Chapter 19

Faking Countermeasure Against Side-Channel Attacks

Rubén Lumbarres-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.
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 \( \text{HW}(11010001) = 4 \), whereas \( \text{HD}(11010001, 01011001) = 2 \).

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:
Chapter 19. Faking Countermeasure against Side-Channel Attacks

\[ u_m = u \oplus m \]
\[ v_m = v \oplus m, \]  \hspace{1cm} (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:

\[ HW(u_m \oplus v_m) = HW(u \oplus m \oplus v \oplus m) = HW(u \oplus v). \]  \hspace{1cm} (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 authors propose the use of products, the square of the sum or even the square of the absolute value [8, 4].

\[ F_{\text{pre-processing}} = \text{abs}(C_{ti} - C_{tk}). \]  \hspace{1cm} (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:

$$C(x) = 3x^3 + x^2 + x + 2,$$

or the polynomial:

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

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.

<table>
<thead>
<tr>
<th>State bytes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15</td>
<td></td>
</tr>
<tr>
<td>15 83 250 3 140 142 46 132 46 7 109 99 162 137 103</td>
<td>AdKey</td>
</tr>
<tr>
<td>118 242 237 45 123 100 25 49 95 49 197 60 251 58 167 133</td>
<td>SBytes</td>
</tr>
<tr>
<td>118 100 197 133 123 49 167 45 95 58 237 49 251 242 25 60</td>
<td>SRow</td>
</tr>
<tr>
<td>0 111 23 42 47 198 104 65 44 54 247 84 197 19 127 133</td>
<td>MCol</td>
</tr>
<tr>
<td>220 255 32 154 180 143 183 168 187 200 133 107 253 146 106 34</td>
<td>AdKey</td>
</tr>
<tr>
<td>134 22 183 184 141 115 169 194 234 232 151 127 84 79 2 147</td>
<td>SBytes</td>
</tr>
<tr>
<td>134 115 151 147 141 232 2 184 234 79 183 194 84 22 169 127</td>
<td>SRow</td>
</tr>
<tr>
<td>134 81 110 72 152 248 178 13 107 116 141 66 68 231 138 189</td>
<td>MCol</td>
</tr>
<tr>
<td>84 152 5 255 209 120 6 83 181 10 75 35 162 24 89 123</td>
<td>AdKey</td>
</tr>
</tbody>
</table>

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:

\[ S_{T_A(i,j)} = T_{(i,j)} \oplus K_{(i,j)}, \]  

(19.6)

where \( S_{T_A(i,j)} \) (\( i = 1..4, j = 1..4 \)) 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 \textit{AddRoundKey}.

<table>
<thead>
<tr>
<th>Key hypothesis and Hamming weight (HW)</th>
<th>Max correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>209 HW(11010001) = 4</td>
<td>0.6282</td>
</tr>
<tr>
<td>210 HW(11010010) = 4</td>
<td>0.6002</td>
</tr>
<tr>
<td>211 HW(11010011) = 5</td>
<td>0.6692</td>
</tr>
<tr>
<td>196 HW(11000011) = 4</td>
<td>0.6545</td>
</tr>
<tr>
<td>148 HW(10010011) = 4</td>
<td>0.6236</td>
</tr>
</tbody>
</table>

The output of function \textit{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 \textit{SBOX}, as follows:

\[
S_{B(i,j)} = SBOX(S_{R(i,j)}),
\]

where \(S_{R(i,j)}\) and \(S_{B(i,j)}\) refer to the value of matrix state at the output of \textit{ShiftRows} and \textit{SubBytes}, respectively. It can be demonstrated that the elements of matrix state \(S_{B(i,j)}\) 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 (\(\rho = 0.8143\)) 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 \textit{SubBytes}.

<table>
<thead>
<tr>
<th>Key hypothesis</th>
<th>Max correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>209 HW(11010001) = 4</td>
<td>0.0632</td>
</tr>
<tr>
<td>210 HW(11010010) = 4</td>
<td>0.0747</td>
</tr>
<tr>
<td>211 HW(11010011) = 5</td>
<td>0.8143</td>
</tr>
<tr>
<td>196 HW(11000011) = 4</td>
<td>0.0591</td>
</tr>
<tr>
<td>148 HW(10010011) = 4</td>
<td>0.0729</td>
</tr>
</tbody>
</table>

After the calculation of function \textit{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 \textit{Mixcolumns} operation is applied on the four columns of the matrix state and it is defined as:

\[
\begin{pmatrix}
S_{M(i,1)} \\
S_{M(i,2)} \\
S_{M(i,3)} \\
S_{M(i,4)}
\end{pmatrix}
= \begin{pmatrix}
2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2
\end{pmatrix}
\begin{pmatrix}
S_{B(i,1)} \\
S_{B(i,2)} \\
S_{B(i,3)} \\
S_{B(i,4)}
\end{pmatrix},
\]

(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 $4 \times 4$ 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.
Let $\text{State}(i,j)$ be the output of function $\text{ShiftRows}$, then the effect of the non-linear function $\text{SubBytes}$ over the matrix state will be:

$$
\text{State}(i,j) = \text{SubBytes} \left( \text{State}(i,j) \right) = \text{SubBytes} \left( \text{Key}_{\text{FAKE}}(i,j) \oplus \text{State}(i,j) \right).
$$

(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:

$$
\text{State}(i,j) = \text{SubBytes} \left( \text{Key}_{\text{REAL}}(i,j) \oplus \text{Key}_{\text{MASK}}(i,j) \oplus \text{State}(i,j) \right)
$$

$$
\text{State}(i,j) = \text{SubBytes} \left( \text{Key}_{\text{REAL}}(i,j) \oplus \text{State}(i,j) \right) \oplus \text{State}_{\text{TRANS}}(i,j).
$$

(19.13)

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

Finally, the result of (19.13) is processed by using function $\text{MixColumns}$. This function operates in the Galois finite field $\text{GF}(2^8)$ 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:
Chapter 19. Faking Countermeasure against Side-Channel Attacks

\[ A \cdot (B + C) = A \cdot B + A \cdot C. \]  \hspace{1cm} (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}
\begin{pmatrix}
St_{bREAL(i,j)} + St_{bTRANS(i,j)} \\
St_{bREAL(i,j)} + St_{bTRANS(i,j)} \\
St_{bREAL(i,j)} + St_{bTRANS(i,j)} \\
St_{bREAL(i,j)} + St_{bTRANS(i,j)}
\end{pmatrix}. \hspace{1cm} (19.15)

Afterward, the distributive property is 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}
\begin{pmatrix}
St_{bTRANS(i,j)} \\
St_{bTRANS(i,j)} \\
St_{bTRANS(i,j)} \\
St_{bTRANS(i,j)}
\end{pmatrix}. \hspace{1cm} (19.16)

The conclusion is that:

\[
St_{M} = \text{MixColumns} (St_{bREAL}) \oplus \text{MixColumns} (St_{bTRANS}), \hspace{1cm} (19.17)
\]

Where \( St_{M} \) is the output of function \( \text{MixColumns} \), \( St_{MREAL} \) is the output of function \( \text{MixColumns} \) if it was processed with the true key, and \( St_{MTRANS} \) 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 \( ST_{MTRANS} \), defined in (19.14), should be calculated in order to neutralize such an effect, since

\[
St_{M} \oplus St_{MTRANS} = St_{MREAL} \oplus St_{MTRANS} \oplus St_{MTRANS} = St_{MREAL}. \hspace{1cm} (19.18)
\]

On the other hand, matrix \( ST_{MTRANS} \) could be obtained by applying function \( \text{MixColumns} \) over matrix \( St_{bTRANS} \), which can be deduced by using (19.13) as:

\[
St_{bTRANS(i,j)} = St_{bREAL(i,j)} \oplus St_{b(i,j)}
\]

\[
St_{bTRANS(i,j)} = SBOX(Key_{REAL(i,j)} \oplus State_{(i,j)}) \oplus S_{b(i,j)}. \hspace{1cm} (19.19)
\]

by substituting (19.12) in (19.19):

\[
St_{bTRANS(i,j)} = SBOX(Key_{REAL(i,j)} \oplus State_{(i,j)}) \oplus
\]
\[ \oplus SBOX(Key_{FAKE(i,j)} \oplus State_{(i,j)}). \quad (19.20) \]

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.

\[ \text{for } j = 0 \text{ to } 255 \\
\quad SBOX_{TRANS}(j) = SBOX(j \oplus Key_{MASK}) \oplus SBOX(j) \quad (19.23) \]

### 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):

\[ St_b(i,j) \oplus SBOX_{TRANS(i,j)} = SBOX(Key_{REAL(i,j)} \oplus State_{(i,j)}). \quad (19.24) \]

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 \((C_l, C_h)\), 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.

![Second-Order DPA (SboxTrans - SubBytes)](image)

**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:

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

where $[m_0, m_1, m_2, m_3]$ represent the mask values for every row of matrix $StM TRANS$.

If matrix $MT_J$ is included, then the calculation of $StM TRANS$ would be:

$$St_{TRANS(i,j)} = St_{bREAL(i,j)} \oplus St_{b(i,j)} \oplus MT_J.$$  \hspace{1cm} (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.

![Second-Order DPA (SboxTrans - SubBytes)](image)

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
\[ S_{b_{TRANS(i,j)}} \oplus MT_j. \]  

Afterwards, function \textit{MixColumns} is applied to (19.27), obtaining:

\[
\text{MixColumns}\left( S_{b_{TRANS(i,j)}} \oplus MT_j \right) = \text{MixColumns}\left( S_{b_{TRANS(i,j)}} \right) \oplus \text{MixColumns}\left( MT_j \right). \tag{19.28}
\]

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.

\[ \text{Fig. 19.5. AES algorithm structure with faking countermeasure and masking protection.} \]
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 plaintexts 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 $StM_{TRANS}$. 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.

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

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 Proyectos de Investigación Fundamental, project TEC2015-68784-R.

References


