# Montgomery's Trick and Fast Implementation of Masked AES

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.** Side Channel Analysis (SCA) is a class of attacks that exploit leakage of information from a cryptographic implementation during execution. To thwart it, masking is a common strategy that aims at hiding correlation between the manipulated secret key and the physical measures. Even though the soundness of masking has often been argued, its application is very time consuming, especially when so-called higher-order SCA (HO-SCA) are considered. Reducing this overhead at the cost of limited RAM consumption increase is a hot topic for the embedded security industry. In this paper, we introduce such an improvement in the particular case of the AES. Our approach consists in adapting a trick introduced by Montgomery to efficiently compute several inversions in a multiplicative group. For such a purpose, and to achieve security against HO-SCA, recent works published at CHES 2010 and ACNS 2010 are involved. In particular, the secure dirac computation scheme introduced by Genelle *et al.* at ACNS is extended to achieve security against SCA at any order. As argued in the second part of this paper, our approach improves in time complexity all previous masking methods requiring little RAM .

**Keywords:** Montgomery's Trick, Side Channel Analysis, Secret Sharing, AES.

## 1 Introduction

In the nineties, a new family of attacks against implementations of cryptographic algorithms in embedded devices has been introduced. The idea of those attacks, called *Side Channel Analysis*, is to take advantage of the correlation between the manipulated secret data (*e.g.* secret keys) and physical measures such as the power consumption of the device. During the last two decades, the development of the smart card industry has urged the cryptographic research community to carry on with SCA and many papers describing either countermeasures or attacks developments have been published. In particular, the original attacks in [2, 9] have been improved and the concept of higher-order SCA (HO-SCA) has been introduced [11]. It consists in targeting the manipulation of several (and not only one) intermediate values at different times or different locations during

A. Nitaj and D. Pointcheval (Eds.): AFRICACRYPT 2011, LNCS 6737, pp. 153–169, 2011.

the algorithm processing to reveal information on secret-dependent data called *sensitive data*. A HO-SCA targeting $d$ intermediate values is usually named $d^{th}$*-order SCA*.

A common countermeasure against SCA is to randomize any sensitive variable appearing during the algorithm processing by *masking* techniques (also known as *secret sharing*)[1, 5, 7, 10, 17, 18, 22]. The principle is to randomly split each sensitive variable into several shares which will be manipulated separately. The shares propagate throughout the algorithm in such a way that no intermediate variable is sensitive. An advantage of $d^{th}$*-order masking* schemes, for which the number of shares per sensitive variable is $d + 1$, is that they perfectly thwart $d^{\text{th}}$-order SCA. Moreover, whatever the kind of attacks (HO-SCA of any order or template attacks [15]), their soundness as a countermeasure has been argued for realistic leakage models in [3], where it is proved that the difficulty of recovering information on a variable shared into several parts grows exponentially with the number of shares. Resistance against HO-SCA is of importance since their effectiveness has been demonstrated against some family of devices [11, 16, 25]. Nowadays it must therefore be possible to easily scale the security of an implementation, starting from a resistance against $1^{\text{st}}$-order SCA and possibly going to resistance against $d^{\text{th}}$-order SCA for any $d$. In the case of block ciphers such as AES, the most critical part to protect when applying masking is the non-linear layer. The latter one involves 16 times a same non-linear function, called *s-box*. Several methods have been proposed in the literature to deal with this issue. We list in the next section those that are privileged, to the best of our knowledge, by the embedded device industry.

### 1.1 Related Work

State-of-the-art methods to protect the AES non-linear layer can be split into two categories. In one part we have methods that involve pre-computed look-up tables in `RAM`[1] to achieve good timing performances [5, 10, 19]. They are moreover particularly dedicated to $1^{\text{st}}$-order SCA. A few attempts have been done to extend these methods to deal with HO-SCA [22, 24]. However the approach did not permit to thwart SCA at order 3 or higher. Moreover, security against $2^{\text{nd}}$-order SCA is only achieved at the cost of a prohibitive memory overhead which excludes its use in low-cost devices. In brief, countermeasures that use pre-computed tables cannot be used to protect algorithms at order $d > 1$ in a `RAM` constrained environment. In a second part, we have methods that achieve SCA security at the cost of a limited amount of `RAM` memory (*e.g.* less than 100 bytes), which is particularly relevant for the smart cards industry [1, 17, 18, 20]. They have in common to exploit the simple algebraic structure of the AES s-box, which is affinely equivalent to a field inversion extended in 0 by setting $0^{-1} = 0$. They are less efficient in terms of timing than the methods in the first category, but can be embedded in constrained devices. Moreover, in contrast with the

[1] `RAM` is a volatile memory. It can be accessed in read/write mode and is usually used to store local or global variables used by the programs.

schemes in the first category, they are more suitable for the extension of the security at order $d$, since they do not rely on the table re-computation principle. Actually a scheme has been proposed in this category that achieves $d^{\text{th}}$-order SCA resistance whatever $d$ [23]. It moreover turns out that even for $d = 2$ this scheme is much more efficient (around 3 times) than the methods based on table re-computations [22, 24].

### 1.2 Our Results

State-of-the-art methods secure the overall AES non-linear layer by separately securing each of the 16 s-boxes computation. Their complexity is hence merely equal to that of securing the AES s-box processing, or equivalently the field inversion over GF($2^8$). In this paper a different approach is introduced, where the masking of the whole non-linear layer is considered at once. This is accomplished by applying Montgomery's trick [13]. The latter one can be applied in any multiplicative group and enables to compute say $n$ multiplicative inverses at the cost of $3(n-1)$ field multiplications and one single inversion. It is relevant when the inversion processing is much more costly than that of a multiplication (*i.e.* at least around 3 times more costly) which is often the case in the context where the inversion includes SCA countermeasures. Even if the context of AES secure implementation seems to be a natural outlet for Montgomery's trick, its application is not straightforward. First, the field multiplications must be secured such that their use in the non-linear layer computation does not decrease the security of the implementation against (HO-)SCA. In other terms, if the inversions were resistant against $d^{\text{th}}$-order SCA, then the multiplications replacing them in the new process must also resist to those attacks. Secondly, since the elements of the AES state are defined over GF($2^8$) whereas Montgomery's trick applies on GF($2^8$)$^\star$, a pre-processing must be defined to map the state elements up to the multiplicative group without modifying the functional behavior of the AES. Moreover, the mapping must not introduce any flaw w.r.t. $d^{\text{th}}$-order SCA. This paper deals with the two issues by using memory as little as possible for any SCA resistance order $d$. First, we suggest to use the multiplication algorithm proposed in [23] which can be specified to thwart HO-SCA of any order at the cost of acceptable timings and without extra large `RAM` memory consumption. To deal with the second issue, we base our approach on the technique suggested in [5], that reduces the problematic to that of securing a Dirac function (which is a function that maps zero values into non-zero ones). We improve the method to use less `RAM` memory than in the original method and we extend it to get a Dirac computation secure at any order $d$. The use of such solutions results in a secure and efficient adaptation of Montgomery's trick in the context of SCA-resistant AES implementations. Since our approach is only relevant when the ratio between the cost of a secure inversion and that of secure multiplication is lower than some threshold, it is not suitable when both functions are tabulated once per algorithm execution. In the other cases (which include all the proposed countermeasures against HO-SCA), our proposal improves the timing performance at the cost of a small `RAM` overhead: $1^{\text{st}}$-order secure methods not based on table

re-computation are improved by at least 21% and we have a timing gain of at least 13% and 9% for $2^{\text{nd}}$-order and $3^{\text{rd}}$-order secure methods respectively.

### 1.3 Paper Organization

The paper is organized as follows. We briefly introduce in Sect. 2 some basics on AES and SCA. Section 3 describes Montgomery's trick in the context of a SCA-resistant AES implementation. Section 4 presents the different tools required for the new generic scheme. Section 5 reports on the efficiency of several implementations of our method in combination with state-of-the-art secure implementations of AES. Eventually Sect. 6 concludes the paper.

## 2 Notations and Basics on AES and SCA

We briefly introduce here the AES algorithm, and we give some notations and definitions used to describe our proposal and to analyze its security.

The AES block cipher algorithm is the composition of several rounds that operate on an internal *state* denoted by $\mathbf{s} = (s_i)_{i \leq 15}$ and viewed in the following either as a $(16 \times 1)$-matrix over $\mathrm{GF}(2^8)$ or as a $(16 \times 8)$-binary matrix (in this case each $s_i$ is considered as a vector in $\mathrm{GF}(2)^8$ whose bit-coordinates are denoted by $s_i[j]$). Field multiplication will be denoted by $\otimes$, whereas field addition will be denoted by $\oplus$. The latter exactly corresponds to the bitwise addition in $\mathrm{GF}(2)^8$. Eventually the bitwise multiplication (`AND`) will be denoted by $\odot$. Each AES round is the composition of a round-key addition, a linear layer and a non-linear layer. The latter one consists in a single AES s-box that is applied to each state element $s_i$ separately. It is defined as the composition of an affine function with the multiplicative inverse function in $\mathrm{GF}(2^8)$, i.e. $s \mapsto s^{-1}$, extended in 0 by setting $0^{-1} = 0$. We call the latter function *extended inversion* and we denote it `Inv`. The global transformation $(s_i)_{i \leq 15} \mapsto (s_i^{-1})_{i \leq 15}$ is denoted by `Inv-Layer`. In this paper, we focus on protecting the processing of `Inv-Layer` against (HO-)SCA, the round-key addition, the linear layer and the affine transformation being straightforward to secure (see for instance [10]). We moreover assume that the masking strategy is followed to protect the overall AES. When such a scheme is specified at order $d$, the state $\mathbf{s}$ is randomly split into $d+1$ shares $(\mathbf{s}^0, \cdots, \mathbf{s}^d)$ such that $\mathbf{s} = \bigoplus_{i=0}^{d} \mathbf{s}^i$. We shall say that $(\mathbf{s}^0, \cdots, \mathbf{s}^d)$ is a $(d+1)$-*sharing* of $\mathbf{s}$. After denoting by $s_i^j$ the $i^{\text{th}}$ line of the $j^{\text{th}}$ share, we can check that $(s_i^0, ..., s_i^d)$ is a $(d+1)$-sharing of the state element $s_i$. In the following, we shall say that a variable is sensitive if it can be expressed as a deterministic function of the plaintext and the secret key and which is not constant with respect to the secret key. Additionally, we shall say that an algorithm achieves $d^{th}$*-order SCA security* if every $d$-tuple of its intermediate variables is independent of any sensitive variable.

In the next section we give the core principle of our proposal to improve the timing efficiency of the state-of-the-art $d^{\text{th}}$-order SCA-secure AES implementations.

# 3 Montgomery's Trick to Secure the AES Inv-Layer: Core Idea

This section is organized as follows: first we introduce the classical approach when masking is involved, secondly we describe Montgomery's trick as introduced in [13] and eventually we show the adaption of the latter to SCA-secure AES implementations.

## 3.1 Classical Approach

Usual implementations of Inv-Layer are protected against $d^{\text{th}}$-order SCA by following a divide-and-conquer approach. The global security is deduced from the local security of each of the sixteen processings of Inv. To achieve local security, a scheme Sec-Inv$(d, \cdot)$ is involved. It applies on the $(d+1)$-sharing $(s_i^0, \cdots, s_i^d)$ of each state element $s_i$ and outputs a $(d+1)$-sharing $(r_i^0, \cdots, r_i^d)$ of the inverse Inv$(s_i)$. Eventually the secure version of Inv-Layer outputs a $(d+1)$-sharing $(\mathbf{r}^0, \cdots, \mathbf{r}^d)$ of Inv-Layer$(\mathbf{s})$. For $d = 1$, the secure inversion algorithm Sec-Inv$(d, \cdot)$ can be chosen among the numerous ones proposed in the literature [1, 5, 7, 10, 17, 18, 22]. For $d > 1$, the choice is much more reduced and the secure inversion algorithm must be one of those proposed in [22, 23, 24].

In the next section we introduce an alternative to the classical approach which starts from a trick introduced by Montgomery[13].

## 3.2 Approach with Montgomery's Trick

The principle of Montgomery's trick is to reduce the total number of field inversions by using field multiplications. Let us consider $n$ field elements $s_i$. With Montgomery's trick the $n$ inverses $(s_i^{-1})_{0 \le i \le n-1}$ are computed by performing two separate passes through the data. In the forward pass, a variable $\text{Prod}_0$ is initialized with $s_0$ and then the following product is computed for $i = 1, \ldots, n-1$:

$$\text{Prod}_i = \text{Prod}_{i-1} \otimes s_i \ .$$

The last product $\text{Prod}_{n-1}$ satisfies $\text{Prod}_{n-1} = \prod_{i=0}^{n-1} \text{Prod}_i$. Then a single field inversion

$$I = (\text{Prod}_{n-1})^{-1}$$

is computed. Next, in the backward pass, $t_{n-1}$ is initialized by $I$ and then for $i = n-1, \ldots, 1$, the two following products are computed

$$s_i^{-1} = t_i \otimes \text{Prod}_{i-1} \qquad \text{and} \qquad t_{i-1} = t_i \otimes s_i \ .$$

To finish $s_0^{-1}$ is set to $t_0$. In total the algorithm requires a single field inverse, and $3(n-1)$ field multiplications.

Montgomery's trick has been applied in many contexts [6, 12, 14]. This paper investigates its application to improve the secure AES Inv-Layer computation. Clearly this application cannot be done directly and we have to deal with two

main issues. The first issue is that the inverse $I$ cannot be computed directly from the product of the AES state elements since the latter ones may equal zero. The second issue is that the application of the trick must be secure against SCA of any order $d$. We propose hereafter a modification of Montgomery's trick that circumvents the first issue. Then, in Sect. 3.3, we explain how it can be efficiently secured at any order.

To deal with the first issue we propose to first transform the elements of the state in such way that their image is always non-zero and to keep track of this transformation. More precisely, prior to the forward pass, we add each state element $s_i$ with its Dirac value $\delta_0(s_i)$ defined by $\delta_0(s_i) = 1$ if $s_i = 0$ and $\delta_0(s_i) = 0$ otherwise. The computation of the products $(\mathrm{Prod}_i)_{0 \leq i \leq 15}$ is let unchanged except that it applies on $(s_i \oplus \delta_0(s_i))_{i \leq 15}$ instead of $(s_i)_{i \leq 15}$. Eventually the potential modification is corrected by adding $\delta_0(s_i)$ to $(s_i \oplus \delta_0(s_i))^{-1}_{i \leq 15}$ after having computed $(t_i)_{i \leq 15}$. The completeness of this treatment holds from $(s_i \oplus \delta_0(s_i))^{-1} = s_i^{-1} \oplus \delta_0(s_i)$. The sequence of those different steps is presented in Alg. 1.

**Algorithm 1.** Montgomery's Trick Applied on AES State Elements

INPUT(S): The AES state $\mathbf{s} = (s_i)_{i \leq 15}$ in $\mathrm{GF}(2^8)^{16}$
OUTPUT(S): $(s_i^{-1})_{i \leq 15} = \texttt{Inv-Layer}(\mathbf{s})$

$**$ *Mapping of the state elements from* $\mathrm{GF}(2^8)$ *to* $\mathrm{GF}(2^8)^\star$.
1. **for** $i = 0$ **to** 15 **do**
   $\delta_0(s_i) \leftarrow \texttt{Dirac}(s_i)$
   $s_i \leftarrow s_i \oplus \delta_0(s_i)$

$**$ *Computation of intermediate products used for the inverses extraction.*
2. $\mathrm{Prod}_0 \leftarrow s_0$
3. **for** $i = 1$ **to** 15 **do**
   $\mathrm{Prod}_i \leftarrow \mathrm{Prod}_{i-1} \otimes s_i$

$**$ *Computation of the single inverse.*
4. $I \leftarrow (\mathrm{Prod}_{15})^{-1}$

$**$ *Extraction of* $s_i^{-1}$ *for every* $i \leq 15$.
5. **from** $i = 15$ **down to** 1 **do**
   $s_i^{-1} \leftarrow I \otimes \mathrm{Prod}_{i-1}$
   $I \leftarrow I \otimes s_i$
6. $s_0^{-1} \leftarrow I$

$**$ *Mapping of the state elements from* $\mathrm{GF}(2^8)^\star$ *to* $\mathrm{GF}(2^8)$.
7. **for** $i = 0$ **to** 15 **do**
   $s_i^{-1} \leftarrow s_i^{-1} \oplus \delta_0(s_i)$

8. **return** $(s_0^{-1}, \ldots, s_{15}^{-1})$

The operation $\texttt{Dirac}(s_i)$ computes the Dirac value of $s_i$. Algorithm 1 could be optimized by doing the calls to $\texttt{Dirac}(\cdot)$ inside the loops in Steps 3, but for a better comprehension of our proposal we described intentionally the different steps separately.

### 3.3 Secure Computation

The application of Alg. 1 in a context where all the sensitive data are represented by a $(d+1)$-sharing requires two modifications. First any intermediate result (including the input and output) must be replaced by a $(d+1)$-sharing representing it. Additionally, operations $\texttt{Dirac}(\cdot)$ and $\otimes$ shall be replaced by secure versions of them, called $\texttt{Secure-Dirac}(d,\cdot)$ and $\texttt{Secure-MUL}(d,\cdot,\cdot)$ and satisfying the following properties:

- $\texttt{Secure-Dirac}(d,\cdot)$ must output a $(d+1)$-sharing $(\Delta^0,\cdots,\Delta^d)$ of $\Delta = (\delta_0(s_0),\cdots,\delta_0(s_{15}))$ from the $(d+1)$-sharing $(\mathbf{s}^0,\cdots,\mathbf{s}^d)$ of $\mathbf{s}$. The processing must moreover be $d^{\text{th}}$-order secure.
- $\texttt{Secure-MUL}(d,\cdot,\cdot)$ must output the $(d+1)$-sharing $(p^0,\ldots,p^d)$ of $p = s_i \otimes s_j$ from the $(d+1)$-sharing $(s_i^0,\cdots,s_i^d)$ and $(s_j^0,\cdots,s_j^d)$ of $s_i$ and $s_j$.

We shall moreover also need a function $\texttt{Add}-\texttt{Dirac}(\cdot,\cdot)$ that applies on the $(d+1)$-sharing of $\Delta$ and $\mathbf{s}$ and simply replaces the first column of each matrix share $\mathbf{s}^i$ by the bitwise addition of this column with the binary column vector $\Delta^i$. Its cost $\mathsf{C}_{\text{A-D}}$ in terms of logical operations is therefore $16(d+1)\times \mathtt{c}_{\oplus}$, where $\mathtt{c}_{\oplus}$ denotes the cost of a bitwise addition over GF($2^8$).

We sum-up hereafter the main steps of our new proposal to implement the AES $\texttt{Inv-Layer}$ in a $d^{\text{th}}$-order SCA-secure way.

*Completeness.* Step 1 computes the $(d+1)$-sharing of the Dirac of each state element $s_i$, the shares of same index being grouped to form the 16-bit vectors $\Delta^0$, ..., $\Delta^d$. It is viewed as a $(16\times(d+1))$-binary matrix whose bit-coordinates are denoted by $\Delta^j[i]$. The second step transforms the $(d+1)$-sharing $(s_i^0,\cdots,s_i^d)$ of each state element $s_i$ into a new one $(s_i^0\oplus\Delta^0[i],\cdots,s_i^d\oplus\Delta^d[i])$. Since the sum $\bigoplus_j \Delta^j[i]$ (resp. $\bigoplus_j s_i^j$) equals $\delta_0(s_i)$ (resp. $s_i$), this step outputs a $(d+1)$-sharing of $s_i\oplus\delta_0(s_i)$. Steps 3 to 7 simply implement the SCA-secure Montgomery's Trick, where each elementary operation is performed thanks to a $d^{\text{th}}$-order secure algorithm. Eventually, the $8^{\text{th}}$ step reverses the mapping (if it has occurred) of a state element $r_i = 0$ into 1. Namely, it processes in a secure way the $(d+1)$-sharing of $(r_i+\delta_0(s_i))^{-1}\oplus\delta_0(s_i)$ which equals $r_i^{-1}$, since $1^{-1}=1$ and $0^{-1}$ equals 0 by assumption.

Algorithm 2 involves four procedures: $\texttt{Sec-Inv}(d,\cdot)$, $\texttt{Add-Dirac}(\cdot,\cdot)$, $\texttt{Secure-Dirac}(d,\cdot)$ and $\texttt{Secure-MUL}(d,\cdot,\cdot)$. The different ways how to choose the function $\texttt{Sec-Inv}(d,\cdot)$ have been presented in Sect. 3.1. Additionally, we have shown in

**Algorithm 2.** Secure `Inv-Layer` with Montgomery's Trick

INPUT(S): The AES state $\mathbf{s}$ split into $d+1$ shares $(\mathbf{s}^0, \cdots, \mathbf{s}^d)$
OUTPUT(S): A new $(d+1)$-sharing $(\mathbf{r}^0, \cdots, \mathbf{r}^d)$ of the AES state $\mathbf{r}$ such that $\mathbf{r} = \texttt{Inv-Layer}(\mathbf{s})$

$**$ *Mapping of the state elements from* $\mathrm{GF}(2^8)$ *to* $\mathrm{GF}(2^8)^\star$.
1. $(\Delta^0, \cdots, \Delta^d) \leftarrow \texttt{Secure-Dirac}(d, (\mathbf{s}^0, \cdots, \mathbf{s}^d))$
2. $(\mathbf{s}^0, \cdots, \mathbf{s}^d) \leftarrow \texttt{Add-Dirac}((\mathbf{s}^0, \cdots, \mathbf{s}^d), (\Delta^0, \cdots, \Delta^d))$

$**$ *Computation of the* $(d+1)$*-sharing* $(\mathrm{Prod}_i^0, \cdots, \mathrm{Prod}_i^d)$ *of the intermediate products* $\mathrm{Prod}_i$ *used for the inverses extraction.*
3. $(\mathrm{Prod}_0^0, \cdots, \mathrm{Prod}_0^d) \leftarrow (s_0^0, \cdots, s_0^d)$
4. **for** $i = 1$ **to** 15 **do**
   $(\mathrm{Prod}_i^0, \cdots, \mathrm{Prod}_i^d) \leftarrow \texttt{Secure-MUL}(d, (\mathrm{Prod}_{i-1}^0, \cdots, \mathrm{Prod}_{i-1}^d), (s_i^0, \cdots, s_i^d))$

$**$ *Secure computation of the* $(d+1)$*-sharing of* $(\mathrm{Prod}_{15})^{-1}$ *from its sharing* $(\mathrm{Prod}_{15}^0, \cdots, \mathrm{Prod}_{15}^d)$.
5. $(\mathrm{Inv}^0, \cdots, \mathrm{Inv}^d) \leftarrow \texttt{Sec-Inv}(d, (\mathrm{Prod}_{15}^0, \cdots, \mathrm{Prod}_{15}^d))$

$**$ *Extraction of the* $(d+1)$*-sharing* $(r_i^0, \cdots, r_i^d)$*of* $s_i^{-1}$*for every* $i \leq 15$.
6. **from** $i = 15$ **down to** 1 **do**
   $(r_i^0, \cdots, r_i^d) \leftarrow \texttt{Secure-MUL}(d, (\mathrm{Prod}_{i-1}^0, \cdots, \mathrm{Prod}_{i-1}^d), (\mathrm{Inv}^0, \cdots, \mathrm{Inv}^d))$
   $(\mathrm{Inv}^0, \cdots, \mathrm{Inv}^d) \leftarrow \texttt{Secure-MUL}(d, (\mathrm{Inv}^0, \cdots, \mathrm{Inv}^d), (s_i^0, \cdots, s_i^d))$
7. $(r_0^0, \cdots, r_0^d) \leftarrow (\mathrm{Inv}^0, \cdots, \mathrm{Inv}^d)$

$**$ *Mapping of the state elements from* $\mathrm{GF}(2^8)^\star$ *to* $\mathrm{GF}(2^8)$.
8. $(\mathbf{r}^0, \cdots, \mathbf{r}^d) \leftarrow \texttt{Add-Dirac}((\mathbf{r}^0, \cdots, \mathbf{r}^d), (\Delta^0, \cdots, \Delta^d))$

9. **return** $(\mathbf{r}^0, \cdots, \mathbf{r}^d)$

this section how to simply process $\texttt{Add-Dirac}(\cdot, \cdot)$. For our presentation to be consistent, procedures $\texttt{Secure-MUL}(d, \cdot, \cdot)$ and $\texttt{Secure-Dirac}(d, \cdot)$ still need to be described and they are actually the most tricky parts of our proposal. The purpose of the following section is to present them. Eventually, the analysis of the complexity and security of the overall proposal (Alg. 2) is done in Sect. 4.3.

# 4 Secure and Efficient Implementations of the Primitives

## 4.1 Field and Logical Multiplications Secure at Any Order

Let $\ell$ be a positive integer and let $a$ and $b$ be two elements of the field $\mathrm{GF}(2^\ell)$ with multiplication law $\otimes$. We denote by $p$ the product $a \otimes b$. In Sect. 3, we have promoted the need for a secure multiplication $\texttt{Secure-MUL}(d, \cdot, \cdot)$ that securely constructs a $(d+1)$-sharing $(p^0, \cdots, p^d)$ of $p$ from the $(d+1)$-sharings $(a^0, \cdots, a^d)$ and $(b^0, \cdots, b^d)$ of $a$ and $b$ respectively. An algorithm to process such a secure multiplication has been proposed in [23] as an extension of Ishaï *et al.* 's work [7]. The main steps of this algorithm are recalled hereafter.

**Algorithm 3.** `Secure-MUL`$(d, \cdot, \cdot)$

INPUT(S): A masking order $d$ and two $(d+1)$-sharings $(a^0, \cdots, a^d)$ and $(b^0, \cdots, b^d)$ of $a$ and $b$ respectively.
OUTPUT(S): A $(d+1)$-sharing $(p^0, \cdots, p^d)$ such that $p = a \otimes b$.

1. Compute the $((d+1) \times (d+1))$-matrix $\mathbf{M} = (a^0, \cdots, a^d)^\intercal \times (b^0, \cdots, b^d)$, where $\times$ denotes the matrix product and where the matrix coordinates are multiplied with the law $\otimes$.
2. Split $\mathbf{M}$ into an upper triangular matrix $\mathbf{M}_1$ and a strictly lower triangular matrix $\mathbf{M}_2$ such that $\mathbf{M} = \mathbf{M}_1 \oplus \mathbf{M}_2$.
3. Generate a strictly upper triangular random matrix $\mathbf{R}_1 = (r_{ij})_{i,j}$ (*i.e.* $j \leq i$ implies $r_{ij} = 0$).
4. Compute $\mathbf{U} = \mathbf{M}_1 \oplus \mathbf{R} \oplus \mathbf{M}_2^\intercal$ from left to right, where $\mathbf{R}$ denotes $\mathbf{R}_1 \oplus \mathbf{R}_1^\intercal$.
5. Return $(p^0, \cdots, p^d) = \mathbf{1} \times \mathbf{U}$, where $\mathbf{1}$ denotes the line vector whose $d+1$ coordinates are all equal to 1.

In the three following paragraphs we discuss the completeness, the security and the complexity of Alg. 3.

*Completeness.* By construction, the sum $p$ of the output shares $p^i$ satisfies $\bigoplus_{i \leq d} p^i = \mathbf{1} \times \mathbf{U} \times \mathbf{1}^\intercal$. On the other hand, we have:

$$\begin{aligned}
\mathbf{1} \times \mathbf{U} \times \mathbf{1}^\intercal &= \mathbf{1} \times (\mathbf{M_1} \oplus \mathbf{R} \oplus \mathbf{M_2}^\intercal) \times \mathbf{1}^\intercal \ , \\
&= \mathbf{1} \times (\mathbf{M_1} \oplus \mathbf{R_1} \oplus \mathbf{M_2}^\intercal \oplus \mathbf{R_1}^\intercal) \times \mathbf{1}^\intercal \ , \\
&= \mathbf{1} \times (\mathbf{M_1} \oplus \mathbf{R_1} \oplus (\mathbf{M_2} \oplus \mathbf{R_1})^\intercal) \times \mathbf{1}^\intercal \ , \\
&= \mathbf{1} \times (\mathbf{M_1} \oplus \mathbf{R_1}) \times \mathbf{1}^\intercal \oplus \mathbf{1} \times (\mathbf{M_2} \oplus \mathbf{R_1}) \times \mathbf{1}^\intercal \ , \\
&= \mathbf{1} \times (\mathbf{M_1}) \times \mathbf{1}^\intercal \oplus \mathbf{1} \times (\mathbf{M_2}) \times \mathbf{1}^\intercal \oplus \mathbf{1} \times (\mathbf{R_1}) \times \mathbf{1}^\intercal \oplus \mathbf{1} \times (\mathbf{R_1}) \times \mathbf{1}^\intercal, \\
&= \mathbf{1} \times \mathbf{M} \times \mathbf{1}^\intercal = \mathbf{1} \times (a^0, \cdots, a^d)^\intercal \times (b^0, \cdots, b^d) \times \mathbf{1}^\intercal \ .
\end{aligned}$$

Since $(a^0, \cdots, a^d)$ and $(b^0, \cdots, b^d)$ are respectively a $(d+1)$-sharing of $a$ and $b$, we have $a = \mathbf{1} \times (a^0, \cdots, a^d)^\intercal$ and $b = \mathbf{1} \times (b^0, \cdots, b^d)^\intercal$. We thus deduce that $\mathbf{1} \times \mathbf{U} \times \mathbf{1}^\intercal$ equals $p = a \otimes b$ which states the completeness of Alg. **??**.

*Security.* The security of `Secure-MUL`$(d, \cdot, \cdot)$ against $d^{\text{th}}$-order SCA has been proved in [23].

*Complexity.* Let us denote by $\mathsf{c}_\otimes$ (resp. $\mathsf{c}_\oplus$) the cost of a field multiplication $\otimes$ (resp. bitwise addition $\oplus$) in terms of logical operations. In [23], it is argued that `Secure-MUL`$(d, \cdot, \cdot)$ algorithm can be processed with $(d+1)^2$ field multiplications $\otimes$ and $2d(d+1)$ bitwise additions $\oplus$. Its cost, denoted by $\mathsf{C}_{\text{S-M}}$, therefore satisfies:

$$\mathsf{C}_{\text{S-M}} = (d+1)^2 \times \mathsf{c}_\otimes + 2d(d+1) \times \mathsf{c}_\oplus \ . \tag{1}$$

It moreover requires the generation of $d(d+1)/2$ random bytes. As an illustration, securing a field multiplication $\otimes$ over GF($2^8$) thanks to Secure-MUL$(d,\cdot,\cdot)$ requires $\mathsf{C}_{\text{S-M}} = 4 \times \mathsf{c}_{\otimes} + 4 \times \mathsf{c}_{\oplus}$ for $d = 1$, and $\mathsf{C}_{\text{S-M}} = 9 \times \mathsf{c}_{\otimes} + 12 \times \mathsf{c}_{\oplus}$ for $d = 2$, and $\mathsf{C}_{\text{S-M}} = 16 \times \mathsf{c}_{\otimes} + 24 \times \mathsf{c}_{\oplus}$ for $d = 3$.

In the following, we shall also need a slightly modified version of Secure-MUL$(d,\cdot,\cdot)$ called Secure-AND$(d,\ell,\cdot,\cdot)$ and enabling to securely process the bitwise multiplication $\odot^{\ell}$ of two $\ell$-bit vectors $a$ and $b$ (*i.e.* a bitwise AND). It applies exactly the same steps as Secure-MUL$(d,\cdot,\cdot)$ algorithm except that the operation $\otimes$ is replaced by $\odot^{\ell}$. It moreover obviously inherits its $d^{\text{th}}$-order SCA security and its complexity from that of Secure-MUL$(d,\cdot,\cdot)$. To be absolutely clear in our argument in the next sections, we shall denote by $\underset{\text{sec}}{\odot^{\ell}}$ the operation $\odot^{\ell}$ when it is processed by applying Secure-AND$(d,\ell,\cdot,\cdot)$.

### 4.2 Dirac Computation Secure at Any Order

In [5], a 1st-order secure implementation of the Dirac function is proposed. It involves a look-up table in RAM whose size (32 or 256 bytes) is chosen according to an expected timing/memory trade-off. This method has two drawbacks in our context. First, it consumes RAM whereas we are looking for a secure AES implementation that uses memory as little as possible. Secondly, the method is only resistant to 1st-order SCA and its extension to achieve higher-order security seems to be an issue. Indeed, it inherits from the same drawbacks w.r.t higher-order SCA than all the methods based on table re-computations techniques [4].

In order to define a Dirac implementation secure at any order, we chose to start from the description of this function in terms of logical instructions.

*Dirac Computation.* Let $\overline{x}$ denote the bitwise complement of a word (or a matrix) $x$ and let $\odot$ be the logical binary AND. The Dirac $\delta_0(s)$ of a $\ell$-bit vector $s = (s[0], \cdots, s[\ell-1])$ satisfies:

$$\delta_0(s) = (\overline{s}[0]) \odot (\overline{s}[1]) \odot \cdots \odot (\overline{s}[\ell-1]) \ . \tag{2}$$

The computation of the Dirac of $\ell$ elements $s_0$,..., $s_{\ell-1}$ in GF$(2)^{\ell}$ can be performed by using a bit-slicing approach (see *e.g.* [8])[2]. The elements are first represented as a $(\ell \times \ell)$-binary matrix $\mathbf{s}$ whose lines are the $s_i$. Denoting by $\mathbf{t}$ the transpose of $\mathbf{s}$, the line $t_j$ of $\mathbf{t}$ satisfies $t_j = (s_0[j], \cdots, s_{\ell-1}[j]))$.

The Dirac values of the $s_i$ are then computed by applying the operation $\odot^{\ell}$ on the bitwise complement of the $t_j$, leading to the following analogous of (2):

$$\Delta = (\delta_0(s_0), \cdots, \delta_0(s_{\ell-1})) = \overline{t}_0 \odot^{\ell} \overline{t}_1 \odot^{\ell} \cdots \odot^{\ell} \overline{t}_{\ell-1} \ . \tag{3}$$

The cost of the Dirac computation (3) per $\ell$-bit vector $s_i$ is around $(\ell-1)/\ell$ computation of $\odot^{\ell}$ plus 1 bitwise complement, to which we have to add the cost of a $(\ell \times \ell)$-matrix transposition over GF(2) (to get $\mathbf{t}$ from $\mathbf{s}$).

[2] To easy the description of the method we assume that there are $\ell$ elements $s_i$ of size $\ell$. This is needed to have a square matrix in the following. The generalization of the method for $n > \ell$ is given at the end of the section.

*Secure Dirac Computation.* In the context of a $d^{\text{th}}$-order masking scheme, (3) must be modified to no longer operate on $\mathbf{t}$ but on a $(d+1)$-sharing $(\mathbf{t}^0, \ldots, \mathbf{t}^d)$ of it (each share $\mathbf{t}^i$ being a binary $(\ell \times \ell)$-matrix). Moreover the computations of the operation $\odot^\ell$ must be secured thanks to the algorithm $\texttt{Secure-AND}(d, \ell, \cdot, \cdot)$ introduced in Sect. 4.1. By applying the latter algorithm to the lines of the $\mathbf{t}^i$, we can construct a $(d+1)$-sharing $(\Delta^0, \cdots, \Delta^d)$ of the $\ell$-bit vector $\Delta$ defined in (3). Actually, if we denote by $t^i_j$ the $j^{\text{th}}$ line of $\mathbf{t}^i$, the algorithm we present in this section aims at processing the following computation:

$$(\Delta^0, \cdots, \Delta^d) = (\overline{t}^0_0, \cdots, t^d_0) \underset{\text{sec}}{\odot^\ell} (\overline{t}^0_1, \cdots, t^d_1) \underset{\text{sec}}{\odot^\ell} \cdots \underset{\text{sec}}{\odot^\ell} (\overline{t}^0_{\ell-1}, \cdots, t^d_{\ell-1}) \ . \quad (4)$$

Comparing (3) and (4), we can observe that each $\overline{t}_i$ has been replaced by its $(d+1)$-sharing, and that the operation $\odot^\ell$ has been replaced by its secure version $\underset{\text{sec}}{\odot^\ell}$. We give hereafter a formal description of $\texttt{Secure-Dirac}(d, \cdot)$.

**Algorithm 4.** $\texttt{Secure-Dirac}(d, \cdot)$

INPUT(S): An order $d$, a length $\ell$ and a $(d+1)$-sharing $(\mathbf{s}^0, \cdots, \mathbf{s}^d)$ of a binary $(\ell \times \ell)$-matrix $\mathbf{s}$ whose lines are the $s_i$.
OUTPUT(S): A $(d+1)$-sharing $(\Delta^0, \cdots, \Delta^d)$ of the $\ell$-bit vector $\Delta = (\delta_0(s_0), \cdots, \delta_0(s_{\ell-1}))$

** *Compute the bitwise complement* $\overline{\mathbf{s}}^0$ *of the* $(\ell \times \ell)$*-matrix* $\mathbf{s}^0$.
1. $\mathbf{s}^0 \longleftarrow \overline{\mathbf{s}}^0$.

** *Transpose the* $(\ell \times \ell)$ *matrices* $\mathbf{s}^i$ *for every* $i \leq d$.
2. **for** $i = 0$ **to** $d$
   **do** $\mathbf{t}^i \longleftarrow (\mathbf{s}^i)^{\mathsf{T}}$.

** *Process the Dirac computations.*
3. $(\Delta^0, \cdots, \Delta^d) \longleftarrow (t^0_0, \cdots, t^d_0)$
4. **for** $i = 1$ **to** $\ell - 1$
   **do** $(\Delta^0, \cdots, \Delta^d) \longleftarrow \texttt{Secure-AND}(d, \ell, (\Delta^0, \cdots, \Delta^d), (t^0_i, \cdots, t^d_i))$

5. **return** $(\Delta^0, \cdots, \Delta^d)$

The $i^{\text{th}}$ call to $\texttt{Secure-AND}(d, \ell, \cdot, \cdot)$ outputs $\overline{t}_0 \odot^\ell \overline{t}_1 \odot^\ell \cdots \odot^\ell \overline{t}_{i-1}$, the operation being performed in a secure way from the $(d+1)$-sharings $(\Delta^0, \cdots, \Delta^d)$ and $(t^0_{i-1}, \cdots, t^d_{i-1})$ which represent $\Delta$ and $\overline{t}_{i-1}$ respectively.

*Security (Sketch of Proof).* The $d^{\text{th}}$-order security of $\texttt{Secure-AND}(d, \ell, \cdot, \cdot)$ implies that of each iteration of the loop. Moreover, the $d$ random values used to construct the $(d+1)$-sharing of the $\texttt{Secure-AND}(d, \ell, \cdot, \cdot)$ output are randomly regenerated at each call. We thus deduce that the local $d^{\text{th}}$-order security implies that of the overall algorithm.

*Complexity.* Let us denote by $\mathtt{c}_{\mathsf{T}}$ (resp. $\mathtt{c}_\odot$) the cost of a $(\ell \times \ell)$-matrix transposition (resp. bitwise multiplication $\odot$). The $d^{\text{th}}$-order secure processing of the Dirac

of $\ell$ elements in $\mathrm{GF}(2)^\ell$ costs $(d+1)\times \mathsf{c}_\intercal + (\ell-1)(d+1)^2 \times \mathsf{c}_\odot + 2(\ell-1)d(d+1)\times \mathsf{c}_\oplus$ (which corresponds to $d+1$ matrix transpositions and $\ell-1$ calls to `Secure-AND`$(d,\ell,\cdot,\cdot)$). We experimented that the cost $\mathsf{c}_\intercal$ is of around 150 logical operations on a $\ell$-bit architecture with bit-addressable memory (see Sect. 5)[3].

Let $n$ be a multiple of $\ell$. Algorithm 4 can be simply extended to compute the $(d+1)$-sharings of the Dirac's of $n$ elements $s_0$, ..., $s_{n-1}$ in $\mathrm{GF}(2^\ell)$. In this case, the matrix $\mathbf{s}$ is a binary $(n\times\ell)$-matrix and its $(d+1)$-sharing is also composed of binary $(n\times\ell)$-matrices. Thus, before applying Alg. 4 the elements of the $(d+1)$-sharing $(\mathbf{s}^0,\cdots,\mathbf{s}^d)$ of $\mathbf{s}$ are split into $n/\ell$ sub-matrices of $\ell$ lines and $\ell$ columns. This results in the definition of a splitting of $(\mathbf{s}^0,\cdots,\mathbf{s}^d)$ into $n/\ell$ sharings $({\mathbf{s}^0}_{j(\ell)},\cdots,{\mathbf{s}^d}_{j(\ell)})$, each corresponding to the sub-matrix $\mathbf{s}_{j(\ell)}$ composed of the $j^{\text{th}}$ block of $\ell$ lines of $\mathbf{s}$. Once this splitting has been done, Alg. 4 is applied to each $(d+1)$-sharing $({\mathbf{s}^0}_{j(\ell)},\cdots,{\mathbf{s}^d}_{j(\ell)})$ separately to output a $(d+1)$-sharing of the Dirac values corresponding to the state elements $s_{j\ell}$, ..., $s_{(j+1)\ell-1}$. The overall procedure is denoted by `Secure-Dirac`$(d,(\mathbf{s}^0,\cdots,\mathbf{s}^d))$ in the following. It inherits its $d^{\text{th}}$-order security from that of `Secure-AND`$(d,\ell,\cdot,\cdot)$ and its cost in terms of elementary operations, denoted by $\mathsf{C}_{\text{S-D}}$, is exactly $n/\ell$ times that of Alg. 4. For instance, in the case of the AES ($n=16$ and $\ell=8$) we have:

$$\mathsf{C}_{\text{S-D}} = 2(d+1)\times\mathsf{c}_\intercal + 14(d+1)^2\times\mathsf{c}_\odot + 28d(d+1)\times\mathsf{c}_\oplus \ . \tag{5}$$

### 4.3 Security and Complexity Analysis of the Proposal

Based on the analysis conducted in the two previous sections, we study hereafter the security and the complexity of our proposal presented in Alg. 2 to secure the AES `Inv-Layer`.

*Security (Sketch of Proof).* `Add-Dirac`$(\cdot,\cdot)$ is a linear function operating on two data masked with independent $d$-tuples of masks. It operates on each share independently. For those two reasons it is $d^{\text{th}}$-order secure. Except the memory allocations (Steps 3 and 7) which are obviously $d^{\text{th}}$-order secure since they always manipulate the shares separately, the other steps process operations (`Secure-Dirac`$(d,\cdot)$, `Sec-MUL`$(d,\cdot,\cdot)$ and `Sec-Inv`$(d,\cdot)$) that have been proved to be $d^{\text{th}}$-order secure either in previous works [23] or in the present paper (see Sect. 4.2). The fact that all operations in Alg. 2 are $d^{\text{th}}$-order SCA-secure straightforwardly implies that Alg. 2 is at least $1^{\text{st}}$-order SCA secure. Actually, we claim here that it is also $d^{\text{th}}$-order SCA-secure. The precise formalization of the $d^{\text{th}}$-order security of Alg. 2 can be done by following the outlines of the proof of [23, Theorem 2] and may possibly require some *mask-refreshing* procedure (such as involved in [23]) to change the $(d+1)$-sharing of an internal state into a new one.

*Complexity.* Algorithm 2 involves 2 calls to the function `Add-Dirac`$(\cdot,\cdot)$, $3\times(16-1)$ calls to `Secure-MUL`$(d,\cdot,\cdot)$, 1 call to `Sec-Dirac`$(d,\cdot)$ and 1 call to `Sec-Inv`$(d,\cdot)$.

[3] Note that we did not took into account the cost of the bitwise complement which is negligible compared to the other costs.

Its complexity $\mathsf{C}_{\text{S-L}}$ therefore satisfies:

$$\mathsf{C}_{\text{S-L}} = 2 \times \mathsf{C}_{\text{A-D}} + 45 \times \mathsf{C}_{\text{S-M}} + \mathsf{C}_{\text{S-D}} + \mathsf{C}_{\text{Inv}} \ .$$

From the complexity analysis conducted in previous sections we hence deduce that the cost $\mathsf{C}_{\text{S-L}}$ of our proposal in terms of elementary operations satisfies:

$$\mathsf{C}_{\text{S-L}} = (d+1)[2 \times \mathsf{c}_{\mathsf{T}} + (32+118d) \times \mathsf{c}_{\oplus} + 14(d+1) \times \mathsf{c}_{\odot} + 45(d+1) \times \mathsf{c}_{\otimes}] + \mathsf{C}_{\text{Inv}} \ .$$

The cost of a classical processing of `Inv-layer` is around 16 times the cost $\mathsf{C}_{\text{Inv}}$ of the secure processing of a field inversion. Hence, our method improves the classical approach if and only if $\mathsf{C}_{\text{Inv}}$ satisfies:

$$\mathsf{C}_{\text{Inv}} \geq \frac{(d+1)[2 \times \mathsf{c}_{\mathsf{T}} + (32+118d) \times \mathsf{c}_{\oplus} + 14(d+1) \times \mathsf{c}_{\odot} + 45(d+1) \times \mathsf{c}_{\otimes}]}{15} \ . \tag{6}$$

For our implementations reported in Sect. 5, we experimented $\mathsf{c}_{\oplus} = \mathsf{c}_{\odot} = 1$, $\mathsf{c}_{\mathsf{T}} = 148$ and $\mathsf{c}_{\otimes} = 22$. In this particular case, (6) becomes $\mathsf{C}_{\text{Inv}} \geq \frac{1122d^2+2454d+1332}{15}$. For $d = 1$, $d = 2$ and $d = 3$ the lower bound respectively equals 328, 717 and 1256.

## 5 Experimentations

The purpose of this section is twofold. First, we experimentally validate the relevance of the SCA-secure Montgomery's trick by improving many methods of literature. Secondly, we quantify in practice the efficiency gain provided by our proposal. Even if this section reports on AES implementation in mode 128, the main conclusions stay valid in all the other modus operandi. Our AES implementations involve the same code to process the round-key addition and the linear/affine steps. Actually, they only differ in the code part dedicated to the processing of `Inv-Layer`. To protect the linear/affine AES steps against (HO-) SCA, the masking scheme (*a.k.a* secret sharing scheme) presented in [11] for order 1 and extended in [23] to any order has been applied. To secure the AES `Inv-Layer`, we first implemented some $1^{\text{st}}$-order SCA-secure methods, then all the existing $2^{\text{nd}}$-order SCA-secure methods, and eventually the single $3^{\text{rd}}$-order SCA-secure method existing in the literature (see Sect. 1.1 for an argumentation of the choices). In what follows, we give more details about the methods we chose in each category.

*For $d = 1$,* we chose to only consider methods requiring a limited amount of `RAM` memory, which excludes the methods proposed in [11] and [19]. Indeed, as mentioned in the introduction, our purpose is to improve the timing efficiency of $1^{\text{st}}$-order SCA-secure implementations in contexts where a limited amount of `RAM` is available. Moreover, we experimented that usually our proposal does not improve $1^{\text{st}}$-order methods optimized by involving `RAM` look-up tables precomputed with part of (or all) the masking material (as *e.g.* in [11] and [19]). In this case indeed, $\mathsf{C}_{\text{Inv}}$ does not satisfy (6). Eventually, we chose to implement the methods in [17, 18, 20, 21, 23].

- In [17, 18], the field $\mathrm{GF}(2^8)$ is represented as an extension of $\mathrm{GF}(2^2)$. Thanks to linear isomorphisms, the AES s-box is evaluated with operations in $\mathrm{GF}(2^2)$ where the extended inversion is linear.
- In [20], the extended inversion over $\mathrm{GF}(2^8)$ is essentially performed by going down to $\mathrm{GF}(2^4)^2$ and by computing a Fourier transform on $\mathrm{GF}(2^4)$.
- In [21], the authors perform the extended inversion by going down to $(\mathrm{GF}(2^4))^2$ and by bitwisely adding 15 elements of a `ROM` look-up table representing a permutation over $\mathrm{GF}(2^4)$.
- In [23], the extended inversion is represented as the power function $x \mapsto x^{254}$ and the evaluation of this function is essentially secured against SCA by decomposing the exponentiation into a minimum number of multiplications which are not squaring and by securing those multiplications. The latter step is done by calling the function `Secure-MUL`$(1, \cdot, \cdot)$ recalled in Sect. 4.

*For $d = 2$,* only a few methods exist that are perfectly SCA-secure. Actually, only the works [24], [22] and [23] propose such kind of schemes (the two first ones working for any s-box and the third one being dedicated to the AES s-box). The method in [24] can be viewed as a generalization of the re-computation table method proposed in [10]. Each time a s-box must be evaluated, a new pair of input/output masks is generated and two new look-up tables in `RAM` are generated from both those masks and a `ROM` look-up table representing the AES s-box. The method [22] is a generalization of [21]. Eventually, the method [23] applied for $d = 2$ protects the evaluation of the power function $x \mapsto x^{254}$ by securing the linear steps in a straightforward way (by applying the computations on each share separately) and by securing the multiplications thanks to `Secure-MUL`$(2, \cdot, \cdot)$.

*For $d = 3$,* only [23] proposes a solution. It involves `Secure-MUL`$(3, \cdot, \cdot)$ to secure the non-linear steps of the exponentiation $x \mapsto x^{254}$.

Table 1 lists the timing/memory performances of the different implementations. Memory performances correspond to the number of bytes allocations and cycles numbers correspond to multiple of $10^3$. The right-hand column gives the performance gain achieved by applying the SCA-secure Montgomery's trick (*e.g.* a gain of 60% signifies that the new timing equals 40% of the timings of the original code). Codes have been written in assembly language for a 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 always were lower than 5 K-bytes which is acceptable in almost all current embedded devices (for comparison a software secure implementation of RSA usually uses more than 10 K-bytes). Eventually, 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

**Table 1.** Comparison of AES implementations

| | Method to secure the s-box | Without trick | | With trick | | Timing Gain |
|---|---|---|---|---|---|---|
| | | Cycles | RAM | Cycles | RAM | |
| | Unprotected Implementation | | | | | |
| 1. | No Masking | 2 | 0 | Na. | Na. | Na. |
| | First-Order SCA | | | | | |
| 2. | Tower Field in GF($2^4$) [17, 18] | 77 | 0 | 55 | 56 | 29% |
| 3. | Masking *on-the-fly* [21] | 82 | 0 | 55 | 56 | 33% |
| 4. | Fourier Transform [20] | 122 | 0 | 58 | 56 | 52% |
| 5. | Secure Exponentiation [23] | 73 | 24 | 58 | $24 + 32$ | 21% |
| | Second-Order SCA | | | | | |
| 6. | Double Recomputations [24] | 594 | 512 | 190 | $512 + 96$ | 68% |
| 7. | Single Recomputation [22] | 672 | 256 | 195 | $256 + 96$ | 70% |
| 8. | Secure Exponentiation [23] | 189 | 48 | 165 | $48 + 48$ | 13% |
| | Third-Order SCA | | | | | |
| 9. | Secure Exponentiation [23] | 326 | 72 | 292 | $72 + 64$ | 9% |

only first-order SCA are considered (*e.g.* the same input/output mask can be used for all the s-box evaluations).

As it can be seen in the last column of Table 1, SCA-secure Montgomery's trick always improves the timing efficiency of the method on which it is applied to. At every order, this gain has been obtained at the cost of a small RAM overhead: $24d$ bytes to implement Secure-MUL$(d, \cdot, \cdot)$ and $14(d+1) + 2(d+1)$ bytes dedicated to Montgomery's trick. For $d = 1$, this overhead is acceptable, even in a very constrained context (we indeed still have a consumption lower than 100 bytes). For $d > 1$, the RAM overhead is either negligible for methods which already consumed a lot of RAM [22, 24] or acceptable for [23] since the total amount of RAM allocation (96 bytes) is not prohibitive in view of the security level ($d = 2, 3$).

For $d = 1$, it can be observed that the timing performances of the methods become very close when the SCA-secure Montgomery's trick is applied. In view of (6), this result was expected since the performances of the inversion method has a small impact on performances of the global algorithm when the trick is involved. Indeed, in this case only 10 secure inversions for the overall AES-128 calculation are performed instead of 160. So, when the trick is involved the timings performances essentially correspond to the cost of 10 applications of Alg. 2 and the cost of Step 5 (the secure inversion) is negligible. For $d = 2$, this remark is less pertinent. This is a consequence of the huge difference between the timings of the secure inversion methods proposed in [22, 24] and in [23] (the latter one being at least 2.2 times faster). In this case, the impact of the method used to protect the inversion (Step 5 in Alg. 2) is still measurable. For $d = 3$, the SCA-secure Montgomery's trick continues to improve the efficiency of the inner method but its impact is less significative than for $d = 1, 2$. Actually, the method used in [23] to secure the inversion involves 4 calls to Secure-MUL$(d, \cdot, \cdot)$ and when $d$ grows the timing efficiency of the method essentially corresponds

to the cost of those 4 calls. When applied, the SCA-secure Montgomery's trick merely replaces 4 calls to $\texttt{Secure-MUL}(d, \cdot, \cdot)$ by 3 calls to $\texttt{Secure-MUL}(d, \cdot, \cdot)$ plus a $d^{\text{th}}$-order secure Dirac computation. The gain in efficiency thus essentially relies on the difference of performances between one execution of $\texttt{Secure-MUL}(d, \cdot, \cdot)$ and the cost per byte of the $d^{\text{th}}$-order secure Dirac computation described in Alg. 4.

## 6 Conclusion

In this paper, we have proposed a different approach for the masking of the non-linear layer of the AES. Instead of sequentially computing the image of masked data through each s-box, we have proposed to evaluate them globally. Our approach is based on Montgomery's trick combined with the use of masked Dirac functions. Our solution allows us to improve significantly in time complexity all previous masking methods requiring a small amount of `RAM` at the cost of a little memory overhead.

## References

1. Blömer, J., Merchan, J.G., Krummel, V.: Provably Secure Masking of AES. In: Matsui, M., Zuccherato, R. (eds.) SAC 2004. LNCS, vol. 3357, pp. 69–83. Springer, Heidelberg (2004)
2. Brier, É., Olivier, F., Clavier, C.: Correlation Power Analysis with a Leakage Model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
3. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Wiener, M.J. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)
4. Coron, J.-S., Prouff, E., Rivain, M.: Side Channel Cryptanalysis of a Higher Order Masking Scheme. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 28–44. Springer, Heidelberg (2007)
5. Genelle, L., Prouff, E., Quisquater, M.: Secure Multiplicative Masking of Power Functions. In: Zhou, J., Yung, M. (eds.) ACNS 2010. LNCS, vol. 6123, pp. 200–217. Springer, Heidelberg (2010)
6. Harris, D. G.: Simultaneous field divisions: an extension of montgomery's trick. Cryptology ePrint Archive, Report 2008/199 (2008), `http://eprint.iacr.org/`
7. Ishai, Y., Sahai, A., Wagner, D.: Private Circuits: Securing Hardware against Probing Attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Heidelberg (2003)
8. Matsui, M., Fukuda, S.: How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors. In: Handschuh, H., Gilbert, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 398–412. Springer, Heidelberg (2005)
9. Messerges, T.S.: Power Analysis Attacks and Countermeasures for Cryptographic Algorithms. PhD thesis, University of Illinois (2000)

10. Messerges, T.S.: Securing the AES Finalists against Power Analysis Attacks. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 150–164. Springer, Heidelberg (2001)
11. Messerges, T.S.: Using Second-order Power Analysis to Attack DPA Resistant Software. In: Koç, Ç.K., Paar, C. (eds.) CHES 2000. LNCS, vol. 1965, pp. 238–251. Springer, Heidelberg (2000)
12. Mishra, P.K., Sarkar, P.: Application of Montgomery's Trick to Scalar Multiplication for Elliptic and Hyperelliptic Curves Using a Fixed Base Point. In: Bao, F., Deng, R.H., Zhou, J. (eds.) PKC 2004. LNCS, vol. 2947, pp. 41–54. Springer, Heidelberg (2004)
13. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 54, 839–854 (1990)
14. Okeya, K., Kurumatani, H., Sakurai, K.: Elliptic Curves with the Montgomery-Form and Their Cryptographic Applications. In: Imai, H., Zheng, Y. (eds.) PKC 2000. LNCS, vol. 1751, pp. 238–257. Springer, Heidelberg (2000)
15. Oswald, E., Mangard, S.: Template Attacks on Masking—Resistance Is Futile. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 243–256. Springer, Heidelberg (2006)
16. Oswald, E., Mangard, S., Herbst, C., Tillich, S.: Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 192–207. Springer, Heidelberg (2006)
17. Oswald, E., Mangard, S., Pramstaller, N.: Secure and Efficient Masking of AES – A Mission Impossible? Cryptology ePrint Archive, Report 2004/134 (2004)
18. Oswald, E., Mangard, S., Pramstaller, N., Rijmen, V.: A Side-Channel Analysis Resistant Description of the AES S-box. In: Handschuh, H., Gilbert, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 413–423. Springer, Heidelberg (2005)
19. Oswald, E., Schramm, K.: An Efficient Masking Scheme for AES Software Implementations. In: Song, J., Kwon, T., Yung, M. (eds.) WISA 2005. LNCS, vol. 3786, pp. 292–305. Springer, Heidelberg (2006)
20. Prouff, E., Giraud, C., Aumônier, S.: Provably Secure S-Box Implementation Based on Fourier Transform. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 216–230. Springer, Heidelberg (2006)
21. Prouff, E., Rivain, M.: A Generic Method for Secure SBox Implementation. In: Kim, S., Yung, M., Lee, H.-W. (eds.) WISA 2007. LNCS, vol. 4867, pp. 227–244. Springer, Heidelberg (2008)
22. Rivain, M., Dottax, E., Prouff, E.: Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis. In: Baignères, T., Vaudenay, S. (eds.) FSE 2008. LNCS, vol. 5086, pp. 127–143. Springer, Heidelberg (2008)
23. Rivain, M., Prouff, E.: Provably Secure Higher-Order Masking of AES. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg (2010)
24. Schramm, K., Paar, C.: Higher Order Masking of the AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 208–225. Springer, Heidelberg (2006)
25. Tillich, S., Herbst, C.: Attacking State-of-the-Art Software Countermeasures—A Case Study for AES. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 228–243. Springer, Heidelberg (2008)