A New Countermeasure Against Side-Channel Attacks Based on Hardware-software Co-design

Ruben Lumbiarres-Lopez[1]
Mariano Lopez-Garcia[1]
Enrique Canto-Navarro[2]

1-Electronic Engineering, Universidad Politècnica de Cataluña, Avda. Victor Balaguer, 08800, Vilanova i la Geltrú, Spain. mariano.lopez@upc.edu

2-Enrique Cantó-Navarro - Universitat Rovira i Virgili, Automatics and Electronic Engineering, Avda. Països Catalans, Tarragona, Spain.

Abstract

This paper aims at presenting a new countermeasure against Side-Channel Analysis (SCA) attacks, whose implementation is based on a hardware-software co-design. The hardware architecture consists of a microprocessor, which executes the algorithm using a false key, and a coprocessor that performs several operations that are necessary to retrieve the original text that was encrypted with the real key. The coprocessor hardly affects the power consumption of the device, so that any classical attack based on such power consumption would reveal a false key. Additionally, as the operations carried out by the coprocessor are performed in parallel with the microprocessor, the execution time devoted for encrypting a specific text is not affected by the proposed countermeasure. In order to verify the correctness of our proposal, the system was implemented on a Virtex 5 FPGA. Different SCA attacks were performed on several functions of AES algorithm. Experimental results show in all cases that the system is effectively protected by revealing a false encryption key.

Keywords: Countermeasure, Side-Channel Analysis, AES algorithm and Hardware-software Co-design

1. Introduction

Since Kocher et al. [1], in the late 1990s, demonstrated the vulnerabilities of cryptographic devices, Side Channel Analysis (SCA) attacks have become the most significant threat related to the security of cryptographic algorithms. These attacks base their success on analyzing the leakage information that is mainly observable through the power consumption or the electromagnetic radiation (EM) emitted by a hardware device. The attack is feasible because either of these two quantities is related to the data being processed by the device, which depends on the value of the cryptographic key.

Once such weakness was revealed, part of the scientific community oriented their efforts in proposing countermeasures that provide resistance against SCA attacks. Although with some differences, almost all proposed solutions attempt to design systems in which the power consumption (or the EM) is independent of the data that they process. This objective is achieved either by providing systems featured with random power consumption or building devices in which such power is constant in each clock cycle. The latter approach, known as hiding, has usually been implemented at cell level based on the Dual-Rail Precharge (DRP) logic style. This style is tailored with signals represented by two complementary wires, in such a way that in every clock cycle only one switch per cycle is produced. Thus, during the pre-charge phase, both the direct and complementary wires are charged,whereas in the evaluation phase only one of them is discharged. Among the more significant proposals of this logic style can be found *Sense Amplifier Based Logic* (SABL) [2] and *Wave Dynamic Differential Logic* (WDDL) [3]. However, the main drawback of such DRP logic styles is that their success depends on the perfect balancing between the capacitive loads related to the complementary wires that form the overall circuit. This requirement implies including some constraints on the placement and routing steps. In contrast, the former approach, known as masking, has been implemented at both algorithm and cell levels. At cell level, the most relevant proposals are *Random Switching Logic* (RSL) [4], *Dual Random Switching Logic* (DRSL)

[5] and *Masked Dual-Rail Precharge Logic* (MDPL) [6]. Masking Boolean approaches base their resistance against SCA attacks on concealing, by means of an exclusive OR operator, all intermediate values v with a random mask m. The masked values $v_m = (v \oplus m)$, which are actually being processed into the hardware device, are statistically independent with respect to v, so that the power consumption and the cryptographic key are completely uncorrelated. Thus, these logic styles are not affected by the imbalance existing between the routing capacitances of complementary wires. Furthermore, approaches based on hiding (i.e., SABL and WDDL) could be implemented in a smaller area than the one needed by the masked logic styles (i.e., MPDL, RSL, DRSL). Additionally, SABL, RSL and DRSL require designing specific cells for their implementation, whereas WDDL or MDPL allow designing such secure logic based on existing standard cells.

Moreover, it has been shown that in general the security of cell level implementations could be compromised due to the effect of the inter-wire capacitances [7] or the so-called *early propagation effect* [8][9]. As these vulnerabilities became known, the previous proposals were updated, including new measures that make systems more secure against most of these harmful effects. For instance, the original MDPL, which inherently is a glitch-free logic style based on majority-gates, was modified to support the early propagation effect (iMDPL) [10]. Other examples of improved DPR styles can be found in [11] and [12]. More recently, a new countermeasure termed SecLib has been proposed [13]. The early evaluation is prevented by designing specific cells based on two stages that avoid such effect. However, as stated by the authors, it also increases the cost in terms of area, delay and power consumption.

Masking is a countermeasure that can be also implemented at algorithm level. In [14], the authors proposed an implementation of the AES encryption algorithm using six independent masks. The algorithm was solved on an 8-bit microcontroller leading to an execution time twice that compared with the unmasked version. There are also some proposals for implementing hiding countermeasures on software. These approaches aim at introducing temporal jitter

in the sequence of operations performed by the microprocessor. This way, the instant at which an effective attack might be produced is distributed over time following an unknown probability distribution function (misalignment of power traces). Some examples of these software countermeasures consist of introducing dummy cycles [15] or a random variation on the execution orders [16].

Other different publications aim at introducing noise to reduce the correlation between the processed data and the cryptographic key. Following this idea an interesting approach was proposed in [17], in which a noise generator correlated with the data that is being processed is included. However, the attack is only effective when the target is the function correlated with the introduced power noise. Additionally, the revealed key is not always the same and it depends on the number of traces captured and used to perform the attack.

As mentioned above, SCA attacks based their success on exploiting the existing dependence between the processed data and the power consumption (or EM). As data depend on the cryptographic key, from a statistical point of view it means that there exists a correlation between such a key and the consumed power. Theoretically, only the correct key is able to produce a correlation with a significant value, whereas the rest of the keys would generate a value close to zero. Countermeasures based on hiding or masking try to eliminate this correlation, in such a way that any SCA attack, performed on any possible key, does not produce any relevant result that could be distinguished among all others. In other words, all correlations between power consumption and guessed keys are equally likely and tend to zero.

The countermeasure proposed in this paper is completely different when compared with previous approaches. The mechanism for protecting the system consists in revealing a false key when a SCA attack is performed. This false key (or fake key) produces the highest correlation coefficient between the data processed and the power consumed by the hardware device. Thus, from the perspective of an attacker, the system behaves as an unprotected implementation that conceals the true key by producing a false positive. Note that, such implementation should be performed affecting as little as possible the power

consumption trace (in amplitude and time) when compared with the original non-protected system.

Although the proposed countermeasure, termed faking, could be entirely implemented in software, the penalty on the execution time would be quite significant. In fact, including all additional calculations needed to conceal the real key, such execution time is almost doubled when compared with the non-protected version. Instead, the implementation presented in this paper is based on a hardware/software co-design. The system consists of a microprocessor which solves via software the classical Advanced Encryption Standard (AES) 128-bit cryptographic algorithm, and a coprocessor specifically designed for implementing the proposed countermeasure. The proposed architecture is intended for applications in which the main task performed by the microprocessor is to solve a specific processing from which a critical information is obtained. The encryption is necessary for storing this confidential data in an external device or for sending such information through a non-secure channel. For instance, the microprocessor could be used for analyzing a fingerprint image from which a confidential biometric feature is obtained and should be stored in an external memory. Although is out the scope of this paper, in applications where the encryption is the main task that should be performed, a complete hardware-implementation would be more suitable and faster. Regardless of the implementation chosen, hardware, hardware-software or pure software, the level of security for all of them is identical and only their features in terms of area and speed are different.

This paper is organized into five sections. Section II presents the fundamentals of the proposed countermeasure. The aim of Section III is to describe the internal architecture of the coprocessor and its main features. Section IV presents the experimental results. Finally, section V presents the conclusions

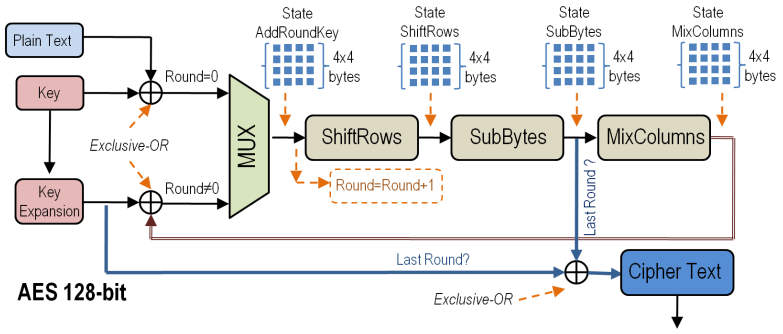

Figure 1: Basic structure of the Advanced Encryption Standard algorithm AES 128-bit

2. Fundamentals

2.1. Introduction

The structure of the AES 128-bit encryption algorithm is represented in Fig:1 As the figure shows, the algorithm consists of four operations that are performed on a matrix of 16 bytes, termed state, in different rounds: *AddRoundKey (exclusive-OR)*, *SubBytes*, *ShiftRows* and *Mixcolumns*. A general description about the principles of this cipher, including such four operations, can be found in [18] [19].

Although in the proposal presented by *Kocher* the cryptographic key was found using the differential-of-means method, currently the most extended statistical method employed for this purpose is based on correlation [14]. This method consists of the following steps:

a) The encryption algorithm is executed M times using a set of M different plain texts. For each one, a current trace is captured and stored for its subsequent processing.

b) It is quite usual to choose as points to be attacked (target) the output of one of the four operations (inputs of the following points) involved in the AES algorithm, since their result (state) is normally written in a memory or register, which creates a distinguishable point at the captured power trace.

c) A theoretical power model, which represents the consumption of the overall set of CMOS cells that form the circuit, should be chosen. Such a theoretical model is normally based on the Hamming distance (HD) or the Hamming weight (HW), that represents the value of a set of bits $v(t_k)$ related to the point to be attacked. Note that, if an intermediate value at instant (t_{k-1}) is $v(t_{k-1})$, then

$$HD(v(t_k)) = HW(v(t_{k-1}) \oplus v(t_k)) \quad (1)$$

Note that, the choice of any of the four operations as target of the attack facilitates the calculation of the value related to the theoretical model of power consumption

d) As the value of such a model depends on the cryptographic key, N values for N possible guessed keys should be calculated, assuming that the plain text (or the cipher text) is known by the attacker. Usually, to make the attack feasible, only a specific byte of an intermediate value at instant t_k is attacked, which reduces the value of N to 256 possibilities.

e) Each of these N particularized power models is correlated with all the M captured current traces at instant t_k. A number of NxM correlation coefficients are obtained. The calculation of such correlation is based on the Pearson's coefficient [14].

f) The highest correlation corresponds to the true encryption key.

2.2. Basis of the faking countermeasure

The underlying idea behind the proposed faking countermeasure is to carry out the encrypting process using a fake key Key_{FAKE}, which is obtained by XORing the real one Key_{REAL} with a mask Key_{MASK} in the following way

$$Key_{FAKE} = Key_{REAL} \oplus Key_{MASK} \quad (2)$$

Note that, in a general case all keys included in (2) consist of 16 different bytes each one, and they can be represented by a matrix of 4x4 bytes. The basic structure of the protected system is presented in Fig:2. The operations

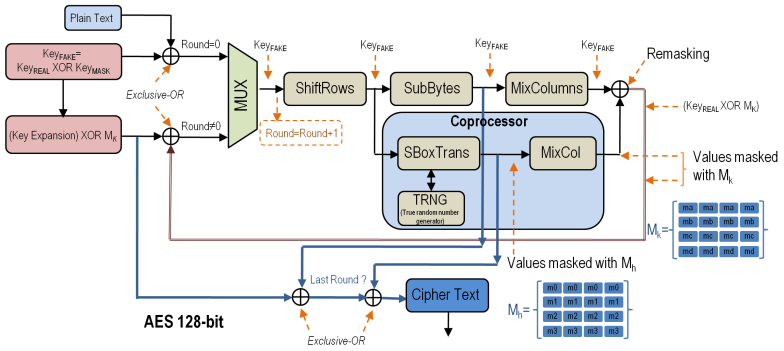

Figure 2: Structure for the implementation of the faking countermeasure applied on the Advanced Encryption Standard algorithm AES 128-bit.

performed during each round on the state matrix (*AddRoundKey, SubBytes, ShiftRows* and *MixColumns*) are processed by using the Key_{FAKE}, so that as the system does not have any additional countermeasure any SCA attack would reveal the false key.

If no additional actions are performed, the cipher text would be encrypted with Key_{FAKE} rather than the real key. Thus, at the end of each round the coprocessor is in charge of making the inverse process in order to retrieve the proper state matrix encrypted with Key_{REAL}. Such a process is performed in two steps, *SBoxTrans* and *MixCol*, in order to break up the operations performed with Key_{FAKE} by *SubBytes* and *MixColumns*, respectively. The effect of *ShiftRows* is not included in the coprocessor, since its output is based on a simple permutation. The block *MixCol* behaves in an identical manner to the *MixColumns* operation defined by the standard AES algorithm. Instead, *SBoxTrans* concentrates the main idea on which the faking countermeasure is based. *SubBytes* is a non-linear function, derived from the multiplicative inverse over $GF(2^8)$, that is implemented by using a substitution box $SBox(b)$ *(b=0..255)* which applies over each byte of state. Such a box could be pre-computed only once at the beginning of the algorithm. Let be $a_F(i,j)$ *(i=0..3, j=0..3)* a particular byte of state at the output of *Shiftrows*. This byte mainly consists in a

combination of the plain text $T_{(i,j)}$ and $Key_{FAKE(i,j)}$ with a XOR operator

$$SBox(a_F(i,j)) = SBox(T(i,j) \oplus Key_{FAKE}(i,j)) \tag{3}$$

Thus, a simple way of recovering the original byte $a_R(i,j)$ encrypted with Key_{REAL} ($a_R(i,j) = T(i,j) \oplus Key_{REAL}(i,j)$) at the output of *SubBytes* would be by defining *SBoxTrans* as follows

$$SBoxTrans(a_F(i,j)) = SBox(a_F(i,j)) \oplus SBox(a_F(i,j) \oplus Key_{MASK}(i,j)) \tag{4}$$

Using (2) and the algebraic properties of the XOR operator, (4) becomes:

$$SBox(a_F(i,j)) = SBoxTrans(a_F(i,j)) \oplus SBox(a_R(i,j)) \tag{5}$$

so that computing (4) and (5) with an exclusive-OR, $SBox(a_R(i,j))$ is directly obtained. In fact such an operation, termed remasking, is performed at the end of each round by computing the output of *MixColumns* and *MixCol* (see Fig:2). It is noteworthy that *SBoxTrans* can be implemented as a simple lookup table, which can be pre-computed like the original *SBox()*. It is important to point out that there will be a clear vulnerability if the same byte $Key_{MASK}(i,j)$ is used to build the matrix Key_{MASK}. As Key_{FAKE} could be revealed through a SCA attack, and since the cipher text is known by the attacker, a simple brute force attack consisting in 256 guessed values (the 256 possible values that could take the byte $Key_{MASK}(i,j)$ *(i=0..3, j=0..3))* would be enough to reveal Key_{REAL}. Hence, there will be a trade-off between the time needed to calculate this table and the level of security, which are both dependent on the number of different bytes $Key_{MASK}(i,j)$ chosen to form the matrix Key_{MASK}. Let be I such a number, being $1 \leq I \leq 16$. Then, as byte $a_F(i,j)$ can take 256 different values, the size of the table used to build *SBoxTrans* will be *Ix256* elements and its implementation, in the worst case, would be possible using a memory of 4Kbytes. In practice, it is advisable to choose the maximum value for I, in order to obtain the same fortress as the original AES algorithm (experimental results were obtained using a Key_{MASK} which consists of 16 different bytes).

2.3. Weakness of faking and use of masking techniques

When implementing the coprocessor, it is important to take into account some weaknesses that are usually related to hardware implementations.

Devices protected by Boolean masking approaches, which leak the Hamming distance, could not be completely secure if internal operations are not performed carefully. Indeed, taking into account (1), a situation of risk occurs when two intermediate values of $v_m = (v \oplus m)$, concealed with the same mask m, are written over a bus or register in two consecutive instants of time *k-1* and k, respectively, since

$$HD(v_m(t_k)) = HW(v_m(t_{k-1}) \oplus v_m(t_k)) = HW(v(t_{k-1}) \oplus v(t_k)) \qquad (6)$$

In devices that leak the Hamming weight distance, a vulnerability is produced when two intermediate masked values u_m and v_m are operated by an exclusive-OR operator $s_m = u_m \oplus v_m$

$$HW(s_m = u_m \oplus v_m) = HW(u \oplus v) = HW(s) \qquad (7)$$

A similar idea is behind the so-called second-order attacks [20]. Let u_m and v_m be two intermediate masked values, which occur at different nodes of the circuit in Fig:1, and at different instants of time t_1 and t_2, respectively. These attacks consist of processing both intermediate masked values with an exclusive-OR operator, in order to create a situation like that described in (7). The result of this processing is used as a model of power consumption.This model is correlated with a combination, based on one of the functions proposed in [14] or [20], of the current traces captured in such intermediate masked values.

Moreover, if no additional measures of protection are taken, there are several points in the structure presented in Fig:2, that are susceptible to being attacked due to some of the following vulnerabilities:

- Since Key_{FAKE} is known, if the attacker is able to determine Key_{MASK}, then Key_{REAL} would be obtained by simply applying (2). Note that, the output of *SBoxTrans* in (4) could be attacked by a first order SCA and,

hence, Key_{MASK} could be revealed. It is noteworthy that *SBoxTrans* applies over each of the 16 bytes *(i,j) (i=0..3, j=0..3)* that represents the state at the output of *ShiftRows*. This vulnerability could be avoided by concealing the output of $SBoxTrans_{(i,j)}(a_F(i,j))$ $(a_F(i,j) = 0..255)$ with a mask $M_h(i,j)$

$$SBoxTrans_{(i,j)}(a(i,j)) = Sbox(a(i,j)) \oplus \\ \oplus Sbox(a(i,j) \oplus Key_{MASK}(i,j)) \oplus M_h(i,j) \quad (8)$$

Equation (8) is the real system implemented for *SBoxTrans* and it will be used to obtain the experimental results. Such a mask is obtained by including a True Random Number Generator (TRNG) used to update its value for each encrypted plain text.

- It is observed that the output of *SubBytes* and the output of *SboxTrans* can be computed with an exclusive OR, leading to a new value S_{COMB}

$$S_{COMB} = Sbox(a_R(i,j)) \oplus M_h(i,j) \quad (9)$$

If such a value was not protected by the mask $M_h(i,j)$, then the system would be vulnerable to a second-order attack, since S_{COMB} reveals the bytes $a_R(i,j)$ encrypted with Key_{REAL}. Note that, although second order attacks are usually performed over masked systems, here we use the same terminology since the attack on S_{COMB} exploits the leakage of two intermediate values (i.e. $Sbox(a_F(i,j))$ and $SBoxTrans(a_F(i,j))$.

- The function *MixCol*, included in the coprocessor, operates with the exclusive-OR on several bytes of different rows obtained at the output of *SBoxTrans*. Note that these bytes are concealed with the mask $M_h(i,j)$. As described in (7), if the same mask m is used for all bytes*(i,j)* of $M_h(i,j) = m$, there will be intermediate values unmasked by the effect of this operation. As shown in Fig:2, to avoid this situation of risk, four different masks m_0, m_1, m_2 and m_3 are used to conceal each row at the output of *SBoxTrans* [14]. Therefore, $M_h(i,j)$ will be represented by a 4x4 matrix in which the value

of all bytes of a row n are identical and fixed by the corresponding mask m_n *(n=0..3)*. Consequently, the rows of state at the output of the coprocessor will be concealed by four new masks m_a, m_b, m_c and m_d, whose values are obtained by the simple application of the *MixColumns* operation on the masked matrix $M_h(i,j)$. Then, a new 4x4 matrix $M_k(i,j)$ formed by such masks is created. Note that, this new matrix can be also pre-computed before executing the encryption algorithm. Finally, it is observed that as, the expanded key and $M_k(i,j)$ are operated with an exclusive-OR before *AddRoundKey* is activated, any SCA attack on such rounds would also reveal the fake key Key_{FAKE}.

- Additionally, as the coprocessor uses a different set of masks at the input and output of MixCol, the possibility of performing a second-order attack between both intermediate values is eliminated.

As will be described in the next section, these and other measures of protection are taken into account when designing the internal architecture of the coprocessor.

3. Architecture of the coprocessor

Some aspects related to the design of the coprocessor depend on how the microprocessor manages the execution of instructions and the access to data stored in memory. Fig:3 shows a simulation that represents the processing of a byte when the microprocessor computes the *SubByte* function over the particular value $v_m = 0x05$. The microprocessor employed for this purpose is the MicroBlaze, which is also used for obtaining the experimental results. This microprocessor consists of five pipeline stages, so that the execution of an instruction is performed five cycles after its fetch is produced. The first two rows of Fig:3 represent both the address (*m_abus*) and the data (*sl_dbus*) bus connected to the instruction memory. The calculation of $SBox(0x05)$ is expanded from cycle *CLK_1* to cycle *CLK_8*. In *CLK_9* a new processing over a different value

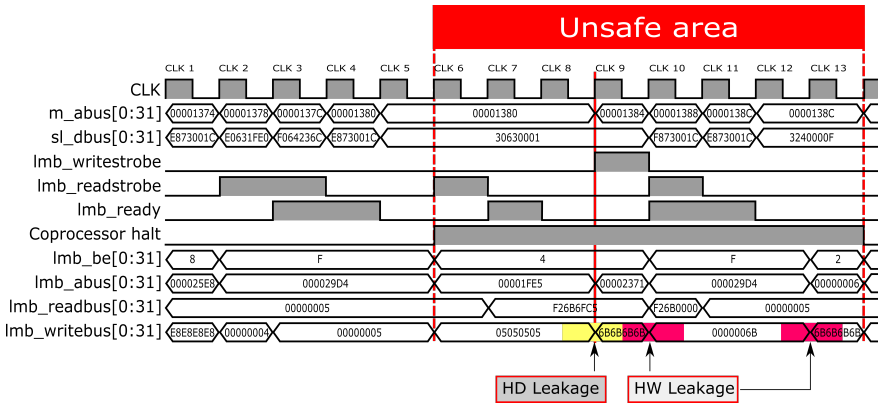

Figure 3: Representation of the behavior of signals (buses) related to MicroBlaze when evaluating the function SubBytes.

is started. The pre-computed table related to *SBox* is stored at base address *0x01FE0*. The last three rows represent the behavior of buses connected to data memory: address (*lmb_abus*), read (*lmb_readbus*) and write (*lmb_writebus*). The address, in which v_m is located (*0x01FE5 = 0x01FE0 + 0x05*), is placed on the *lmb_abus* line at cycle *CLK_6*, according to the specific pipeline structure of the microprocessor. Note that, in such a cycle the *lmb_writebus* is also charged with v_m, which is the output of function *ShiftRows*. Such a byte is processed by the *SubBytes* function providing as result the value $u_m = 0x6B$, which is placed in bus *lmb_writebus* during several clock cycles. Thus, the time interval that covers from cycle *CLK_6* to cycle *CLK_13* is the period in which the microprocessor is vulnerable to an SCA attack, since during this period of time *lmb_writebus* contains either the output of functions *ShiftRows* or *SubBytes*. Particularly, in cycle *CLK_9* the leakage of information can be utilized to evaluate the Hamming distance (HD), whereas in cycles *CLK_10* to *CLK_13* the actual leakage is useful for calculating the Hamming weight distance (HW). Thus, if the idea of the proposed countermeasure is to reveal a false key, during such time interval, described as unsafe area in Fig:3, the coprocessor must be disabled in order to facilitate an SCA attack that produces the highest correlation coefficient. These conclusions are experimentally corroborated by Fig:4, which shows the calcula-

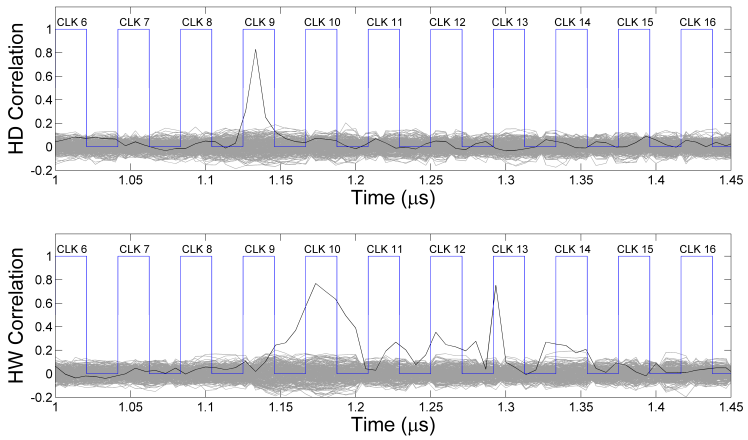

Figure 4: correlation coefficient based on HD (upper) and HW (lower) distances, at different instants of time included inside the unsafe area.

tion of the correlation coefficient during all instants of time included into the unsafe area and using the power models based on HD and HW, respectively. These figures were obtained by applying to each clock cycle the attack process described in section 2.1, and representing only the value obtained for the highest correlation coefficient related to the false key. The HD model is quite effective in *CLK_9*, and it could be easily calculated since the previous ($v_m = 0x05$, output of *ShiftRows*) and the subsequent ($u_m = 0x6B$, output of *SubBytes*) values written in *lmb_writebus* are known. In contrast, note that when a byte is written on such a bus, its 24 most significant bits are pre-charged to 0 during three clock cycles. This behavior makes the HW the most effective model for the rest of the unsafe area. As can be seen in Fig:4 (lower trace), in accordance with the simulation results, in cycles *CLK_10* and *CLK_13* the value of the correlation coefficient is quite significant.

The coprocessor knows the area of memory in which the *SBox* table is stored, so that it could be disabled by simply monitoring the address bus *lmb_abus*. In Fig:3, this situation is represented by the signal *Coprocessor_halt*, which is activated when such memory space is addressed.

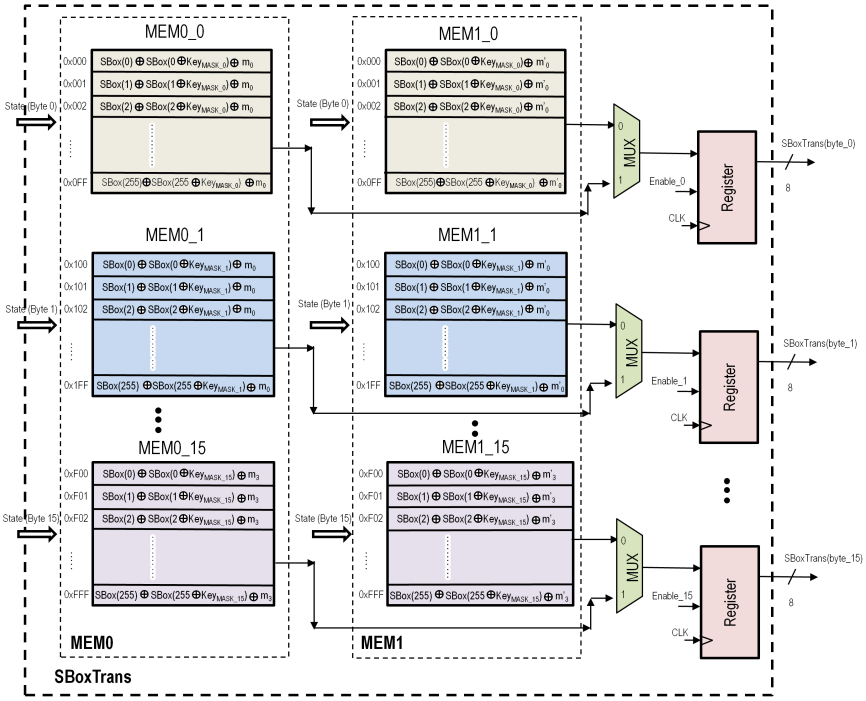

Figure 5: Internal architecture for *SboxTrans*.

The coprocessor is designed including two independent blocks for implementing the operations *SboxTrans* and *MixCol*; Fig:5 and Fig:6 represent their internal structure.

As can be seen, *SBoxTrans* consists of two groups of 16 memories implemented using blocks of BRAM included in the FPGA. Each of these two groups of memories, denoted as MEM0 and MEM1, are concealed with 2 different sets of masks $M_h(i,j)$ *(i=0..3, j=0..3)* and $M_{h'}(i,j)$ *((i=0..3, j=0..3))*, respectively. Each individual memory stores the 256 pre-computed values obtained by applying (8). These two groups are included in order to facilitate the updating of masks $M_h(i,j)$ (or $M_{h'}(i,j)$) used to protect the output of *SBoxTrans*, in such a way that each new plaint text is encrypted using a different set of masks $M_h(i,j)$ or $M_{h'}(i,j)$. The process for updating the value of such masks is as follows:

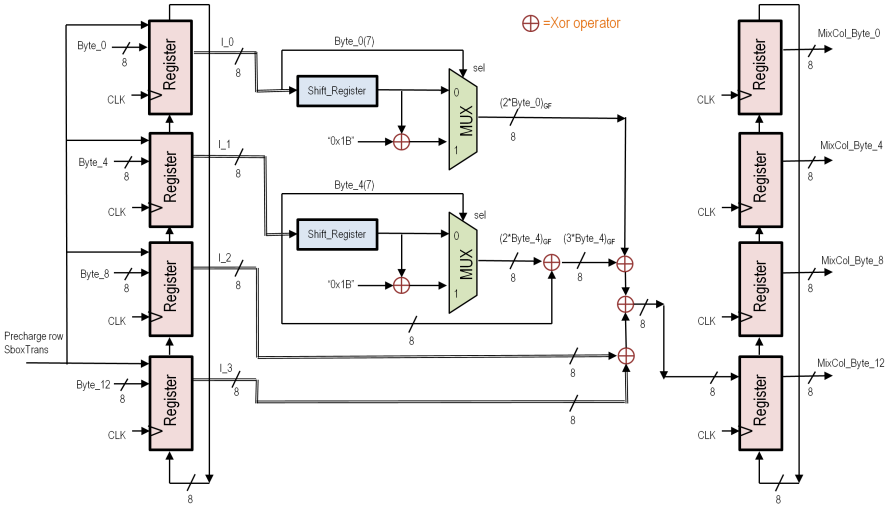

Figure 6: Internal architecture for block MixCol included in the coprocessor.

- If during the encryption of plain text T_i the output of *SBoxTrans* is concealed with masks $M_h(i,j)$ (included in MEM0), then in the following plain text T_{i+1} such output will be concealed with masks $M_{h'}(i,j)$ (included in MEM1) and vice versa.

- The True Random Number Generator (TRNG), included as part of the coprocessor, creates a new set of masks $M_h(i,j)_{CREATED}$ and $M_{h'}(i,j)_{CREATED}$ for each encrypted plain text T_i and T_{i+1}, respectively.

- The created mask $M_h(i,j)_{CREATED}$, and the actual values of its corresponding group of memories concealed with $M_h(i,j)_{OLD}$, are operated with an exclusive-OR. Thus, the elements of such memories will be concealed with a new mask $M_h(i,j)_{NEW}$ whose value is $M_h(i,j)_{CREATED} \oplus M_h(i,j)_{OLD}$. The second group of memories related to $M_{h'}(i,j)$ are updated in a similar way.

- The operations of creating and updating the masks are performed during the execution of rounds 5 and 6. Note that, when such rounds are

processed the state depends on the 128 bits of the cryptographic key, so that performing an SCA attack is considered unpractical. This way, the interfering noise created when updating these masks does not affect any potential attack performed on rounds in which the system is vulnerable and can reveal the false key (first and last round).

On the other hand, the signal *Enable_t (t=0..15)* is activated according to the byte of state that is being processed. The time needed by *SBoxTrans* to process the 16 input bytes is 17 clock cycles (T_{CLK}).

The block *MixCol* is implemented by multiplying each column of the output of *SboxTrans* by a fixed polynomial $3x^3 + x^2 + x + 2$ modulo $x^4 + 1$. The operation sum in $GF(2^8)$ can be performed by a simple XOR gate, whereas the multiplication is defined by many authors using the so-called function *xtime()* [19]. As Fig:6 shows, such a function can be implemented by including a shift-register, a multiplexor and a XOR gate. Thus, firstly a complete 32-bit column *(j=0..3)* of *SboxTrans* is pre-charged on a circular shift register formed by 4 synchronous 8-bit registers. Afterwards, in each of the following four clock cycles, one element of the output column is calculated based on function *xtime()*, and such element is stored in a second set of circular shift registers. Once this process is finished, the following column is pre-charged and the process, for computing the rest of columns of *SboxTrans*, is initiated again. The coprocessor *MixCol* performs all these computations in $24 \cdot T_{CLK}$.

4. Experimental and simulation results

4.1. Area and correlation results

In order to prove the correctness of our proposal, the complete system was implemented on a Virtex-5 FPGA clocked at 24 MHz. Power traces were measured using a Tektronix CT-1 current probe featuring a bandwidth range of 25 kHz to 1 GHz. The current probe was connected to an Agilent DSO1024A oscilloscope, which captures and stores current traces using a sample rate of 2 GS/s.

Table 1: Area and maximum clock frequency F_{max} Percentage (%) against the total number of resources in the FPGA

Subsystem	LUT (Lookup table)	Flip-Flop (Reg)	Slices	BRAM (36Kb)	F_{max} (MHz)
Microblaze	1628 (5.7%)	1498 (5.2%)	641 (8.9%)	16	160.94
Coprocessor	516 (1.8%)	567 (1.9%)	253 (3.5%)	2	227.17
Rest of peripherals	920 (3.2%)	709 (2.5%)	462 (6.4%)	–	180.64
Embedded System	**3064 (10.7%)**	**3100 (10.6%)**	**1356 (18.8%)**	**18 (37.5%)**	**160.94**

Table 2: Execution time of function *AddRoundkey*, *ShiftRows*, *SubBytes* and *MixColumns* when solved by Microblaze at 24 MHz and including coprocessor

Function	Execution time (μs)
AddRoundkey	44.75 μs (1074$\cdot$ T_{CLK})
ShiftRows + FSL Write	5.79 μs (139$\cdot$ T_{CLK})
SubBytes	26.58 μs (638$\cdot$ T_{CLK})
MixColumns + FSL Read + Remasking	48.95 μs (1175$\cdot$ T_{CLK})
Execution time for one round)	**126.07 μs (3026$\cdot$ T_{CLK})**

The implemented system includes a MicroBlaze microprocessor, a specific hardware (coprocessor) that synthetizes both the *SboxTrans* and *MixCol* blocks, and finally a set of peripherals used for debugging the application and providing the synchronization signals needed to capture properly the current traces. The microprocessor, the hardware and the peripherals are connected together by means of a Processor Local Bus (PLB), although the communication between the specific hardware and the microprocessor is performed through the Fast Simplex Link (FSL) bus. The logic resources needed for implementing the overall system, and the maximum frequency given by the critical path, are represented in Table 1.

Basically, the MicroBlaze executes the AES 128-bit algorithm by encrypting the plain text with Key_{FAKE}. Table 2 shows the execution time of each of the

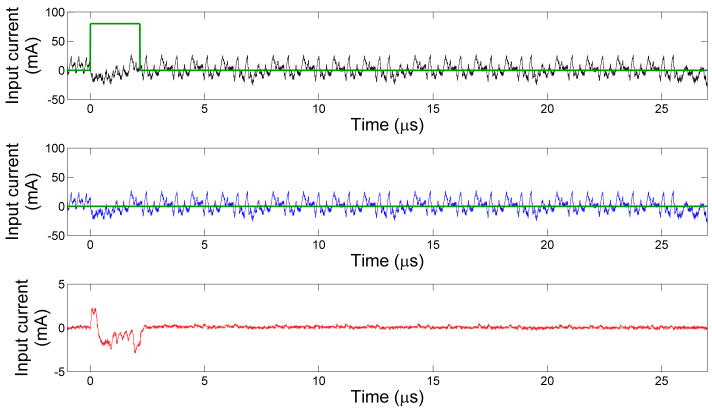

Figure 7: Input current when the coprocessor concentrates (no wait-state cycles were introduced) its activity at the beginning of the processing of *SubBytes* by the microprocessor: a) Coprocessor activated (upper trace), b) Coprocessor disabled (middle trace) and c) Difference between upper and middle traces.

four operations for a specific round. As can be seen, the *SubBytes* operation is solved in $638 \cdot T_{CLK}$, whereas *MixColumns* is executed in $1175 \cdot T_{CLK}$. However, as blocks *SubTrans* and *MixCol* are implemented in hardware, their resolution, according to their internal structure shown in Fig:5 and Fig:6, is faster and could be performed in $17 \cdot T_{CLK}$ and $24 \cdot T_{CLK}$, respectively. These blocks are activated by the microprocessor, which sends an activation signal through the FSL bus. In fact, as Table 2 shows, the delay related to the communication between both systems (FSL bus write or FSL bus read) is included as part of the operations *ShiftRows* (writing) and *MixColumns* (reading). Such processes of writing and reading are used for transferring through the FSL bus the actual 16 bytes of state.

Fig:7 compares the total input current consumed by the device when the coprocessor is activated (Fig:7a) or disabled (Fig:7b). The additional power consumption provided by its activation is concentrated on the next $51 \cdot T_{CLK}$ (2.12 μs), which includes both the processing and the communication delays. The difference between the power consumption in both situations, activated or

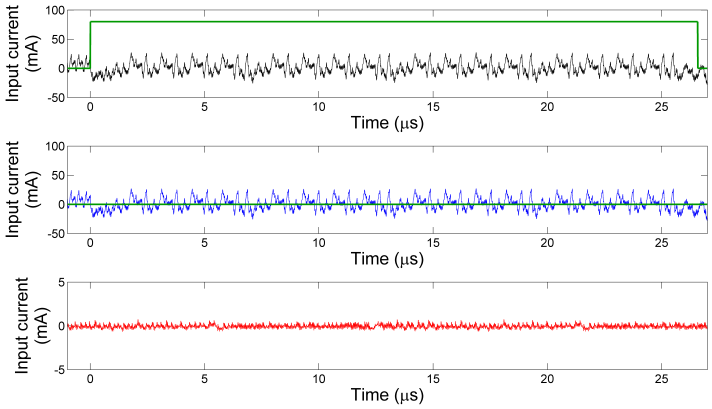

Figure 8: Input current when the coprocessor introduces wait-state cycles and distributes its activity along the interval of time in which *SubBytes* is processed by the microprocessor: a) Coprocessor activate (upper trace), b) Coprocessor disabled (middle trace) and c) Difference between upper and middle traces.

disabled, is perfectively distinguishable and measurable. Such difference, shown in Fig:7c, is higher than 5 mA (absolute value). Moreover, the energy consumed by the coprocessor is added on the global power, so that such energy behaves as a noise that deteriorates the correlation coefficient related to Key_{FAKE}. In order to conceal such information, the coprocessor is slowed down by introducing wait-state cycles, in such a way that its activity is distributed along the total time needed by the microprocessor for executing the individual operation *SubBytes*. This proposal is shown in Fig:8, in which it can be observed as the coprocessor distributes the calculation of functions *SubTrans* and *MixCol* during the $638 \cdot T_{CLK}$ (26.58 μs) employed for executing *SubBytes*. Thus, a total of 587 wait-state cycles have been introduced. Furthermore, the difference between the input current traces when the coprocessor is activated or disabled (Fig:8.c) is nearly unnoticeable (0.5 mA), so that the correlation coefficient will be unaffected by the addition of the faking countermeasure. In fact, the value represented in Fig:8.c for such difference is almost constant whatever the state of the coprocessor is. Additionally, taking into account that the actual values

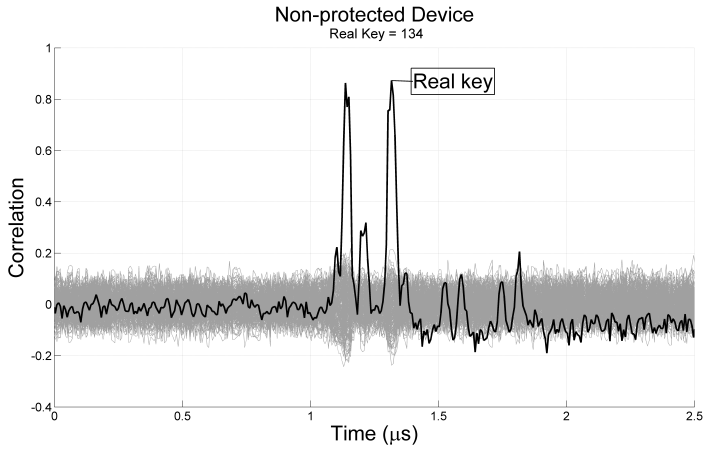

Figure 9: correlation coefficient calculated when processing operation SubBytes by the microprocessor for a non-protected system.

processed by the coprocessor are concealed with $M_h(i,j)$, an SCA attack on *SBoxTrans* or *MixCol* would be unsuccessful even in the hypothetical situation in which its power consumption could be isolated from the rest of the system.

Fig:9 and Fig:10 show the result of an SCA attack performed during the execution of the operation *SubBytes*. Such figures compare the behavior of the correlation coefficient when the coprocessor is disabled (Fig:9) or activated (Fig:10) using the power model based on the HW. Note that, according to Fig:3 and 4, there are two points in which the leakage information provides a successful result using such a model of power. Moreover, the power consumption of the coprocessor does not affect the calculation of the correlation, which corroborates the benefits of distributing its activity along the time interval in which the function *SubBytes* is processed.

Fig:11 and 12 show a simulation that confirms the need of protecting *SboxTrans* with a mask $M_h(i,j)$. The simulation represents a second-order SCA attack performed by combining both the output of *SboxTrans* and the output of the *SubBytes* by processing the absolute difference of both signals [14]. Results show whether the system is vulnerable if the attack is carried out without concealing the data, and how such vulnerability disappears when mask $M_h(i,j)$

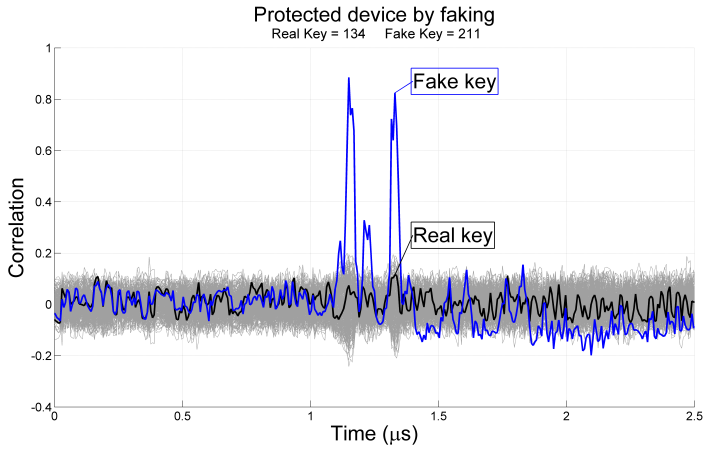

Figure 10: correlation coefficient calculated when processing operation *SubBytes* by the microprocessor using the faking countermeasure. The Key_{FAKE} is plotted in blue and the Key_{REAL} in bold.

is included.

The experiments shown in Fig:13 (non-protected) and Fig:14 (protected) represent the evolution of the maximum correlation coefficient over an increasing number of plain texts for an attack performed on function *SubBytes*. These results are almost identical when the coprocessor is activated or disabled, so that the attacker is unable to find out if the system is protected by the faking countermeasure or not. As can be seen, capturing about 25 power traces is enough to reveal the false key. Similarly, Fig:15 shows the same attack performed on *MixColumns* function which leads to identical results.

Finally, Fig:16 shows an attack based on the differential-of-means method proposed by Kocher. Unlike the original attack, which was performed on a single bit, our proposal is targeted on a complete byte following a similar strategy that introduces some modifications:

- The process is applied on each bit j included in the byte to be analyzed.
- For the specific bit j, in which the attack is initially focused, the N current traces are separated into two groups, depending on the value that such a

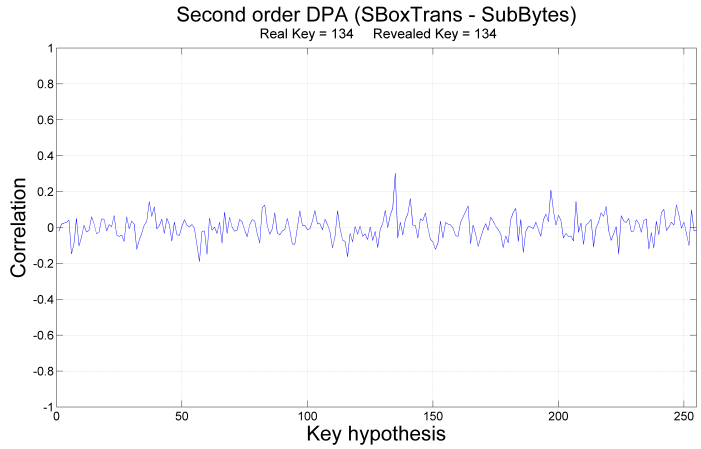

Figure 11: Second-order attack on functions *SubBytes* and *SboxTrans*. System non-protected by mask $M_h(i, j)$.

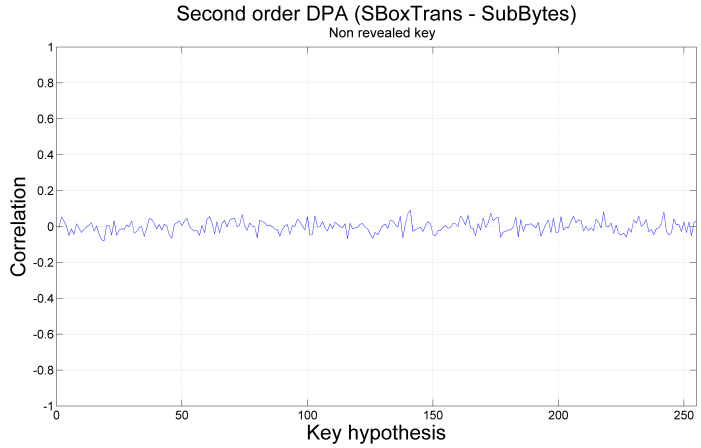

Figure 12: Second-order attack on functions *SubBytes* and *SboxTrans*. System protected by mask $M_h(i, j)$.

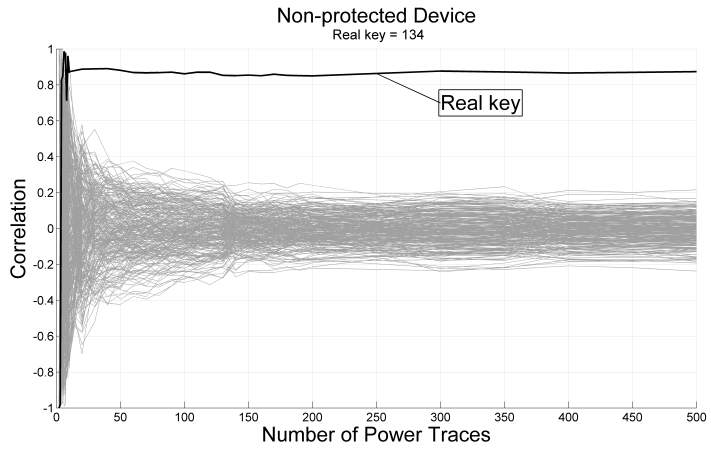

Figure 13: Experimental attack on *SubBytes* for a non-protected system. Evolution of the correlation over an increasing number of plain texts.

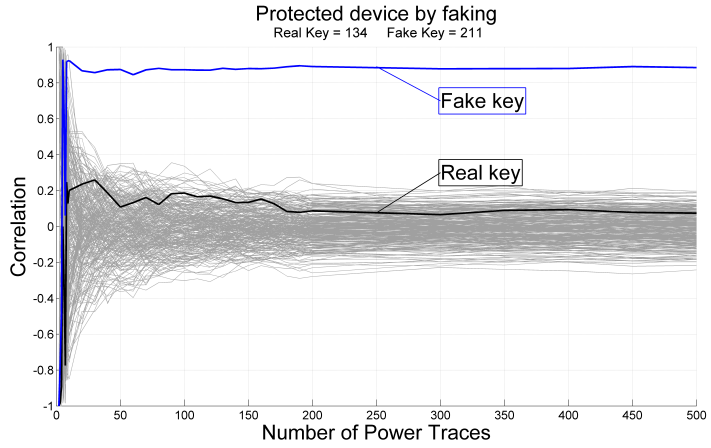

Figure 14: Experimental attack on *SubBytes* for a system protected by the faking countermeasure. Evolution of the correlation over an increasing number of plain texts. The Key_{FAKE} is plotted in blue and the Key_{REAL} in bold.

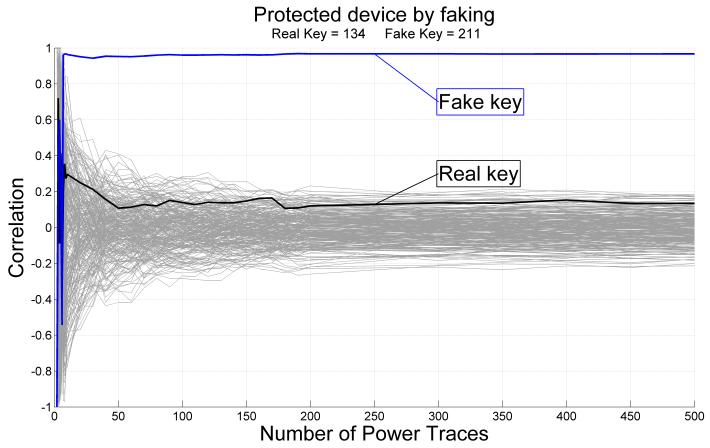

Figure 15: Experimental attack on *MixColumns* for a system protected by the faking countermeasure. Evolution of the correlation over an increasing number of plain texts. The Key_{FAKE} is plotted in blue and the Key_{REAL} in bold.

bit takes on the power consumption model for a particular plain text and a specific key K_n *(n=0..255)*.

- For each key K_n, the average of each group is calculated and the difference between each average is assigned to the element *d(j,n) (j=0..7, n=0..255)* of a matrix D.

- The process is repeated for all bits and keys until matrix D is completed.

- For each column n of matrix D, its average value D_n *(n=0..255)* is calculated. The maximum value of D_n indicates the correct key.

In order to obtain a comparison between the hardware software implementation presented in this paper and a completely software execution, several plain texts have been encrypted by executing the faking countermeasure using only the microprocessor (the coprocessor is disabled). The results are shown in Table 3 . In this case, the execution time for one round is $4847 \cdot T_{CLK}$ (about 202μs), which is almost twice when compared with the time needed by our proposal shown in Table 2. Note that, the execution time of the same function

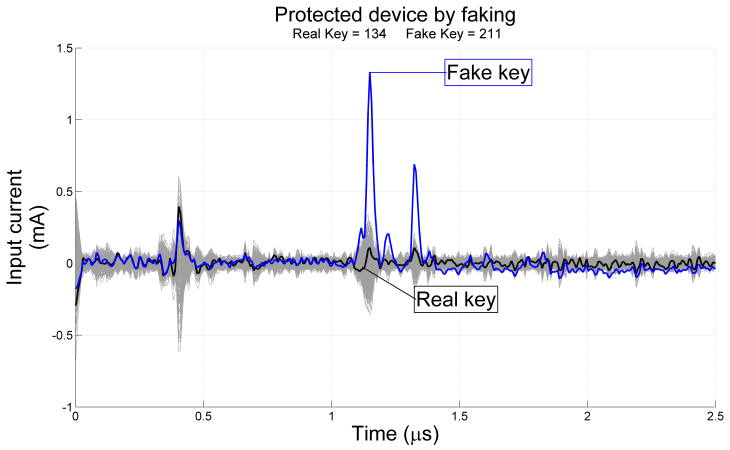

Figure 16: Experimental attack on *SubBytes* using the differential-of-means method. System protected by the faking countermeasure. The Key_{FAKE} is plotted in blue and the Key_{REAL} in bold.

performed by the microprocessor in both experiments shows some slight differences depending on whether the coprocessor is or not activated. For instance, in the complete software implementation the function *MixColumns* is used for both processing the state and calculating the block *MixCol*. Thus, in order to differentiate both situations an additional processing should be included, which produces such difference in the execution time. Finally, the third column of Table 3 presents the results for one round when the faking countermeasure is disabled and the plain text is encrypted by the microprocessor (reference system). As can be seen, the execution time is almost identical when compared with the hardware/software implementation, being the main difference due to the delay communications created when writing or reading on the FSL bus (communication between the Microblaze and the coprocessor).

4.2. Comparison with other proposals

A fair comparison of our proposal against previous publications should be carefully performed. The results, in terms of area and speed, depend strongly on the FPGA family used for implementing the system. Thus, the coprocessor was

Table 3: Execution time of function *AddRoundkey*, *ShiftRows*, *SubBytes* and *MixColumns* when solved by Microblaze at 24 MHz and including coprocessor

Function	Execution time (μs) (including faking)	Execution time (μs) (without faking)
AddRoundkey	52.76 μs (1266· T_{CLK})	36.16 μs (940· T_{CLK})
ShiftRows	4.04 μs (97· T_{CLK})	4.04 μs (97· T_{CLK}
SubBytes	26.58 μs (638· T_{CLK})	26.58 μs (638· T_{CLK})
MixColumns	54.03 μs (1297· T_{CLK})	43.17 μs (1036· T_{CLK})
SBoxTrans	4.46 μs (107· T_{CLK})	–
MixCol	54.03 μs (1297· T_{CLK})	–
Remasking	6.04 μs (145· T_{CLK})	–
Execution timefor one round	**201.94 μs (4847· T_{CLK})**	**112.95 μs (2711· T_{CLK})**

additionally synthetized on a Virtex 4 and a Spartan 6 FPGA. Such FPGAs were used by those publications in which the comparison performed in this section is based. On the other hand, almost all such publications are based on a pure hardware implementation, in contrast to our proposal which is based on a hardware-software co-design. Thus, as the coprocessor only performs the countermeasure to protect the real key, its performance is usually higher.

Table 4 shows such comparison for different protected implementations of the AES algorithm performed on several FPGAs. As the coprocessor is masked, only designs protected by masking were included.

The implementation proposed by *Reggazoni et al.* [21] is performed on a Virtex 5. Authors presented two different structures based on a datapath of 32-bit and 128-bit, respectively.As the *MixCol* block included in the coprocessor is based on a 32-bit datapath, results are compared according to this design. As can be seen, our implementation is carried out in 253 slices, so that only the 40% of the logical resources used in that publication are needed for implementing the coprocessor. However, the maximum throughput provided by the implementation of *Reggazoni* is higher leading to a better result. Note that, our coprocessor was designed aiming at minimizing its power consumption per clock cycle. Such a strategy provides the highest correlation coefficient related

Table 4: Comparative study. Previous implementations on different FPGAs.

Proposal	LUT (Lookup tables)	Flip-Flop (Registers)	Slices	BRAM	F_{max} (MHz)	Maximum Throughput (Mbits/s)
Coprocessor designed (Virtex 5)	516	567	253	2	227	98
Coprocessor designed (Virtex 4)	1233	693	810	4	175	75
Coprocessor designed (Spartan 6)	518	690	303	4	151	65
Reggazoni el at. [21] (Virtex 5) (32-bit datapath)	1429	643	637	–	100	290
Kaumon et at. [17] (Virtex 4)	–	–	1491	–	143	–
Güneysu et al. [22] (implemented by [23] in Spartan 6)	2888	2351	–	16	147	35
Sasdrich et at. [23] (Spartan 6)	1284	415	–	8	148	68

to the false key, but at the expense of producing a lower throughput.

The protected system proposed by *Kaumon et al.* [17] is implemented in a Virtex 4. Only results about slices are presented by the authors.Using the same FPGA the coprocessor needs about the 54% (810 slices) of the logical resources used in that publication (1491 slices).

On the other hand, *Güneysu et al.* [22] presented the concept of Block Memory content Scrambling (BMS); a structure based on dual-port BRAM primitives that offers protection at the cost of increasing the reconfiguration time for the mask update. When compared with the coprocessor synthetized on a Spartan 6, the number of LUTs, Flip-Flops and BRAM memories used by their design is larger.

The paper presented by *Sasdrich et al.* [23] shows an alternative implementation of BMS, but using distributed RAM memory based on Slice-M LUTs available in modern FPGAs. They reduce by half the resources needed by the original proposal of *Güneysu*, and additionally, the throughput is increased by two. Our coprocessor, designed in a Spartan 6, could be implemented using less LUTs (518 against 1284) but more Flip-Flops (690 against 415 FF) than this improved proposal. Results for the maximum frequency and throughput are almost identical.

5. Conclusions

A new countermeasure against SCA attacks and its implementation based on a software/hardware co-design was presented. The effectiveness of such countermeasure relies on revealing a false key, rather than eliminating the statistical dependence between data and power consumption that is usually performed by classical approaches. Functions implemented in hardware by the coprocessor hardly affect the power consumption, so that their effect on revealing the false key by means of a correlation attack is unnoticeable. In contrast, as the countermeasure is solved in parallel with the execution of the original algorithm by the microprocessor, there is no penalty on the execution time. The complete system was implemented on a Virtex 5 FPGA in order to obtain experimental results that corroborated the efficiency of our proposal.

Acknowledgment

This work was supported by the Ministerio de Economía y Competitividad in the framework of the Programa Nacional de Proyectos de Investigación Fundamental, project TEC2012-38329-C02-02.

References

[1] J. J. P. C. Kocher, B. Jun, Differential power analysis, Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA vol. 1666 (1999) pp. 388–397.

[2] M. A. Kris Tiri, I. Verbauwhede, A dynamic and differential cmos logic with signal independent power consumption to withsand differential power analysis on smart cards, ESSCIRC (2002) pp. 403–406.

[3] K. Tiri, I. Verbauwhede, A logic level design methodology for a secure dpa resistant asic or fpga implementation, Automation and Test in Europe Conference and Exposition, Paris, France. IEEE Computer Society vol. 1 (2004) pp. 246–251.

[4] M. S. PD. Suzuki, T. Ichikawa, Random switching logic: A countermeasure against dpa based on transition probability, Cryptology ePrint Archive (http: //eprint.iacr.org/) Report 2004/346.

[5] Z. Chen, Y. Zhou, Dual-rail random switching logic: A countermeasure to reduce side channel leakage, Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan vol. 4249 of Lecture Notes in Computer Science (2006) pp. 242–254.

[6] T. Popp, S. Mangard, Masked dual-rail pre-charge logic: Dpa-resistance without routing constraints, Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK vol. 3659 of Lecture Notes in Computer Science (2005) pp. 172–186.

[7] H. Z Chen-S, P. Schaumont, Side-channel leakage in masked circuits caused by higher-order circuit effects, International Conference on Information Security and Assurance (ISA2009) (2009) pp 327–336.

[8] A. Moradi, V. Immler, Early propagation and imbalanced routing, how to diminish in fpgas, Cryptographic and Embedded Systems - CHES 2014 Lecture Notes in Computer Science, Vol. 8731 (2014) pp 598–615.

[9] D. Suzuki, M. Saeki, Security evaluation of dpa countermeasures using dual-rail pre-charge logic style, Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan vol. 4249 of Lecture Notes in Computer Science (2006) pp 255–269.

[10] T. Z. T. Popp, M. Kirschbaum, S. Mangard, Evaluation of the maked logic style mdpl on a prototype chip, Proc. Cryptographic Hardware and Embedded Systems (CHES 07), Vienna, Austria.

[11] W. P. M. R. P. McEvoy, C. C. Murphy, M. Tunstall, Isolated wddl: A hiding countermeasure for differential power analysis on fpgas, ACM Trans. Reconfigurable Technol. Syst. vol. 2 (num. 1) (2009) pp. 1–23.

[12] E. d. l. T. W. He, T. Riesgo, A precharge-absorbed dpl logic for reducing early propagation effects on fpga implementations, In ReConFig2011. IEEE Computer Society (2011) pages 217–222.

[13] P. H. e. a. S. Guilley, L. Sauvage, Security evaluation of wdll and seclib countermeasures against power attacks, IEEE Transactions on Computers vol. 57 (no. 11) (2008) pp. 1482–1497.

[14] E. O. Stefan Mangard, T. Popp, Power Analysis Attacks – Revealing the Secrets of Smart Cards, Springer, 2007.

[15] J. S. C. C. Clavier, N. Dabbous, Differential power analysis in the presence of hardware countereasures, Proceedings of CHES vol. 1965 of Lecture Notes in Computer Science (2000) pp. 253–263.

[16] C. H. S. Tillich, S. Mangard, Protecting aes software implementations on 32-bit processors against power analysis, Proceedings of ACNS vol. 4521 of Lecture Notes in Computer Science (2007) pages 141–157.

[17] L. B. Najeh Kamoun, A. Ghazel, Correlated power noise generator as low cost dpa countermeasures to secure hardware aes chiper, 3er. International Conference on Signals, Circuits and Systems (2009) pp. 1–6.

[18] J. Daemen, V. Rijmenc, The design of Rijndael: AES - The advanced Encryption Standard, Vol. 19, Springer-Verlag, 2002.

[19] V. R. Joan Daemen, Aes proposal: Rijndael, Available at http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[20] T. S. Messerges, Using second-order power analysis to attack first dpa resistance software, Cryptographic Hardware and Embedded Systems - CHESS 2000. Lecture Notes in Computer Science vol. 1965 (2000) pp 238–251.

[21] F. Regazzoni, Y. Wang, F.-X. Standaert, Fpga implementations of the aes masked against power analysis attacks, Proceedings of COSADE 2011, International Workshop on Side-Channel Analysis and Secure Design, Darmstadt.

[22] T. Gneysu, A. Moradi, Generic side-channel counter-measures for reconfigurable devices, Cryptographic Hardware and Embedded Systems - CHES 2011, Nara, Japan vol. 6917 of LNCS (1-12) (2011) pag. 33–48.

[23] P. Sasdrich, O. Mischke, A. Moradi, T. Gneysu, Constructive side-channel analysis and secure design, 6th International Workshop, COSADE 2015 Revised Selected Papers: LNCS 9064 Springer International Publishing.