Secure Multiple SBoxes Implementation with Arithmetically Masked Input

Luk Bettale

Oberthur Technologies
71-73 rue des Hautes Pâtures
92726 Nanterre Cedex - France
l.bettale@oberthur.com

Abstract The building blocks of several block ciphers involve arithmetic operations, bitwise operations and non-linear functions given as SBoxes. In the context of implementations secure against Side Channel Analysis, these operations shall not leak information on secret data. To this end, masking is a widely used protection technique. Propagating the masks through non-linear functions is a necessary task to achieve a sound and secure masked implementation. This paper describes an efficient method to securely access N SBoxes when the N inputs are encoded as a single word arithmetically masked. This problematic arises for instance in a secure implementation of the standard block ciphers GOST or SEED. A method using state of the art algorithms would be to first perform an arithmetic to boolean mask conversion before independently accessing the N SBoxes. Compared to this method, the algorithm proposed in this paper needs less code, less random generation and no extra memory. This makes our algorithm particularly suitable for very constrained devices. As a proof of concept, we compare an implementation in 8051 assembly language of our algorithm to the existing solutions.

Keywords: Side Channel Analysis, Differential Power Analysis, Block Cipher, SBox, Arithmetic Masking, Boolean Masking, Mask Conversion

1 Introduction

During the execution of a cryptographic algorithm, power consumption, execution timings, or electro-magnetic radiation may give information on secret data. The techniques using these leakages to attack cryptographic primitives are called Side Channel Analysis (SCA). Different techniques appeared in the literature. Among them, Differential Power Analysis (DPA) [13,5,15] turns out to be a powerful tool to attack implementations [3,1,18]. To recover a small part k of a secret, DPA consists in recording the power consumption – the leakage – of d moments where a sensitive variable (i.e. a value depending on k and the input) is involved, for a large number of different inputs. Then, an attacker makes predictions on a combination of these leakages for all possible hypotheses on k according to a well-chosen leakage model. Finally, the attacker outputs the hypothesis $\widehat{k}$ for

which the prediction and the actual leakages are the most “similar”. This degree of similarity is usually measured using statistical tools such as the Pearson correlation coefficient [5]. The number d of moments denotes the order of the DPA.

In parallel, countermeasures were soon developed to secure implementations against DPA. In this specific context, a countermeasure aims at preventing all sensitive data from leaking information, or more precisely making their leakages independent of the secret. A commonly used countermeasure is to *mask* – or split – a sensitive variable with one (resp. d) random value(s) [11,6]. This is called first order masking (resp. d-th order masking). With this countermeasure, as the random masks change at each execution, the leakage of the masked variable is statistically independent of the secret. Only the combined leakages of the masked variable and all the masks could reveal some information. Thus, a $d+1$-th order DPA is necessary to attack a d-th order masked implementation.

Sensitive data shall then remain masked through every step of a cryptographic algorithm. The critical task is to adapt the algorithm to keep this state, especially for non-linear parts. In block ciphers, the use of highly non-linear functions (w.r.t. bitwise addition) is mandatory to prevent classical cryptanalysis. One way to represent such functions is to use a so-called *SBox*. SBoxes have been used to design many symmetric encryption schemes such as the DES or the AES. Several masking schemes have been developed for these functions to prevent DPA of the first order [11,2,4,9] or higher order [22,21,20]. In general, for block ciphers built upon SBoxes and linear layers, a masking scheme based on *boolean masking* is chosen and an adapted secure SBox access [16,19] is implemented.

Some algorithms also use modular addition as a non-linear function. The block cipher IDEA [14] is an example. For this kind of algorithm, boolean masking is not always suitable. Indeed, a boolean mask propagates easily through linear functions, but with modular addition, *arithmetic masking* is preferred. To switch from one masking to another, a mask conversion is required. Several methods have been proposed [10,7,17,8], and an application to IDEA is suggested for instance in [17].

Finally, both modular addition and SBoxes may be used together as in the Korean standard block cipher SEED [23], or in the Russian standard block cipher GOST 28147-89 [24]. In these algorithms, multiple small SBoxes are used simultaneously to compute the image of a larger input. More precisely, arithmetic operations are performed on the large input before accessing these SBoxes. It is a natural requirement to protect such algorithms against DPA. This paper focuses on protecting this kind of block ciphers against first order DPA. More specifically, given N SBoxes (ℓ bits $\rightarrow \ell'$ bits), we need to securely access these SBoxes when the N inputs are encoded as a single word of $n = N\,\ell$ bits, arithmetically masked – addition modulo 2^n – with a random n-bit mask. Though, protecting an implementation usually comes with a cost, either in memory or in performances. Often, devices that are more likely to suffer Side Channel Analysis are also very constrained in terms of memory (smart-cards, embedded devices, . . .).

A method has been proposed in [12] and applied to the SEED algorithm. The method is highly efficient but uses a non-negligible amount of RAM. It is then interesting to have an alternative solution requiring lower memory resources.

Contributions. In this paper, we study the protection of block ciphers mixing arithmetic operations, boolean operations and multiple SBoxes against first order DPA. More specifically, we focus on the secure access to multiple SBoxes when arithmetic masking is used. We propose a novel algorithm to perform this operation. Compared to existing methods, our algorithm needs less memory, less code and less random generation.

Organization of the Paper. This paper is organized as follows. In Sect. 2, we give some notations and review existing algorithms to securely access one SBox. We also give some useful background on masking conversion. In Sect. 3, we present in detail the main concern of this paper, namely the problem of accessing simultaneously N SBoxes with an arithmetically masked input. In this section we also propose a first solution using state of the art techniques. In Sect. 4, we present our new algorithm as well as a careful security analysis. We compare an implementation of both solutions in Sect. 5. Section 6 concludes this article.

2 Background and Related Work

This paper deals with two kinds of masking:

- Boolean masking: a sensitive variable $a \in \mathbb{F}_2^n$ is masked with a random value $m \in \mathbb{F}_2^n$ by computing $\widetilde{a} = a \oplus m$, where $\oplus$ denotes bitwise exclusive or.
- Arithmetic masking: a sensitive variable $a \in \mathbb{F}_2^n$ is masked with a random value $m \in \mathbb{F}_2^n$ by computing $\widetilde{a} = a \boxplus m$, where $\boxplus$ denotes addition modulo 2^n.

When the masking operation is not precised, we may use operators $\star$ and $\diamond$ to denote either $\oplus$ or $\boxplus$. The unmasking operation for $\star$ is denoted by $\star^{-1}$. For instance if $\star$ is $\boxplus$, then $\star^{-1}$ is $\boxminus$. If $\star$ is $\oplus$, then $\star^{-1}$ is also $\oplus$.

Sometimes, a value $a \in \mathbb{F}_2^n$ is viewed as N chunks of ℓ bits where $n = N\,\ell$. The notation a_i, $0 \leqslant i < N$, is used to denote the i-th chunk where a_0 is the least significant chunk. Then, we use the notation $a = a_{N-1} \| \ldots \| a_0$. If a modular operation is performed on a small chunk, the operator is sub-scripted by the corresponding bit-size. For instance, we write $\boxplus_\ell$ to denote the addition modulo 2^ℓ. If the operation is done on n bits, the subscript is omitted to ease reading.

2.1 Substitution Boxes

A non-linear function $f : \mathbb{F}_2^n \to \mathbb{F}_2^p$ is often described as an SBox. An SBox is a table which gives for an n-bit index a the corresponding p-bit output $f(a)$. If S is

an SBox, we use the notation $S[a]$ to denote the image of a by the corresponding function f.

In a secure implementation context, it is necessary that a masked input $\widetilde{a}$ remains masked during the SBox access. Moreover, the output of the SBox shall also be masked, possibly with a different mask. The input masking is denoted by the operation $\star$ and the output masking is denoted by the operation $\diamond$.

In [16], Messerges proposes to recompute a table corresponding to a masked version of the SBox, that is, to compute an SBox T such that $T[a \star m] = S[a] \diamond m'$. This method has been chosen in [2] to secure the DES and AES SBoxes. The method is efficient but requires a large amount of memory ($p \cdot 2^n$ bits). In a context where memory is a limited resource, this method could be a bottleneck if n is too large. In [19], the authors proposed an alternative method, more suitable to such environment. The method does not require extra memory space. We describe it in Alg. 1.

Algorithm 1 Secure SBox implementation [19]

Inputs: $S : \mathbb{F}_2^n \to \mathbb{F}_2^p,\ \widetilde{a} = (a \star m) \in \mathbb{F}_2^n,\ m \in \mathbb{F}_2^n,\ m' \in \mathbb{F}_2^p,$
$\star : (\mathbb{F}_2^n \times \mathbb{F}_2^n) \to \mathbb{F}_2^n,\ \diamond : (\mathbb{F}_2^p \times \mathbb{F}_2^p) \to \mathbb{F}_2^p$

Output: $\widetilde{b} = S[a] \diamond m'$

1: **function** SecureSBox($S, \widetilde{a}, m, m', \star, \diamond$)
2:     **for** $k = 0$ **to** $2^n - 1$ **do**
3:         cmp $\leftarrow (k \stackrel{?}{=} m)$ ▷ if k and m are equal, cmp is 1, else 0
4:         $t \leftarrow \widetilde{a} \star^{-1} k$ ▷ unmask $\widetilde{a}$ with the loop index
5:         $R_{\text{cmp}} \leftarrow S[t] \diamond m'$
6:     **end for**
7:     **return** R_1
8: **end function**

Using Alg. 1, the real unmasking operation is dissimulated among $2^n - 1$ other dummy unmasking. Note that the masking type may be the same for both the input and the output. In the rest of the paper, this method is preferred to the table re-computation as we focus on implementation on memory-constrained devices.

Remark 1. Contrary to this paper, the method proposed in [12] is built upon the table re-computation algorithm of [16].

2.2 Masking Conversion

A sensitive variable has to remain masked throughout the cryptographic algorithm. A boolean (resp. arithmetic) masking propagates easily through boolean (resp. arithmetic) operations. When a specific algorithm mixes boolean operations (rotations, bitwise and/or, . . .) and arithmetic operations (modular addition/subtraction), *mask conversion* is needed. An efficient method to perform

boolean to arithmetic masking conversion (BMtoAM) has been proposed by Goubin in [10]. It uses a constant number of operations with respect to the bit-size of the data. The converse operation (AMtoBM) is more costly to achieve. A first method has also been proposed in [10] but it requires a number of operations linear in the bit-size. Later, Coron and Tchulkine introduced in [7] a more efficient method which pre-computes a conversion table for "small" input and output masks r and s (say λ bits over n), and processes the input by λ-bit chunks. It appears that the method fails for some inputs because of a problem in the carry propagation between the chunks [8]. In 2004, Neiße and Pulkus proposed a sound and efficient method to perform AMtoBM conversion in [17]. Their method is similar to [7], as it also uses a conversion table for a smaller λ-bit mask, but they handle the carries in a different way. A pre-computation step first generates two uniformly distributed λ-bit masks r and s. Then, two tables T and C are initialized. For every possible λ-bit inputs k ($0 \leqslant k < 2^\lambda$) the value $(k \boxminus_\lambda r) \oplus s$ is stored in $T[k]$. The carry resulting from $(k \boxminus_\lambda r)$ is stored in $C[k]$. The whole process is described in Alg. 2.

Algorithm 2 Secure AMtoBM implementation [17]: pre-computation

```
1: procedure AMtoBMPrecomp()
2:     r ← Random({0, ..., 2^λ})
3:     s ← Random({0, ..., 2^λ})
4:     for k = 0 to 2^λ − 1 do
5:         T[k] ← (k ⊟_λ r) ⊕ s                ▷ subtraction may generate a carry
6:         C[k] ← carry(step 5)
7:     end for
8: end procedure
```

To convert the masking without leaking information, the authors use the following useful property. For any $u, v \in \mathbb{F}_2^\lambda$, it holds that

$$\neg(u \boxplus_\lambda v) = \neg u \boxplus_\lambda \neg v \boxplus_\lambda 1 \ ,$$

where $\neg x$ denotes the bitwise complement of x. Let (-1) be the value $2^\lambda - 1$ (all λ bits are set to 1). Then, for any $z \in \{0, 1\}$, it holds that

$$(u \boxplus_\lambda v) \oplus (-z) = (u \oplus (-z)) \boxplus_\lambda (v \oplus (-z)) \boxplus_\lambda z \ . \tag{1}$$

If the bit z is chosen uniformly at random for each execution, it can be used to mask the propagating carries. Indeed, whenever $u \boxplus_\lambda v$ generates a carry, $\neg u \boxplus_\lambda \neg v \boxplus_\lambda 1$ does not, and conversely. The pre-computed tables T and C can then be used to convert and propagate the carry by λ-bit chunks. This is the purpose of the conversion algorithm from [17] described in Alg. 3.

Remark 2. The authors of [17] also propose several optimizations to their algorithm such as storing $C[k]$ in place of the least significant bit (LSB) of $T[k]$ after

Algorithm 3 Secure AMtoBM implementation [17]: conversion

Inputs: $\widetilde{a} = (a \boxplus m) \in \mathbb{F}_2^n$, $m \in \mathbb{F}_2^n$, $m' \in \mathbb{F}_2^n$
Output: $\widetilde{b} = a \oplus m'$
1: **function** AMtoBM($\widetilde{a}, m, m'$)
2: AMtoBMPrecomp( ) ▷ only if not already done
3: $z \leftarrow$ Random($\{0,1\}$)
4: $f \leftarrow \widetilde{a} \oplus (-z)$ ▷ complement (or not) the input
5: $g \leftarrow (m \oplus (-z)) \boxminus (r\|\dots\|r)$ ▷ change mask to $r\|\dots\|r$
6: $h \leftarrow ((s\|\dots\|s) \oplus (-z)) \oplus m'$ ▷ change mask from $s\|\dots\|s$
7: $c \leftarrow z$
8: **for** $i = 0$ **to** $N-1$ **do**
9: $f \leftarrow f \boxminus g_i \boxminus c$
10: $\widetilde{b}_i \leftarrow T[f_0] \oplus h_i$
11: $c \leftarrow C[f_0]$
12: $f \leftarrow f \gg \lambda$
13: **end for**
14: **return** $\widetilde{b} = \widetilde{b}_{N-1}\|\dots\|\widetilde{b}_0$
15: **end function**

remarking that if the LSB of r and s are the same, the LSB of k and $T[k]$ are also the same (for $0 \leqslant k < 2^\lambda$). Then this bit does not need to be stored when s is chosen accordingly, for instance when $s = r$.

Recently, Debraize proposed an alternative method in [8]. The method is similar to [17], but the carry is protected by computing two sets of tables $T^{(0)}, C^{(0)}$ and $T^{(1)}, C^{(1)}$. These tables can be described using table T and C of Alg. 2 by:

$$T^{(\rho)}[i] = T[i] \ , \qquad T^{(\rho\oplus 1)}[i] = T[i \boxminus_\lambda 1] \ ,$$
$$C^{(\rho)}[i] = C[i] \oplus \rho \ , \qquad C^{(\rho\oplus 1)}[i] = C[i \boxminus_\lambda 1] \oplus \rho \ ,$$

where ρ is a bit randomly chosen at each execution. Whether to access $T^{(0)}$ or $T^{(1)}$ is decided according to the value of the input carry masked by ρ. The output carry is masked again by ρ. This way, the input and output carries are always blinded. Compared to Alg. 3, the resulting algorithm would require twice the amount of memory. To suit our low-memory requirements, in the rest of this paper Alg. 3 is preferred.

3 Multiple SBoxes

As described in Sect. 1, in some algorithms, several "small" SBoxes are used simultaneously to compute a non-linear function on a "big" input. Let $\ell, N \in \mathbb{N}$ and let $n = N\,\ell$. Let $a \in \mathbb{F}_2^n$ be an SBox input. It can be written as $a = a_{N-1}\|\dots\|a_0$, where a_i are elements of $\mathbb{F}_2^\ell$, for $0 \leqslant i < N$. Consider N SBoxes $S_0, \dots, S_{N-1}$ which map ℓ bits to ℓ' bits ($\mathbb{F}_2^\ell \to \mathbb{F}_2^{\ell'}$). We define S to be the "SBox" mapping n bits to $N\,\ell'$ bits such that

$$S[a] = S_{N-1}[a_{N-1}]\|\dots\|S_0[a_0] \ .$$

For instance, the DES has $N = 8$ SBoxes with $\ell = 6$ and $\ell' = 4$. For the sake of clarity, in what follows, we assume that $\ell = \ell'$. Nevertheless, the results below can be extended to the case $\ell \neq \ell'$.

Boolean Masking. When the input a of multiple SBoxes is masked with boolean masking, the application of Alg. 1 to each small SBox is straightforward because each part m_i of a boolean mask m is a boolean mask for the corresponding part a_i of an input a:

$$\widetilde{a} = a \oplus m = a_{N-1} \oplus m_{N-1} \| \dots \| a_0 \oplus m_0 \ .$$

One may then securely compute $S[a]$ using table re-computation or using Alg. 1 by computing

$$\textsc{SecureSBox}\left(S_{N-1}, \widetilde{a}_{N-1}, m_{N-1}, m'_{N-1}, \oplus, \oplus\right) \| \dots \\ \dots \| \textsc{SecureSBox}\left(S_0, \widetilde{a}_0, m_0, m'_0, \oplus, \oplus\right) \ , \quad (2)$$

where the m'_i's are ℓ-bit parts of an output mask m'. This method is the one used in [2] to secure the DES SBoxes with table re-computation. However, an issue appears when the input is arithmetically masked.

3.1 Multiple SBoxes on Arithmetically Masked Input

In some algorithms, it can be necessary that the input of multiple SBoxes is masked with an arithmetic mask. In Fig. 1, we give as an example the round function of a typical block cipher. At round t, the current sub-key $\text{key}^{(t)}$ is added to the state $v^{(t)}$ to constitute the N ℓ-bit input a. The output b of the $N = 4$ SBoxes is then rotated to produce the next state $v^{(t+1)}$.

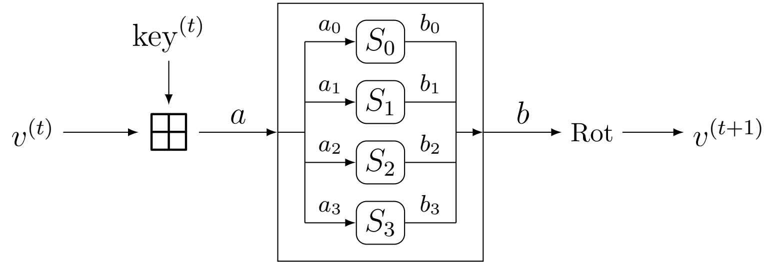

Figure 1. A block cipher round using an arithmetic operation and multiple SBoxes

This kind of round function is used for instance in SEED where $N = 4$ and $\ell = 8$, or in GOST 28147-89 where $N = 8$ and $\ell = 4$. In both cases, the input a and the output b are 32-bit words.

To protect such a round against DPA, $v^{(t)}$, $v^{(t+1)}$ as well as the intermediate values a and b should be masked. In this example, $v^{(t)}$ should be preferably

masked with modular addition to allow the mask to propagate through the operation $v^{(t)} \boxplus \text{key}^{(t)}$. As for b, it should be masked with a boolean mask which easily propagates through the rotation. In these conditions, a simple solution as in the boolean case cannot be achieved.

When the input a is arithmetically masked with a mask m, the following property holds:

$$\widetilde{a} = a \boxplus m = (a_{N-1} \boxplus_\ell m_{N-1} \boxplus_\ell c_{N-1}) \,\|\, \ldots \,\|\, (a_0 \boxplus_\ell m_0 \boxplus_\ell c_0) \quad , \qquad (3)$$

where $c_0 = 0$ and for all i, $1 \leqslant i < N$,

$$c_i = \begin{cases} 1 & \text{if} \quad (a_{i-1} + m_{i-1} + c_{i-1}) \geqslant 2^\ell \\ 0 & \text{otherwise} \end{cases} \quad .$$

The value c_i corresponds to the carry propagating through the modular addition $\boxplus$. Because of this carry, the direct application of (2) is not possible. We have then to address the following problem.

Problem 1. Given an arithmetically masked n-bit input $\widetilde{a} = a \boxplus m$, and N ℓ-bit SBoxes $S = (S_0, \ldots, S_{N-1})$ such that $n = N\ell$ as defined above, we want to securely compute $\widetilde{b} = b \diamond m'$ such that $b = S[a]$, where m and m' are uniformly distributed n-bit masks, and $\diamond$ is either $\boxplus$ or $\oplus$.

Assume that an algorithm called SecureMulSBox solves Prob. 1 with $\diamond = \oplus$, a secure implementation of the round function of Fig. 1 would be achieved in Fig. 2. Our goal is then to find an implementation of SecureMulSBox.

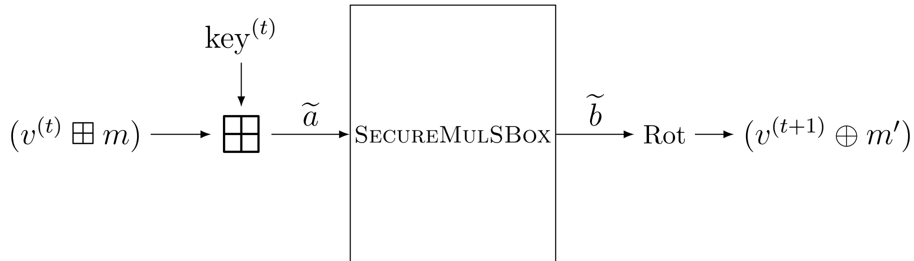

Figure 2. A masked block cipher round using an arithmetic operation and a secure implementation of multiple SBoxes

3.2 A Solution Using Mask Conversion

We have seen that accessing N SBoxes when boolean masking is used can be done independently. Then, an answer to Prob. 1 could be to first perform an

Algorithm 4 Secure multiple SBox with AMtoBM implementation

Inputs: $S = (S_0, \ldots, S_{N-1}) \in (\mathbb{F}_2^\ell \to \mathbb{F}_2^\ell)^N$, $\widetilde{a} = (a \boxplus m) \in \mathbb{F}_2^n$, $m \in \mathbb{F}_2^n$, $m' \in \mathbb{F}_2^n$
Output: $\widetilde{b} = S[a] \oplus m' = (S_{N-1}[a_{N-1}] \oplus m'_{N-1}) \| \ldots \| (S_0[a_0] \oplus m'_0)$

1: **function** SecureMulSBox($S, \widetilde{a}, m, m'$)
2: $\quad \widetilde{t} \leftarrow$ AMtoBM($\widetilde{a}, m, m$) ▷ keep m as output mask
3: $\quad$ **for** $i = 0$ **to** $N-1$ **do**
4: $\quad\quad \widetilde{b}_i \leftarrow$ SecureSBox($S_i, \widetilde{t}_i, m_i, m'_i, \oplus, \oplus$)
5: $\quad$ **end for**
6: $\quad$ **return** $\widetilde{b} = \widetilde{b}_{N-1} \| \ldots \| \widetilde{b}_0$
7: **end function**

AMtoBM conversion and then apply N times Alg. 1 on each small SBox. This process is described in Alg. 4.

The output of Alg. 4 is masked with boolean masking. To obtain an arithmetic masking instead, a BMtoAM conversion could be added at the end. As mentioned in Sect. 2.2, the conversion in this direction is not very costly. In particular, it does not require extra memory. Thus, in the rest of this paper, we focus on solving Prob. 1 with $\diamond = \oplus$.

Algorithm 4 needs to pre-compute 2^λ elements in a table (to use AMtoBM from [17]). A large value of λ ensures a faster conversion time, but to fit our low memory requirements, λ would have to be quite small. This implies a more expensive conversion. In the next section, we discuss a new solution which integrates the conversion within the secure SBoxes computation.

4 Secure Multiple SBoxes with Arithmetic Masking

4.1 Algorithm

As pointed out in Sect. 2.2, the propagation of the carry coming from the arithmetic masking is an issue. In the solution of Prob. 1 presented in Alg. 4, it is carried out by the masking conversion algorithm. If one could turn an arithmetic mask m on $n = N\ell$ bits into N arithmetic masks on ℓ bits for a given masked input, then, similarly to (2), one could compute

$$\begin{aligned}\text{SecureSBox}\left(S_{N-1}, \widetilde{a}_{N-1}, m_{N-1}, m'_{N-1}, \boxplus_\ell, \oplus\right) \| \ldots \\ \ldots \| \text{SecureSBox}\left(S_0, \widetilde{a}_0, m_0, m'_0, \boxplus_\ell, \oplus\right) \quad . \quad (4)\end{aligned}$$

We describe in Alg. 5 a naive, though non-secure algorithm to “remove carries” from an arithmetic mask.

Algorithm 5 is insecure against DPA as it manipulates the sensitive variable a (more precisely, parts a_i of the sensitive variable) unmasked. This is done at step 4 when the i-th input block $\widetilde{a}_i$ is unmasked to compute the carry for the $i\,\ell$-th bit. However if we use (4), in the algorithm SecureSBox presented in Alg. 1 a loop for every possible masks is performed to securely access the i-th SBox.

Algorithm 5 Non-secure removing of carries

Inputs: $\widetilde{a} = (a \boxplus m) \in \mathbb{F}_2^n,\ \ m \in \mathbb{F}_2^n$
Output: $\widetilde{b} = (a_{N-1} \boxplus_\ell m_{N-1}) \| \dots \| (a_0 \boxplus_\ell m_0)$

```
1: function RemoveCarries(ã, m)
2:     c = 0                                   ▷ C propagates the carries
3:     for i = 0 to N − 1 do
4:         t ← ã_i ⊟_ℓ m_i ⊟_ℓ c
5:         c ← carry(step 4)                   ▷ save next carry
6:         b_i ← t ⊞_ℓ m_i
7:     end for
8:     return b̃ = b̃_{N−1} ‖ . . . ‖ b̃_0
9: end function
```

The idea is to use this loop to dissimulate not only the unmasking operation, but also the next carry computation.

Still, each output carry has to be masked. This may be performed using a bit z as in the mask conversion algorithm from [17] described in Alg. 3. Indeed, consider (3) when masked by a random bit z. It holds then that

$$\begin{aligned}\widetilde{a} \oplus (-z) &= (a \boxplus m) \oplus (-z)\\ &= (a_{N-1} \boxplus_\ell m_{N-1} \boxplus_\ell c_{N-1}) \oplus (-z) \| \dots \| (a_0 \boxplus_\ell m_0 \boxplus_\ell c_0) \oplus (-z)\ \ .\end{aligned}$$

For each i, $0 \leqslant i < N$, let $a_i^z = a_i \oplus (-z)$ and $m_i^z = m_i \oplus (-z)$. Let $c_i^z = c_i \oplus (-z) = c_i \oplus z$, the i-th carry masked by the bit z. Then, according to (1), we obtain that

$$\widetilde{a} \oplus (-z) = \left(a_{N-1}^z \boxplus_\ell m_{N-1}^z \boxplus_\ell c_{N-1}^z\right) \| \dots \| (a_0^z \boxplus_\ell m_0^z \boxplus_\ell c_0^z)\ \ . \tag{5}$$

Equation (5) together with Alg. 5 are used to derive the main algorithm of this paper presented in Alg. 6.

To prove the correctness of Alg. 6, we study the internal variables when $i = 0$. It can be noticed that only when k is equal to the mask chunk m_0^z, step 12 becomes

$$R_1 = S_0[(\widetilde{a}_0^z \boxminus_\ell m_0^z \boxminus_\ell z) \oplus (-z)] \oplus m_0'\ \ .$$

Using (5), as $\widetilde{a}_0^z = a_0^z \boxplus_\ell m_0^z \boxplus_\ell z$, it holds that

$$\begin{aligned}R_1 &= S_0[((a_0^z \boxplus_\ell m_0^z \boxplus_\ell z) \boxminus_\ell m_0^z \boxminus_\ell z) \oplus (-z)] \oplus m_0'\\ &= S_0[a_0^z \oplus (-z)] \oplus m_0'\\ &= S_0[a_0] \oplus m_0'\ \ .\end{aligned}$$

At step 11, we also have

$$\begin{aligned}B_1 &= \textsc{carry}(\text{step } 10)\\ &= \begin{cases}1 & \text{if}\quad (a_0^z + m_0^z + z) \geqslant 2^\ell\\ 0 & \text{otherwise}\end{cases}\ \ .\end{aligned}$$

Algorithm 6 Secure multiple SBox with arithmetically masked input

Inputs: $S = (S_0, \ldots, S_{N-1}) \in (\mathbb{F}_2^\ell \to \mathbb{F}_2^\ell)^N$, $\widetilde{a} = (a \boxplus m) \in \mathbb{F}_2^n$, $m \in \mathbb{F}_2^n$, $m' \in \mathbb{F}_2^n$
Output: $\widetilde{b} = S[a] \oplus m' = (S_{N-1}[a_{N-1}] \oplus m'_{N-1}) \| \ldots \| (S_0[a_0] \oplus m'_0)$

1: **function** SecureMulSBox($S, \widetilde{a}, m, m'$)
2: $\quad z \leftarrow$ Random($\{0,1\}$)
3: $\quad \widetilde{a}^z \leftarrow \widetilde{a} \oplus (-z)$
4: $\quad m^z \leftarrow m \oplus (-z)$
5: $\quad B_1 \leftarrow z$ ▷ B_1 propagates the correct carries
6: $\quad$ **for** $i = 0$ **to** $N-1$ **do**
7: $\quad\quad c \leftarrow B_1$
8: $\quad\quad$ **for** $k = 0$ **to** $2^\ell - 1$ **do**
9: $\quad\quad\quad \text{cmp} \leftarrow (k \stackrel{?}{=} m_i^z)$ ▷ if k and m are equal, cmp is 1, else 0
10: $\quad\quad\quad t \leftarrow \widetilde{a}_i^z \boxminus_\ell k \boxminus_\ell c$
11: $\quad\quad\quad B_{\text{cmp}} \leftarrow$ carry(step10) ▷ save next carry
12: $\quad\quad\quad R_{\text{cmp}} \leftarrow S_i[t \oplus (-z)] \oplus m'_i$
13: $\quad\quad$ **end for**
14: $\quad\quad \widetilde{b}_i \leftarrow R_1$
15: $\quad$ **end for**
16: $\quad$ **return** $\widetilde{b} = \widetilde{b}_{N-1} \| \ldots \| \widetilde{b}_0$
17: **end function**

This corresponds to the next carry masked by the bit z:

$$B_1 = c_1 \oplus z \ .$$

For $i \geqslant 0$, assume that B_1 contains the current masked carry at the beginning of step 8. For each i, $0 < i < N$, it holds then that

$$\begin{aligned} R_1 &= S_i[(\widetilde{a}_i^z \boxminus_\ell m_i^z \boxminus_\ell c_i^z) \oplus (-z)] \oplus m'_i \\ &= S_i[((a_i^z \boxplus_\ell m_i^z \boxplus_\ell c_i^z) \boxminus_\ell m_i^z \boxminus_\ell c_i^z) \oplus (-z)] \oplus m'_i \\ &= S_i[a_i^z \oplus (-z)] \oplus m'_i \\ &= S_i[a_i] \oplus m'_i \ . \end{aligned}$$

Consequently, we also obtain

$$B_1 = c_{i+1}^z = c_{i+1} \oplus z \ ,$$

which is the next masked carry. At the end of Alg. 6, $\widetilde{b}_i = S_i[a_i] \oplus m'_i$ for each i, $0 \leqslant i < N$, which is the expected output.

4.2 Security Analysis

To achieve security against first order DPA, each step of our algorithm shall not depend on any unmasked secret data. Before discussing the security in details, we need to precisely define the attacker model.

Attacker: We assume that the attacker has full access to a device performing an encryption using a block-cipher implementing Alg. 6 as a countermeasure in the

same context as Fig. 2. The attacker can choose any input to the block-cipher, and we assume that she is able to predict the value of the input a of Alg. 6 for any hypothesis on the secret key$^{(t)}$. The attacker can only perform first-order DPA, meaning that only one moment of the execution can be targeted at each execution. But, the attacker is able to acquire as many power traces as needed for the targeted moment.

Leakage: We assume that the device is leaking information on the values involved during every operation. This information may be their Hamming weight. Every operation is assumed to leak similarly.

Recall that a and $S[a]$ are sensitive data. Algorithm 6 is secure against the previously defined model only if each operation involved behaves as it was manipulating only random or constant values. No operation should involve both a masked value and the corresponding mask at the same time. We analyze each step of Alg. 6 involving a sensitive data in Table 1. Steps 2, 4, 5 and 9 are omitted as they manipulate only constant or random value. The notation k^z is used to denote $k \oplus (-z)$.

Table 1. Sensitive values manipulated in Alg. 6

Step	Instruction	Manipulated value	Sensitive value	Mask(s)
3.1	$t \leftarrow \widetilde{a}$	$a \boxplus m$	a	m
3.2	$\widetilde{a}^z \leftarrow t \oplus (-z)$	$(a \boxplus m) \oplus (-z)$	a	m, z
7	$c \leftarrow B_1$	c_i^z	c_i	z
10.1	$t \leftarrow \widetilde{a}_i^z$	$(a_i \boxplus_\ell m_i \boxplus_\ell c_i) \oplus (-z)$	$a_i \boxplus_\ell c_i$	m_i, z
10.2	$t \leftarrow t \boxminus_\ell k \boxminus_\ell c$	$a_i \oplus (-z) \boxplus_\ell m_i^z \boxminus_\ell k$	a_i	z, $m_i^z \boxminus_\ell k$
11	$B_{\text{cmp}} \leftarrow \textsc{carry}(\text{step}10)$	c_{i+1}^z	c_{i+1}	z
12.1	$t \leftarrow t \oplus (-z)$	$a_i \boxplus_\ell m_i \boxminus_\ell k^z$	a_i	$m_i \boxminus_\ell k^z$
12.2	$t \leftarrow S_i[t]$	$S_i[a_i \boxplus_\ell m_i \boxminus_\ell k^z]$	a_i	$m_i \boxminus_\ell k^z$
12.3	$R_{\text{cmp}} \leftarrow t \oplus m'_i$	$S_i[a_i \boxplus_\ell m_i \boxminus_\ell k^z] \oplus m'_i$	a_i	$m_i \boxminus_\ell k^z$, m'_i
14	$\widetilde{b}_i \leftarrow R_1$	$S_i[a_i] \oplus m'_i$	$S_i[a_i]$	m'_i

We further detail the steps presented in Table 1:

- Step 3 manipulates a masked variable, and the operation only adds another independent random bit.
- Steps 7 and 14 are assignments of masked data: B_1 is the next carry masked by z and R_1 is $S_i[a_i]$ masked by m'_i.
- At steps 10 and 12, k is constant, then the masks $m_i^z \boxminus_\ell k$ or $m_i \boxminus_\ell k^z$ are uniformly distributed on ℓ bits. These steps are performed once for every possible value of k. Next, the carry read at Step 11 is already masked by the bit z and gives no information on the sensitive variable.

According to Table 1, each sensitive value is masked with a uniformly distributed mask of at least as many bits as the sensitive value.

5 Implementation

We implemented our method (Alg. 6) as well as the method using the mask conversion from [17], then the secure SBox algorithm of [19] (Alg. 4). The implementations were done in 8051 assembly with the same optimizations – favoring small code size. The implemented example is $N = 8$ SBoxes of $\ell = 4$ bits input and output which are the parameters used in the GOST block-cipher. For Alg. 4, we used $\lambda \in \{2, 4, 8\}$ as chunk size for the AMtoBM conversion algorithm. In the cases $\lambda < 8$, each element of precomputed tables T and C has been stored on one byte. In the case $\lambda = 8$, we also used the optimization for Alg. 4 described in Remark 2. We give in Table 5 a comparison between these methods regarding several parameters. Our method has not been compared to methods based on table re-computation because we targeted low memory devices. For these parameters, such methods would require at least 64 extra bytes of memory.

Table 2. Comparison of 8051 implementations of Alg. 4 and Alg. 6

	Alg. 4 ([17] + [19])			Alg. 6
	$\lambda = 2$	$\lambda = 4$	$\lambda = 8$	(this paper)
rand. gen. (in bits)	3	5	9	1
pre-comp. time (in cy)	72	201	3349	$\emptyset$
algorithm time (in cy)	3013	2773	2633	3334
XRAM (in bytes)	4	16	256	0
Code (in bytes)	276	271	256	151

It can be noticed that our new algorithm outperforms the others in terms of memory requirements (RAM and code). This makes it particularly suitable on devices with limited resources. If only one call is needed, the execution time of our algorithm is similar to the algorithms from [17] and [19] with $\lambda = 2$, a bit slower with $\lambda = 4$, and better with $\lambda = 8$. Then, for the implementation of a full block cipher, depending on the number of secure rounds needed, our method may be less efficient than those using pre-computation, but still has the advantage of using no extra memory. Furthermore, the random generation time has not been taken into account in the provided timings. On a device where no fast random is available, the cost of the additional random generation needed by Alg. 4 may make it even slower.

6 Conclusion

We have introduced a new method that answers the problem of accessing so-called multiple SBoxes with arithmetically masked input in the context of a

software implementation secure against first order DPA. One advantage of the proposed solution is that no extra RAM is needed to securely compute the output of the SBoxes. Besides, the code size of the proposed method is relatively low, and the execution speed remains competitive with other methods when only few computations are needed. All in all, our method is particularly suitable for extremely constrained devices with tight requirements on memory (RAM and ROM). We have demonstrated the security of our method against first-order DPA. An extension of our algorithm to second-order masking as in [21] is the next step of this work.

References

1. Akkar, M.L., Bévan, R., Goubin, L.: Two Power Analysis Attacks against One-Mask Method. In: Roy, B., Meier, W. (eds.) Fast Software Encryption – FSE 2004. LNCS, vol. 3017, pp. 332–347. Springer (2004)
2. Akkar, M.L., Giraud, C.: An Implementation of DES and AES, Secure against Some Attacks. In: Koç, Ç., Naccache, D., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2001. LNCS, vol. 2162, pp. 309–318. Springer (2001)
3. Biham, E., Shamir, A.: Power Analysis of the Key Scheduling of the AES Candidates. In: Second AES Candidate Conference – AES 2 (Mar 1999), `http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm`
4. Blömer, J., Merchan, J.G., Krummel, V.: Provably Secure Masking of AES. In: Matsui, M., Zuccherato, R. (eds.) Selected Areas in Cryptography – SAC 2004. LNCS, vol. 3357, pp. 69–83. Springer (2004)
5. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: Joye, M., Quisquater, J.J. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer (2004)
6. Chari, S., Jutla, C., Rao, J., Rohatgi, P.: Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Wiener, M. (ed.) Advances in Cryptology – CRYPTO '99. LNCS, vol. 1666, pp. 398–412. Springer (1999)
7. Coron, J.S., Tchulkine, A.: A New Algorithm for Switching from Arithmetic to Boolean Masking. In: Walter, C., Koç, Ç., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2003. LNCS, vol. 2779, pp. 89–97. Springer (2003)
8. Debraize, B.: Efficient and provably secure methods for switching from arithmetic to boolean masking. In: Cryptographic Hardware and Embedded Systems – CHES 2012. LNCS, vol. 7428. Springer (2012)
9. Genelle, L., Prouff, E., Quisquater, M.: Secure multiplicative masking of power functions. In: Zhou, J., Yung, M. (eds.) Applied Cryptography and Network Security - 8th International Conference, ACNS 2010. LNCS, vol. 6123, pp. 200–217. Springer (2010)
10. Goubin, L.: A Sound Method for Switching between Boolean and Arithmetic Masking. In: Koç, Ç., Naccache, D., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2001. LNCS, vol. 2162, pp. 3–15. Springer (2001)
11. Goubin, L., Patarin, J.: DES and Differential Power Analysis – The Duplication Method. In: Koç, Ç., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems – CHES '99. LNCS, vol. 1717, pp. 158–172. Springer (1999)
12. Kim, H., Cho, Y.I., Choi, D., Han, D.G., Hong, S.: Efficient masked implementation forr SEED based on combined masking. ETRI Journal 33(2), 267–274 (April 2011)

13. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) Advances in Cryptology – CRYPTO '99. LNCS, vol. 1666, pp. 388–397. Springer (1999)
14. Lai, X., Massey, J.: A proposal for a new block encryption standard. In: Damgård, I. (ed.) Advances in Cryptology – EUROCRYPT '90. LNCS, vol. 473, pp. 389–401. Springer (1990)
15. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks – Revealing the Secrets of Smartcards. Springer (2007)
16. Messerges, T.: Securing the AES Finalists against Power Analysis Attacks. In: Schneier, B. (ed.) Fast Software Encryption – FSE 2000. LNCS, vol. 1978, pp. 150–164. Springer (2000)
17. Neiße, O., Pulkus, J.: Switching Blindings with a View Towards IDEA. In: Joye, M., Quisquater, J.J. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2004. LNCS, vol. 3156, pp. 230–239. Springer (2004)
18. 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.) Topics in Cryptology – CT-RSA 2006. LNCS, vol. 3860, pp. 192–207. Springer (2006)
19. 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 (2008)
20. Rivain, M., Prouff, E.: Provably Secure Higher-order Masking of AES. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2010. LNCS, vol. 6225. Springer (2010)
21. Rivain, M., Dottax, E., Prouff, E.: Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis. In: Baignères, T., Vaudenay, S. (eds.) Fast Software Encryption – FSE 2008. pp. 127–143. LNCS, Springer (2008)
22. Schramm, K., Paar, C.: Higher Order Masking of the AES. In: Pointcheval, D. (ed.) Topics in Cryptology – CT-RSA 2006. LNCS, vol. 3860, pp. 208–225. Springer (2006)
23. Telecommunications Technology Association: 128-bit symmetric block cipher (SEED) (1998), Seoul, Korea
24. Zabotin, I.A., Glazkov, G.P., Isaeva, V.B.: Cryptographic protection for information processing systems, government standard of the USSR, GOST 28147-89. Government Committee of the USSR for Standards (1989)