# Thwarting Higher-Order Side Channel Analysis with Additive and Multiplicative Maskings

## Extended Version

Laurie Genelle[1], Emmanuel Prouff[1] and Michaël Quisquater[2]

[1] Oberthur Technologies
{l.genelle,e.prouff}@oberthur.com
[2] University of Versailles
michael.quisquater@prism.uvsq.fr

**Abstract.** Higher-order side channel attacks is a class of powerful techniques against cryptographic implementations. Their complexity grows exponentially with the order, but for small orders (*e.g.* 2 and 3) recent studies have demonstrated that they pose a serious threat in practice. In this context, it is today of great importance to design software countermeasures enabling to counteract higher-order side channel attacks for any arbitrary chosen order. At CHES 2010, Rivain and Prouff have introduced such a countermeasure for the AES. It works for any arbitrary chosen order and benefits from a formal resistance proof. Until now, it was the single one with such assets. By generalizing at any order a countermeasure introduced at ACNS 2010 by Genelle *et al.* , we propose in this paper an alternative to Rivain and Prouff's solution. The new scheme can also be proven secure at any order and has the advantage of being at least 2 times more efficient than the existing solutions for orders 2 and 3, while maintaining the RAM consumption lower than 200 bytes.

## 1 Introduction

In the late nineties, attacks called *Side Channel Analysis* (SCA for short) have been exhibited against cryptosystems implemented in embedded devices. Since then, they have been refined and, in particular, their initial principle has been generalized in order to exploit several leakage points simultaneously. This led to the introduction of the *higher-order SCA* concept, which exploit leakage observations resulting from the handling of several intermediate variables during the cryptosystem processing. One way to make them ineffective at *order $d$* is to randomize the algorithm such that the probability distribution of any vector of $d$ observations is independent of the key. To perform this randomization, a standard technique is based on *secret sharing* [21] and is often called *masking* in the context of side channel analysis. We talk about *additive* (resp. *multiplicative*) masking when any value is expressed as a sum (resp. product) of *shares*.

As side channel attacks, masking can be characterized by the number of random shares per variable. This number is called the *masking order*. A $d^{\text{th}}$-order masking can always be theoretically defeated by a $(d+1)^{\text{th}}$-order SCA, but noise effects imply that the difficulty of carrying out such an attack in practice increases exponentially with its order [3,20]. For this reason, the masking order is today a well accepted security criterion and many works have studied how to apply $d^{\text{th}}$-order masking to protect any kind of cryptosystem at any order $d$. In particular, block cipher software implementations have been a privileged target either to demonstrate the efficiency of an attack [14] or to argue on the effectiveness of a countermeasure [4,6,15,18,19]. It is actually a matter of fact that any improvement of an attack against, or a countermeasure for, a standard block cipher such as AES has an important and direct impact on the (public or military) embedded security industry.

### 1.1 Related Works

Protecting a block cipher software implementation by masking at any order $d$ reveals some issues which are very close to those tackled out in the *Multi-Party Computation* or *Private Circuits* area [2,5]. The main difficulty lies in performing all the algorithm steps by manipulating the shares separately, while being able to re-build 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 in the context of embedded security. For $d = 2$, there only exist three methods that perfectly thwart 2O-SCA [18–20]. For $d > 2$, several methods have been proposed [19,20], but except [19] all those attempts have been shown to be flawed, which has raised the need for solutions with formal resistance proof. Solution in [19], which is dedicated to the AES, benefits from such a proof and, when applied for $d = 2$, it is much more efficient than [20] and [18]. However, the time efficiency is still low (around 200 K-cycles in a classical smart card 8-bit architecture) and, even, becomes prohibitive when $d = 3$ (greater than 300 K-cycles). Alternative solutions are therefore missing, which would have equivalent security but would be more efficient. It is all the more important that second and third order SCA have been substantially improved during the last years and have even been successfully put into practice [11,13,14,17,23].

### 1.2 Our Results

In this paper, we are interested in masking to the order $d$, block ciphers whose design involves affine operations and power functions defined on a finite field. The classic strategy is to mask the message additively and to calculate the masks propagation through the various transformations. While calculating the propagation of additive masks through affine operations is easy, this is no longer the case for the power functions. The approach proposed in [19] is to express a power function in terms of squares and multiplications. The computation of the propagation of the additive masks through a multiplication requires little memory and can be managed regardless of the order $d$. However, this step is very time consuming (quadratic in the order $d$). A natural idea to achieve better performance is to mask affine functions additively and power functions multiplicatively. In this case, the calculation of the masks spreading is fast and requires little memory. When applied at order $d$, the only potentially costly part lies in the conversion of additives masks into multiplicative ones (and *vice versa*) since this conversion must be done without manipulating $d$-tuple of shares dependent on sensitive data. This strategy has already been followed to define implementations with assumed security at order $d = 1$ [1,9,22]. Unfortunately, none of them was perfectly thwarting first-order SCA and, even, [1] and [22] were shown to be flawed. Finally, Genelle *et al.* have proposed in [7] a satisfactory solution with formal security proof and good performances. This is an encouraging step but the extension of [7] to any order poses several problems. Firstly, it requires to calculate a Dirac function in a secure manner w.r.t. higher-order SCA. Secondly, it implies to generalize the conversion functions that map additive maskings into multiplicative ones and conversely. In a recent work, the authors of [8] have solved the first issue efficiently. In this paper, we solve the second one and we prove the security of our proposal. Having solved the two issues related to the generalization of Genelle *et al.* 's work at any order, we are now able to design a masking scheme for any block cipher combining affine transformations and power functions. When applied to secure the software implementation of the AES at order $d = 2$ (resp. $d = 3$), we achieve a time efficiency around 70K cycles (resp. 180K cycles) at the cost of a `RAM` memory consumption lower than 200 bytes in both cases. Since this amount of `RAM` is almost always acceptable in the nowadays embedded systems, this secure AES implementation is, to the best of our knowledge, the first one that makes $2^{\text{nd}}$ and $3^{\text{rd}}$ order security achievable, even in very constraint contexts.

### 1.3 Road Map

In Sect. 2, we first introduce a few basics and notations related to the additive and multiplicative maskings in finite fields. Then, in Sect. 3 we present the core principle of our approach, we recall how the computation of a Dirac function can be secured at any order $d$ and we present two new conversion algorithms enabling to securely convert an additive masking into a multiplicative one and conversely. Eventually, in Sect. 4 we apply our masking scheme to the AES and compare its efficiency with that of the state of the art solutions.

## 2 Basics and Notations

### 2.1 Notations

The bit-length of the elements involved in the algorithmic description of the cryptosystem will be denoted by $n$. By default, any variables in this paper are assumed to be in a vector space of some dimension $m$ over $\mathrm{GF}(2^n)$. The field addition in $\mathrm{GF}(2^n)$ is denoted by $\oplus$ and the field multiplication by $\otimes$. To operate on elements of $\mathrm{GF}(2^n)^m$, the two previous laws are extended: the addition continues to be a bitwise addition and the multiplication between two vectors in $\mathrm{GF}(2^n)^m$ corresponds to the *componentwise product*. For two vectors $(x_1, \ldots, x_m)$ and $(y_1, \ldots, y_m)$ in $\mathrm{GF}(2^n)^m$, the result of the latter product is a vector $(z_1, \ldots, z_m)$ in $\mathrm{GF}(2^n)^m$ whose coordinates satisfy $z_i = x_i \otimes y_i$. The inverse of an element $(x_1, \ldots, x_m) \in (\mathrm{GF}(2^n)^\star)^m$ for the componentwise product will be simply defined as the vector $(x_1^{-1}, \ldots, x_m^{-1})$, where for every $i$, $x_i^{-1}$ is the inverse of $x_i$ for the multiplicative law $\otimes$ of $\mathrm{GF}(2^n)^\star$. For convenience, we will keep the notations $\oplus$ and ${}^{-1}$ for the extensions of the field operations $\oplus$ and ${}^{-1}$. On the other hand and to avoid any ambiguity, we will denote by $\dot{\otimes}$ the extension of the field operations $\otimes$ into a componentwise multiplication. To differentiate vectors in $\mathrm{GF}(2^n)^m$ from elements of $\mathrm{GF}(2^n)$, we shall write the vector in bold. Namely, by convention $\mathbf{x}$ shall denote a vector in $\mathrm{GF}(2^n)^m$, whereas $x$ shall denote an element of $\mathrm{GF}(2^n)$.

### 2.2 Basics on Masking

When higher-order masking is involved to secure the physical implementation of a cryptographic algorithm, every sensitive variable $\mathbf{x}$ occurring during the computation is randomly split into $d+1$ shares $\mathbf{x}_0$, $\ldots$, $\mathbf{x}_d$ in such a way that the following relation is satisfied for a group operation $\perp$:

$$\mathbf{x}_0 \perp \mathbf{x}_1^{-1} \perp \ldots \perp \mathbf{x}_d^{-1} = \mathbf{x} \ , \tag{1}$$

where, $\mathbf{x}_i{}^{-1}$ denotes the inverse of $\mathbf{x}_i$ w.r.t. to $\perp$.

Usually, the $d$ shares $\mathbf{x}_1$, $\ldots$, $\mathbf{x}_d$ (called *the masks*) are randomly picked up and the last one $\mathbf{x}_0$ is processed such that (1) is satisfied. When $d$ random masks are involved per sensitive variable, the masking is said to be *of order d*. The so-called *additive masking* assumes that $\perp$ is the addition $\oplus$ in $\mathrm{GF}(2^n)^m$. In this case, we have $\mathbf{x}_i^{-1} = \mathbf{x}_i$ for every $i$. The $(d+1)$-tuple $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ is called a $(d+1)$-*additive sharing* of $\mathbf{x}$ and the transformation $(\mathbf{x}, (\mathbf{x}_i)_{i>1}) \mapsto \mathbf{x}_0 = \mathbf{x} \oplus \mathbf{x}_1 \oplus \ldots \oplus \mathbf{x}_d$ is called $d^{th}$*-order additive masking*. In *multiplicative masking*, the operation $\perp$ is the componentwise product $\dot{\otimes}$ in the group $(\mathrm{GF}(2^n)^\star)^m$. The $(d+1)$-tuple $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ is called $(d+1)$-*multiplicative sharing* of $\mathbf{x}$ and the transformation $(\mathbf{x}, (\mathbf{x}_i)_{i>1}) \mapsto \mathbf{x}_0 = \mathbf{x} \dot{\otimes} \mathbf{x}_1 \dot{\otimes} \ldots \dot{\otimes} \mathbf{x}_d$ is the $d^{th}$*-order multiplicative masking* of $\mathbf{x}$. Note that the multiplicative masking is only defined for vectors $\mathbf{x}$ with only non-zero coordinates. In what follows, we shall simply say masking if there is no ambiguity on the nature of the operation or if the text is applicable for the two kinds of maskings.

When $d^{\mathrm{th}}$-order masking is involved to secure an implementation composed of elementary transformations in the form $\mathbf{y} \leftarrow Op(\mathbf{x})$, a so-called $d^{th}$*-order masking scheme* must be designed to replace them by new transformations taking at input a sharing $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ of $\mathbf{x}$ and returning a sharing $(\mathbf{y}_0, \ldots, \mathbf{y}_d)$ of $\mathbf{y}$. The $d^{\mathrm{th}}$-order security of such a design holds if and only if it can be proved that every $d$-tuple of manipulated intermediate results during the computation is independent of any sensitive variable of the implementation (including $\mathbf{x}$ and $\mathbf{y}$).

## 3 Higher-Order Masking

We formally define a block cipher as a cryptographic algorithm that transforms a plaintext block into a ciphertext block from a secret key. The transformation is done by operating several elementary operations on a so-called *internal state*, viewed as a vector in $\mathrm{GF}(2^n)^m$ and initially filled with the plaintext. In this section, we show how to secure at any order $d$ a block cipher composed of transformations $Op$ that are either affine or are bijective power functions defined w.r.t. to the same field operation laws $\oplus$ and $\otimes$ over $\mathrm{GF}(2^n)$. Affine transformations will be assumed to be defined over the vector space $\mathrm{GF}(2^n)^m$. Power functions will be assumed to operate on a vector in $\mathrm{GF}(2^n)^m$ coordinate by coordinate.

### 3.1 Core Idea.

As usually done when applying masking, each calculation $\mathbf{y} \leftarrow Op(\mathbf{x})$ is replaced by a sequence of elementary calculations that securely construct a $(d+1)$-sharing $(\mathbf{y}_0, \ldots, \mathbf{y}_d)$ of $\mathbf{y}$ from the $(d+1)$-sharing $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ of $\mathbf{x}$. To define those sequences of elementary operations we use the fact that linear transformations are automorphisms of $(\mathrm{GF}(2^n)^m, \oplus)$, while bijective power functions are automorphisms of $(\mathrm{GF}(2^n)^m, \dot{\otimes})$. Hence, depending of its (affine or multiplicative) nature, we involve either an additive or a multiplicative sharing of the internal state to secure the operation $Op$.

*Affine Transformations processing.* If $Op$ is a linear transformation defined over $\mathrm{GF}(2^n)^m$, then the sensitive variable $\mathbf{x}$ is assumed to be represented by a $(d+1)$-additive sharing $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$. In this case, securing the calculation $\mathbf{y} \leftarrow Op(\mathbf{x})$ simply consists in replacing it by $d+1$ applications of $Op$, one for each share $\mathbf{x}_i$. After denoting by $\mathbf{y}_i$ the value $Op(\mathbf{x}_i)$, we have $\bigoplus_{j=0}^{d} \mathbf{y}_j = \mathbf{y}$. We conclude that $(\mathbf{y}_0, \ldots, \mathbf{y}_d)$ is a $(d+1)$-sharing of $\mathbf{y}$. Moreover, it is obvious that no $d$-tuple of intermediate data is sensitive during this processing. For affine transformations the processing is done similarly, except for $d$ even where only the linear part of $Op$ is applied to the last share $\mathbf{x}_d$.

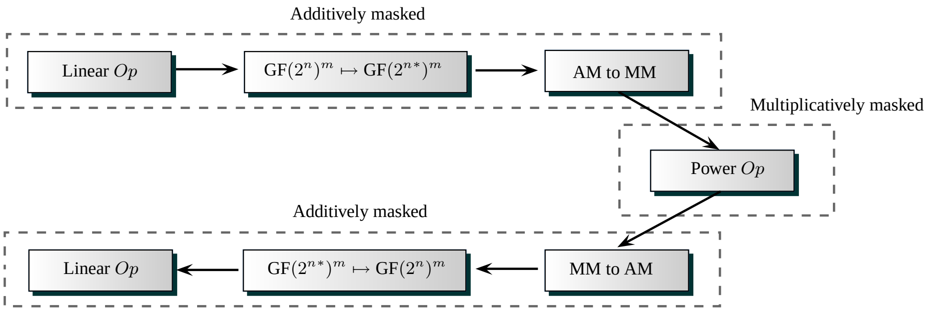
*Power Functions processing.* If $Op$ is a power function over $\mathrm{GF}(2^n)^m$, then the sensitive variable $\mathbf{x}$ is assumed to be non-zero and represented by a $(d+1)$-multiplicative sharing $(\mathbf{z}_0, \ldots, \mathbf{z}_d)$. In this case, $\mathbf{y} \leftarrow Op(\mathbf{x})$ is simply replaced by $d+1$ elementary calculations of $Op$, one on each multiplicative share $\mathbf{z}_i$. This results in $d+1$ shares $\mathbf{y}_i = Op(\mathbf{z}_i)$ that satisfy $\mathbf{y}_0 \dot{\otimes} \dot{\bigotimes}_{j=1}^{d} \mathbf{y}_j^{-1} = \mathbf{y}$ and are thus a $(d+1)$-multiplicative sharing of $\mathbf{y}$. It can be easily checked that every $d$-tuple of intermediate variables involved in the processing is independent of any sensitive variable, since all the $\mathbf{z}_i$ (and $\mathbf{y}_i$) are manipulated independently.

The application of the most appropriate masking for each elementary operation enables to efficiently secure each (affine or non-linear) layer of the block cipher. Nevertheless, the mix of additive and multiplicative maskings arises the two following issues:

Issue 1: the proposed power functions processing involves multiplicative sharings and the latter ones can only be defined for an element $\mathbf{x}$ in $(\mathrm{GF}(2^n)^\star)^m$, whereas the block cipher internal state is defined in $\mathrm{GF}(2^n)^m$. A $d^{\mathrm{th}}$-order secure scheme for the mapping of an element of $\mathrm{GF}(2^n)^m$ into an element of $(\mathrm{GF}(2^n)^\star)^m$ (and *vice versa*) must therefore be defined. Moreover, the mapping must be reversible at any time during the block cipher processing.

Issue 2: since affine functions and power functions are processed alternatively, special transformations must be defined to convert additive sharings into multiplicative ones and conversely. Moreover those transformations must themselves be $d^{\mathrm{th}}$-order secure to not decrease the overall security of the block cipher implementation.

The first issue has been solved in [8]. We give in Sect. 3.2 the outlines of the solution that essentially relies on the secure processing of a *Dirac function*. The second issue is tackled out in Sect. 3.3, where we propose two algorithms that transform an additive masking (AM for short) into a multiplicative masking (MM for short) and conversely. All those transformations are eventually combined in Sect. 3.4 to secure a block cipher round transformation according to the following diagram.


Additively masked
Linear Op
$\mathrm{GF}(2^n)^m \mapsto \mathrm{GF}(2^{n*})^m$
AM to MM
Multiplicatively masked
Power Op
Additively masked
Linear Op
$\mathrm{GF}(2^{n*})^m \mapsto \mathrm{GF}(2^n)^m$
MM to AM

### 3.2 Issue 1: Mapping Elements of $\mathrm{GF}(2^n)^m$ Into $(\mathrm{GF}(2^n)^\star)^m$

The solution of Issue 1 proposed in [8] consists in transforming any zero value into a non-zero one, keeping track of this modification if applied. Let us denote by $\delta_0$ the Dirac function defined in $\mathrm{GF}(2^n)$ by $\delta_0(x) = 1$ if $x = 0$ and $\delta_0(x) = 0$ otherwise. To map any $x \in \mathrm{GF}(2^n)$ into $\mathrm{GF}(2^n)^\star$, the element is simply added with its dirac value $\delta_0(x)$. After extending the Dirac function to $\mathrm{GF}(2^n)^m$ by setting $\delta(\mathbf{x}) = (\delta(x_0), \ldots, \delta(x_{m-1}))$, we get a function $\mathbf{x} \mapsto \mathbf{x} \oplus \delta(\mathbf{x})$ mapping any element of $\mathrm{GF}(2^n)^m$ into an element of $(\mathrm{GF}(2^n)^\star)^m$. To secure the processing of the latter transformation against $d^{\mathrm{th}}$-order SCA, the vector $\mathbf{x}$ is represented by a $(d+1)$-additive sharing $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ and a secure processing is applied to output an additive sharing $(\Delta_0, \ldots, \Delta_d)$ of $\delta(\mathbf{x})$. The details of the processing, as long a proof of its security against $d^{\mathrm{th}}$-order SCA are given in [8]. We call this processing `SecDirac` in the following.

### 3.3 Issue 2: Conversion Functions.

In this section, we show how to build $d^{th}$-order secure transformations passing from the $(d+1)$-additive sharing $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ of $\mathbf{x} \in (\mathrm{GF}(2^n)^\star)^m$ to its $(d+1)$-multiplicative sharing $(\mathbf{z}_0, \ldots, \mathbf{z}_d)$ and conversely. These transformations are respectively called $\mathtt{AMtoMM}(\cdot)$ and $\mathtt{MMtoAM}(\cdot)$ and act as follows:

- $\mathtt{AMtoMM}(\mathbf{x} \oplus \bigoplus_{i=1}^{d} \mathbf{x}_i, \mathbf{x}_1, \ldots, \mathbf{x}_d) \rightarrow (\mathbf{x} \dot{\otimes} \dot{\bigotimes}_{i=1}^{d} \mathbf{z}_i, \mathbf{z}_1, \ldots, \mathbf{z}_d)$,
- $\mathtt{MMtoAM}(\mathbf{x} \dot{\otimes} \dot{\bigotimes}_{i=1}^{d} \mathbf{z}_i, \mathbf{z}_1, \ldots, \mathbf{z}_d) \rightarrow (\mathbf{x} \oplus \bigoplus_{i=1}^{d} \mathbf{x}_i, \mathbf{x}_1, \ldots, \mathbf{x}_d)$.

To process the `AMtoMM` transformation, the general strategy developed hereafter consists in converting sequentially each additive mask of $\mathbf{x}$ into a multiplicative one. To preserve the security order of the scheme at each step, an additive mask is added to the multiplicatively masked representation of $\mathbf{x}$ prior to remove one of the remaining multiplicative masks. The strategy followed for the `MMtoAM` is exactly the same, except that the roles of the additive and multiplicative masks are reversed.

In the hereafter detailed descriptions of the transformations we will use three ordered sets $\mathcal{S}_{MV}$, $\mathcal{S}_{AM}$ and $\mathcal{S}_{MM}$ that will be respectively dedicated to the storage of the masked value, the additive shares and the multiplicatives shares.

At the beginning of the `AMtoMM` processing, let us associate the $(d+1)$-additive sharing $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ of $\mathbf{x}$ with the sets $\mathcal{S}_{MV} = \{\mathbf{x}_0\}$, $\mathcal{S}_{AM} = \{\mathbf{x}_1, \ldots, \mathbf{x}_d\}$ and $\mathcal{S}_{MM} = \emptyset$. The conversion of the $(d+1)$-additive sharing to a multiplicative one $(\mathbf{z}_0, \ldots, \mathbf{z}_d)$ may be viewed as a sequence of updatings of those three sets such that, at the end, $\mathcal{S}_{MV} = \{\mathbf{z}_0\}$, $\mathcal{S}_{AM} = \emptyset$ and $\mathcal{S}_{MM} = \{\mathbf{z}_1, \ldots, \mathbf{z}_d\}$. To perform such a conversion, the following treatment is repeated for every $i \in [1; d]$:

1. Masking multiplicatively the element in $\mathcal{S}_{MV}$ and all the shares in $\mathcal{S}_{AM}$ by $\mathbf{z}_i$.
2. Inserting the multiplicative mask $\mathbf{z}_i$ at the end of $\mathcal{S}_{MM}$.
3. Removing the first element of $\mathcal{S}_{AM}$ and adding this value to the masked value in $\mathcal{S}_{MV}$.

For the `MMtoAM` method, the $(d+1)$-multiplicative sharing $(\mathbf{z}_0, \ldots, \mathbf{z}_d)$ of $\mathbf{x}$ is associated with the sets $\mathcal{S}_{MV} = \{\mathbf{z}_0\}$, $\mathcal{S}_{MM} = \{\mathbf{z}_1, \ldots, \mathbf{z}_d\}$ and $\mathcal{S}_{AM} = \emptyset$ and the conversion consists in repeating the following treatment for every $i \in [1; d]$:

1. Masking additively the element of $\mathcal{S}_{MV}$ with $\mathbf{x}_i$.
2. Inserting the mask $\mathbf{x}_i$ at the end of $\mathcal{S}_{AM}$.
3. Removing the first component, i.e. $\mathbf{z}_i$, of $\mathcal{S}_{MM}$ and multiplying by $\mathbf{z}_i^{-1}$ the element of $\mathcal{S}_{MV}$ and all the additive shares in $\mathcal{S}_{AM}$.

This straightforward strategy is $d^{\text{th}}$-order secure when $d = 1$ or $d = 2$ but not when $d$ is higher. Indeed, it can be observed that the process of `AMtoMM` (resp. `MMtoAM`) leads to the computation of the value $\mathbf{x}_d \dot{\otimes} \dot{\bigotimes}_{i=1}^{d} \mathbf{z}_i$ (resp. $\mathbf{x}_1 \dot{\otimes} \dot{\bigotimes}_{i=1}^{d} \mathbf{z}_i^{-1}$). Hence, if $\mathbf{x}_d \neq 0$ (resp. $\mathbf{x}_1 \neq 0$), then the secret value $\mathbf{x}$ may be recovered from $\mathbf{x}_d$, $\mathbf{x}_d \dot{\otimes} \dot{\bigotimes}_{i=1}^{d} \mathbf{z}_i$ and $\mathbf{x} \dot{\otimes} \dot{\bigotimes}_{i=1}^{d} \mathbf{z}_i$ (resp. $\mathbf{x}_1$, $\mathbf{x}_1 \dot{\otimes} \dot{\bigotimes}_{i=1}^{d} \mathbf{z}_i^{-1}$ and $\mathbf{x} \dot{\otimes} \dot{\bigotimes}_{i=1}^{d} \mathbf{z}_i$). In both cases, this means that 3 shares are sufficient to recover $\mathbf{x}$ which implies that the straightforward schemes are never $3^{\text{rd}}$-order secure.

In order to solve this issue, we slightly modify our approach. In place of the third step in the sequence related to `AMtoMM`, we mask at order 1 all the shares in $\mathcal{S}_{AM}$ with new fresh random values, except for the last share which stays unchanged. We remove all those elements from $\mathcal{S}_{AM}$ and we add them to the element in $\mathcal{S}_{MV}$. Finally, we insert all the new fresh random masks into $\mathcal{S}_{AM}$. For the `MMtoAM` transformation, we do not replace the third step of the sequence and instead, we add a fourth step during which all the shares in $\mathcal{S}_{AM}$ are masked at order 1 with new fresh random values. We remove then all those values from $\mathcal{S}_{AM}$ and we add them to the element in $\mathcal{S}_{MV}$. Finally, we insert all the new fresh random masks into $\mathcal{S}_{AM}$.

We present in Alg. 1 the sequence of the different steps required for the conversion of an additive masking into a multiplicative one.

**Algorithm 1.** Secure `AMtoMM`$(\cdot)$

Input(s): A $(d+1)$-additive sharing $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ of $\mathbf{x}$
Output(s): A $(d+1)$-multiplicative sharing $(\mathbf{z}_0, \ldots, \mathbf{z}_d)$ of $\mathbf{x}$

```
1.      $\mathbf{z}_0 \leftarrow \mathbf{x}_0$
2.      for $i = 1$ to $d$ do
2.1.        $\mathbf{z}_i \leftarrow \mathbf{rand}((\mathrm{GF}(2^n)^\star)^m)$
2.2.        $\mathbf{z}_0 \leftarrow \mathbf{z}_0 \dot{\otimes} \mathbf{z}_i$
2.3.        for $j = 1$ to $d - i$ do
2.3.1.          $U \leftarrow \mathbf{rand}(\mathrm{GF}(2^n)^m)$
2.3.2.          $\mathbf{x}_j \leftarrow \mathbf{z}_i \dot{\otimes} \mathbf{x}_j$
                ** Refreshing of the additive share
2.3.3.          $\mathbf{x}_j \leftarrow \mathbf{x}_j \oplus U$
2.3.4.          $\mathbf{z}_0 \leftarrow \mathbf{z}_0 \oplus \mathbf{x}_j$
2.3.5.          $\mathbf{x}_j \leftarrow U$
2.4.        $\mathbf{x}_{d-i+1} \leftarrow \mathbf{z}_i \dot{\otimes} \mathbf{x}_{d-i+1}$
2.5.        $\mathbf{z}_0 \leftarrow \mathbf{z}_0 \oplus \mathbf{x}_{d-i+1}$
3.      return $(\mathbf{z}_0, \mathbf{z}_1, \ldots, \mathbf{z}_d)$
```

Alg. 2 describes the different steps to convert a multiplicative sharing into an additive one.

**Algorithm 2.** Secure MMtoAM($\cdot$)

INPUT(S): A $(d+1)$-multiplicative sharing $(\mathbf{z}_0, \dots, \mathbf{z}_d)$ of $\mathbf{x}$
OUTPUT(S): A $(d+1)$-additive sharing $(\mathbf{x}_0, \dots, \mathbf{x}_d)$ of $\mathbf{x}$

```
1.      $\mathbf{x}_0 \leftarrow \mathbf{z}_0$
2.      for $i = 1$ to $d$ do
2.1.        $\mathbf{x}_i \leftarrow \mathbf{rand}(\mathrm{GF}(2^n)^m)$
2.2.        $\mathbf{x}_0 \leftarrow \mathbf{x}_0 \oplus \mathbf{x}_i$
2.3.        $\mathbf{x}_0 \leftarrow \mathbf{x}_0 \dot{\otimes} \mathbf{z}_i^{-1}$
2.4.        for $j = 1$ to $i$ do
2.4.1.          $\mathbf{x}_j \leftarrow \mathbf{x}_j \dot{\otimes} \mathbf{z}_i^{-1}$
2.4.2.          $U \leftarrow \mathbf{rand}(\mathrm{GF}(2^n)^m)$
                ** Refreshing of the additive share
2.4.3.          $\mathbf{x}_j \leftarrow \mathbf{x}_j \oplus U$
2.4.4.          $\mathbf{x}_0 \leftarrow \mathbf{x}_0 \oplus \mathbf{x}_j$
2.4.5.          $\mathbf{x}_j \leftarrow U$
3.      return $(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_d)$
```

*Remark 1.* The security of AMtoMM and MMtoAM algorithms is not affected if additive masks are not refreshed during the two first steps. This optimization was not presented for the sake of clarity. Also, in our application (see Sect. 4), we will only handle the inverse of the multiplicative shares $(\mathbf{z}_1, \dots, \mathbf{z}_d)$. Therefore, AMtoMM and MMtoAM can be input with $(\mathbf{z}_0, \mathbf{z}_1^{-1}, \dots, \mathbf{z}_d^{-1})$ instead of $(\mathbf{z}_0, \dots, \mathbf{z}_d)$, so that the inverse of the $\mathbf{z}_i$ does not need to be computed inside the algorithms.

The following proposition states the completeness of AMtoMM and MMtoAM algorithms.

**Proposition 1 (Completeness).** *If* $(\mathbf{x}_0, \dots, \mathbf{x}_d)$ *is a* $(d+1)$*-additive sharing of* $\mathbf{x}$*, then algorithm* AMtoMM$(\mathbf{x}_0, \dots, \mathbf{x}_d)$ *outputs a* $(d+1)$*-multiplicative sharing of* $\mathbf{x}$. *If* $(\mathbf{z}_0, \dots, \mathbf{z}_d)$ *is a* $(d+1)$*-multiplicative sharing of* $\mathbf{x}$*, then* MMtoAM$(\mathbf{z}_0, \dots, \mathbf{z}_d)$ *outputs a* $(d+1)$*-additive sharing of* $\mathbf{x}$.

*Proof.* The proof is given in Appendix A.

In the following proposition we prove that the processings of AMtoMM and MMtoAM are $d^{\mathrm{th}}$-order secure, meaning that they perfectly thwart $d^{\mathrm{th}}$-order SCA.

**Proposition 2 (Security).** AMtoMM($\cdot$) *and* MMtoAM($\cdot$) *are* $d^{th}$*-secure.*

*Proof.* The proof is given in Appendix B.

### 3.4 Full scheme.

In this section we apply the principle presented in Sect. 3.1 and the functions introduced in Sect(s) 3.2 and 3.3 to secure the processing of a block cipher round. We assume that this round is parameterized by a secret round key $\mathbf{k} \in \mathrm{GF}(2^n)^m$ and operates a transformation of the form $\lambda' \circ \gamma \circ \lambda$ on an internal state $\mathbf{x} \in \mathrm{GF}(2^n)^m$. Functions $\lambda$ and $\lambda'$ are assumed to be automorphisms of $(\mathrm{GF}(2^n)^m, \oplus)$ and function $\gamma$ is assumed to be an automorphism of $(\mathrm{GF}(2^n)^m, \dot{\otimes})$ (*e.g.* a transformation processing bijective power functions – not necessarily the same – to the $n$-bit coordinates of the input vector). In the following algorithm, we assume that the round key $\mathbf{k} \in \mathrm{GF}(2^n)^m$ and the internal state $\mathbf{x} \in \mathrm{GF}(2^n)^m$ have been previously additively shared into $(\mathbf{k}_0, \ldots, \mathbf{k}_d)$ and $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ respectively. In the right-hand column of the following algorithm description, we added an expression of the form $\cdot \leftarrow \cdot$ to explicit to which variable (on the left) relies the sharing (on the right).

**Algorithm 3.** $d^{\text{th}}$-order secure processing of $\lambda' \circ \gamma \circ \lambda(\mathbf{x} \oplus \mathbf{k})$

INPUT(S): A $(d+1)$-additive sharing $(\mathbf{k}_0, \ldots, \mathbf{k}_d)$ of $\mathbf{k}$ and a $(d+1)$-additive sharing $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ of $\mathbf{x}$
OUTPUT(S): A $(d+1)$-additive sharing $(\mathbf{x}_0, \ldots, \mathbf{x}_d)$ of $\mathbf{x} = \lambda \circ \gamma \circ \lambda'(\mathbf{x} \oplus \mathbf{k})$

```
** Secure processing of the round-key addition
1. for $i = 0$ to $d$ do
       $\mathbf{x}_i \leftarrow \mathbf{x}_i \oplus \mathbf{k}_i$
                                        $[\mathbf{x} \oplus \mathbf{k} \leftarrow (\mathbf{x}_0, \ldots, \mathbf{x}_d)]$
** Secure processing of $\lambda$
2. for $i = 0$ to $d$ do
       $\mathbf{x}_i \leftarrow \lambda(\mathbf{x}_i)$
                                        $[\lambda(\mathbf{x} \oplus \mathbf{k}) \leftarrow (\mathbf{x}_0, \ldots, \mathbf{x}_d)]$
** Secure mapping from $\mathrm{GF}(2^n)^m$ into $(\mathrm{GF}(2^n)^\star)^m$
** The $(d+1)$-additive sharing $(\Delta_0, \ldots, \Delta_d)$ of $\delta(\mathbf{x} \oplus \mathbf{k})$ is saved in memory
3. $(\mathbf{x}_0, \ldots, \mathbf{x}_d) \leftarrow$ SecDirac$(\mathbf{x}_0, \ldots, \mathbf{x}_d)$
                                        $[\lambda(\mathbf{x} \oplus \mathbf{k}) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k})) \leftarrow (\mathbf{x}_0, \ldots, \mathbf{x}_d)]$
** Secure conversion of the additive masking into a multiplicative one
4. $(\mathbf{z}_0, \ldots, \mathbf{z}_d) \leftarrow$ AMtoMM$(\mathbf{x}_0, \ldots, \mathbf{x}_d)$
                                        $[\lambda(\mathbf{x} \oplus \mathbf{k}) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k})) \leftarrow (\mathbf{z}_0, \ldots, \mathbf{z}_d)]$
** Secure processing of $\gamma$
5. $\mathbf{z}_0 \leftarrow \gamma(\mathbf{z}_0)$
6. for $i = 1$ to $d$ do
       $\mathbf{z}_i \leftarrow \gamma(\mathbf{z}_i)$
                                        $[\gamma \circ \lambda(\mathbf{x} \oplus \mathbf{k}) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k})) \leftarrow (\mathbf{z}_0, \ldots, \mathbf{z}_d)]$
** Secure conversion of the multiplicative masking into an additive one
7. $(\mathbf{x}_0, \ldots, \mathbf{x}_d) \leftarrow$ MMtoAM$(\mathbf{z}_0, \ldots, \mathbf{z}_d)$
                                        $[\gamma \circ \lambda(\mathbf{x} \oplus \mathbf{k}) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k})) \leftarrow (\mathbf{x}_0, \ldots, \mathbf{x}_d)]$
** Secure mapping from $(\mathrm{GF}(2^n)^\star)^m$ into $\mathrm{GF}(2^n)^m$
8. for $i = 0$ to $d$ do
```

```
        $\mathbf{x}_i \leftarrow \mathbf{x}_i \oplus \Delta_i$
                                        $[\gamma \circ \lambda(\mathbf{x} \oplus \mathbf{k}) \leftarrow (\mathbf{x}_0, \dots, \mathbf{x}_d)]$
** Secure processing of $\lambda'$
9. for $i = 0$ to $d$ do
        $\mathbf{x}_i \leftarrow \lambda'(\mathbf{x}_i)$
                                        $[\lambda' \circ \gamma \circ \lambda(\mathbf{x} \oplus \mathbf{k}) \leftarrow (\mathbf{x}_0, \dots, \mathbf{x}_d)]$
10. return $(\mathbf{x}_0, \dots, \mathbf{x}_d)$
```

**Completeness** Steps 1 and 2 process linear operations/transformations. They have been specified according to the Affine Transformations processing described in Sect. 3.1. Namely, each elementary transformation is processed to each share separately and the completeness holds thanks to the linearity of $\lambda$ (namely $\bigoplus_{i=0}^{d} \lambda(\mathbf{x}_i) = \lambda(\mathbf{x})$). Step 3 securely transforms the $(d+1)$-additive sharing of $\lambda(\mathbf{x} \oplus \mathbf{k})$ into a $(d+1)$-additive sharing of $\lambda(\mathbf{x} \oplus \mathbf{k}) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k}))$ and the $(d+1)$-additive sharing $(\Delta_0, \dots, \Delta_d)$ of $\delta(\lambda(\mathbf{x} \oplus \mathbf{k}))$ is reserved for future use (at Step 8). Then, the additive masking $(\mathbf{x}_0, \dots, \mathbf{x}_d)$ of $\lambda(\mathbf{x} \oplus \mathbf{k}) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k}))$ is converted into a multiplicative one $(\mathbf{z}_0, \dots, \mathbf{z}_d)$. Since $\lambda(\mathbf{x} \oplus \mathbf{k}) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k}))$ is now masked multiplicatively, the computation of $\gamma(\lambda(\mathbf{x} \oplus \mathbf{k}) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k})))$ is simply performed by applying $\gamma$ to each multiplicative share $\mathbf{z}_i$. After Step 6, $(\mathbf{z}_0, \dots, \mathbf{z}_d)$ is a multiplicative sharing of $\gamma(\lambda(\mathbf{x} \oplus \mathbf{k}) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k})))$, that is a multiplicative sharing of $\gamma(\lambda(\mathbf{x} \oplus \mathbf{k})) \oplus \delta(\lambda(\mathbf{x} \oplus \mathbf{k}))$ since $\gamma$ maps any coordinate of $\mathbf{x} \in \mathrm{GF}(2^n)^m$ into 0 if it equals 0 and into 1 if it equals 1. The multiplicative sharing is hence transformed into an additive one at Step 7. Eventually at Step 8, the additive sharing $(\Delta_0, \dots, \Delta_d)$ of $\delta(\lambda(\mathbf{x} \oplus \mathbf{k}))$ is added to get the additive sharing of $\gamma(\lambda(\mathbf{x} \oplus \mathbf{k}))$. Step 9 updates the additive sharing to take the linear transformation $\lambda'$ into account, which results in the expected additive sharing of $\lambda'(\gamma(\lambda(\mathbf{x} \oplus \mathbf{k})))$.

**Security** The security of Alg. 3 w.r.t. $d^{\text{th}}$-order SCA can be deduced from the local resistance of its main steps. Steps 1, 2, 5-6, 8 and 9 operate a transformation or an operation on each share of the $(d+1)$-sharing of the internal state independently. They are therefore secure against $d^{\text{th}}$-order SCA. The security of `SecDirac` has been proved in [8] and is a direct consequence of the security proof in [10]. Eventually, transformations `AMtoMM` and `MMtoAM` have been proved to be secure against $d^{\text{th}}$-order SCA in Sect. 3.3. We deduce that Alg. 3 thwarts $d^{\text{th}}$-order SCA for any $d$.

**Complexity** We list in Tables 1 and 2 the complexity of Alg. 3 in terms of the following elementary operations: $(n,n)$-matrix transpositions $\mathbf{M}^{\intercal}$ required for the secure Dirac computation (see [8]), $n$-bit operations `AND`, `XOR` and $\otimes$ and transformations $\lambda$, $\gamma$ and $\lambda'$ over $\mathrm{GF}(2^n)^m$. To have interpretable results we consider separately the operations related to: (1) the secure mapping from $\mathrm{GF}(2^n)^m$ to $(\mathrm{GF}(2^n)^\star)^m$ (*i.e.* `SecDirac`), (2) the conversion functions (Algorithms 1 and

2) and (3) the remaining transformations in Alg. 3 (right-hand column of Table 2).

**Table 1.** Complexity of Algorithm SecDirac.

| Order | SecDirac | | |
|---|---|---|---|
| | $\mathbf{M}^\intercal$ | XOR | AND |
| 1 | $2m/n$ | $28m/n+2m$ | $28m/n$ |
| 2 | $3m/n$ | $84m/n+3m$ | $63m/n$ |
| 3 | $4m/n$ | $168m/n+4m$ | $112m/n$ |
| $d$ | $(d+1)m/n$ | $(14d+n)(d+1)m/n$ | $7(d+1)^2m/n$ |

**Table 2.** Complexity of the masking conversions and Algorithm 3.

| Order | AMtoMM | | MMtoAM | | Algorithm 3 | | | |
|---|---|---|---|---|---|---|---|---|
| | XOR | $\otimes$ | XOR | $\otimes$ | $\lambda$ | $\gamma$ | $\lambda'$ | XOR |
| 1 | $m$ | $2m$ | $3m$ | $2m$ | 2 | 2 | 2 | $4m$ |
| 2 | $4m$ | $5m$ | $8m$ | $5m$ | 3 | 3 | 3 | $6m$ |
| 3 | $9m$ | $9m$ | $15m$ | $9m$ | 4 | 4 | 4 | $8m$ |
| $d$ | $md^2$ | $\frac{md}{2}(3+d)$ | $md(2+d)$ | $\frac{md}{2}(3+d)$ | $d+1$ | $d+1$ | $d+1$ | $2m(d+1)$ |

*Remark 2.* For our implementations reported in Sect. 4 (in this case we had $m = 16$ and $n = 8$), we experimented that the cost of the $n$-bit operations XOR and AND was equal to 1 clock cycle. The cost of $\otimes$ was equal to 22 and that of $\mathbf{M}^\intercal$ was equal to 148. Moreover, we implemented the functions $\lambda$, $\lambda'$ and $\gamma$ thanks to ROM lookup tables and hence, each computation costed around $m$ clock cycles (considering that one table access costs one clock cycle).

As it can be checked in Table 2, the complexity of the secure processing of the non-linear function $\gamma$ (Steps 3 to 8 in Alg. 3) essentially corresponds to the sum of the complexities of SecDirac and Algorithms AMtoMM and MMtoAM. Neglecting the cost of the matrix transposition and assuming $m = 16$ and $n = 8$ (as it is the case for the AES), our secure processing of $\gamma$ requires $74d^2+104d+30$ operations[3] XOR or AND and $16d^2+48d$ operations $\otimes$. For comparison, the cost of Rivain and Prouff's solution [19] to secure the AES non-linear layer (which corresponds to our transformation $\gamma$) is $128d^2+192d$ operations XOR and $64d^2+128d+64$ operations $\otimes$, neglecting the cost of the look-up table accesses. In view of the two

[3] The two operations are considered globally since they have the same cost.

costs above, our solution is clearly more efficient than that in [19]. In particular, the number of operations $\otimes$ is divided by around 4, which is an important improvement considering that the latter operation is costly (around 20 times more costly than a XOR or a AND).

## 4 Application to the AES

The AES-128 is a cryptosystem that iterates 10 times a same round transformation on a 16-bytes internal state initially filled with the plaintext (*i.e.* parameters $n$ and $m$ in Sect. 3.4 equal here 8 and 16 respectively). The round is composed of a key addition AddRoundKey, a nonlinear layer SubBytes which applies the same *substitution-box* (s-box) to every byte of the internal state and linear transformations ShiftRows and MixColumns. The s-box is defined as the left-composition of a linear transformation $\lambda_A$ over GF(256) with the power function $f : x \in \mathrm{GF}(256) \mapsto x^{254} \in \mathrm{GF}(256)$, followed by the addition of a constant term. The SubBytes transformation can thus also be represented as the left composition of the two transformations $A = (\lambda_A, \ldots, \lambda_A)$ and $\mathtt{Inv} = (f, \ldots, f)$, both defining a componentwise transformation of the internal state, followed by the bitwise addition of a constant term $\mathbf{c} \in \mathrm{GF}(2^n)^m$. While $A$, ShiftRows and MixColumns are automorphisms of $(\mathrm{GF}(2^n)^m, \oplus)$, the transformation Inv defines an automorphism of $((\mathrm{GF}(2^n)^\star)^m, \dot{\otimes})$. In view of this description, it is clear that the AES round can be rewritten as a composition of transformations satisfying the assumptions done in the introduction of Sect. 3: $\lambda$ is defined as the identity function over $\mathrm{GF}(2^n)^m$, $\gamma$ is the function Inv and $\lambda'$ is the function $\mathtt{MixColumns} \circ \mathtt{ShiftRows} \circ A$. The masking scheme presented in Sect. 3.4 can thus be applied to protect the AES rounds.

In this section, we compare the efficiency of our proposal with that of the state of the art solutions when applied to secure the AES. All the implementations presented below involve the same code to process the linear transformations $\lambda'$ and AddRoundKey. Namely, we use a $(d + 1)$-additive masking such as presented previously in this article. We chose to protect all the rounds of the AES processing. To secure the $\gamma$ transformation, we chose to select few methods from the literature for $d = \{1, 2, 3\}$. In what follows, we give details on the methods we chose in each category.

*For $d = 1$,* we selected four methods. First we chose the *table re-computation method* [12] since it achieves the best timing performance. The second chosen method is the *tower field method* [15], which offers the best memory performance. Then, since the work of this paper is the generalization of the 1st-order multiplicative masking scheme proposed in [7], we implemented it as well. Eventually, we chose to implement the $d^{\mathrm{th}}$-order SCA secure scheme proposed in [19]. Though it is less efficient than the others for $d = 1$, choosing it enables to compare our proposal with another method which can be applied generally for any order $d$.

*For* $d = 2$, only few methods exist that are perfectly SCA secure. Actually, only the works [20], [18] and [19] propose such kind of schemes. We chose to implement all of them.

*For* $d = 3$, only [19] proposes a solution in this category.

Table 3 lists the timing/memory performances of the different implementations. We wrote the codes in assembly language for an `8051` based 8-bit architecture with bit-addressable memory. `RAM` consumption related to implementation choices (*e.g.* use of some local variables, use of pre-computed values to speed-up some computations, etc.) are not taken into account in the performances reporting. Also, `ROM` consumptions (*i.e.* code sizes) are not listed since they are not prohibitive for almost all current embedded devices. Eventually, cycles numbers are multiple of $10^3$.

**Table 3.** Comparison of secure AES implementations

| Method | | Reference | cycles ($10^3$) | RAM (bytes) |
|---|---|---|---|---|
| Unprotected Implementation | | | | |
| 1. | No Masking | Na. | 2 | 32 |
| First-Order Masking | | | | |
| 2. | Re-computation | [12] | 10 | 256 |
| 3. | Tower Field in GF($2^2$) | [15, 16] | 77 | 42 |
| 4. | Multiplicative Masking | [7] | 22 | 256 |
| 5. | Secure exponentiation for $d = 1$ | [19] | 73 | 24 |
| 6. | Our scheme for $d = 1$ | This paper | 25 | 50 |
| Second-Order Masking | | | | |
| 7. | Double Re-computations | [20] | 594 | $512 + 28$ |
| 8. | Single Re-computation | [18] | 672 | $256 + 22$ |
| 9. | Secure exponentiation for $d = 2$ | [19] | 189 | 48 |
| 10. | Our scheme for $d = 2$ | This paper | 69 | 86 |
| Third-Order Masking | | | | |
| 11. | Secure exponentiation for $d = 3$ | [19] | 326 | 72 |
| 12. | Our scheme for $d = 3$ | This paper | 180 | 128 |

*Remark 3.* For $d = 1$ (Implementations 2 to 5), improvements have been added to the original proposals. They essentially amount to preprocess a part of the masking material, which is possible since the latter one does not need to be changed during the algorithm processing when only first-order SCA are considered.

We observe that only two methods achieve better timing performances than our proposal and that this occurs only in the case $d = 1$. As expected, the re-computation remains the most efficient method when 256 bytes of RAM are available. We can also note that the original countermeasure involving multiplicative masking [7] stays better than our countermeasure (which merely generalizes it at any order). The difference is due to the tabulation of the Dirac function used in [7] which implies a faster processing than the algebraic implementation of this function but at the cost of memory. Except those two particular cases, it turns out that our proposal is the most efficient one: it is at least 2.9 times faster for $d = 1, 2$ and 1.8 times faster for $d = 3$. Even if our scheme requires more RAM than [19], the consumption stays lower than 200 bytes and is therefore acceptable for almost all embedded systems (even the low cost ones).

Memory and timing performances of the solution [19] and those of our proposal progress similarly as soon as the order increases. This is explained by the fact that both methods use the same approach to thwart SCA, that is to replace each transformation calculation by a sequence of elementary calculations. To secure them, the solution [19] involves additive maskings while our solution mixes additive and multiplicative maskings. Memory allocation differences between the two methods are merely due to the fact that additional vectors are required in our scheme since it involves more shares (multiplicative shares, dirac shares, etc.). The differences of timing performances come from the fact that solution [19] involve much more field multiplications than in our proposal (see Tables 1 and 2).

## 5 Conclusion

In this paper, we have introduced a new higher-order masking scheme dedicated to block ciphers mixing affine transformations with power functions. It is provably secure at any chosen order and can be implemented in software at the cost of a reasonable overhead. In particular, it is an efficient alternative to [19] in order secure the AES implementation at any order. For our construction, we have introduced conversion functions that can securely transform an additive masking into a multiplicative one. We think that those transformations could be interesting as secure primitives in other contexts where security against higher-order side channel attacks must be achieved and power functions are involved.

## References

1. Mehdi-Laurent Akkar and C. Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Ç.K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.
2. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1988. ACM.

3. S. Chari, C.S. Jutla, J.R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M.J. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
4. Jean-Sébastien Coron. A New DPA Countermeasure Based on Permutation Tables. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *Security and Cryptography for Networks, 6th International Conference, SCN 2008*, volume 5229 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2008.
5. Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting Circuits from Leakage: the Computationally-Bounded and Noisy Cases. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 135–156. Springer, 2010.
6. Guillaume Fumaroli, Ange Martinelli, Emmanuel Prouff, and Matthieu Rivain. Affine masking against higher-order side channel analysis. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 262–280. Springer, 2010.
7. Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Secure multiplicative masking of power functions. In Jianying Zhou and Moti Yung, editors, *ACNS*, volume 6123 of *Lecture Notes in Computer Science*, pages 200–217, 2010.
8. Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Montgomery's trick and fast implementation of masked aes. In Abderrahmane Nitaj and David Pointcheval, editors, *AFRICACRYPT*, volume 6737 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2011.
9. J. Golić and C. Tymen. Multiplicative Masking and Power Analysis of AES. In B.S. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2002.
10. Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In D. Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.
11. M. Joye, P. Paillier, and B. Schoenmakers. On Second-order Differential Power Analysis. In J.R. 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. T.S. 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.
13. Elisabeth Oswald and Stefan Mangard. Template Attacks on Masking—Resistance is Futile. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 243–256. Springer, 2007.
14. Elisabeth Oswald, Stefan Mangard, Christoph Herbst, and Stefan 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*, pages 192–207. Springer, 2006.
15. Elisabeth Oswald, Stefan Mangard, and Norbert Pramstaller. Secure and Efficient Masking of AES – A Mission Impossible ? Cryptology ePrint Archive, Report 2004/134, 2004.
16. Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A Side-Channel Analysis Resistant Description of the AES S-box. In H. Handschuh and H. Gilbert, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2005.

17. Eric Peeters, Francois-Xavier Standaert, Nicolas Donckers, and Jean-Jacques Quisquater. Improved Higher-order Side-Channel Attacks with FPGA Experiments. In J.R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 309–323. Springer, 2005.
18. Matthieu Rivain, Emmanuelle Dottax, and Emmanuel Prouff. Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis. In T. Baignères and S. Vaudenay, editors, *Fast Software Encryption – FSE 2008*, Lecture Notes in Computer Science, pages 127–143. Springer, 2008.
19. Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.
20. Kai Schramm and Christof 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.
21. Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, November 1979.
22. Elena Trichina, D. DeSeta, and L. Germani. Simplified Adaptive Multiplicative Masking for AES. In B.S. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 187–197. Springer, 2002.
23. J. Waddle and D. Wagner. Toward Efficient Second-order Power Analysis. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

## A Proof of Proposition 1

*Proof of Proposition 1.* In the following we use $\cdot$ and $\prod$ in place of $\dot{\otimes}$ and $\dot{\bigotimes}$ to simplify the notations and because there is no ambiguity. They both refer to the componentwise product over $\mathrm{GF}(2^n)^m$.

In what follows $\mathcal{S}^i_{MV}, \mathcal{S}^i_{AM}$ and $\mathcal{S}^i_{MM}$ denote the state of the sets $\mathcal{S}_{MV}$, $\mathcal{S}_{AM}$ and $\mathcal{S}_{MM}$ at the output of the loop $i$ for $i = 1 \dots d$. By definition, $\mathcal{S}^0_{MV}$, $\mathcal{S}^0_{AM}$ and $\mathcal{S}^0_{MM}$ are the states of the sets $\mathcal{S}_{MV}$, $\mathcal{S}_{AM}$ and $\mathcal{S}_{MM}$ at the input of the algorithm. The uniform random variable $U$ generated during the $i^{\text{th}}$ iteration of the loop at Step 2 and the $j^{\text{th}}$ iteration of the loop at Step 2.3 of `AMtoMM` (resp. at Step 2.4 of `MMtoAM`) will be denoted by $U^{(i)}_{i+j}$.

Let us prove the completeness of `AMtoMM`. By definition, $U^{(0)}_j = \mathbf{x}_j$ for $j = 1 \dots d$. We claim that for $i = 0 \dots d$, we have:

$$\mathcal{S}^i_{MV} = \{\mathbf{x} \cdot \prod_{j=1}^{i} \mathbf{z}_j \oplus \bigoplus_{j=i+1}^{d} U^{(i)}_j\}, \mathcal{S}^i_{AM} = \{U^{(i)}_{i+1}, \dots, U^{(i)}_d\} \tag{2}$$

and

$$\mathcal{S}^i_{MM} = \{\mathbf{z}_1, \dots, \mathbf{z}_i\} \, . \tag{3}$$

For $i = 0$, we easily check that these sets are reduced to the sets associated to the $(d+1)$-additive sharing $(\mathbf{x}_0, \dots, \mathbf{x}_d)$ of $\mathbf{x}$, if we take into account the conventions and the definitions. Let us now assume that (2) and (3) are satisfied for $i$ and let us prove that it is still valid for $i+1$. We first multiply the value of $\mathcal{S}_{MV}$ and the elements of $\mathcal{S}_{AM}$ by $\mathbf{z}_{i+1}$ (resp. Step 2.2 and Step 2.3.2 in Alg. 1). Also, we insert $\mathbf{z}_{i+1}$ into $\mathcal{S}_{MM}$. We get

$$\mathcal{S}^i_{MV} = \left\{ \left( \mathbf{x} \cdot \prod_{j=1}^{i} \mathbf{z}_j \oplus \bigoplus_{j=i+1}^{d} U_j^{(i)} \right) \cdot \mathbf{z}_{i+1} \right\} ,$$

$$\mathcal{S}^i_{AM} = \{ U_{i+1}^{(i)} \cdot \mathbf{z}_{i+1}, \ldots, U_d^{(i)} \cdot \mathbf{z}_{i+1} \} ,$$

and

$$\mathcal{S}^i_{MM} = \{ \mathbf{z}_1, \ldots, \mathbf{z}_i, \mathbf{z}_{i+1} \} .$$

We mask the $d-(i+1)$ first shares of $\mathcal{S}^i_{AM}$ with $U_{i+2}^{(i+1)}$, $U_{i+3}^{(i+1)}$, ..., and $U_d^{(i+1)}$ respectively (Step 2.3.3). We add all the elements of $\mathcal{S}^i_{AM}$ to the element of $\mathcal{S}_{MV}$ (Step 2.3.4). Hence, we get:

$$\mathcal{S}^i_{MV} = \left\{ \left( \mathbf{x} \cdot \prod_{j=1}^{i} \mathbf{z}_j \oplus \bigoplus_{j=i+1}^{d} U_j^{(i)} \right) \cdot \mathbf{z}_{i+1} \oplus \bigoplus_{j=i+1}^{d-1} (U_j^{(i)} \cdot \mathbf{z}_{i+1} \oplus U_{j+1}^{(i+1)}) \oplus U_d^{(i)} \cdot \mathbf{z}_{i+1} \right\} ,$$

that is

$$\mathcal{S}^i_{MV} = \{ \mathbf{x} \cdot \prod_{j=1}^{i+1} \mathbf{z}_j \oplus \bigoplus_{j=i+2}^{d} U_j^{(i+1)} \} .$$

Finally, we replace the elements of $\mathcal{S}_{AM}$ with the masks $U_{i+2}^{(i+1)}$, $U_{i+3}^{(i+1)}$, ..., and $U_d^{(i+1)}$ (Step 2.3.5) and we get:

$$\mathcal{S}^i_{MV} = \{ \mathbf{x} \cdot \prod_{j=1}^{i+1} \mathbf{z}_j \oplus \bigoplus_{j=i+2}^{d} U_j^{(i+1)} \},$$

$$\mathcal{S}^i_{AM} = \{ U_{i+2}^{(i+1)}, \ldots, U_d^{(i+1)} \}$$

and

$$\mathcal{S}^i_{MM} = \{ \mathbf{z}_1, \ldots, \mathbf{z}_i, \mathbf{z}_{i+1} \} .$$

This is exactly the definition of the sets $\mathcal{S}^{i+1}_{MV}$, $\mathcal{S}^{i+1}_{AM}$ and $\mathcal{S}^{i+1}_{MM}$. Finally, it is not hard to check that $\mathcal{S}^d_{MV}$, $\mathcal{S}^d_{AM}$ and $\mathcal{S}^d_{MM}$ are the sets associated to the $(d+1)$-multiplicative sharing $(\mathbf{z}_0, \ldots, \mathbf{z}_d)$ of $\mathbf{x}$. The proof of the completeness of the algorithm `MMtoAM` is similar. ◇

## B Proof of Proposition 2

*Proof of Proposition 2.*

In this proof, we use the notations introduced in Appendix A. We moreover add the notation $\mathcal{S}^{i,j}_{MV}$ to denote the set containing the value of the masked variable ($\mathbf{z}_0$) after the $j^{\text{th}}$ iteration of the loop at Step 2.3 at the $i^{\text{th}}$ iteration of the loop at Step 2 of the Algorithm `AMtoMM` (it corresponds to the value of $\mathbf{z}_0$ after Step 2.3.4). To prove the security of `AMtoMM`$(\cdot)$, we demonstrate by recurrence that no $d$-tuple of intermediate variables during its processing depends on $\mathbf{x}$. With the notation introduced in the completeness proof, this is equivalent to prove the following property that we call Property($d$): no $d$-tuple of elements in $\cup_{0\leq j\leq d}\mathcal{S}^{j}_{AM}$, $\cup_{0\leq j\leq d}\mathcal{S}^{j}_{MM}$ and $\cup_{0\leq i,j\leq d}\mathcal{S}^{i,j}_{MV}$ depends on $\mathbf{x}$. In our recurrence proof, we will show that Property($d$) is trivially satisfied for $d=0$. Then, we will prove that if Property($\ell$) is satisfied, then so does Property($\ell+1$).

At the beginning of the `AMtoMM` processing, the $d$ additive shares $\mathbf{x}_1$, ..., $\mathbf{x}_d$ of $\mathbf{x}$ and the masked variable $\mathbf{x}\oplus\bigoplus_{i=1}^{d}\mathbf{x}_i$ are manipulated. They correspond respectively to the variables $U_1^{(0)},\ldots,U_d^{(0)}$ in the set $\mathcal{S}^0_{AM}$ and to the initial content of $\mathcal{S}^0_{MV}$. We hence have:

$$\mathcal{S}^0_{AM}=\{U_1^{(0)},\ldots,U_d^{(0)}\},\ \mathcal{S}^0_{MM}=\emptyset \text{ and } \mathcal{S}^0_{MV}=\{\mathbf{x}\oplus\bigoplus_{i=1}^{d}U_i^{(0)}\}\ .$$

Since $(U_1^{(0)},\ldots,U_d^{(0)},\mathbf{x}\oplus\bigoplus_{i=1}^{d}U_i^{(0)})$ is a $(d+1)$-sharing of $\mathbf{x}$, Property(0) is satisfied.

Let us now assume that Property($\ell$) is satisfied for $\ell\geq 0$ and let us prove that it is still valid for $\ell+1$. During the $(\ell+1)^{\text{th}}$ processing of the loop at Step 2, the element of $\mathcal{S}_{MV}$ and the elements of $\mathcal{S}_{AM}$ are all multiplied by $\mathbf{z}_{\ell+1}$ (resp. Step 2.2 and Step 2.3.2 in Alg. 1). Also, we insert $\mathbf{z}_{\ell+1}$ into $\mathcal{S}_{MM}$. We get

$$\mathcal{S}^{\ell}_{MV}=\left\{\left(\mathbf{x}\cdot\prod_{i=1}^{\ell}\mathbf{z}_i\oplus\bigoplus_{i=1}^{d-\ell}U_{\ell+i}^{(\ell)}\right)\cdot\mathbf{z}_{\ell+1}\right\}\ ,$$

$$\mathcal{S}^{\ell}_{AM}=\{U_{\ell+1}^{(\ell)}\cdot\mathbf{z}_{\ell+1},\ldots,U_d^{(\ell)}\cdot\mathbf{z}_{\ell+1}\}\ ,$$

and

$$\mathcal{S}^{\ell}_{MM}=\{\mathbf{z}_1,\ldots,\mathbf{z}_{\ell},\mathbf{z}_{\ell+1}\}\ .$$

Since $\mathbf{z}_{\ell+1}$ is new fresh random value independent of all the previously generated shares and since the multiplication by $\mathbf{z}_{\ell+1}$ is done on each share separately, Property($\ell$) stays satisfied. After Step 2.3.3, some shares in $\mathcal{S}^{(\ell)}_{AM}$ are masked with new random values $U_{\ell+2}^{(\ell+1)}$, $U_{\ell+3}^{(\ell+1)}$, ..., and $U_d^{(\ell+1)}$ mutually independent on $\{U_{t+j}^{(t)};t\leq\ell,j\leq d-t\}$ and $\{\mathbf{z}_\ell\}$. Once again, Property($\ell$) stays trivially satisfied and $\mathcal{S}_{AM}$ became:

$$\mathcal{S}^{\ell}_{AM}=\{U_{\ell+1}^{(\ell)}\cdot\mathbf{z}_{\ell+1}\oplus U_{\ell+2}^{(\ell+1)},\ldots,U_{d-1}^{(\ell)}\cdot\mathbf{z}_{\ell+1}\oplus U_d^{(\ell+1)},U_d^{(\ell)}\cdot\mathbf{z}_{\ell+1}\}\ .$$

During the $j^{\text{th}}$ processing of Step 2.3.4, the $j^{\text{th}}$ element of $\mathcal{S}^{(\ell)}_{AM}$ is added to $\mathcal{S}^{\ell}_{MV}$, resulting in the following set $\mathcal{S}^{\ell,j}_{MV}$:

$$\mathcal{S}_{MV}^{\ell,j} = \{(\mathbf{x} \cdot \prod_{i=1}^{\ell+1} \mathbf{z}_i) \oplus (\mathbf{z}_{\ell+1} \cdot \bigoplus_{i=j+1}^{d-\ell} U_{\ell+i}^{(\ell)}) \oplus \bigoplus_{i=1}^{j} U_{\ell+1+i}^{(\ell+1)}\} \ .$$

In $\mathcal{S}_{MV}^{\ell,j}$, the value $(\mathbf{x} \cdot \prod_{i=1}^{\ell+1} \mathbf{z}_i)$ is $(d-\ell)^{\text{th}}$-order additively masked (recalling that $\mathbf{z}_{\ell+1}$ is a random non-zero value). Since this level of additive masking was assumed to be sufficient to protect $(\mathbf{x} \cdot \prod_{j=1}^{\ell} \mathbf{z}_j$ in $\mathcal{S}_{MV}^{\ell}$ (by Property($\ell$)) and since $\mathbf{z}_{\ell+1}$ is a fresh non-zero random value, Property($\ell$) stays satisfied after each processing of a value $\mathcal{S}_{MV}^{\ell,j}$ for $j \leq d-\ell$.

Let us now check whether Property($\ell$) stays satisfied after the processing of Step 2.5. After this step, we have:

$$\mathcal{S}_{MV}^{\ell+1} = \{(\mathbf{x} \cdot \prod_{i=1}^{\ell+1} \mathbf{z}_i) \oplus \bigoplus_{i=1}^{d-(\ell+1)} U_{\ell+1+i}^{(\ell+1)}\} \ ,$$

and

$$\mathcal{S}_{AM}^{\ell+1} = \{U_{\ell+2}^{(\ell+1)}, \ldots, U_{d}^{(\ell+1)}\} \, .$$

Namely, the value $\mathbf{x} \cdot \prod_{j=1}^{\ell+1} \mathbf{z}_j$ is $(d-\ell-1)^{\text{th}}$-order additively masked, where the masks $U_{\ell+i}^{(\ell+1)}$ have always been manipulated separately. Since $\mathbf{x}$ and the $\mathbf{z}_i$ are all independent and non-zero, $(\mathbf{x} \cdot \prod_{j=1}^{\ell+1} \mathbf{z}_i, \mathbf{z}_1, \ldots \mathbf{z}_{\ell+1})$ is a $(\ell+1)$-multiplicative sharing of $\mathbf{x}$. The $\mathbf{z}_i$ have always been manipulated separately and the refreshing with the $U_{\ell+i}^{(\ell+1)}$ ensures that no tuple of data in $\mathcal{S}_{AM}^{(\ell)}$ depends an $(\mathbf{z}_1, \ldots, \mathbf{z}_{\ell+1})$. Eventually, this proves that $\mathbf{x}$ is still $(d+1)^{\text{th}}$-order shared (multiplicatively and additively) which implies that Property($\ell+1$) is satisfied and concludes the proof by recurrence.

The proof of the security of `MMtoAM`$(\cdot)$ is similar. $\diamond$