Key_{MASK}) \oplus SBOX(val). \quad (19.22)$$

The use of this expression allows to calculate all the elements of $SBOX_{TRANS}$ by using the pseudocode shown in (19.23). Finally, a suitable value for a specific input could be calculated in a similar way to that which is performed when searching in *SBOX* during the execution of function *SubBytes* included in AES.

$$\begin{aligned} &for\ j = 0\ to\ 255 \\ &\quad SBOX_{TRANS}(j) = SBOX(j \oplus Key_{MASK}) \oplus SBOX(j) \\ &next\ j \end{aligned} \quad (19.23)$$

19.3.1. Added Vulnerabilities

It is necessary to analyze if the real implementation of the faking countermeasure adds itself new vulnerabilities that can be used to reveal the true key. Note that, the application of SCA analysis allows to find the false key Key_{FAKE}. So that, if the attacker does focus its target on revealing the key-mask Key_{MASK}, then it would be easy to find Key_{REAL} just by using (19.11). There are two weak points in the implementation presented in Fig. 19.2, which can be used to reveal Key_{REAL}.

19.3.1.1. Second Order Attack

A second order attack between the output of functions *SboxTrans* and *SubBytes* can be used to find Key_{REAL}. The model to be used is based on the combination, by means of an exclusive-OR operator, between the output of *SubBytes* (19.13) and the output of *SboxTrans* (19.19):

$$St_{b(i,j)} \oplus SBOX_{TRANS(i,j)} = SBOX\left(Key_{REAL(i,j)} \oplus State_{(i,j)}\right). \quad (19.24)$$

In the first round, the state corresponds directly to the plain-text. Additionally, SBOX is defined by the algorithm, so that both data are known by the attacker and can be used to perform SCA analysis for revealing Key_{REAL}.

The power consumption can be pre-processed by using the function defined in (19.3), being the pair (C_{ti}, C_{tk}), the power consumed at the output of functions *SubBytes* and *SboxTrans*, respectively.

Fig. 19.3 shows the result of a theoretical second order attack following this procedure and which is performed over the first byte of the key. It can be observed as the obtained correlations are quite reduced, but the maximum is perfectly distinguishable among the rest of the values and it is in accordance with the real key.

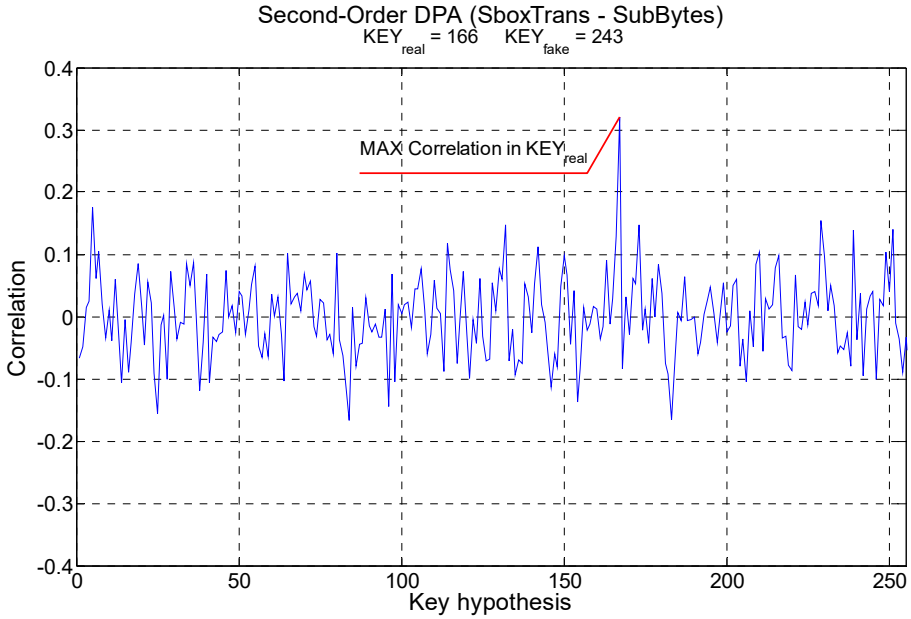

Fig. 19.3. Simulation of a second order attack based on the correlation.

19.3.1.2. Attack on the Output of *SboxTrans*

Eq. (19.20) depends only on the state, the false key and the real key. Note that, if both function *SBOX* and the KEY_{FAKE} are known by the attacker and then it is possible to perform SCA on *SboxTrans* to reveal the real key.

19.3.2. Protecting the StM $_{TRANS}$ Array

The solution for avoiding the attacks presented above is to protect the matrix St_{MTRANS} using a random mask. Such a matrix is combined, by using an exclusive-OR operator, with a mask that is changing its bytes at every encryption cycle.

Masking St_{MTRANS} has a significant effect on function *MixColumns*, since this function operates over different bytes that are located in several rows. It is not recommended to use the same mask for masking the 16 bytes of state, since the protection can be removed momentarily when one of the exclusive-OR operations involved in *Mixcolumns* is performed. In [4], it is proposed to use different masks for each row of state, in order to protect the integrity of data when *Mixcolumns* is executed.

Then, the values of bytes included in the mask are randomly generated for every encryption (decryption) cycle. Such values form a matrix named MT_J and defined as:

$$MT_J = \begin{pmatrix} m_0 & m_0 & m_0 & m_0 \\ m_1 & m_1 & m_1 & m_1 \\ m_2 & m_2 & m_2 & m_2 \\ m_3 & m_3 & m_3 & m_3 \end{pmatrix}, \tag{19.25}$$

where [m_0, m_1, m_2, m_3] represent the mask values for every row of matrix $St_{M\ TRANS.}$

If matrix MT_J is included, then the calculation of $St_{M\ TRANS}$ would be:

$$St_{b\,TRANS(i,j)} = St_{b\,REAL(i,j)} \oplus St_{b\,(i,j)} \oplus MT_J. \tag{19.26}$$

Note that, the system is protected since now a random term that is unknown by the attacker is included. Besides, it should be pointed out that this random term cannot be cancelled by means of a second order attack, since it is only applied on function *SboxTrans*.

If in this new scenario a second order attack is performed, the results shown in Fig. 19.4 demonstrate the effectiveness of the protection that is added by the mask. It is impossible to distinguish between the real hypothesis and the false hypothesis that correspond to the true key.

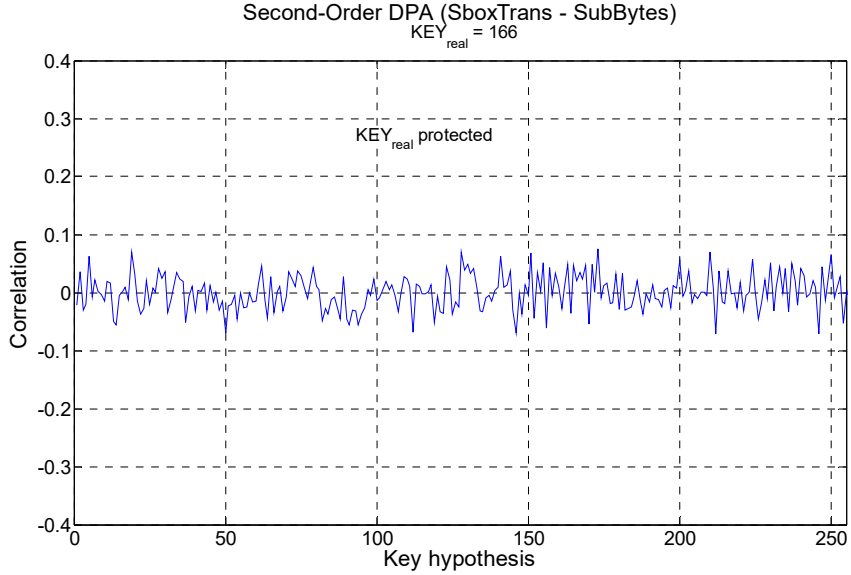

Fig. 19.4. Second order attack including masking.

Obviously, before starting a new round, the effect of masking should be removed in order to perform the encryption process correctly. The cancelation matrix is named MT_K.

After the application of function *SboxTrans,* Eq.(19.27) is obtained

$$St_{b\,TRANS(i,j)} \oplus MT_J. \tag{19.27}$$

Afterwards, function *MixColumns* is applied to (19.27), obtaining:

$$MixColumns\left(St_{b\,TRANS(i,j)} \oplus MT_J\right) =$$
$$= MixColumns\left(St_{b\,TRANS(i,j)}\right) \oplus MixColumns\left(MT_J\right). \tag{19.28}$$

From (19.28), it can be deduced that the elements of matrix MT_K can be calculated by using expression (19.29):

$$MT_K = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} m_0 & m_0 & m_0 & m_0 \\ m_1 & m_1 & m_1 & m_1 \\ m_2 & m_2 & m_2 & m_2 \\ m_3 & m_3 & m_3 & m_3 \end{pmatrix} =$$
$$= \begin{pmatrix} m_a & m_a & m_a & m_a \\ m_b & m_b & m_b & m_b \\ m_c & m_c & m_c & m_c \\ m_d & m_d & m_d & m_d \end{pmatrix} \tag{19.29}$$

Finally, Fig. 19.5 shows the complete structure of the system, including the random number generator used to calculate matrix MT_J, and the operations needed to perform the cancelation of masks before starting a new round.

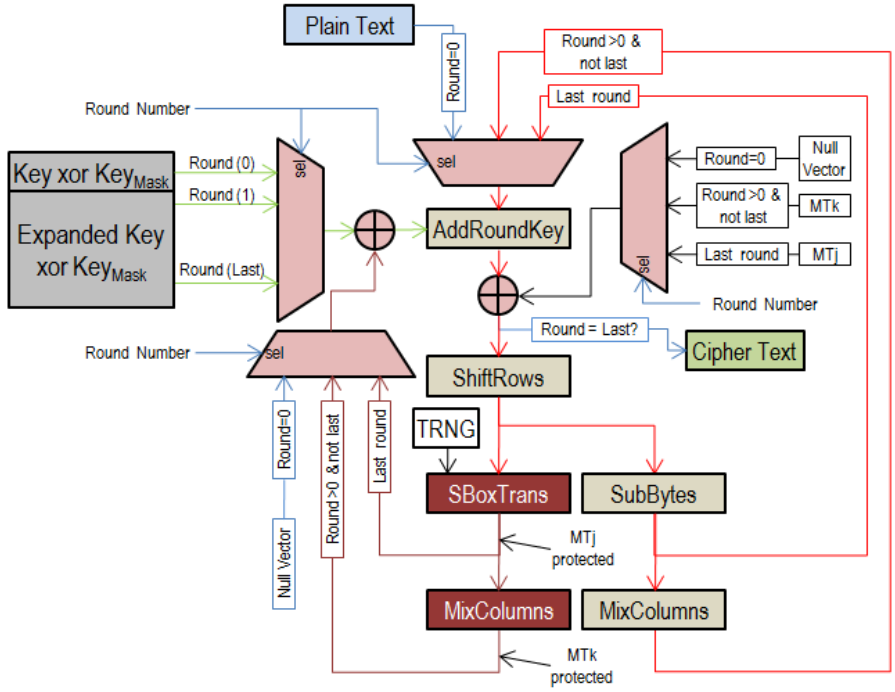

Fig. 19.5. AES algorithm structure with faking countermeasure and masking protection.

19.3.3. Computational Cost

Additional operations included to perform the faking countermeasure increase the processing time. These operations can be divided into three groups:

- Those operations that are carried out at the beginning of the encryption process for changing KEY_{REAL} or KEY_{MASK};
- Operations that are performed at every encryption cycle;
- Operations made at every round of the algorithm.

In the first group, only the masking of the expanded key is involved. The impact of this operation is not very important, since the execution time is distributed among the plain-texts that are processed using the same key.

The operations included in the second group are intended for calculating matrix MT_J and matrix MT_K.Eq. (19.27) is applied on the 16 bytes that form the state, so that it is necessary to calculate 16 $SBOX_{TRANS}$ tables for calculating *SboxTrans*. In order to minimize the impact of these operations, it is possible to perform their calculation concurrently with the communication of the device when reading or writing the original and cipher texts, respectively.

The third group of operations is necessary for calculating matrix $St_{M\ TRANS}$. Since these operations are made in each round, they have an important impact on the execution time. As the key is completely scattered after round 2, and then performing a successful SCA is almost impossible, the time could be reduced if after round 2 the countermeasure is disabled (the countermeasure should be activated again after round 9, because SCA analysis could be focus on the last rounds [5]).

19.4. Experimental Results

This section shows the experimental results obtained when a software implementation of the proposed countermeasure is included in the AES 128-bit encryption algorithm. The execution is performed on a 32-bit soft-core Microblaze V8.10 microprocessor provided by Xilinx. In [10] it is shown that it is possible to undertake a successful SCA analysis on this microprocessor. The system was implemented on a Virtex-5 FPGA clocked at 24 MHz [3]. The current traces were captured using a Tektronic CT-1 current probe connected to an Agilent DSO1024A oscilloscope.

The presented results are based on the analysis of the correlation and the attack proposed in [11], based on the differences-in-means. In order to observe the efficiency of the proposal, both results are shown when the countermeasure is activated and disabled. The target of the attack was focussed on function *SubBytes*, when byte 0 of the key is processed in the first round. The value chosen for KEY_{REAL} is 134, KEY_{MASK} is 85, and therefore we take into account that in (19.27) KEY_{FAKE} is 211.

The upper traces in Fig. 19.6 shows SCA analysis based on the value of the correlation when the countermeasure is disabled. Clearly, the maximum value for such correlation is about 0.8 and it corresponds with the true key. When the countermeasure is activated (lower traces of Fig. 19.6), the maximum correlation is given for the false key. The true key is completely protected and has a very low correlation that is non-distinguishable between the rests of the keys.

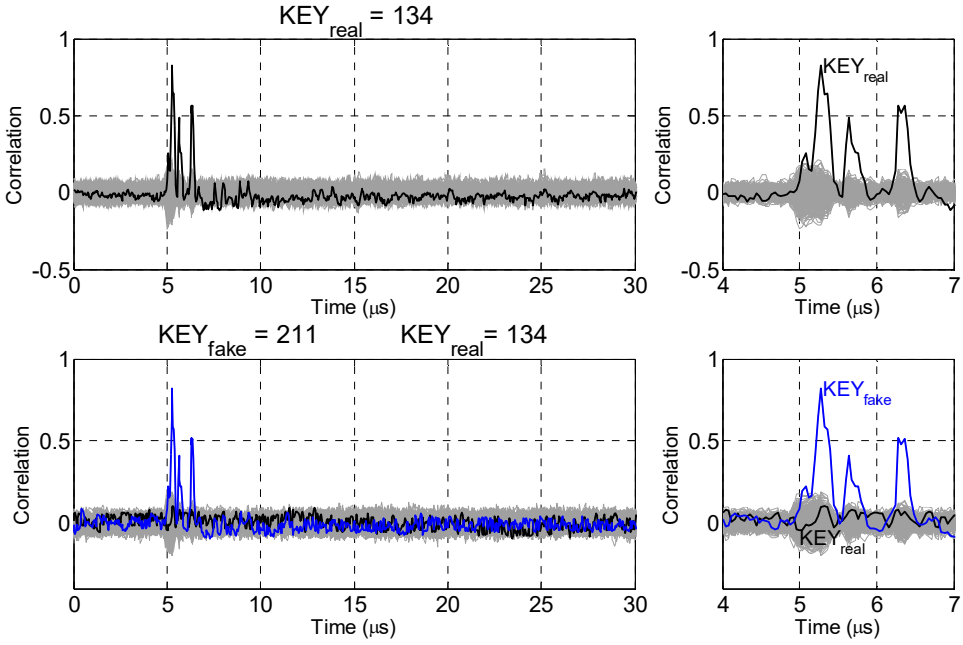

Fig. 19.6. Correlation attack. Upper traces correspond to an unprotected system; lower traces are obtained by activating the faking countermeasure.

Fig. 19.7 shows the values of the correlation against an increasing number of captured traces. The SCA is able to reveal the key with only 50 traces, independently, if the system is protected or if the countermeasure is disabled.

The attack based on differences-in-means was performed following the proposal of R. Lumbiarres in [11], which is based partially on the original publication of P. Kocher in [1]. Results shown in Fig. 19.8 again demonstrate that the system is completely protected when the faking countermeasure is activated by revealing a false key.

Generally, any countermeasure leads to increasing the number of hardware resources and/or the total execution time. Table 19.4 shows the penalty generated due to the inclusion of the proposed countermeasure. The execution time is increased by about 39 %, whereas, in accordance with this value, the throughput is reduced by 28 %. On the other hand, the memory needed to execute the process is also increased due to the extra memory required to store the 16 $SBOX_{TRANS}$ tables. Compared with other implementations, these penalties are quite acceptable. For instance, a masked implementation of an AES

128-bit algorithm that is executed on a microprocessor is also presented in [4]. The difference in the execution time between the non-protected and masked implementations was doubled.

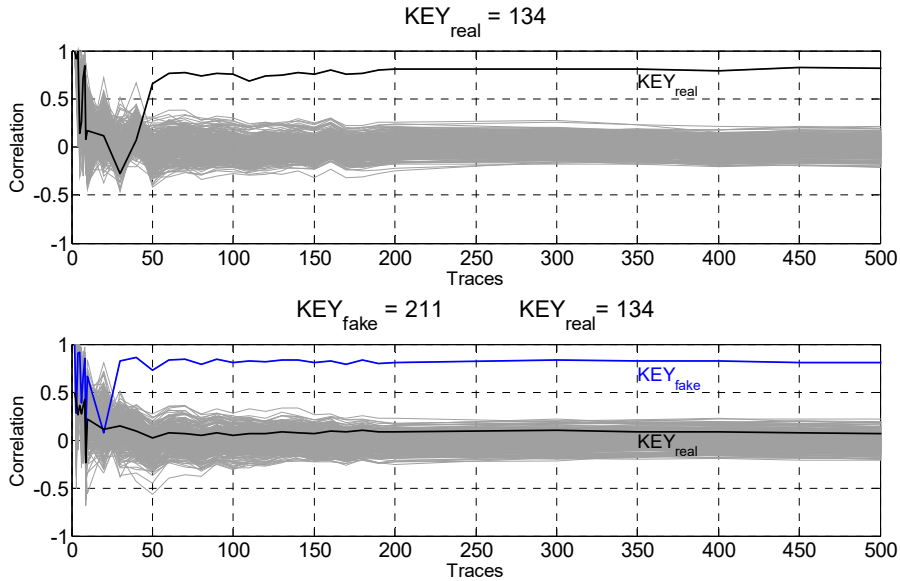

Fig. 19.7. Correlation attack over an increasing number of traces. Upper traces correspond to a non-protected system; lower traces are obtained by activating the faking countermeasure.

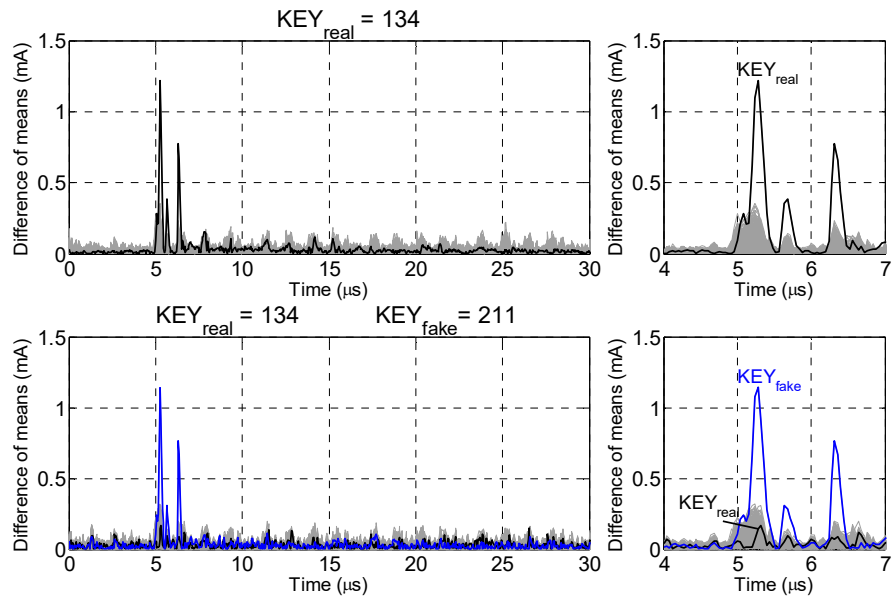

Fig. 19.8. Differences-in-means. Upper traces correspond to a non-protected system; lower traces are obtained by activating the faking countermeasure.

Table 19.4. Execution time and memory used.

Parameter	Non-protected	Fake protected	Improvement
Execution time	1.14 ms	1.59 ms	39 %
Time expended in pre-calculating the tables $SBOX_{TRANS}$	----	1.90 ms	---
Throughput	112 kb/s	80kb/s	-28 %
Memory needed for executing the algorithm	11770 bytes	19094 bytes	62 %

19.5. Conclusions

The proposed countermeasure conceals the true key by presenting a strong correlation related to a false key. Experimental results showed that the method is effective when either the correlation or differences-in-means attacks are performed.

When compared with a non-protected system, the execution time and the memory needed to execute the encryption algorithm are increased. However, such an increase is lower than the penalty related to previous proposals made by different authors.

Another interesting feature of the faking countermeasure is that its particular structure allows to include additional actions that increase the difficulty to find the true key. Thus, it is possible to modify the real key without introducing any modification in the false key. Then, the maximum correlation is the same although the encrypting key was changed. Also, it is possible to introduce modifications into the false key, while keeping the value of the real key. In this case, the attacker may conclude that the encryption key has changed.

Acknowledgements

This work was supported by the Ministerio de Economía y Competitividad in the framework of the Programa Nacional de Proyectosde Investigación Fundamental, project TEC2015-68784-R.

References

[1]. J. Jaffe, B. Jun, P, Kocher, Differential power analysis, in *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO'99)*, Santa Barbara, California, 1999, pp. 388-397.

[2]. Advanced Encryption Standard (AES), FIPS PUB 197, *National Institute of Standards and Technology*, 2001.

[3]. Side-channel Attack Standard Evaluation Board SASEBO-GII Specification, Research Center for Information Security, *National Institute of Advanced Industrial Science and Technology*, 2009, http://satoh.cs.uec.ac.jp/SAKURA/index.html

[4]. E. Oswald, T. Popp, S. Mangard, Power Analysis Attacks - Revealing the Secrets of Smart Cards, *Springer Science+Busines Media*, Graz, Austria, 2007.

[5]. L. Pan, J. den Hartog, J. Lu, Security of AES against first and second-order differential power analysis, in *Proceedings of the 4th Benelux Workshop on Information and System Security (WISSec'09)*, Louvain-la-Neuve, Belgium, 2009.

[6]. K. Wu, B. Peng, Y. Zhang, X. Zheng, F. Yu, H. Li, Enhanced correlation power analysis attack on Smart Card, in *Proceedings of the 9th International Conference for Young Computer Scientists (ICYCS'08)*, 2008, pp. 2143-2148.

[7]. T. S. Messerges, Using second-order power analysis to attack DPA resistant software, in *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES'2000),* Vol. 1, Worcester, 2000, pp. 238-251.

[8]. M. Rivain, R. Bévan, E. Prouff, Statistical analysis of second order differential power analysis, *IEEE Transactions on Computers*, Vol. 58, No. 6, June 2009, pp. 799-811.

[9]. M. López-García, E. F. Cantó-Navarro, R. Lumbiarres-López, Implementation on MicroBlaze of AES algorithm to reveal fake keys against side-channel attacks, in *Proceedings of the IEEE 23rd International Symposium on Industrial Electronics (ISIE'14),* Istanbul, 2014, pp. 1882-1887.

[10]. M. López-García, E. F. Cantó-Navarro, R. Lumbiarres-López, Ataques por canal lateral sobre el algoritmo de encriptación AES implementado en MicroBlaze, in *Proceedings of the XIII Jornadas de Computación Reconfigurable y Aplicaciones (JCRA'13)*, Madrid, 2013, pp. 105-112.

[11]. M. Lopez-Garcia, E. Canto-Navarro, R. Lumbiarres-Lopez, Hardware architecture implemented on FPGA for protecting cryptographic keys against side-channel attacks*, IEEE Transactions on Dependable and Secure Computing*, Vol. 99, September 2016, pp. 1-10.