Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis

Matthieu Rivain[1,2], Emmanuelle Dottax[2], Emmanuel Prouff[2]

[1] University of Luxembourg
[2] Oberthur Card Systems
{m.rivain,e.dottax,e.prouff}@oberthurcs.com

Abstract. In the recent years, side channel analysis has received a lot of attention, and attack techniques have been improved. Side channel analysis of second order is now successful in breaking implementations of block ciphers supposed to be effectively protected. This progress shows not only the practicability of second order attacks, but also the need for provably secure countermeasures. Surprisingly, while many studies have been dedicated to the attacks, only a few papers have been published about the dedicated countermeasures. In fact, only the method proposed by Schramm and Paar at CT-RSA 2006 enables to thwart second order side channel analysis. In this paper, we introduce two new methods which constitute a worthwhile alternative to Schramm and Paar's proposal. We prove their security in a strong security model and we exhibit a way to significantly improve their efficiency by using the particularities of the targeted architectures. Finally, we argue that the introduced methods allow us to efficiently protect a wide variety of block ciphers, including AES.

1 Introduction

Side Channel Analysis (SCA) is a cryptanalytic technique that consists in analyzing the physical leakage (called *side channel leakage*) produced during the execution of a cryptographic algorithm embedded on a physical device. SCA exploits the fact that this leakage is statistically dependent on the intermediate variables that are processed, these variables being themselves related to secret data. Different kinds of leakage can be exploited. Most of the time SCA focuses on the execution time [12], the power consumption [13] or the electromagnetic emanations [8].

Block ciphers implementations are especially vulnerable to a powerful class of SCA called *Differential* SCA (DCSA) [4, 13]. Based on several leakage observations, a DSCA-attacker estimates a correlation between the leakage and different predictions on the value of a sensitive variable. According to the obtained correlation values, this attacker is able to (in)validate some hypotheses on the secret key. An alternative to DSCA exists when profiling the side channel leakage is allowed. The so-called *Profiling* SCA [6, 24] is more powerful than DSCA, but it assumes a stronger adversary model. Indeed, a Profiling SCA attacker has a device under control, which he uses to evaluate the distribution of the side channel leakage according to the processed values. These estimated distributions are then involved in a maximum likelihood approach to recover the secret data of the attacked device. Profiling attacks are not only more efficient than DSCA but they are also more effective since they can target the key manipulations.

A very common countermeasure against SCA is to randomize sensitive variables by masking techniques [5, 9]. The principle is to add one or several random value(s) (called *mask(s)*) to each sensitive variable. Masks and masked variables (together called the *shares*) propagate throughout the cipher in such a way that every intermediate variable is independent of any sensitive variable. This strategy ensures that the instantaneous leakage is independent of any sensitive variable, thus rendering SCA difficult to perform. Two kinds of masking can be distinguished: the hardware masking (that is included at the logic gate level during the design of the device) and the software masking (that is included at the algorithmic level). Hardware masking is expensive in terms of silicium area and it has some security flaws. In particular, the shares are usually processed at the same time. As a consequence the instantaneous leakage is actually dependent on the sensitive variables, which makes some dedicated attacks possible [19,28]. Other vulnerabilities come from physical phenomena such as glitches [16] or propagation delays [27]. Compared to hardware masking, software masking does not imply any overhead in silicium area, but it usually impacts the timing performances and the memory requirements. Regarding security, it does not suffer from the previous flaws and it is therefore widely used to protect block ciphers implementations.

The masking can be characterized by the number of random masks that are used per sensitive variable. A masking that involves d random masks is called a d^{th} *order masking*. When a d^{th} order masking is used, it can be broken by a $(d+1)^{th}$ *order SCA*, namely an SCA that targets $d+1$ intermediate variables at the same time. Indeed, the leakages resulting from the $d+1$ shares (*i.e.* the masked variable and the d masks) are jointly dependent on the sensitive variable. Whatever the order d, such an attack theoretically bypasses a d^{th} order masking [21]. However, the noise effects imply that the difficulty of carrying out a d^{th} order SCA in practice increases exponentially with its order [5, 25] and the d^{th} *order*

SCA resistance (for a given d) is thus a good security criterion for block cipher implementations.

Though block ciphers can theoretically be protected against d^{th} order SCA by using a d^{th} order masking, the actual implementation reveals some issues. The main difficulty lies in performing all the steps of the algorithm by manipulating the shares separately, while being able to rebuild the expected result. As we will see, non-linear layers – crucial for the block cipher security – are particularly difficult to protect. Only a few proposals have been made regarding this issue and actually none of them provides full satisfaction. A first attempt has been made by Akkar and Goubin for the DES algorithm [2] – and improved in [1, 15] – but it rests on an ad-hoc security and it is not provably secure against second order SCA. A second proposal has been made by Schramm and Paar in [25] to secure an AES implementation against d^{th} order SCA but it has been broken in [7] for $d \geq 3$. Even if it seems to be resistant for $d = 2$, its security has not been proved so that there is nowadays no countermeasure provably secure against second order SCA.

The lack of solutions implies that the higher order SCA resistance still needs to be investigated. As a first step, resistance against second order SCA (2O-SCA) is of importance since it has been substantially improved and successfully put into practice [11, 14, 17–19, 28].

In this paper, we focus on block ciphers implementations provably secure against 2O-SCA. We first introduce in Sect. 2 notions about block ciphers. We recall how they are usually protected and we introduce the security model. We show that in this model, the whole cipher security can be simply reduced to the security of the S-box implementation. Then, two new generic S-box implementations are described in Sect. 3. We analyze their efficiency and we prove their security against 2O-SCA. In this section, we also propose an improvement that allows us to substantially speed up our solutions when several S-box outputs can be stored on one microprocessor word.

Because of length constraints, some results could not be included in the paper. They are given in the extended version [23]. In particular, in [23, Sect. 4] we compare our new proposal with the existing solutions, we give practical implementation results, and we discuss their requirements and their efficiency.

2 Block Ciphers Implementations Secure Against 2O-SCA

In this section, we introduce some basics about block ciphers and we explain how to implement such algorithms in order to guarantee the security against 2O-SCA. Then, we introduce a security model to prove the security of the proposed implementations.

2.1 Block Ciphers

A block cipher is a cryptographic algorithm that, from a secret key K, transforms a plaintext block P into a ciphertext block C through the repetition of key-dependent permutations, called *round transformations*. Let us denote by p, and call *cipher state*, the temporary value taken by the ciphertext during the algorithm. In practice, the cipher is *iterative*, which means that it applies R times the same round transformation ρ to the cipher state. This round transformation is parameterized by a *round key* k that is derived from K by iterating a key scheduling function α. We shall use the notations p^r and k^r when we need to precise the round r during which the variables p and k are involved: we have $k^{r+1} = \alpha(k^r, r)$ and $p^{r+1} = \rho[k^r](p^r)$, with $p^0 = P, p^R = C$ and $k^0 = \alpha(K, 0)$. Moreover, we shall denote by $(p)_j$ the j^{th} part of the state p.

The round transformation ρ can be further modeled as the composition of different operations: a key addition layer σ, a non-linear layer γ, and a linear layer λ:

$$\rho[k] = \lambda \circ \gamma \circ \sigma[k].$$

The whole cipher transformation can thus be written[3]:

$$C = \bigcirc_{r=0}^{R-1} \quad \lambda \circ \gamma \circ \sigma[k^r] \; (P).$$

Remark 1. The key scheduling function α can also be modeled as the composition of linear and non-linear layers.

The key addition layer is usually a simple bitwise addition between the round key and the cipher state and we have $\sigma[k](p) = p \oplus k$. The non-linear layer consists of several, say N, non-linear vectorial functions S_j, called *S-boxes*, that operate independently on a limited number of input bits: $\gamma(p) = \big(S_1((p)_1), \cdots, S_N((p)_N)\big)$. For efficiency reasons, S-boxes are

[3] This is not strictly the case for all iterated block ciphers. For instance, the last round of AES slightly differs from the iterated one. But this restriction does not impact on our purpose.

most of the time implemented as look-up tables (LUT). We will consider in this paper that the layer λ, that mixes the outputs of the S-boxes, is linear with respect to the bitwise addition.

As an illustration, Fig. 1 represents a typical round transformation with a non-linear layer made of four S-boxes. Note that this description is not restrictive regarding the structure of recent block ciphers. In particular, this description can be straightforwardly extended to represent the AES algorithm.

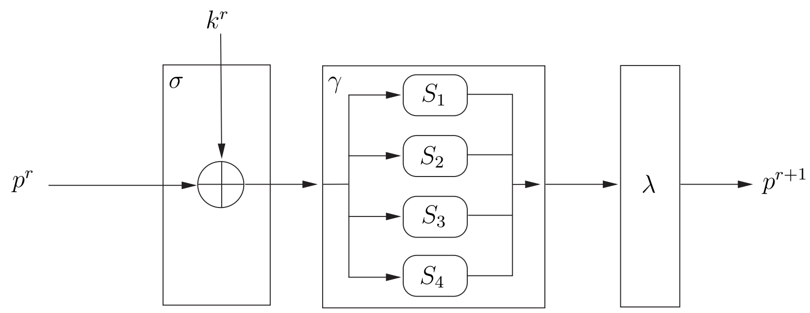

Fig. 1. A typical round transformation with a non-linear layer composed of four S-boxes.

2.2 Securing Block Ciphers Against 2O-SCA

In order to obtain a 2O-SCA resistant implementation of a block cipher, we use masking techniques [5, 9]. To prevent any second order leakage, random shares are introduced: the cipher state p and the secret key k are represented by three shares – (p_0, p_1, p_2) and (k_0, k_1, k_2) respectively – that satisfy the following relations:

$$p = p_0 \oplus p_1 \oplus p_2 \ , \tag{1}$$

$$k = k_0 \oplus k_1 \oplus k_2 \ . \tag{2}$$

In order to ensure the security, shares (p_1, p_2) and (k_1, k_2) – called the masks – are randomly generated. And in order to keep track of the correct values of p and k, shares p_0 and k_0 – called the masked state and the masked key – are processed according to Relations (1) and (2).

Remark 2. For an implementation to be secure against 2O-DSCA only, the key does not need to be masked. This amounts in our description to fix the values of k_1 and k_2 at zero. In such a case, the key scheduling function can be normally implemented.

At the end of the algorithm, the expected ciphertext (which corresponds to the final value p^R) is re-built from the shares $\left(p_0^R, p_1^R, p_2^R\right)$. To preserve the security throughout the cipher and to avoid any second order leakage, the different shares must always be processed separately. Thus, the point is to perform the algorithm computation by manipulating the shares separately, while maintaining Relations (1) and (2) in such a way that the unmasked value can always be re-established. To maintain these relations along the algorithm, we must be able to maintain them throughout the three layers λ, σ and γ.

For the linear layer λ, maintaining Relations (1) and (2) is simply done by applying λ to each share separately. Indeed, by linearity, $\lambda(p)$ and the new shares $\lambda(p_i)$ satisfy the desired relation:

$$\lambda(p) = \lambda(p_0) \oplus \lambda(p_1) \oplus \lambda(p_2) \ .$$

An easy relation stands also for the key addition layer σ where each k_i can be separately added to each p_i since we have:

$$\sigma[k](p) = \sigma[k_0](p_0) \oplus \sigma[k_1](p_1) \oplus \sigma[k_2](p_2) \ .$$

For the non-linear layer, no such a relation exists and maintaining Relation (1) is a much more difficult task. This makes the secure implementation of such a layer the principal issue while trying to protect block ciphers.

Because of the non-linearity of γ, new random masks p_1', p_2' must be generated. Then a masked output state p_0' has to be processed from p_0, p_1, p_2 and p_1', p_2' in such a way that:

$$\gamma(p) = p_0' \oplus p_1' \oplus p_2'.$$

Since γ is composed of several S-boxes, each operating on a subpart of p, the problem can be reduced to securely implement one S-box. The underlying problem is therefore the following.

Problem 1. Let S be an (n, m)-function (that is a function from $\mathbb{F}_2^n$ in $\mathbb{F}_2^m$). From a masked input $x \oplus r_1 \oplus r_2 \in \mathbb{F}_2^n$, the pair of masks $(r_1, r_2) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$ and a pair of output masks $(s_1, s_2) \in \mathbb{F}_2^m \times \mathbb{F}_2^m$, compute $S(x) \oplus s_1 \oplus s_2$ without introducing any second order leakage.

If the problem above can be resolved by an algorithm *SecSbox*, then the masked output state p_0' can be constructed by performing each S-box computation through this algorithm. Let us now assume that we have

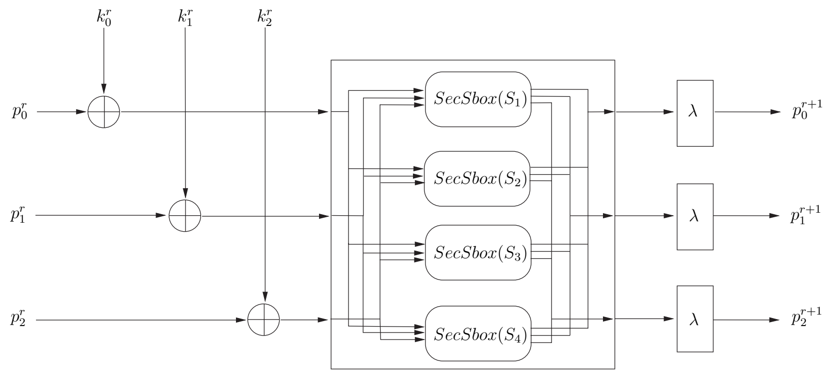

Fig. 2. A 2O-SCA resistant round transformation.

such a secure S-box implementation. Then, the scheme described in Fig. 2 can be viewed as the protected version of the round transformation described in Fig. 1. Finally, the whole block cipher algorithm protected against 2O-SCA can be implemented as depicted in Algorithm 1.

Remark 3. We have described above a way to secure a round transformation ρ. The secure implementation α_{sec} of the key scheduling function α – necessary to thwart Profiling 2O-SCA – can be straightforwardly deduced from this description since it is also composed of linear and non-linear layers.

Algorithm 1 Block Cipher secure against 2O-SCA

INPUT: a plaintext P, a masked key $k_0 = K \oplus k_1 \oplus k_2$ and the masks (k_1, k_2)
OUTPUT: the ciphertext C

1. $(p_1, p_2) \leftarrow rand()$
2. $p_0 \leftarrow P \oplus p_1 \oplus p_2$
3. **for** $r = 0$ **to** $R - 1$ **do**
4. $\quad (k_0, k_1, k_2) \leftarrow \alpha_{sec}\left((k_0, k_1, k_2), r\right)$
5. $\quad (p_0, p_1, p_2) \leftarrow (p_0 \oplus k_0, p_1 \oplus k_1, p_2 \oplus k_2)$
6. $\quad (p'_1, p'_2) \leftarrow rand()$
7. $\quad$ **for** $j = 1$ **to** N **do** $(p'_0)_j \leftarrow SecSbox\left(S_j, (p_0)_j, (p_1)_j, (p_2)_j, (p'_1)_j, (p'_2)_j\right)$
8. $\quad (p_0, p_1, p_2) \leftarrow \left(\lambda\left(p'_0\right), \lambda\left(p'_1\right), \lambda\left(p'_2\right)\right)$
9. **return** $p_0 \oplus p_1 \oplus p_2$

This paper aims to design implementations that are provably secure against any kind of 2O-SCA. We will show how it can be achieved by using masking only. However, as stated in [5,26], masking must be combined with hiding-like countermeasures (such as noise addition, pipelining, operations order randomization *etc.*) to provide a satisfying resistance[4] against SCA of any order. Otherwise a higher order SCA may be easy to carry out (see for instance [17, 18]). As a consequence, to offer a good level of resistance against SCA of order greater than 2, the implementations we propose hereafter should be combined with classical hiding techniques (*e.g.* the operations order randomization described in [10] for the AES).

2.3 Security Model

In order to prove the security of our implementations, we need to introduce a few definitions. We shall say that a variable is *sensitive* if it is a function of the plaintext and the secret key (that is not constant with respect to the secret key). Additionally, we shall call *primitive random values* the intermediate variables that are generated by a random number generator (RNG) executed during the algorithm processing. In the rest of the paper, the primitive random values are assumed to be uniformly distributed and mutually independent.

In the security analysis of our proposal, we will make the distinction between a statistical dependency and what we shall call a functional dependency. Every intermediate variable of a cryptographic algorithm can be expressed as a discrete function of some sensitive variables and some primitive random values (generated by a RNG). When this function involves (*resp.* does not involve) a primitive or sensitive variable X, the intermediate variable is said to be *functionally dependent* (*resp. independent*) of X. If the distribution of an intermediate variable I varies (*resp.* does not vary) according to the value of a variable X then I is said to be *statistically dependent* (*resp. independent*) of X. It can be checked that the two notions are not equivalent since the functional independency implies the statistical independency but the converse is false. We give hereafter an example that illustrates the difference between these notions.

Example 1. Let X be a sensitive variable and let R be a primitive random value. The variable $I = X \oplus R$ is functionally dependent on X and on R. On the other hand, it is statistically independent of X since the probability $P[X = x | I = i]$ is constant for every pair (x, i).

In the rest of the paper, the term (in)dependent will be used alone to refer to the statistical notion.

[4] By resistance, we mean the computational difficulty of the attack.

Definition 1 (2O-SCA). *A* second order SCA *is an SCA that simultaneously exploits the leakages of at most 2 intermediate variables.*

From Definition 1 and according to [3, 7], we formally define hereafter the security against 2O-SCA.

Definition 2. *A cryptographic algorithm is said to be* secure against 2O-SCA *if every pair of its intermediate variables is independent of any sensitive variable.*

Conversely, an algorithm is said to admit a *second order leakage* if two of its intermediate variables jointly depend on a sensitive variable.

Remark 4. Usually a second order SCA refers to an SCA that simultaneously targets two different leakage points in the time-indexed leakage vector corresponding to the algorithm execution. Thus Definitions 1 and 2 implicitly assume that an instantaneous leakage gives information on at most one intermediate variable. However, a non-careful implementation may imply that an instantaneous leakage jointly depends on two intermediate variables. This may result from physical transitions occurring at the hardware level (*e.g.* in a register or on a bus [4, 20]). The different algorithms proposed in this paper fulfill security according to Definition 2 and assume a careful implementation to preclude this kind of phenomena.

Due to Definition 2, proving that an algorithm is secure against 2O-SCA can be done by listing all pairs of its intermediate variables and by showing that they are all independent of any sensitive variable. In order to simplify our security proofs, we introduce the notion of independency for a set.

Definition 3. *A set E is said to be* independent *of a variable X if every element of E is independent of X.*

By extension, Definition 3 implies that the cartesian product of two sets E_1 and E_2 is independent of a variable X if any pair in $E_1 \times E_2$ is independent[5] of X. Thus, according to Definition 2, an algorithm processing a set $\mathcal{I}$ of intermediate variables is secure against 2O-SCA if and only if $\mathcal{I} \times \mathcal{I}$ is independent of any sensitive variable.

Based on the definitions above, our security proofs make use of the two following lemmas.

[5] Unlike for a set, a pair is independent of a variable X if its two elements are jointly independent of X.

Lemma 1. *Let X and Z be two independent random variables. Then, for every family $(f_i)_i$ of measurable functions the set $E = \{f_i(Z); i\}$ is independent of X.*

Remark 5. In the sequel we will say that an intermediate variable I is a function of a variable Z (namely $I = f(Z)$), if its expression involves Z and (possibly) other primitive random values of which Z is functionally independent.

Lemma 2. *Let X be a random variable defined over $\mathbb{F}_2^n$ and let R be a random variable uniformly distributed over $\mathbb{F}_2^n$ and independent of X. Let Z be a variable independent of R and functionally independent of X. Then the pair $(Z, X \oplus R)$ is independent of X.*

When a sensitive variable is masked with two primitive random values, then Lemmas 1 and 2 imply that no second order leakage occurs if the three shares are always processed separately.

According to the definitions and lemmas we have introduced, we get the following proposition.

Proposition 1. *Algorithm 1 is secure against 2O-SCA if and only if SecSbox is secure against 2O-SCA.*

Sketch of Proof. Let us denote by $\mathcal{I}$ the set of intermediate variables processed during one execution of Algorithm 1. Moreover, let us denote by $\mathcal{S}$ the set of intermediate variables processed in the different executions of *SecSbox*, and by $\mathcal{O}$ the set of the other intermediate variables of Algorithm 1 (namely $\mathcal{I} = \mathcal{O} \cup \mathcal{S}$). We will argue that $\mathcal{I} \times \mathcal{I}$ admits a leakage (namely a pair of $\mathcal{I} \times \mathcal{I}$ depends on a sensitive variable) if and only if $\mathcal{S} \times \mathcal{S}$ admits a leakage.

If $\mathcal{S} \times \mathcal{S}$ admits a leakage then it is straightforward that so does $\mathcal{I} \times \mathcal{I}$. Let us now show that the converse is also true. In Algorithm 1, all the operations except the S-box computations are performed independently on the three shares of every sensitive variable. This implies that $\mathcal{O} \times \mathcal{O}$ is independent of any sensitive variable *i.e.* that it admits no leakage. Since one execution of *SecSbox* takes as parameter a tuple $\big((p_0)_j, (p_1)_j, (p_2)_j, (p'_1)_j, (p'_2)_j\big)$, every intermediate variable in $\mathcal{S}$ can be expressed as a function of such a tuple. Hence, if $\mathcal{O} \times \mathcal{S}$ depends on a sensitive variable then this one is either $(p)_j$ or $(p')_j = S\big((p)_j\big)$. We deduce that the intermediate variable in $\mathcal{O}$ that jointly leaks with the one in $\mathcal{S}$ is either a share $(p_i)_j$ or a share $(p'_i)_j$. Since we have

$\{(p_0)_j, (p_1)_j, (p_2)_j, (p'_0)_j, (p'_1)_j, (p'_2)_j\} \subset \mathcal{S}$ we deduce that if a leakage occurs in $\mathcal{O} \times \mathcal{S}$ then it also occurs in $\mathcal{S} \times \mathcal{S}$.

Finally, we can conclude that if a leakage occurs in $\mathcal{I} \times \mathcal{I}$ then it occurs in $\mathcal{S} \times \mathcal{S}$. $\diamond$

In the next section, we propose two new methods to implement any S-box in a way which is provably secure against 2O-SCA. Using one of these methods as *SecSbox* in Algorithm 1 guarantees a global 2O-SCA security.

3 Generic S-box Implementations Secure Against 2O-SCA

In this section, we first describe two methods (Sect. 3.1 and Sect. 3.2) to implement any (n, m)-function S and we prove their security against 2O-SCA. Then we propose an improvement (Sect. 3.3) that allows us to substantially reduce the complexity of both methods.

3.1 A First Proposal

The following algorithm describes a method to securely process a second order masked S-box output from a second order masked input.

Algorithm 2 Computation of a 2O-masked S-box output from a 2O-masked input

INPUT: a pair of dimensions (n, m), a masked value $\tilde{x} = x \oplus r_1 \oplus r_2 \in \mathbb{F}_2^n$, the pair of input masks $(r_1, r_2) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$, a pair of output masks $(s_1, s_2) \in \mathbb{F}_2^m \times \mathbb{F}_2^m$, a LUT for the (n, m)-function S

OUTPUT: the masked S-box output $S(x) \oplus s_1 \oplus s_2 \in \mathbb{F}_2^m$

1. $r_3 \leftarrow rand(n)$
2. $r' \leftarrow (r_1 \oplus r_3) \oplus r_2$
3. **for** $a = 0$ **to** $2^n - 1$ **do**
4. $\quad a' \leftarrow a \oplus r'$
5. $\quad T[a'] \leftarrow (S(\tilde{x} \oplus a) \oplus s_1) \oplus s_2$
6. **return** $T[r_3]$

Remark 6. In the description of Step 5, we used brackets to point out that the introduction of the two output masks s_1 and s_2 is done in this very order (otherwise a second order leakage would occur).

The random value r_3 is used to mask the sum $r_1 \oplus r_2$ and to avoid any second order leakage. The value returned at the end of the algorithm satisfies: $T[r_3] = S(\tilde{x} \oplus r_3 \oplus r') \oplus s_1 \oplus s_2 = S(x) \oplus s_1 \oplus s_2$, which proves the correctness of Algorithm 2.

Complexity. Algorithm 2 requires the allocation of a table of 2^n m-bit words in RAM. It involves 4×2^n (+2) `XOR` operations, 2×2^n (+1) memory transfers and the generation of n random bits.

Security Analysis. We prove hereafter that Algorithm 2 is secure against 2O-SCA.

Security Proof. Algorithm 2 involves four primitive random values r_1, r_2, s_1 and s_2. These variables are assumed to be uniformly distributed and mutually independent together with the sensitive variable x.

The intermediate variables of Algorithm 2 are viewed as functions of the loop index a and are denoted by $I_j(a)$. The set $\{I_j(a); 0 \leq a \leq 2^n - 1\}$ is denoted by I_j. If an intermediate variable $I_j(a)$ does not functionally depend on a, then the set I_j is a singleton. The set $\mathcal{I} = I_1 \cup \cdots \cup I_{15}$ of all the intermediate variables of Algorithm 2 is listed in Table 1.

Remark 7. In Table 1, the step values refer to the lines in the algorithm description (where Step 0 refers to the input parameters manipulation). Note that one step (in the algorithm description) can involve several intermediate variables. However, these ones are separately processed and do not leak information at the same time.

Table 1. Intermediate variables of Algorithm 2.

j	I_j	Steps
1	r_1	0,2
2	r_2	0,2
3	s_1	0,2
4	s_2	0,2
5	r_3	1,6
6	$r_1 \oplus r_3$	2
7	$r_1 \oplus r_2 \oplus r_3$	2,4
8	a	3,4,5
9	$a \oplus r_1 \oplus r_2 \oplus r_3$	4,5
10	$x \oplus r_1 \oplus r_2$	0,5
11	$x \oplus r_1 \oplus r_2 \oplus a$	5
12	$S(x \oplus r_1 \oplus r_2 \oplus a)$	5
13	$S(x \oplus r_1 \oplus r_2 \oplus a) \oplus s_1$	5
14	$S(x \oplus r_1 \oplus r_2 \oplus a) \oplus s_1 \oplus s_2$	5
15	$S(x) \oplus s_1 \oplus s_2$	6

In order to prove that Algorithm 2 is secure against 2O-SCA, we need to show that $\mathcal{I} \times \mathcal{I}$ is independent of x. For this purpose, we split $\mathcal{I}$ into the three subsets $E_1 = I_1 \cup \cdots \cup I_9$, $E_2 = I_{10} \cup \cdots \cup I_{14}$ and I_{15}. First, the sets $E_1 \times E_1$, $E_2 \times E_2$ and $I_{15} \times I_{15}$ are shown to be independent of x. Then, we show that $E_1 \times E_2$, $E_1 \times I_{15}$ and $E_2 \times I_{15}$ are also independent of x, thus proving the independency between $\mathcal{I} \times \mathcal{I}$ and x.

The set $E_1 \times E_1$ is independent of x since E_1 is functionally independent of x. Moreover, since $x \oplus r_1 \oplus r_2$ (*resp.* $S(x) \oplus s_1 \oplus s_2$) is independent of x and since each element in $E_2 \times E_2$ (*resp.* $I_{15} \times I_{15}$) can be expressed as a function of $x \oplus r_1 \oplus r_2$ (*resp.* $S(x) \oplus s_1 \oplus s_2$), then Lemma 1 implies that $E_2 \times E_2$ (*resp.* $I_{15} \times I_{15}$) is independent of x.

One can check that E_1 is independent of $r_1 \oplus r_2$ and is functionally independent of x. Hence, we deduce from Lemma 2 that $E_1 \times \{x \oplus r_1 \oplus r_2\}$ is independent of x, which implies (from Lemma 1) that $E_1 \times E_2$ and x are independent. Similarly, E_1 is independent of $s_1 \oplus s_2$ so that $E_1 \times \{I_{15}\}$ (namely $E_1 \times \{S(x) \oplus s_1 \oplus s_2\}$) is independent of $S(x)$ and hence of x.

To prove the independency between $E_2 \times I_{15}$ and x, we split E_2 into two subsets: $I_{10} \cup \cdots \cup I_{13}$ and I_{14}. One can check that $(x \oplus r_1 \oplus r_2, S(x) \oplus s_2)$ is independent of x and that every element of $(I_{10} \cup \cdots \cup I_{13}) \times I_{15}$ can be expressed as a function of this pair. Hence one deduces from Lemma 1 that $(I_{10} \cup \cdots \cup I_{13}) \times I_{15}$ is independent of x. In order to prove that $I_{14} \times I_{15}$ is also independent of x, let us denote $u_1 = S(x) \oplus s_1 \oplus s_2$ and $u_2 = S(x \oplus a \oplus r_1 \oplus r_2)$. The variables u_1 and u_2 are uniformly distributed[6], independent and mutually independent of x. Since $I_{14} \times I_{15}$ equals $\{S(x) \oplus u_2 \oplus u_1\} \times \{u_1\}$, we deduce that it is independent of x. $\diamond$

3.2 A Second Proposal

In this section, we propose an alternative to Algorithm 2 for implementing an S-box securely against 2O-SCA. This second solution requires more logical operations but less RAM allocation, which can be of interest for low cost devices.

The algorithm introduced hereafter assumes the existence of a masked function $compare_b$ that extends the classical Boolean function (defined by $compare(x, y) = 0$ iff $x = y$) in the following way:

$$compare_b(x, y) = \begin{cases} b & \text{if } x = y \\ \bar{b} & \text{if } x \neq y \end{cases} . \quad (3)$$

Based on the function above, the second method is an adaptation of the first order secure S-box implementation which has been published in [22].

[6] This holds for u_2 if and only if the S-box S is balanced (namely every element in $\mathbb{F}_2^m$ is the image under S of 2^{n-m} elements in $\mathbb{F}_2^n$). As it is always true for cryptographic S-boxes we implicitly make this assumption.

Algorithm 3 Computation of a 2O-masked S-box output from a 2O-masked input

INPUT: a pair of dimensions (n, m), a masked value $\tilde{x} = x \oplus r_1 \oplus r_2 \in \mathbb{F}_2^n$, the pair of input masks $(r_1, r_2) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$, a pair of output masks $(s_1, s_2) \in \mathbb{F}_2^m \times \mathbb{F}_2^m$, a LUT for the (n, m)-function S

OUTPUT: the masked S-box output $S(x) \oplus s_1 \oplus s_2 \in \mathbb{F}_2^m$

1. $b \leftarrow rand(1)$
2. **for** $a = 0$ **to** $2^n - 1$ **do**
3. $\quad cmp \leftarrow compare_b(r_1 \oplus a, r_2)$
4. $\quad R_{cmp} \leftarrow (S(\tilde{x} \oplus a) \oplus s_1) \oplus s_2$
5. **return** R_b

Let $indif$ denote any element in $\mathbb{F}_2^m$. Steps 3 and 4 of Algorithm 3 perform the following operations:

$$\begin{cases} cmp \leftarrow b\ ; \quad R_b \leftarrow S(x) \oplus s_1 \oplus s_2 & \text{if } a = r_1 \oplus r_2\ , \\ cmp \leftarrow \bar{b}\ ; \quad R_{\bar{b}} \leftarrow indif & \text{otherwise.} \end{cases}$$

We thus deduce that the value returned by Algorithm 3 is $S(x) \oplus s_1 \oplus s_2$.

Complexity. The method involves 4×2^n XOR operations, 2^n memory transfers and the generation of 1 random bit. Since it also involves 2^n $compare_b$ operations, the overall complexity relies on the $compare_b$ implementation. As explained in the next paragraph, the implementation of this function must satisfy certain security properties. We propose such a secure implementation in [23, Appendix A] which – when applied to Algorithm 3 – implies a significant timing overhead compared to Algorithm 2 but requires less RAM allocation.

Security Analysis. Let δ_0 denote the Boolean function defined by $\delta_0(z) = 0$ if and only if $z = 0$. For security reasons, $compare_b(x, y)$ must be implemented in a way that prevents any first order leakage on $\delta_0(x \oplus y)$ that is, on the result of the unmasked function $compare(x, y)$ (and more generally on $x \oplus y$). Otherwise, Step 3 would provide a first order leakage on $\delta_0(r_1 \oplus r_2 \oplus a)$ and an attacker could target this leakage together with $\tilde{x} \oplus a$ (Step 4) to recover information about x. Indeed, the joint distribution of $\delta_0(r_1 \oplus r_2 \oplus a)$ and $\tilde{x} \oplus a$ depends on x which can be illustrated by the following observation: $\tilde{x} \oplus a = x$ if and only if $\delta_0(r_1 \oplus r_2 \oplus a) = 0$. In particular, the straightforward implementation $compare_b(x, y) = compare(x, y) \oplus b$ is not valid since it processes $compare(x, y)$ directly. A possible implementation of a secure function $compare_b$ is given in [23, Appendix A]. With such a function, Algorithm 3 is secure against 2O-SCA as we prove hereafter.

Security Proof. As done in Sect. 3.1, we denote by $\mathcal{I}$ the set of intermediate variables that are processed during an execution of Algorithm 3. Table 2 lists these variables. The primitive random values r_1, r_2, s_1, s_2 and b are assumed to be uniformly distributed and mutually independent together with the sensitive variable x. The following security proof is quite similar to the one done in Sect. 3.1.

Table 2. Intermediate variables of Algorithm 3.

j	I_j	Steps
1	r_1	0,3
2	r_2	0,3
3	s_1	0,4
4	s_2	0,4
6	b	1,3
7	a	2-4
8	$r_1 \oplus a$	3
10	$\delta_0(a \oplus r_1 \oplus r_2) \oplus b$	3
11	$x \oplus r_1 \oplus r_2$	0,4
12	$x \oplus r_1 \oplus r_2 \oplus a$	4
13	$S(x \oplus r_1 \oplus r_2 \oplus a)$	4
14	$S(x \oplus r_1 \oplus r_2 \oplus a) \oplus s_1$	4
15	$S(x \oplus r_1 \oplus r_2 \oplus a) \oplus s_1 \oplus s_2$	4
16	$S(x) \oplus s_1 \oplus s_2$	5

In order to prove that Algorithm 3 is secure against 2O-SCA, we need to show that $\mathcal{I} \times \mathcal{I}$ is independent of x. As in Sect. 3.1 we split $\mathcal{I}$ into three subsets $E_1 = I_1 \cup \cdots \cup I_{10}$, $E_2 = I_{11} \cup \cdots \cup I_{15}$ and I_{16}. First, we show that $E_1 \times E_1$, $E_2 \times E_2$ and $I_{16} \times I_{16}$ are independent of x and then, we show that $E_1 \times E_2$, $E_1 \times I_{16}$ and $E_2 \times I_{16}$ are independent of x (thus proving that $\mathcal{I} \times \mathcal{I}$ is independent of x).

As in Sect. 3.1, $E_1 \times E_1$ is straightforwardly independent of x and the independency between $x \oplus r_1 \oplus r_2$ (*resp.* $S(x) \oplus s_1 \oplus s_2$) and x implies, by Lemma 1, that $E_2 \times E_2$ (*resp.* $I_{16} \times I_{16}$) is independent of x.

Since E_1 is independent of $r_1 \oplus r_2$ (*resp.* $s_1 \oplus s_2$) and functionally independent of x, Lemma 2 implies that $E_1 \times \{x \oplus r_1 \oplus r_2\}$ (*resp.* $E_1 \times \{S(x) \oplus s_1 \oplus s_2\}$) is independent of x. Hence, since every element of E_2 (*resp.* I_{16}) can be written as a function of $x \oplus r_1 \oplus r_2$ (*resp.* $S(x) \oplus s_1 \oplus s_2$), Lemma 1 implies that $E_1 \times E_2$ (*resp.* $E_1 \times I_{16}$) is independent of x.

Every pair in $(E_2 \backslash I_{15}) \times I_{16}$ can be expressed as a function of $(x \oplus r_1 \oplus r_2, S(x) \oplus s_2)$ which is independent of x. Hence, by Lemma 1, $(E_2 \backslash I_{15}) \times I_{16}$ is independent of x. Finally, $I_{15} \times I_{16}$ can be rewritten as $\{S(x) \oplus u_2 \oplus u_1\} \times \{u_1\}$, where u_1 $(= S(x) \oplus s_1 \oplus s_2)$ and u_2 $(= S(x \oplus r_1 \oplus r_2 \oplus a))$ are uniformly distributed, mutually independent and mutually independent of x. This implies that $I_{15} \times I_{16}$ is independent of x. $\diamond$

3.3 Improvement

This section aims at describing an improvement of the two previous methods which can be used when the device architecture allows the storage of 2^w S-box outputs on one q-bit word (namely m, w and q satisfy $2^w m \leq q$). This situation may happen for 8-bit architectures when the S-boxes to implement have small output dimensions (*e.g.* $m = 4$ and $w = 1$) or for q-bit architectures when $q \geq 16$ (and $m \leq 8$).

In the following, we assume that the S-box is represented by a LUT having 2^{n-w} elements of bit-length $2^w m$ (instead of 2^n elements of bit-length m). This LUT, denoted by $LUT(S')$, can then be seen as the table representation of the $(n-w, 2^w m)$-function S' defined for every $y \in \mathbb{F}_2^{n-w}$ by: $S'(y) = (S(y,0), S(y,1), \cdots, S(y, 2^w - 1))$, where each $i = 0, \cdots, 2^w - 1$ must be taken as the integer representation of a w-bit value.

For every $x \in \mathbb{F}_2^n$, let us denote by $x[i]$ the i-th most significant bit of x and by x_H (*resp.* x_L) the vector $(x[1], \cdots, x[n-w])$ (*resp.* the vector $(x[n-w+1], \cdots, x[n])$). According to these notations, the S-box output $S(x)$ is the m-bit coordinate of $S'(x_H)$ whose index is the integer representation of x_L.

In order to securely compute the masked output $S(x) \oplus s_1 \oplus s_2$ from the 3-tuple $(\tilde{x}, r_1, r_2)$, our improvement consists in the two following steps. In the first step we securely compute the masked vector $S'(x_H) \oplus z_1 \oplus z_2$ (where z_1 and z_2 are $(2^w m)$-bit random masks). Then, the second step consists in securely extracting $S(x) \oplus s_1 \oplus s_2$ from $S'(x_H) \oplus z_1 \oplus z_2$.

To securely compute the masked vector $S'(x_H) \oplus z_1 \oplus z_2$, we perform Algorithm 2 (or 3) with as inputs the pair of dimensions $(n-w, 2^w m)$, the 3-tuple $(\tilde{x}_H, r_{1,H}, r_{2,H})$, the pair of output masks (z_1, z_2) and the table $LUT(S')$. This execution returns the value $S'(x_H) \oplus z_1 \oplus z_2$. Moreover, as proved in Sect. 3.1 (or Sect. 3.2), it is secure against 2O-SCA.

At this point, we need to securely extract $S(x) \oplus s_1 \oplus s_2$ from $S'(x_H) \oplus z_1 \oplus z_2$ as well as s_1 and s_2 from z_1 and z_2. Namely, we need to extract the m-bit coordinate of $S'(x_H) \oplus z_1 \oplus z_2$, and of z_1 and z_2 whose index corresponds to the integer representation of x_L. For such a purpose, we propose a process that selects the desired coordinate by dichotomy.

For every word y of even bit-length, let $H_0(y)$ and $H_1(y)$ denote the most and the least significant half part of y. At each iteration our process calls an algorithm *Select* that takes as inputs a dimension l, a 2O-masked $(2l)$-bit word $z_0 = z \oplus z_1 \oplus z_2$ (and the corresponding masking words z_1 and z_2) and a 2O-masked bit $c_0 = c \oplus c_1 \oplus c_2$ (and the corresponding masking bits c_1 and c_2). This algorithm returns a 3-tuple of l-bit words (z'_0, z'_1, z'_2) that satisfies $z'_0 \oplus z'_1 \oplus z'_2 = H_c(z)$. We detail hereafter the global process that enables to extract the 3-tuple $(S(x) \oplus s_1 \oplus s_2, s_1, s_2)$ from $(S'(x_H) \oplus z_1 \oplus z_2, z_1, z_2)$.

1. $z_0 \leftarrow S'(x_H) \oplus z_1 \oplus z_2$
2. **for** $i = 0$ **to** $w - 1$
3. $\quad (c_0, c_1, c_2) \leftarrow (\tilde{x}_L[w-i], r_{1,L}[w-i], r_{2,L}[w-i])$
4. $\quad (z'_0, z'_1, z'_2) \leftarrow \mathit{Select}\ 2^w m/2^{i+1}, (z_0, z_1, z_2), (c_0, c_1, c_2)$
4. $\quad (z_0, z_1, z_2) \leftarrow (z'_0, z'_1, z'_2)$
6. **return** (z_0, z_1, z_2)

To be secure against 2O-SCA, this process requires that *Select* admits no second order leakage on z nor on c. A solution for such a secure algorithm is given hereafter (Algorithm 4). It requires three l-bit addressing registers (A_0, A_1), (B_0, B_1) and (C_0, C_1).

Algorithm 4

INPUT: a dimension l, a masked word $z_0 = z \oplus z_1 \oplus z_2 \in \mathbb{F}_2^{2l}$, the pair of masks $(z_1, z_2) \in \mathbb{F}_2^{2l} \times \mathbb{F}_2^{2l}$, a masked bit $c_0 = c \oplus c_1 \oplus c_2 \in \mathbb{F}_2$ and the pair of masking bits $(c_1, c_2) \in \mathbb{F}_2 \times \mathbb{F}_2$

OUTPUT: a 3-tuple $(z'_0, z'_1, z'_2) \in (\mathbb{F}_2^l)^3$ that satisfies $z'_0 \oplus z'_1 \oplus z'_2 = z[c]$

1. $t_1, t_2 \leftarrow rand(l)$
2. $b \leftarrow rand(1)$
3. $c_3 \leftarrow (c_1 \oplus b) \oplus c_2$
4. $A_{c_3} \leftarrow H_{c_0}(z_0) \oplus t_1$
5. $B_{c_3} \leftarrow H_{c_0}(z_1) \oplus t_2$
6. $C_{c_3} \leftarrow H_{c_0}(z_2) \oplus t_1 \oplus t_2$
7. $A_{\overline{c_3}} \leftarrow H_{\overline{c_0}}(z_0) \oplus t_1$
8. $B_{\overline{c_3}} \leftarrow H_{\overline{c_0}}(z_1) \oplus t_2$
9. $C_{\overline{c_3}} \leftarrow H_{\overline{c_0}}(z_2) \oplus t_1 \oplus t_2$
10. **return** (A_b, B_b, C_b)

One can verify that Algorithm 4 performs the following operations for every value of (c_1, c_2):

$$\begin{cases} (A_b, B_b, C_b) \leftarrow (H_c(z_0) \oplus t_1, H_c(z_1) \oplus t_2, H_c(z_2) \oplus t_1 \oplus t_2) \\ (A_{\overline{b}}, B_{\overline{b}}, C_{\overline{b}}) \leftarrow (H_{\overline{c}}(z_0) \oplus t_1, H_{\overline{c}}(z_1) \oplus t_2, H_{\overline{c}}(z_2) \oplus t_1 \oplus t_2) \end{cases} .$$

Thus the three returned variables satisfy $A_b \oplus B_b \oplus C_b = z[c]$.

Complexity. Algorithm 4 involves 10 XOR operations and the generation of $2l + 1$ random bits.

The improvement allows to divide the execution time of Algorithm 2 (or 3) by approximately 2^w since it performs a loop of 2^{n-w} iterations instead of 2^n. Additionally, the improvement involves w calls to Algorithm 4 which implies an overhead of approximately $10 \times w$ XOR operations and the generation of $2m \times (2^w - 1) + w$ random bits. For instance, for an 8×8 S-box on a 16-bit architecture, the improvement applied to Algorithm 2 allows to save 512 XOR operations and 128 memory transfers for an overhead of 10 XOR operations and the generation of 33 random bits (16 more for (z_1, z_2) than for (s_1, s_2) and $16 + 1$ for Algorithm 4).

Security Analysis. The random values t_1 and t_2 are introduced to avoid any second order leakage on c. Otherwise, if the algorithm simply returns $(H_c(z_0), H_c(z_1), H_c(z_2))$, an inherent second order leakage (*i.e.* independent of the algorithm operations) occurs. Indeed, by targeting one of the inputs z_i and one of the outputs $H_c(z_i)$, an attacker may recover information on c since $\big(z_i, H_c(z_i)\big)$ depends on c (even if z_i is random).

The security proof of Algorithm 4 is given in the extended version of this paper [23].

4 Conclusion

In this paper, we have detailed how to implement block ciphers in a way that is provably protect against second order side channel analysis. We have introduced two new methods to protect an S-box implementation and we have proved their security in a strong and realistic security model. Furthermore, we have introduced an improvement of our methods, that can be used when several S-box outputs can be stored on one processor word. Implementation results for an 8×8 S-box on 16-bit and 32-bit architectures have demonstrated its practical interest [23].

Considering the today feasibility of second order attacks, our proposals constitute an interesting contribution in the field of provably secure countermeasures, as being the sole alternative to Schramm and Paar's method [25] and achieving lower memory requirements and possibly better efficiency [23].

Acknowledgements

The authors would like to thank Christophe Giraud for his valuable contribution to this work.

References

1. M.-L. Akkar, R. Bévan, and L. Goubin. Two Power Analysis Attacks against One-Mask Method. In *FSE 2004*, vol. 3017 of *LNCS*, pp. 332–347.
2. M.-L. Akkar and L. Goubin. A Generic Protection against High-Order Differential Power Analysis. In *FSE 2003*, vol. 2887 of *LNCS*, pp. 192–205.
3. J. Blömer, J. Guajardo, and V. Krummel. Provably Secure Masking of AES. In *SAC 2004*, vol. 3357 of *LNCS*, pp. 69–83.
4. E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In *CHES 2004*, vol. 3156 of *LNCS*, pp. 16–29.
5. S. Chari, C. Jutla, J. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *CRYPTO '99*, vol. 1666 of *LNCS*, pp. 398–412.
6. S. Chari, J. Rao, and P. Rohatgi. Template Attacks. In *CHES 2002*, vol. 2523 of *LNCS*, pp. 13–29.
7. J.-S. Coron, E. Prouff, and M. Rivain. Side Channel Cryptanalysis of a Higher Order Masking Scheme. In *CHES 2007*, vol. 4727 of *LNCS*, pp. 28–44.
8. K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In *CHES 2001*, vol. 2162 of *LNCS*, pp. 251–261.
9. L. Goubin and J. Patarin. DES and Differential Power Analysis – The Duplication Method. In *CHES '99*, vol. 1717 of *LNCS*, pp. 158–172.
10. P. Herbst, E. Oswald, and S. Mangard. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In *ANCS 2006*, vol. 3989 of *LNCS*, pp. 239–252.
11. M. Joye, P. Paillier, and B. Schoenmakers. On Second-Order Differential Power Analysis. In *CHES 2005*, vol. 3659 of *LNCS*, pp. 293–308.
12. P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO '96*, vol. 1109 of *LNCS*, pp. 104–113.
13. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO '99*, vol. 1666 of *LNCS*, pp. 388–397.
14. K. Lemke-Rust and C. Paar. Gaussian Mixture Models for Higher-Order Side Channel Analysis. In *CHES 2007*, vol. 4727 of *LNCS*, pp. 14–27.
15. J. Lv and Y. Han. Enhanced DES Implementation Secure Against High-Order Differential Power Analysis in Smartcards. In *ACISP 2005*, vol. 3574 of *LNCS*, pp. 195–206.
16. S. Mangard, T. Popp, and B. M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In *CT-RSA 2005*, vol. 3376 of *LNCS*, pp. 351–365.
17. E. Oswald and S. Mangard. Template Attacks on Masking–Resistance is Futile. In *CT-RSA 2007*, vol. 4377 of *LNCS*, pp. 562–567.
18. E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In *CT-RSA 2006*, vol. 3860 of *LNCS*.
19. E. Peeters, F.-X. Standaert, N. Donckers, and J.-J. Quisquater. Improving Higher-Order Side-Channel Attacks with FPGA Experiments. In *CHES 2005*, vol. 3659 of *LNCS*, pp. 309–321.

20. E. Peeters, F.-X. Standaert, and J.-J. Quisquater. Power and Electromagnetic Analysis: Improved Model, Consequences and Comparisons. *Integration*, 40(1):52–60, 2007.
21. G. Piret and F.-X. Standaert. Security Analysis of Higher-Order Boolean Masking Schemes for Block Ciphers (with Conditions of Perfect Masking). To Appear in IET Information Security.
22. E. Prouff and M. Rivain. A Generic Method for Secure SBox Implementation. In *WISA 2007*, vol. 4867 of *LNCS*, pp. 227–244.
23. M. Rivain, E. Dottax, and E. Prouff. Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis. Cryptology ePrint Archive, Report 2008/021. `http://eprint.iacr.org/`.
24. W. Schindler, K. Lemke, and C. Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In *CHES 2005*, vol. 3659 of *LNCS*.
25. K. Schramm and C. Paar. Higher Order Masking of the AES. In *CT-RSA 2006*, vol. 3860 of *LNCS*, pp. 208–225.
26. F.-X. Standaert, E. Peeters, C. Archambeau, and J.-J. Quisquater. Towards Security Limits of Side-Channel Attacks. In *CHES 2006*, vol. 4249 of *LNCS*, pp. 30–45.
27. D. Suzuki and M. Saeki. Security Evaluation of DPA Countermeasures Using Dual-Rail Pre-charge Logic Style. In *CHES 2006*, vol. 4249 of *LNCS*, pp. 255–269.
28. J. Waddle and D. Wagner. Toward Efficient Second-order Power Analysis. In *CHES 2004*, vol. 3156 of *LNCS*, pp. 1–15.