Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis

– Extended Version* –

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 proposition. 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 to efficiently protect a wide variety of block ciphers, including AES.

Keywords: Side Channel Analysis, Second Order SCA, Block Ciphers Implementations, Masking Countermeasure.

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].

* The abridged version of this paper will appear at FSE 2008.

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 hypothesis 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 admits some security flaws. In particular, the shares are usually processed at the same time. As a consequence the instantaneous leakage is actually dependent of the sensitive variables, which allows some dedicated attacks [20,28]. 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 can impact 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 allows to bypass a d^{th} order masking [22]. 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 proposition 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, 18–20, 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 to substantially speed up our solutions when several S-box outputs can be stored on one microprocessor word. Finally, we list in Sect. 4 the existing solutions and we compare them with our new proposition. We also 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 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 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 most

[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.

of the time implemented as look-up tables (LUT). We will consider in this paper that the linear 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.

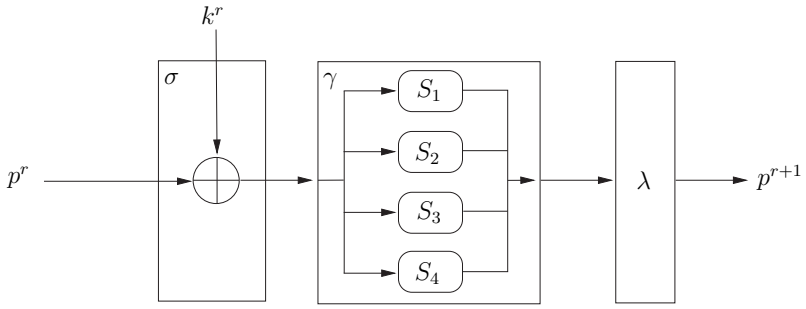

Fig. 1. A typical round transformation with a non-linear layer composed of four S-boxes.

2.2 Securing Block Ciphers Against 2O-SCA

In order to obtain a 2O-SCA resistant implementation of a block cipher, we will 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 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 desired 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 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 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 protecting 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

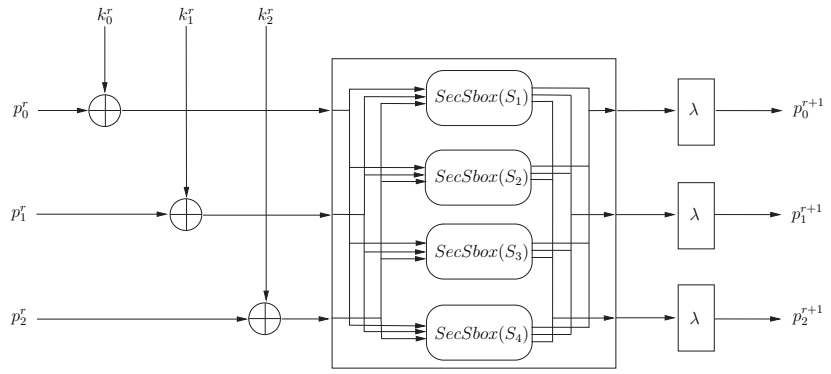

Fig. 2. A 2O-SCA resistant round transformation.

computation through this algorithm. Let us now assume that we have 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. $\quad (k_0, k_1, k_2) \leftarrow \alpha_{sec}((k_0, k_1, k_2), r)$
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** $(p'_0)_j \leftarrow SecSbox(S_j, (p_0)_j, (p_1)_j, (p_2)_j, (p'_1)_j, (p'_2)_j)$
8. $\quad (p_0, p_1, p_2) \leftarrow (\lambda(p'_0), \lambda(p'_1), \lambda(p'_2))$
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 us-

ing 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 [18,19]). Consequently, to offer a good level of resistance against SCA of order greater than 2, our implementations 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 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 one (or several) instantaneous leakage that 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, 21]). The different algorithms proposed in this paper fulfill security according to Definition 2 and assume a careful implementation to provide practical security.

Due to Definition 2, proving that an algorithm is secure against 2O-SCA can be done by listing every pair among 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 introduced, we have 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.

It is straightforward that if $\mathcal{S} \times \mathcal{S}$ admits a leakage then 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.* 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 one of the shares $(p_i)_j$ and $(p'_i)_j$. 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 in the above described block cipher implementation guarantees a global 2O-SCA resistance.

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 to substantially reduce the complexity of both methods.

3.1 A First Proposition

In the following algorithm we describe 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:

$$T[r_3] = S(\tilde{x} \oplus r_3 \oplus r') \oplus s_1 \oplus s_2 = S(x) \oplus s_1 \oplus s_2 \ , \tag{3}$$

which shows 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 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 Proposition

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:

$$compare_b(x, y) = \begin{cases} b & \text{if } x = y \\ \bar{b} & \text{if } x \neq y \end{cases} . \tag{4}$$

[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.

Based on the function above, the second method is an adaptation of the first order secure S-box implementation which has been published in [23].

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 some value in $\mathbb{F}_2^m$. Steps 3 and 4 of Algorithm 3 perform the following operations:

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

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 satisfies certain security properties. We propose such a secure implementation in 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 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. And finally, as in Sect. 3.1, $I_{15} \times I_{16}$ 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:

$$S'(y) = (S(y,0), S(y,1), \cdots, S(y, 2^w - 1)), \tag{5}$$

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. $\quad (z'_0, z'_1, z'_2) \leftarrow Select\big(2^w m/2^{i+1}, (z_0, z_1, z_2), (c_0, c_1, c_2)\big)$
4. $\quad (z_0, z_1, z_2) \leftarrow (z'_0, z'_1, z'_2)$
6. **return** (z_0, z_1, z_2)

In order 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 satisfiy $A_b \oplus B_b \oplus C_b = z[c]$.

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

When it is used, 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).

Security Proof. Table 3 lists all the intermediate variables that are processed by Algorithm 4. To prove that Algorithm 4 is secure against 2O-SCA, we must show that $\mathcal{I} \times \mathcal{I}$ is independent of z and of c.

The independency between $\mathcal{I} \times \mathcal{I}$ and z is straightforward. Indeed, from Table 3, one can verify that $z \oplus z_1 \oplus z_2$, z_1 and z_2 are always processed separately.

Now, let us show that $\mathcal{I} \times \mathcal{I}$ is also independent of c. For such a purpose, we split $\mathcal{I}$ into the three subsets $E_1 = I_1 \cup \cdots \cup I_{10}$, $E_2 =$

Table 3. Intermediate variables of Algorithm 4.

j	I_j	Steps
1	$z \oplus z_1 \oplus z_2$	0-4-7
2	z_1	0,5,8
3	z_2	0,6,9
4	c_1	0,3
5	c_2	0,3
6	t_1	1,4,6,7,9
7	t_2	1,5,6,8,9
8	b	2-3-10
9	$c_1 \oplus b$	3
10	$c_1 \oplus c_2 \oplus b$	3-9
11	$c \oplus c_1 \oplus c_2$	0,4-6
12	$H_{c\oplus c_1\oplus c_2}(z \oplus z_1 \oplus z_2)$	4
13	$H_{c\oplus c_1\oplus c_2}(z \oplus z_1 \oplus z_2) \oplus t_1$	4
14	$H_{c\oplus c_1\oplus c_2}(z_1)$	5
15	$H_{c\oplus c_1\oplus c_2}(z_1) \oplus t_2$	5
16	$H_{c\oplus c_1\oplus c_2}(z_2)$	6
17	$H_{c\oplus c_1\oplus c_2}(z_2) \oplus t_1$	6
18	$H_{c\oplus c_1\oplus c_2}(z_2) \oplus t_1 \oplus t_2$	6
19	$H_{\overline{c\oplus c_1\oplus c_2}}(z \oplus z_1 \oplus z_2)$	7
20	$H_{\overline{c\oplus c_1\oplus c_2}}(z \oplus z_1 \oplus z_2) \oplus t_1$	7
21	$H_{\overline{c\oplus c_1\oplus c_2}}(z_1)$	8
22	$H_{\overline{c\oplus c_1\oplus c_2}}(z_1) \oplus t_2$	8
23	$H_{\overline{c\oplus c_1\oplus c_2}}(z_2)$	9
25	$H_{\overline{c\oplus c_1\oplus c_2}}(z_2) \oplus t_1$	9
26	$H_{\overline{c\oplus c_1\oplus c_2}}(z_2) \oplus t_1 \oplus t_2$	9
27	$H_c(z \oplus z_1 \oplus z_2) \oplus t_1$	10
28	$H_c(z_1) \oplus t_2$	10
29	$H_c(z_2) \oplus t_1 \oplus t_2$	10

$I_{11} \cup \cdots \cup I_{26}$ and $E_3 = I_{27} \cup I_{28} \cup I_{29}$. As done in Sect. 3.1 and Sect. 3.2, we first show that $E_i \times E_j$ is independent of x for $i = j$ and then for $i < j$.

E_1 is mathematically independent of c which implies that $E_1 \times E_1$ is independent of c. Since $c \oplus c_1 \oplus c_2$ is independent of c and since every element of $E_2 \times E_2$ is a function of $c \oplus c_1 \oplus c_2$, Lemma 1 implies that $E_2 \times E_2$ is independent of c. On the other hand, each element of $E_3 \times E_3$ can be rewritten $(H_c[u_1] \oplus v_1, H_c[u_1] \oplus v_1)$ or $(H_c[u_1] \oplus v_1, H_c[u_2] \oplus v_2)$ where u_1, u_2 v_1 and v_2 are uniformly distributed random variables that are mutually independent and mutually independent of c. This implies (by Lemma 2) that $E_3 \times E_3$ is independent of c.

One can verify that E_1 is independent of $c_1 \oplus c_2$. Moreover E_1 is mathematically independent of c. This implies, according to Lemma 2, that $E_1 \times \{c \oplus c_1 \oplus c_2\}$ is independent of c. Moreover, since every element in E_2 is a function of $c \oplus c_1 \oplus c_2$ Lemma 1 implies that $E_1 \times E_2$ is independent of c.

To prove that $E_1 \times E_3$ is independent of x, we split E_1 into two subsets: $E_1' = E_1 \setminus (I_6 \cup I_7)$ and $I_6 \cup I_7$. The set E_1' is mathematically independent of c and is independent of t_1, t_2 and $t_1 \oplus t_2$. Since every element of E_3 is a function of c that masked wether by t_1, t_2 or $t_1 \oplus t_2$, then, Lemma 2 implies that $E_1' \times E_3$ is independent of c. On the other hand, every element of E_3 is a function of $H_c(z_i)$ for $i \in \{0, 1, 2\}$ (recalling $z_0 = z \oplus z_1 \oplus z_2$). Since z_i is uniformly distributed and independent of c, $H_c(z_i)$ is independent of c. And since $I_6 \cup I_7$ is mathematically independent of z, z_1, z_2 and c, we can deduce that $(I_6 \cup I_7) \times E_3$ is independent of c.

To prove that $E_2 \times E_3$ is independent of x, we split E_2 into two subsets: I_{11} and $E_2' = E_2 \setminus I_{11}$. Every element of $\mathcal{I}_{11} \times E_3$ is a pair $(c, H_c(z_i))$ that is masked with an independent and uniformly distributed pair $(c_1 \oplus c_2, t)$ (where t is in $\{t_1, t_2, t_1 \oplus t_2\}$). This implies (by Lemma 2) that $I_{11} \times E_3$ is independent of c. On the other hand, every element of $E_2' \times E_3$ is a function of a pair $(H_{c_0}(u_1), H_c(u_2))$ (or $(H_{\overline{c_0}}(u_1), H_c(u_2))$) where c_0 equals $c \oplus c_1 \oplus c_2$ and where u_1 and u_2 are two random variables (possibly equal) both uniformly distributed and independent of c and c_0. Then, since c and c_0 are independent, one can verify that $(H_{c_0}(u_1), H_c(u_2))$ (and $(H_{\overline{c_0}}(u_1), H_c(u_2))$) is independent of c, thus implying, by Lemma 1, that $E_2' \times E_3$ is independent of c. $\diamond$

4 Comparison and Application

The purpose of this section is twofold. First, we list all the methods that have been proposed in the Literature to secure an S-box implementation against 2O-SCA. We explain that only the proposal of Schramm and Paar in [25] may be considered as secure against 2O-SCA and we compare the complexity of their two methods with the one of Algorithm 2. Secondly, we present several implementations of the AES protected as described in Sect. 2.2 and where the *SecSbox* algorithm is implemented either with one of Schramm and Paar's methods or with our new proposal (Algorithm 2). We also implemented the different *SecSbox* algorithms on 8-bit, 16-bit and 32-bit architectures. To demonstrate the practical interest of the improvement proposed in Sect. 3.3, we implemented the improved version of Algorithm 2 for the 16-bit and 32-bit architectures.

4.1 Comparison

State of The Art. We found four papers in the Literature that focus on secure implementation against higher order SCA. It seems that the first attempt has been proposed by Akkar and Goubin in [2] for securing the DES algorithm. The proposed solution had some flaws that were fixed in two steps, firstly by Akkar *et al.* in [1] and secondly by Lv and Han in [15]. All these works are more or less based on the same principle. At each DES execution, a few 32-bit masks are generated (2 masks in [1,2], and 3 masks in [15]). The masks are used to derive several new masked S-boxes from each of the 8 DES S-boxes and those are combined in different ways by the DES implementation. In the security discussion conducted in [2] and implicitly assumed in [1,15], the masks manipulations and the table re-computations are supposed to leak no information about the masks values. Although the different methods proposed in [2] to process the mask values and the masked S-boxes allow to minimize the instantaneous leakages on the mask values, they do not perfectly prevent it. Therefore, according to Definition 1, the implementations proposed in [1, 2, 15] are not secure against 2O-SCA.

In fact, only the paper by Schramm and Paar presents an implementation of AES that seems to be provably secure against 2O-SCA according to Definition 2.

Remark 8. Initially [25] aimed to present an AES implementation secure against any d^{th} order SCA but the paper by Coron *et al.* [7] showed that a 3^{rd} order SCA is always possible when the method is performed for $d \geq 3$.

Schramm and Paar's S-box Implementations are based on the table re-computation method [17]. Namely (for the 2O-SCA resistance), the principle is to compute from a 4-tuple of masks (r_1, r_2, s_1, s_2), the LUT of a masked S-box S^* that satisfies $S^*(x) = S(x \oplus r_1 \oplus r_2) \oplus s_1 \oplus s_2$ for every $x \in \mathbb{F}_2^n$. Then, the masked output $S(x) \oplus s_1 \oplus s_2$ is obtained by simply accessing the value $S^*(\tilde{x})$ in the re-computed LUT. The tricky part in such a method is to construct the LUT of S^* without never manipulating the sum of masks $r_1 \oplus r_2$ and $s_1 \oplus s_2$ directly (since it would introduce a second order leakage together with $\tilde{x}$ or with $S(x) \oplus s_1 \oplus s_2$).

In their paper, Schramm and Paar present two solutions. The first one (the generic) involves two table re-computations[7].

Algorithm 5 Schramm and Paar's Generic Solution

INPUT: $LUT(S)$, (r_1, r_2, s_1, s_2), the masked input $\tilde{x} = x \oplus r_1 \oplus r_2$
OUTPUT: the masked output $S(x) \oplus s_1 \oplus s_2$

/*** First table re-computation ***/
1. **for** a **from** 0 **to** $2^n - 1$ **do**
2. $\quad S_{temp_1}(a) \leftarrow S(a \oplus r_1) \oplus s_1$
/*** Second table re-computation ***/
3. **for** a **from** 0 **to** $2^n - 1$ **do**
4. $\quad S^*(a) \leftarrow S_{temp_1}(a \oplus r_2) \oplus s_2$
5. **return** $S^*(\tilde{x})$

As noticed by the authors in [25], the algorithm above is quite costly as it involves two table re-computations for each S-box computation for each round of the cipher. To reduce this overhead, Schramm and Paar propose in [25] an improvement. In the new solution, two successive table re-computations are still preformed to process the first masked S-box in the first round of the cipher but all the other S-box computations are protected with a single table re-computation. Before describing the method, let us assume that the previous S-box computation has been protected with the 4-tuple of masks (r'_1, r'_2, s'_1, s'_2) and with a masked S-box S^*_{prev} (satisfying $S^*_{prev}(y) = S(y \oplus r'_1 \oplus r'_2) \oplus s'_1 \oplus s'_2$ for every $y \in \mathbb{F}_2^n$). To securely compute the new masked output $S(x) \oplus s_1 \oplus s_2$ from the masked input $\tilde{x}$, the masked S-box S^* is derived from S^*_{prev}. Then, the value $S^*(\tilde{x})$ is accessed to get $S(x) \oplus s_1 \oplus s_2$.

[7] We consider that the table re-computations are performed using the straightforward algorithm since, as argued in [7], other proposals of [25] include some flaws.

Algorithm 6 Schramm and Paar's Improved Solution

INPUT: $LUT(S^*_{prev})$, the 4-tuples (r_1, r_2, s_1, s_2) and (r'_1, r'_2, s'_1, s'_2), the masked input $\tilde{x} = x \oplus r_1 \oplus r_2$

OUTPUT: the masked output $S(x) \oplus s_1 \oplus s_2$ (and $LUT(S^*)$)

1. $ICM \leftarrow (r_1 \oplus r'_1) \oplus r_2 \oplus r'_2$
2. $OCM \leftarrow (s_1 \oplus s'_1) \oplus s_2 \oplus s'_2$
3. **for** a **from** 0 **to** $2^n - 1$ **do**
4. $\quad S^*(a) \leftarrow S^*_{prev}(a \oplus ICM) \oplus OCM$
5. **return** $S^*(\tilde{x})$

Schramm and Paar's improved solution is clearly much more efficient than the generic solution and it seems to be secure against 2O-SCA. However, its use as algorithm *SecSbox* in a scheme such as described in Sect. 2.2 may potentially introduce a second order leakage in the implementation. In fact, the proof given in Sect. 2.3 does not apply in this case since Algorithm 6 does not fulfill the definition of *SecSbox* (see Problem 1). Indeed, Algorithm 6 takes additional parameters (r'_1, r'_2, s'_1, s'_2) which may induce a second order leakage. For instance, let us assume that the linear layer λ operates on a pair of successive S-box outputs $(S(x'), S(x))$ and that it computes a value taking the form $S(x') \oplus S(x)$ (this is the case in AES). When second order masking is used, this sum of successive S-box outputs takes the form $(S(x') \oplus s'_1 \oplus s'_2) \oplus (S(x) \oplus s_1 \oplus s_2)$ that is $(S(x') \oplus S(x)) \oplus (s'_1 \oplus s'_2 \oplus s_1 \oplus s_2)$. Consequently, if Algorithm 6 is used to implement the S-box computations, then targeting the later sum together with $OCM = s'_1 \oplus s'_2 \oplus s_1 \oplus s_2$ (which is processed during Step 2) reveals some information about the sensitive variable $S(x') \oplus S(x)$. To circumvent such a flaw, additional countermeasures have to be added to the implementation of the linear layer λ. The resulting overhead depends on the definition of λ and, in some cases, can be non-negligible. In the rest of our analysis we do not take this overhead into account for simplicity reasons (namely, the above implementations involving Algorithm 6 may have such a flaw). However we point out that the issue described above must not be neglected when it comes to implement Schramm and Paar's improved solution.

Complexity Comparisons. The Schramm and Paar's improved solution (Algorithm 6) requires $2 \times 2^n(+6)$ `XOR` operations and $2 \times 2^n(+2)$ memory transfers. It is therefore almost 1.5 times faster than our Algorithm 2 (if one considers that the execution timings of a `XOR` and of a memory transfer are equal) and 2.5 times faster than Algorithm 3 (when

this last one implements the $compare_b$ function given in Appendix A). However, Algorithm 6 requires the allocation of at least $2 \times 2^n \times m$ bits of RAM for each $n \times m$ S-box involved in the algorithm, whereas Algorithms 2 and 3 respectively require $2^n m$ and 2^n bits of RAM whatever the number of involved S-box(es). RAM memory being a sensitive resource in low cost devices, the memory gain provided by Algorithms 2 and 3 can often be of great interest even if it is mitigated by a timing overhead. This is especially true when the S-box input dimension is high (namely greater than or equal to 8) and/or when the number of S-box(es) to protect is high (as it is for instance the case for DES).

4.2 Application to AES

We compare hereafter several AES implementations protected against 2O-DSCA (*i.e.* we do not mask the key and we do not protect the key scheduling function). We wrote the codes in assembly language for an 8151-based 8-bit architecture. The implementations only differ in their approaches to protect the S-box computations. The linear steps of the AES have been implemented in the same way, by following the outlines of the method presented in Sect. 2.2. To secure the S-box computation, we implemented the generic and the improved solutions of Schramm and Paar and the solution that is the fastest among our two new ones (namely Algorithm 2). Table 4 lists the timing and memory performances of each implementation.

Table 4. Comparison of AES implementations secure against 2O-DSCA

Method	Reference	cycles	RAM (bytes)	ROM (bytes)
AES with Algorithm 5	SP1	10830×10^3	$512 + 86$	2247
AES with Algorithm 6	SP2	5943×10^3	$512 + 90$	2336
AES with Algorithm 2	RDP	6723×10^3	$256 + 86$	2215

As expected, Implementations SP2 (Schramm and Paar improved solution) and RDP (our solution) are much more efficient than the SP1 (Schramm and Paar generic solution) which performs two table re-computations for each S-box computation. The SP2 Implementation is slightly faster than RDP (around 1.13 times faster), essentially because it involves less logical operations (as shown in the complexity analysis conducted in

Sect. 3.1 and 4.1). However, our solution (RDP) only requires the allocation of 256 bytes of RAM, which is two times smaller than the RAM memory used by SP1 and SP2. Since RAM memory is a sensitive resource in the area of embedded devices and since the timing performances of SP2 and RDP are close, our tests show that our solution represents a good alternative to the proposal of Schramm and Paar in applications where memory constraints are strong.

4.3 Implementation of the Improvement

To demonstrate the practical interest of the improvement proposed in Sect. 3.3, we implemented the algorithms of Schramm and Paar and our new proposal (improved or not) on a 16-bit architecture with a proprietary assembly language and on a 32-bit ARM architecture.

Table 5. Comparison of 8×8 S-box implementations secure against 2O-SCA on 8-bit, 16-bit and 32-bit architectures.

Method	Reference	Cycles	RAM (bytes)	ROM (bytes)
8-bit architecture				
Algorithm 5	SP1-8	6703	$512 + 3$	$119 + 256$
Algorithm 6	SP2-8	3638	$512 + 7$	$89 + 256$
Algorithm 2	RDP-8	4142	$256 + 3$	$88 + 256$
16-bit architecture				
Algorithm 5	SP1-16	6418	512	$96 + 512$
Algorithm 6	SP2-16	3090	512	$56 + 256$
Algorithm 2	RDP-16	4125	256	$98 + 512$
Algorithm 2 + Improvement	RDP*-16	2099	256	$260 + 256$
32-bit architecture				
Algorithm 5	SP2-32	3359	512	na.
Algorithm 6	RDP-32	4143	256	na.
Algorithm 2 + Improvement	RDP*-32	1415	256	na.

Remark 9. In our implementations of SP1-16 and RDP-16, we represented each element of the 8×8 S-box by a 16-bit word whose LSB is the S-box element and whose MSB is the zero element. This representation multiplies by 2 the size of the LUT in ROM, but avoids the conversions of 16-bit words into 8-bit words during the loop execution (thus speeding up the entire S-box calculation by around 1.25).

In all the cases, the implementation of Algorithm 6 is more efficient than the implementation of Algorithm 2. In average, it is 1.20 times faster.

The use of the improvement (Algorithm 4) allows a gain of 50% for the 16-bit architecture and of 65% for the 32-bit architecture. With this improvement our method becomes much faster than SP2. For the implementation on ARM 32-bit architecture, it may be noticed that the gain is smaller than the one resulting from our theoretical complexity analysis (Sect. 3.3). This is merely due to the fact that the assembly implementation of Algorithm 4 involves costly registers and data pointers manipulations.

5 Conclusion

In this paper, we have detailed how to implement block ciphers in a way that is provably secure against second order side channel analysis. We have introduced two new methods to secure an S-box implementation and we have proved their security in a strong and realistic security model. Moreover, we have compared their complexity with the ones of Schramm and Paar's solutions (the generic one and the improved one), which are the only other methods that enable to securely implement S-boxes regarding second order side channel analysis. We have pointed out that Schramm and Paar's improved solution can require some adaptations, depending on the block cipher particularities, whereas our proposal is completely generic. To compare the efficiency of the different methods, we have implemented them to protect AES on an 8-bit architecture. With comparable timings and code sizes, our proposal requires nearly half RAM size. This is of interest since RAM is a scarce resource in embedded devices, which are the privileged targets of side channel attacks. 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. To illustrate its practical interest, we have given implementation results for an 8×8 S-box on 16-bit and 32-bit architectures. The comparison between the different implementations shows that our improved method is always faster than Schramm and Paar's solution.

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 and achieving lower memory requirements and possibly better efficiency.

Acknowledgements

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

References

1. M.-L. Akkar, R. Bévan, and L. Goubin. Two Power Analysis Attacks against One-Mask Method. In B. Roy and W. Meier, editors, *Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 332–347. Springer, 2004.
2. M.-L. Akkar and L. Goubin. A Generic Protection against High-Order Differential Power Analysis. In T. Johansson, editor, *Fast Software Encryption – FSE 2003*, volume 2887 of *Lecture Notes in Computer Science*, pages 192–205. Springer, 2003.
3. J. Blömer, J. Guajardo, and V. Krummel. Provably Secure Masking of AES. In M. Matsui and R. Zuccherato, editors, *Selected Areas in Cryptography – SAC 2004*, volume 3357 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2004.
4. E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.
5. S. Chari, C. Jutla, J. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
6. S. Chari, J. Rao, and P. Rohatgi. Template Attacks. In B. Kaliski Jr., Ç. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2002.
7. J.-S. Coron, E. Prouff, and M. Rivain. Side Channel Cryptanalysis of a Higher Order Masking Scheme. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 28–44. Springer, 2007.
8. K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In Ç. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.
9. L. Goubin and J. Patarin. DES and Differential Power Analysis – The Duplication Method. In Ç. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES '99*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.
10. P. Herbst, E. Oswald, and S. Mangard. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In J. Zhou, M. Yung, and F. Bao, editors, *Applied Cryptography and Network Security – ANCS 2006*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 2006.
11. M. Joye, P. Paillier, and B. Schoenmakers. On Second-Order Differential Power Analysis. In J. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.
12. P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
13. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

14. K. Lemke-Rust and C. Paar. Gaussian Mixture Models for Higher-Order Side Channel Analysis. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 14–27. Springer, 2007.
15. J. Lv and Y. Han. Enhanced DES Implementation Secure Against High-Order Differential Power Analysis in Smartcards. In C. Boyd and J. M. G. Nieto, editors, *Information Security and Privacy, 10th Australasian Conference – ACISP 2005*, volume 3574 of *Lecture Notes in Computer Science*, pages 195–206. Springer, 2005.
16. S. Mangard, T. Popp, and B. M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In A. Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.
17. T. Messerges. Securing the AES Finalists Against Power Analysis Attacks. In B. Schneier, editor, *Fast Software Encryption – FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2000.
18. E. Oswald and S. Mangard. Template Attacks on Masking–Resistance is Futile. In M. Abe, editor, *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2007.
19. E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*. Springer, 2006.
20. E. Peeters, F.-X. Standaert, N. Donckers, and J.-J. Quisquater. Improving Higher-Order Side-Channel Attacks with FPGA Experiments. In J. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 309–321. Springer, 2005.
21. E. Peeters, F.-X. Standaert, and J.-J. Quisquater. Power and Electromagnetic Analysis: Improved Model, Consequences and Comparisons. *Integration*, 40(1):52–60, 2007.
22. 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.
23. E. Prouff and M. Rivain. A Generic Method for Secure SBox Implementation. To Appear in WISA 2007.
24. W. Schindler, K. Lemke, and C. Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In J. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*. Springer, 2005.
25. K. Schramm and C. Paar. Higher Order Masking of the AES. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.
26. F.-X. Standaert, E. Peeters, C. Archambeau, and J.-J. Quisquater. Towards Security Limits of Side-Channel Attacks. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 2006.
27. D. Suzuki and M. Saeki. Security Evaluation of DPA Countermeasures Using Dual-Rail Pre-charge Logic Style. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2006.
28. J. Waddle and D. Wagner. Toward Efficient Second-order Power Analysis. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Sys-*

tems – CHES 2004, volume 3156 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

A A Masked Implementation of the Function *compare*

In this section, we describe a way to implement the function $compare_b$ defined as:

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

that prevent any first order leakage on $compare(x, y)$.

The method requires a table T of 2^n bits in RAM. At the beginning of the algorithm in which the $compare_b$ function is involved (for instance Algorithm 3), the following pre-computation is processed:

1. $r_3 \leftarrow rand(n)$
2. $T \leftarrow \{\bar{b}, \bar{b}, \cdots, \bar{b}\}$
3. $T[r_3] \leftarrow b$

At the end of this pre-computation, the table T satisfies:

$$T[x] = \begin{cases} b & \text{if } x = r_3 \\ \bar{b} & \text{otherwise} \end{cases}.$$

Then, the function $compare_b$ is simply implemented as:

return $T[(x \oplus r_3) \oplus y]$

It is straightforward to observe that all intermediate variables of these computations (namely $\{b, \bar{b}, r_3, x \oplus r_3, x \oplus y \oplus r_3, compare_b(x, y)\}$) are independent of $compare(x, y)$ (as well as on $x \oplus y$).

Complexity. The method requires 2^n bit of RAM. The pre-computation involves the generation of n random bits and the initialization of the (2^n)-bit table T (which is roughly $2^n/q + 1$ memory transferts – q being the bit-size of the microprocessor). Then each call to $compare_b$ involves 2 `XOR` operations and 1 memory transfert.

Application to Algorithm 3. With such a method, the total complexity (in operations) of Algorithm 3 is: 6×2^n `XOR` operations, $(2 \times 2^n + 2^n/q + 1)$ memory transferts and the generation of $n + 1$ random bits. Compared to Algorithm 2, this implies a significant timing overhead but it consumes m times less RAM which is of interest for low cost devices.