Faster Mask Conversion with Lookup Tables

Praveen Kumar Vadnala and Johann Großschädl

University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security,
6, rue Richard Coudenhove-Kalergi, L–1359 Luxembourg
{praveen.vadnala,johann.groszschaedl}@uni.lu

Abstract. Masking is an effective and widely-used countermeasure to thwart Differential Power Analysis (DPA) attacks on symmetric cryptosystems. When a symmetric cipher involves a combination of Boolean and arithmetic operations, it is necessary to convert the masks from one form to the other. There exist algorithms for mask conversion that are secure against first-order attacks, but they can not be generalized to higher orders. At CHES 2014, Coron, Großschädl and Vadnala (CGV) introduced a secure conversion scheme between Boolean and arithmetic masking of any order, but their approach requires $d = 2t + 1$ shares to protect against attacks of order t. In the present paper, we improve the algorithms for second-order conversion with the help of lookup tables so that only three shares instead of five are needed, which is the minimal number for second-order resistance. Furthermore, we also improve the first-order secure addition method proposed by Karroumi, Richard and Joye, again with lookup tables. We prove the security of all presented algorithms using well established assumptions and models. Finally, we provide experimental evidence of our improved mask conversion applied to HMAC-SHA-1. Simulation results show that our algorithms improve the execution time by 85% at the expense of little memory overhead.

Keywords: Side-channel analysis (SCA), arithmetic masking, Boolean masking, provably secure masking, HMAC-SHA-1

1 Introduction

Ever since the introduction of Side-Channel Analysis (SCA) attacks in the late 1990s, there has been much interest in finding countermeasures to thwart this form of "physical cryptanalysis," in particular the Differential Power Analysis (DPA) attacks [8]. From a high-level perspective, DPA countermeasures aim to either randomize the power consumption (which can be done in both the time and amplitude domain) or make it completely independent from the processed data. The goal of both approaches is to eliminate (or, at least, reduce) the correlation between the power consumption and the key-dependent intermediate variables processed during the execution of a cryptographic algorithm. Concrete examples for randomization in the time domain include various "hiding"-style countermeasures like the insertion of random delays or shuffling of operations

[9]. On the other hand, a classical example of randomization in the amplitude domain is masking, which aims to conceal each sensitive intermediate variable x with a random value x_2, called mask [2,9]. This means that x is represented by two shares, namely the masked variable $x_1 = x \oplus x_2$ and the mask x_2. The two shares need to be manipulated separately throughout the execution of the algorithm to ensure that the instantaneous power consumption of the device does not leak any information about x. A conventional DPA attack may reveal x_1 or x_2 (both of which appear as random numbers to the attacker), but the knowledge of x_1 alone or x_2 alone does not give the attacker any information about the sensitive variable x.

One of the major challenges when applying masking to a block cipher is to implement the round functions in such a way that the shares can be processed independently from each other, while it still must be possible to recombine them at the end of the execution to get the correct result. This is fairly easy for all linear operations, but can introduce massive overheads for the non-linear parts of a cipher, i.e. the S-boxes. In addition, all round transformations need to be executed twice (namely for x_1 and for x_2, where $x = x_1 \oplus x_2$), which entails a further performance penalty. Another problem is that a basic masking scheme as described above is vulnerable to a so-called second-order DPA attack where an attacker combines information from two leakage points (i.e. he exploits the side-channel leakage originating from x_1 and x_2 simultaneously [11]). Such a second-order DPA attack can, in turn, be thwarted by second-order masking, in which each sensitive variable is concealed with two random masks and, consequently, represented by three shares. In general, a d-th order masking scheme uses d random masks to split a sensitive intermediate variable into $d+1$ shares $x_1, x_2, \ldots, x_{d+1}$ satisfying $x_1 \oplus x_2 \oplus \cdots \oplus x_{d+1} = x$, which are then processed independently. In this way, it is guaranteed that the joint leakage of any subset of up to d shares is independent of the secret key. Only a combination of all $d+1$ shares (i.e. the masked variable $x_1 = x \oplus x_2 \oplus \cdots \oplus x_{d+1}$ and the d masks $x_2, \ldots, x_{d+1}$) is jointly dependent on the sensitive variable. However, given the presence of noise, the cost for attacking a higher-order masked implementation increases exponentially with d [2].

Depending on the algorithmic properties of a cipher, a masking scheme can have to protect Boolean operations (e.g. xors, shifts) or arithmetic operations (e.g. modular additions). When a cipher involves both Boolean and arithmetic operations, it is necessary to convert the masks from one form to the other to obtain the correct ciphertext (or plaintext). Examples of symmetric algorithms that involve arithmetic as well as Boolean operations include the widely-used hash functions SHA-1, SHA-2, Blake and Skein, some ARX-based block ciphers (e.g. XTEA, Threefish) and all four finalists for the eSTREAM software portfolio. Given the widespread deployment of these cryptosystems in various kinds of application (including some with a need for sophisticated countermeasures against DPA), it is important to develop efficient techniques for the conversion between Boolean and arithmetic masks. However, almost all secure conversion techniques reported in the literature are only applicable to first-order masking

[4–7, 10]. Among the few exceptions is the second-order conversion scheme due to Vadnala and Großschädl [13] and the recent higher-order conversion scheme by Coron, Großschädl and Vadnala [3]. We outline both schemes below.

Vadnala-Großschädl Scheme [13]. The foundation of this technique is the generic second-order countermeasure that Rivain, Dottax and Prouff proposed at FSE 2008 [12]. We recall their algorithm for computing a second-order secure masked S-box output from a second-order secure masked input below.

Algorithm 1. Sec2O-masking [12]

```
Input: Three input shares: $(x_1 = x \oplus x_2 \oplus x_3, x_2, x_3) \in \mathbb{F}_{2^n}$, two output shares:
    $(y_1, y_2) \in \mathbb{F}_{2^m}$, and an $(n, m)$ S-box lookup function $S$
Output: Masked S-box output: $S(x) \oplus y_1 \oplus y_2$
1: $r \leftarrow \mathsf{Rand}(n)$
2: $r' \leftarrow (r \oplus x_2) \oplus x_3$
3: for $a := 0$ to $2^n - 1$ do
4:     $a' \leftarrow a \oplus r'$
5:     $T[a'] \leftarrow ((S(x_1 \oplus a) \oplus y_1) \oplus y_2)$
6: end for
7: return $T[r]$
```

In Algorithm 1, a lookup table is generated for all possible values of x. The index to the lookup table is masked using a random number r. Then, the correct value of the share is obtained by retrieving the table entry corresponding to the index r. The main idea here is that the actual computation of the third arithmetic share is hidden among other dummy calculations for all the possible values. Since the value of r changes for every iteration, the attacker is not able to guess the point in time at which the actual value of x is being leaked. The authors of [12] proved the security of the algorithm by demonstrating that no pair of intermediate variables leaks any sensitive information.

The goal of a second-order Boolean to arithmetic conversion is to compute arithmetic shares from a set of Boolean shares without introducing any second or first-order leakage. In order to achieve second-order DPA resistance, we need three Boolean shares x_1, x_2, and x_3 so that the sensitive variable x is given as $x = x_1 \oplus x_2 \oplus x_3$. The goal is to find three arithmetic shares A_1, A_2, A_3 satisfying $x = A_1 + A_2 + A_3$ without leaking any first or second-order information about x. The solution given by Vadnala and Großschädl [13] is to modify the masked lookup table in Algorithm 1 to store $((x_1 \oplus a) - A_2) - A_3$ instead of a masked S-box output; the rest of the algorithm is very similar to the original one. They followed the same approach for arithmetic to Boolean conversion.

Coron-Großschädl-Vadnala Scheme [3]. Recently, Coron, Großschädl and Vadnala proposed conversion algorithms that are secure against attacks of any order [3]. They first proposed a secure solution to add Boolean shares directly by generalizing Goubin's recursion formula [6]. Their solution has a complexity $\mathcal{O}(d^2 \cdot n)$ to secure against t-th order attacks, where $d \geq 2t + 1$ and n is the size of the masks. Then, they used this addition as subroutine to derive algorithms for conversion between Boolean and arithmetic masking, again with complexity $\mathcal{O}(d^2 \cdot n)$.

Our Contributions. The generic solution of Coron, Großschädl and Vadnala [3] requires five shares to protect against second-order attacks, which entails a significant overhead in terms of the required amount of random numbers and execution time. Although the algorithms proposed by Vadnala and Großschädl [13] require only three shares to achieve second-order resistance, they become infeasible for implementation on low-resource devices (e.g. smart cards) when $n > 10$ (the additions are performed modulo 2^n), as they require a lookup table of size 2^n.

In the present paper, we propose second-order secure conversion algorithms that overcome said limitations and can, thus, be easily applied to cryptographic constructions with arbitrary n, e.g. HMAC-SHA-1 with $n = 32$. The proposed algorithms use only three shares and are, therefore, significantly faster than the state-of-the-art. Our solution follows the basic idea of Vadnala and Großschädl (which, in turn, is based on work of Rivain, Dottax and Prouff [12]), but uses a divide and conquer approach to prevent that the lookup tables become prohibitively large. In the case of Boolean to arithmetic conversion, we divide the Boolean shares into words of $l \leq 8$ bits each and then compute the words of the corresponding arithmetic shares independently in a word-by-word fashion. Part of this procedure is to handle all the carries propagating from less to more significant words, which also need to be protected by masking to prevent any first or second-order leakage. We show that this can be achieved in an efficient and secure fashion by using separate lookup tables for the carries. Furthermore, we prove the security of our conversion schemes in the same model as [12]. Using similar techniques, we show that the efficiency of the first-order secure masked addition due to Karroumi, Richard and Joye [7] can be improved as well.

2 Efficient Second-Order Secure Boolean to Arithmetic Masking

In this section, we give the efficient Boolean to arithmetic conversion algorithm secure against attacks of second order. The idea is to split the n-bit shares into p words (of l bits each) and convert each word independently.

2.1 Boolean to Arithmetic Masking of second-order

We are given three Boolean shares x_1, x_2, x_3 so that the sensitive variable x is obtained through $x = x_1 \oplus x_2 \oplus x_3$. The goal is to find three arithmetic shares

A_1, A_2, A_3 that satisfy $x = A_1 + A_2 + A_3$ without leaking any first or second-order information on x. This can be achieved by generating two shares A_2 and A_3 randomly and computing the third share as $A_1 = x - A_2 - A_3$, as done in [13] using the approach of Rivain, Dottax and Prouff from [12]. But as stated earlier, this scheme becomes infeasible for use in practice when $n > 10$ since it requires a lookup table of size 2^n. To obtain a solution for $n > 10$, we apply a divide and conquer approach. That is, we split each share into p words of l bits each and compute $(A_1^i)_{(0 \le i \le p-1)}$ independently, where $A_1 = A_1^{p-1} || \cdots || A_1^0$. In this case, we also need to handle the carries propagating from word i to word $i+1$ properly. More precisely, these carries must be protected by masking, as otherwise they would leak information about the sensitive variable. Below, we describe our method to protect the sensitive variables along with carries and demonstrate its security with a formal proof.

We differentiate between two sets of carries: input carries (i.e. carries used for the computation of A_1^i) and output carries (i.e. carries generated while computing A_1^i). Since the computation of A_1^i involves two subtractions, there are two output carries from each word A_1^i, which become input carries for the word A_1^{i+1}. For the first word A_1^0, the input carries are initialized to 0, i.e. $c_1^0 = 0$ and $c_2^0 = 0$. We compute A_1^i from the input x^i and carries c_1^i, c_2^i as follows:

$$A_1^i = (x^i -_l c_1^i -_l A_2^i -_l c_2^i -_l A_3^i)$$

An operation of the form $a -_l b$ represents $a - b \bmod 2^l$. Similarly, the output carries c_1^{i+1}, c_2^{i+1} are computed as follows:

$$c_1^{i+1} = \mathsf{Carry}(x^i, c_1^i) \oplus \mathsf{Carry}(x^i -_l c_1^i, A_2^i) \tag{1}$$
$$c_2^{i+1} = \mathsf{Carry}(x^i -_l c_1^i -_l A_2^i, c_2^i) \oplus \mathsf{Carry}(x^i -_l c_1^i -_l A_2^i -_l c_2^i, A_3^i) \tag{2}$$

where $\mathsf{Carry}(a, b)$ represents the carry from the subtraction $a - b$. As specified by Equation (1) and (2), each carry computation involves two subtractions: one with the input carry (c_1^i, c_2^i) and the other with a random share (A_2^i, A_3^i). In the simplest case, a subtraction $a - b$ produces a carry when $a < b$. However, in our scenario, we have operations of the form $(a -_l c) -_l b$, whereby a and b are l-bit integers and c is either 0 or 1. In the case of $c = 0$, the above operation generates a carry if $a < b$. On the other hand, when $c = 1$, we have to take into account another case, namely $a < c$, which can only happen when $a = 0$ and $c = 1$. In this special case, the difference $a -_l c$ becomes $2^l - 1$ and a carry is generated that needs to be processed as well. However, the second subtraction can not generate a carry as $b \le 2^{l-1}$. Namely, the carries from these two cases are mutually exclusive; hence, the output carry is set to 1 when either of them produces a carry as shown in Equation (1) and (2). For simplicity, we define the functions $F_1 :\ \{0,1\}^{l+1} \to \{0,1\}^{l+1}$ and $F_2 :\ \{0,1\}^{2l} \to \{0,1\}^{l+1}$ as follows.

$$F_1(a, b) \ = \ a -_l b \,||\, (\mathsf{Carry}(a, b)) \tag{3}$$
$$F_2(a, b) \ = \ a -_l b \,||\, (\mathsf{Carry}(a, b)) \tag{4}$$

For a given word with index i, we can compute A_1^i as well as the output carries c_1^{i+1}, c_2^{i+1} using F_1 and F_2 according to the following equations:

$$(B_1^i||d_1^i) = F_1(x^i, c_1^i)$$
$$(B_2^i||d_2^i) = F_2(B_1^i, A_2^i)$$
$$(B_3^i||d_3^i) = F_1(B_2^i, c_2^i)$$
$$(B_4^i||d_4^i) = F_2(B_3^i, A_3^i)$$

where $A_1^i = B_4^i$ and $c_1^{i+1} = d_1^i \oplus d_2^i$, $c_2^{i+1} = d_3^i \oplus d_4^i$. As pointed out in [12], the S-box in Rivain, Dottax and Prouff's scheme must be balanced in order to be secure[1]. In our case, the function F_1 plays the same role and is balanced; consequently, the security guarantee is preserved. We first present non-randomized version of our solution below for simplicity.

Algorithm 2. Insecure 2OB→A

```
Input: Sensitive variable: $x = x_1 \oplus x_2 \oplus x_3$
Output: Arithmetic shares: $x = A_1 + A_2 + A_3$
1: $c_1^0, c_2^0 \leftarrow 0$                                ▷ Initially carry is zero
2: for $i := 0$ to $p-1$ do
3:     $A_2^i, A_3^i \leftarrow \mathsf{Rand}(l)$              ▷ Generate output masks randomly
4:     $(B_1^i, d_1^i) \leftarrow F_1(x^i, c_1^i)$
5:     $(B_2^i, d_2^i) \leftarrow F_2(B_1^i, A_2^i)$
6:     $(B_3^i, d_3^i) \leftarrow F_1(B_2^i, c_2^i)$
7:     $(B_4^i, d_4^i) \leftarrow F_2(B_3^i, A_3^i)$
8:     $(A_1^i, c_1^{i+1}, c_2^{i+1}) \leftarrow (B_4^i, d_1^i \oplus d_2^i, d_3^i \oplus d_4^i)$
9: end for
10: return $A_1, A_2, A_3$
```

The challenge is to implement Algorithm 2 so that it does not leak any first or second-order information about the sensitive variable x as well as the carries c_1^i, c_2^i for $0 \leq i \leq p-1$. We present our solution in two parts: we first give the algorithm to securely compute the result for one word (namely A_1^i), and then we use this as a "subroutine" to compute A_1. Our solution, given in Algorithm 3, employs a similar technique as [12] (recalled in Algorithm 1) in combination with Algorithm 2. Algorithm 3 expects as input three Boolean shares, six input carry shares (three each for the two carries), two output arithmetic shares, and four output carry shares. It returns as result the third arithmetic share and the remaining two output carry shares. Similar to Algorithm 1, we create a lookup table T for all the possible values in $[0, 2^{l+2}-1]$. Here, l bits are used to store A_1^i and two bits for the two carries correspondingly. The rest of the algorithm is very similar to the original one, except that we have to handle two extra bits for the carry[2].

[1] An S-box $S : \{0,1\}^n \to \{0,1\}^m$ is said to be balanced if every element in $\{0,1\}^m$ is image of exactly 2^{n-m} elements in $\{0,1\}^n$ under S.

[2] We use different tables to store the actual value and the carries so that the security proof can be easily obtained as in [12].

Algorithm 3. Sec2OB→A_Word

```
Input: Three input shares: $(x_1^i = x^i \oplus x_2^i \oplus x_3^i, x_2^i, x_3^i) \in \mathbb{F}_{2^l}$, Six input carry shares:
    $g_1^i = c_1^i \oplus g_2^i \oplus g_3^i, g_2^i, g_3^i$, $g_4^i = c_2^i \oplus g_5^i \oplus g_6^i, g_5^i, g_6^i \in \mathbb{F}_2$, Output arithmetic shares:
    $A_2^i, A_3^i$, Output carry shares: $h_1^i, h_2^i, h_3^i, h_4^i$
Output: Masked Arithmetic share: $(x^i -_l A_2^i) -_l A_3^i$ and masked output carries
 1: $r_1 \leftarrow \mathsf{Rand}(l)$; $r_2 \leftarrow \mathsf{Rand}(1)$; $r_3 \leftarrow \mathsf{Rand}(1)$
 2: $r_1' \leftarrow (r_1 \oplus x_2^i) \oplus x_3^i$; $r_2' \leftarrow (r_2 \oplus g_2^i) \oplus g_3^i$; $r_3' \leftarrow (r_3 \oplus g_5^i) \oplus g_6^i$;
 3: for $a_1 := 0$ to $2^l - 1$, $a_2 := 0$ to $1$, $a_3 := 0$ to $1$ do
 4:     $a_1' \leftarrow a_1 \oplus r_1'$; $a_2' \leftarrow a_2 \oplus r_2'$; $a_3' \leftarrow a_3 \oplus r_3'$
 5:     $(B_1^i, d_1^i) \leftarrow F_1((x_1^i \oplus a_1), (g_1^i \oplus a_2))$
 6:     $(B_2^i, d_2^i) \leftarrow F_2(B_1^i, A_2^i)$
 7:     $(B_3^i, d_3^i) \leftarrow F_1(B_2^i, (g_4^i \oplus a_3))$
 8:     $(B_4^i, d_4^i) \leftarrow F_2(B_3^i, A_3^i)$
 9:     $e_1^i \leftarrow ((d_1^i \oplus h_1^i) \oplus d_2^i) \oplus h_2^i$
10:     $e_2^i \leftarrow ((d_3^i \oplus h_3^i) \oplus d_4^i) \oplus h_4^i$
11:     $(T_1[a_1'||a_2'||a_3'], T_2[a_1'||a_2'||a_3'], T_3[a_1'||a_2'||a_3']) \leftarrow (B_4^i, e_1^i, e_2^i)$
12: end for
13: return $T_1[r_1||r_2||r_3], T_2[r_1||r_2||r_3], T_3[r_1||r_2||r_3]$
```

Finally, we give our second-order secure technique to obtain three arithmetic shares corresponding to the three Boolean shares in Algorithm 4. For the first word (i.e. $i = 0$), there are no input carries and, consequently, the three shares for both carries are set to zero (Step 1), i.e. we have $g_1^0 = g_2^0 = g_3^0 = c_1^0 = 0$ and $g_4^0 = g_5^0 = g_6^0 = c_2^0 = 0$. To protect the output carries, we use four uniformly generated random bits: h_1^i, h_2^i, h_3^i, h_4^i; two each for the two carries. The third share for the carries as well as A_1^i are computed recursively using the function Sec2OB→A_Word (Algorithm 3)[3]. Note that for word i, $g_1^i \oplus g_2^i \oplus g_3^i = c_1^i$ and $g_4^i \oplus g_5^i \oplus g_6^i = c_2^i$. The time complexity of the overall solution is $\mathcal{O}(2^{l+2} \cdot p)$ and the memory requirements amount to $(2^{l+2} \cdot (l+2))$ bits.

Algorithm 4. Sec2OB→A

```
Input: Boolean shares: $x_1 = x \oplus x_2 \oplus x_3, x_2, x_3$
Output: Arithmetic shares: $A_1, A_2, A_3$ so that $x = A_1 + A_2 + A_3$
 1: $g_1^0, g_2^0, g_3^0, g_4^0, g_5^0, g_6^0 \leftarrow 0$                         ▷ Initially carry is zero
 2: for $i := 0$ to $p - 1$ do
 3:     $A_2^i, A_3^i \leftarrow \mathsf{Rand}(l)$                      ▷ Generate output masks randomly
 4:     $h_1^i, h_2^i, h_3^i, h_4^i \leftarrow \mathsf{Rand}(1)$
 5:     $(A_1^i, g_1^{i+1}, g_4^{i+1}) \leftarrow$ Sec2OB→A_Word $((x_j^i)_{1\le j\le 3}, (g_j^i)_{1\le j\le 6}, A_2^i, A_3^i, (h_j^i)_{1\le j\le 4})$
 6:     $g_2^{i+1}, g_3^{i+1}, g_5^{i+1}, g_6^{i+1} \leftarrow h_1^i, h_2^i, h_3^i, h_4^i$
 7: end for
 8: return $A_1, A_2, A_3$
```

[3] Every call to the function Sec2OB→A_Word creates a new table and is useful for that particular word only. Hence, unlike the original method in [12], we do not reuse the table.

2.2 Security Analysis

For an algorithm to be secure against second-order DPA attacks, no pair of intermediate variables appearing in the algorithm should jointly leak the sensitive variable. In [12], the authors prove the security by enumerating all the possible pairs of intermediate variables and showing that the joint distribution of none of these pairs is dependent on the distribution of the sensitive variable. We use a similar method to prove the security of Algorithm 3. Thereafter, we prove the security of Algorithm 4 through induction.

Lemma 1. *Algorithm 3 is secure against second-order DPA.*

Proof. We list all intermediate variables used in Algorithm 1 and Algorithm 3 in Table 1. The intermediate variables computed through a similar technique appear in the same row. The only difference is that we have three intermediate variables instead of one for each row[4]. Hence, the security of Algorithm 3 can be derived from the same arguments as in the case of Algorithm 1. □

Table 1. Comparison of intermediate variables used in Algorithm 1 and Algorithm 3

Intermediate variables in Algorithm 1	**Intermediate variables in Algorithm 3**
x_2	x_2^i, g_2^i, g_5^i
x_3	x_3^i, g_3^i, g_6^i
y_1	A_2^i, h_1^i, h_3^i
y_2	A_3^i, h_2^i, h_4^i
r	r_1, r_2, r_3
$x_2 \oplus r$	$x_2^i \oplus r_1, g_2^i \oplus r_2, g_5^i \oplus r_3$
$x_2 \oplus r \oplus x_3$	$x_2^i \oplus r_1 \oplus x_3^i, g_2^i \oplus r_2 \oplus g_3^i, g_5^i \oplus r_3 \oplus g_5^i$
a	a_1, a_2, a_3
$a \oplus r \oplus x_2 \oplus x_3$	$a_1 \oplus r_1', a_2 \oplus r_2', a_3 \oplus r_3'$
$x_1 = x \oplus x_2 \oplus x_3$	$x_1^i = x^i \oplus x_2^i \oplus x_3^i, g_1^i = c_1^i \oplus g_2^i \oplus g_3^i, g_4^i = c_2^i \oplus g_3^i \oplus g_6^i$
$x_1 \oplus a$	$x_1^i \oplus a, g_1^i \oplus a_2, g_4^i \oplus a_3$
$S(x_1 \oplus a)$	$(B_1^i \Vert d_1^i) = F_1((x_1^i \oplus a), g_1^i \oplus a_2)$ $(B_3^i \Vert d_3^i) = F_1((x_1^i \oplus a) -_l g_1^i \oplus a_2 -_l A_2^i, g_4^i \oplus a_3)$ $d_2^i = \mathsf{Carry}((x_1^i \oplus a) -_l (g_1^i \oplus a_2), A_2^i)$ $d_4^i = \mathsf{Carry}((x_1^i \oplus a) -_l (g_1^i \oplus a_2) -_l A_2^i -_l (g_4^i \oplus a_3), A_3^i)$
$S(x_1 \oplus a) \oplus y_1$	$B_2^i = (x_1^i \oplus a) -_l (g_1^i \oplus a_2) -_l A_2^i,$ $d_1^i \oplus h_1^i \oplus d_2^i, d_3^i \oplus h_3^i \oplus d_4^i$
$S(x_1 \oplus a) \oplus y_1 \oplus y_2$	$B_4^i = (x_1^i \oplus a) -_l (g_1^i \oplus a_2) -_l A_2^i -_l (g_4^i \oplus a_3) -_l A_3^i,$ $d_1^i \oplus h_1^i \oplus d_2^i \oplus h_2^i, d_3^i \oplus h_3^i \oplus d_4^i \oplus h_4^i$
$S(x) \oplus y_1 \oplus y_2$	$x^i -_l c_1^i -_l A_2^i -_l c_2^i -_l A_3^i,$ $c_1^{i+1} \oplus h_1^i \oplus h_2^i, c_2^{i+1} \oplus h_3^i \oplus h_4^i$

Theorem 1. *Algorithm 4 is secure against second-order DPA.*

Proof. To prove the security of Algorithm 4, we apply mathematical induction on the number of words p. When $p = 1$, we already know that the algorithm is

[4] The only exception is for the row $S(x_1 \oplus a)$, where we have four variables.

secure due to the proof of Lemma 1. Now, assume that the algorithm is secure for $p = n$. Let E_i be the set that represents the collection of all intermediate variables corresponding to the word i. Then, by the induction hypothesis, the set $\{E_1, \cdots E_n\} \times \{E_1, \cdots E_n\}$ is independent of the sensitive variables x, c_1 and c_2. For the algorithm to be secure when $p = n + 1$, the set $\{E_1, \cdots E_n, E_{n+1}\} \times \{E_1, \cdots E_n, E_{n+1}\}$ should be independent of the sensitive variables x, c_1 and c_2. Without loss of generality, we divide this set into three subsets as follows: $\{E_{n+1} \times E_{n+1}\}$, $\{E_1, \cdots E_n\} \times \{E_1, \cdots E_n\}$, and $\{E_{n+1}\} \times \{E_1, \cdots E_n\}$. The security of $\{E_{n+1} \times E_{n+1}\}$ can be established directly from the base case, and the security of $\{E_1, \cdots E_n\} \times \{E_1, \cdots E_n\}$ follows from the induction hypothesis (see above). All the intermediate variables in E_{n+1} fall into two categories: (i) the variables that are generated randomly and are independent of any variables in $\{E_1, \cdots E_n\}$, and (ii) the variables that are a function of one or more of the following: x^{n+1}, c_1^{n+1}, c_2^{n+1}. Any pair of intermediate variables involving the former category is independent of the sensitive variables by definition and the first-order resistance of the set $\{E_1, \cdots E_n\}$. The two carry shares for the word $n + 1$, namely $(c_i^{n+1})_{1 \le i \le 2}$, are computed from the word n. Thus, the security of $(c_i^{n+1})_{1 \le i \le 3} \times \{E_1, \cdots E_n\}$ is already established in $\{E_n\} \times \{E_1, \cdots E_n\}$. One can easy see that the set $(x^{n+1}) \times \{E_1, \cdots E_n\}$ is independent of any sensitive variable. Hence, the set $\{E_{n+1}\} \times \{E_1, \cdots E_n\}$ is also independent of any sensitive variable, which proves the theorem. □

3 Efficient Second-Order Secure Arithmetic to Boolean Masking

In arithmetic to Boolean conversion, the problem is to find three shares x_1, x_2 and x_3 satisfying $x = x_1 \oplus x_2 \oplus x_3$, where the sensitive variable x is represented by three arithmetic shares A_1, A_2, A_3 with $x = A_1 + A_2 + A_3$. To tackle this problem, we follow the same strategy as in Section 2.1. We first generate two Boolean shares x_2 and x_3 randomly, and compute the third share by using the relation $x_1 = ((A_1 + A_2 + A_3) \oplus x_2 \oplus x_3)$, without leaking the value of x in a first or second-order DPA. To achieve this, we take the following approach: we firstly obtain a method to convert a single arithmetic share-word, and then we employ this procedure recursively to all words. For each word, we have to deal with two carries corresponding to the two additions, namely the carry from the addition of the two shares corresponding to A_2, A_3 and its subsequent addition with A_1. Our solution is described in Algorithm 5 and Algorithm 6.

Algorithm 5 provides our solution for converting one word of Boolean shares to corresponding arithmetic shares. We again use the technique from Algorithm 1 as in Algorithm 3. Since the input shares here are masked using arithmetic masking instead of Boolean masking, we have to modify the operations accordingly. Hence, the computation of r'_1 (in Step 2) and a'_1 (in Step 5) are replaced with additive operations. However, we can still mask the carries using Boolean masking as previously and, hence, the corresponding operations do not change (Step 3, Step 7). We create a table for all possible values in $[0, 2^{l+2} - 1]$, where

Algorithm 5. Sec20A→B_Word

```
Input: Three input shares: $(A_1^i = (x^i - A_2^i) - A_3^i, A_2^i, A_3^i) \in \mathbb{F}_{2^l}$, Six input carry shares: $g_1^i = c_1^i \oplus g_2^i \oplus g_3^i, g_2^i, g_3^i$, $g_4^i = c_2^i \oplus g_5^i \oplus g_6^i, g_5^i, g_6^i \in \mathbb{F}_2$, Output Boolean shares: $x_2^i, x_3^i$, Output carry shares: $h_1^i, h_2^i, h_3^i, h_4^i$
Output: Third Boolean share: $x_1^i = x^i \oplus x_2^i \oplus x_3^i$ and masked output carries
1: $r_1 \leftarrow \mathsf{Rand}(l)$; $r_2 \leftarrow \mathsf{Rand}(1)$; $r_3 \leftarrow \mathsf{Rand}(1)$
2: $r_1' \leftarrow (A_2^i - r_1) + A_3^i$    ▷ Mask two arithmetic shares
3: $r_2' \leftarrow (r_2 \oplus g_2^i) \oplus g_3^i$; $r_3' \leftarrow (r_3 \oplus g_5^i) \oplus g_6^i$
4: for $a_1 := 0$ to $2^l - 1$ do
5:     $a_1' \leftarrow a_1 -_l r_1'$    ▷ $a_1' = r_1 \implies a = A_2^i + A_3^i$
6:     for $a_2 := 0$ to 1, $a_3 := 0$ to 1 do
7:         $a_2' \leftarrow a_2 \oplus r_2'$; $a_3' \leftarrow a_3 \oplus r_3'$
8:         $(B_1^i || d_2^i) \leftarrow F_3(A_1^i + a_3 + (a_2 +_l a_1))$
9:         $d_1^i \leftarrow \mathsf{Carry}(a_1, r_1') \oplus \mathsf{Carry}(a_1, -a_2)$
10:        $x_1^i \leftarrow (B_1 \oplus x_2^i) \oplus x_3^i$    ▷ Apply Boolean masking to the result
11:        $e_1^i \leftarrow (d_1^i \oplus h_1^i) \oplus h_2^i$    ▷ Apply masking to the carries
12:        $e_2^i \leftarrow (d_2^i \oplus h_3^i) \oplus h_4^i$
13:        $T_1[a_1'||a_2'||a_3'], T_2[a_1'||a_2'||a_3'], T_3[a_1'||a_2'||a_3'] \leftarrow (x_1^i, e_1^i, e_2^i)$
14:    end for
15: end for
16: return $T_1[r_1||r_2||r_3], T_2[r_1||r_2||r_3], T_3[r_1||r_2||r_3]$
```

l bits are used for x_1^i and the two extra bits for carries. From $a_1' = a_1 -_l r_1'$, we have $a_1 = a_1' +_l r_1'$. However, $a_1 - r_1'$ could generate a carry, which needs to be taken care of while computing x_1^i. Hence, we compute the value of a_1 as $a_1 = (a_1' +_l r_1' +_l a_2)$, whereby a_2 is the carry from the previous word. This ensures that, for $a_1' = r_1$, we have:

$$a_1 = (r_1 +_l ((A_2^i - r_1) + A_3^i) +_l a_2) = A_2^i + A_3^i$$

The output carry d_1^i (which becomes a_2 for the next word) can occur in two scenarios: when $a_1 < r_1'$ or when $(a_1 + a_2) \geq 2^l$ (Step 9). It is easy to see that these two cases are mutually exclusive. Now, to compute x_1^i, we use a function $F_3 : \{0,1\}^{l+1} \to \{0,1\}^{l+1}$, which is defined as:

$$F_3(a) = a \bmod 2^l \,||\, \mathsf{Carry}(2^l, a)$$

We call F_3 with $(A_1^i + a_3 + (a_2 +_l a_1))$, where a_3 represents the second carry. In this case, the first part returned by F_3 gives x^i. The second part corresponds to the second carry, which becomes a_3 in the next word[5]. Namely, when $a_1' = r_1$,

$$\begin{aligned} F_3(A_1^i + a_3 + (a_2 +_l a_1)) &= (A_1^i + a_3 + (a_2 +_l a_1)) \bmod 2^l \,|| \\ \mathsf{Carry}(2^l, (A_1^i + a_3 + (a_2 +_l a_1))) &= (x^i + a_3) \bmod 2^l \,||\, \mathsf{Carry}(2^l, (x^i + a_3)) \end{aligned}$$

[5] Note that, even though x^i and the carries are computed in clear, they are hidden amongst $2^{l+2} - 1$ dummy computations, which is the basis for the security of the original algorithm of Rivain, Dottax and Prouff [12].

Once we have x^i and the carries d_1^i, d_2^i, we can simply apply boolean masks on them to obtain x_1^i and the masked carries (in Step 10, 11 and 12). Finally, we specify the full solution for conversion from arithmetic to Boolean masking in Algorithm 6. It is similar to Algorithm 4, except that the Boolean shares and arithmetic shares are interchanged.

Algorithm 6. Sec2OA→B

```
Input: Arithmetic shares: $A_1 = x - A_2 - A_3, A_2, A_3$
Output: Boolean shares: $x_1, x_2, x_3$ so that $x = x_1 \oplus x_2 \oplus x_3$
1: $g_1^0, g_2^0, g_3^0, g_4^0, g_5^0, g_6^0 \leftarrow 0$          ▷ Initially carry is zero
2: for $i := 0$ to $p - 1$ do
3:     $x_2^i, x_3^i \leftarrow \mathsf{Rand}(l)$          ▷ Generate output masks randomly
4:     $h_1^i, h_2^i, h_3^i, h_4^i \leftarrow \mathsf{Rand}(1)$
5:     $(x_1^i, g_1^{i+1}, g_4^{i+1}) \leftarrow \mathsf{Sec2OA{\to}B\_Word}\ ((A_j^i)_{1 \le j \le 3}, (g_j^i)_{1 \le j \le 6}, x_2^i, x_3^i, (h_j^i)_{1 \le j \le 4})$
6:     $g_2^{i+1}, g_3^{i+1}, g_5^{i+1}, g_6^{i+1} \leftarrow h_1^i, h_2^i, h_3^i, h_4^i$
7: end for
8: return $x_1, x_2, x_3$
```

Theorem 2. *Algorithm 6 is secure against second-order DPA.*

Proof. The proof of Algorithm 6 can be obtained similar to Algorithm 4 and is omitted. □

4 Efficient First-Order Secure Masked Addition

This paper considers the general problem of dealing with arithmetic operations on Boolean masks. Till now, we solved this problem by converting the Boolean masks to arithmetic masks. The basic idea is that, once we have the arithmetic masks, we can perform any arithmetic operation directly and then convert the result back to Boolean masks. But there also exists an alternative approach to the original problem, namely to perform an arithmetic operation (e.g. addition) directly on Boolean masks. This idea was first studied for first-order masking in [1] and then detailed in [7]. In this section, we provide a more efficient method using lookup tables based on the conversion technique by Debraize [5].

The problem here can be described as follows: we are given Boolean shares of two n-bit sensitive variables x: x_1, r and y: y_1, s. We need to compute z_1 so that $z_1 \oplus r \oplus s = x + y$, without any first-order leakage of x and y. To achieve this, we follow the same divide-and-conquer strategy we used in Section 2 and Section 3. Namely, we divide n-bit shares into p words of l-bit each and perform addition on the words independently. Furthermore, our method also masks the carry from word i to word $i + 1$. The addition of each word is carried out with the help of a lookup table, which can be reused for all the words[6].

[6] We use different tables in the case of second-order masking, but we can re-use the table for first-order masking.

Our method to generate the lookup table is given in Algorithm 7. It creates a table with 2^{2l+1} entries, each requiring $l+1$ bit of memory. Here, $2l$ bits are used for two l-bit inputs x^i, y^i and one bit for the input carry. The output consists of l-bit z^i and one bit carry. We run through all the possible 2^{2l+1} values and store the masked value of sum and carry in the lookup table. Note that the inputs masks are t_1, t_2 and ρ (carry), and out masks are t_1 and ρ (carry).

Algorithm 7. GenTable

```
Input:
Output: Table $T$, $t_1$, $t_2$, $\rho$
1: $t_1, t_2 \leftarrow \mathsf{Rand}(l)$; $\rho \leftarrow \mathsf{Rand}(1)$
2: for $A = 0$ to $2^l - 1$ do
3:     for $B = 0$ to $2^l - 1$ do
4:         $T[\rho||A||B] \leftarrow ((A \oplus t_1) + (B \oplus t_2)) \oplus (\rho||t_1)$
5:         $T[\rho \oplus 1||A||B] \leftarrow ((A \oplus t_1) + (B \oplus t_2) + 1) \oplus (\rho||t_1)$
6:     end for
7: end for
8: return $T, t_1, t_2, \rho$
```

The full technique to compute addition on Boolean shares is given in Algorithm 8. Initially, the carry is zero, which is masked with the carry mask ρ from Algorithm 7. We distinguish between carry and no-carry cases as follows: when $\beta = \rho$, then there is no carry; otherwise, $\beta = \rho \oplus 1$. Before accessing the lookup table, we change the input masks to t_1 and t_2 (step 3, 4). After we obtain the masked sum, we change the mask back to $r^i \oplus s^i$ from t_1 (step 6). Finally, the output can be obtained as $z_1 = z_1^{p-1}||\cdots||z_1^0 = (x+y) \oplus r \oplus s$.

Algorithm 8. Sec10A

```
Input: $x_1 = x \oplus r, r, y_1 = y \oplus s, s, T, t_1, t_2, \rho$
Output: $z_1 = (x + y) \oplus r \oplus s$
1: $\beta \leftarrow \rho$
2: for $i = 0$ to $p - 1$ do
3:     $x_1^i \leftarrow x_1^i \oplus t_1 \oplus r^i$
4:     $y_1^i \leftarrow y_1^i \oplus t_2 \oplus s^i$
5:     $(\beta||z_1^i) \leftarrow T[\beta||x_1^i||y_1^i]$
6:     $z_1^i \leftarrow (z_1^i \oplus r^i \oplus s^i) \oplus (t_1)$
7: end for
8: return $z_1$
```

Lemma 2. *Algorithm 8 is secure against first-order DPA.*

Proof. It is easy to see that the distribution of all the intermediate variables in Algorithm 8 is independent of the sensitive variables x and y. Consequently, the proof is straightforward. □

5 Implementation Results

We implemented all the proposed algorithms in ANSI C and executed them on a 32-bit ARM microcontroller. The results are summarized in Table 2. We used three different word sizes (namely $\ell = 1, 2, 4$) for the second-order conversion algorithm and a word size of $\ell = 4$ for first-order masked addition[7]. In order to compare our results with that of existing techniques, we also implemented the Coron-Großschädl-Vadnala (CGV) method [3] for second-order conversion and the Karroumi-Richard-Joye (KRJ) method [7] for first-order secure addition. As expected, the improvement in case of the second-order conversion algorithms is significant due to the reduction of the number of shares from five to three. We notice that the conversion algorithms perform best when $\ell = 2$. Our Boolean to arithmetic conversion algorithm with negligible memory requirements (between 8 and 64 bytes) is some 86% faster than the CGV method. Similarly, our arithmetic to Boolean conversion algorithm improves the running time by 83%, with equivalent memory footprint. On the other hand, we improve the performance of the first-order algorithms by roughly 20%.

Table 2. Implementation results for $n = 32$ on a 32-bit microcontroller. The column Time specifies the running time in clock cycles, rand gives the number of calls to the random number generator function, while column ℓ and Memory refer to the word size and memory (in bytes) required for the table-based algorithms.

Algorithm	ℓ	Time	Memory	rand
second-order conversion				
Algorithm 4	1	12186	8	226
Algorithm 4	2	11030	16	114
Algorithm 4	4	19244	64	58
Algorithm 6	1	10557	8	226
Algorithm 6	2	9059	16	114
Algorithm 6	4	15370	64	58
CGV $A \to B$ [3]	-	54060	-	484
CGV $B \to A$ [3]	-	81005	-	822
first-order addition				
KRJ addition [7]	-	371	-	1
Algorithm 8	4	294	512	3

To study the implications of our new techniques in practice, we applied them to secure HMAC-SHA-1. The achieved results are summarized in Table 3. We can see that, in the best case scenario (i.e. $\ell = 2$), our new algorithms perform 85% better than the existing approaches. In the case of first-order masking, the improvement amounts to roughly 6%, taking into account the pre-computation time spent on the generation of the table.

[7] We observed that, for $\ell < 4$, the algorithm from [7] performs better than ours.

Table 3. Running time (in thousands of clock cycles) and penalty factor compared to the unmasked HMAC-SHA-1 implementation

Algorithm	ℓ	Time	PF
HMAC-SHA-1	-	104	1
second-order conversion			
Algorithm 4, 6	1	9715	95
Algorithm 4, 6	2	8917	85
Algorithm 4, 6	4	15329	147
CGV [3]	-	62051	596
first-order addition			
KRJ addition [7]	-	328	3.1
Algorithm 8	4	308	2.9

6 Conclusions

In this paper, we presented new time-memory trade-off solutions for conversion between Boolean and arithmetic masking for first and second order. In the case of second-order conversion, we reduced the number of required shares from five to three compared to the CGV method. We demonstrated that, with negligible memory consumption (up to 64 bytes), we can improve the performance of the existing algorithms by up to 85%.

An open research problem is to find a way to perform additions on Boolean shares directly that is secure against attacks of second order. We can not apply the generic method of [12] in this case since the S-box is not balanced. Such an S-box would require an input of size $2l + 1$ bits (i.e. l bits for each of the two arguments to add and one bit for input carry) and output the $(l + 1)$-bit sum including the carry. For this function to be balanced, each of the 2^{l+1} possible outputs must be an image of exactly 2^l elements. However, this is not the case and, consequently, a second-order attack can be mounted. Finding a solution to this problem could further improve the efficiency of second-order masking.

References

1. Y.-J. Beak and M.-J. Noh. Differetial power attack and masking method. *Trends in Mathematics*, 8(1):53–67, June 2005.
2. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, 1999.
3. J. Coron, J. Großschädl, and P. K. Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, pages 188–205, 2014.
4. J.-S. Coron and A. Tchulkine. A new algorithm for switching from arithmetic to Boolean masking. In *CHES*, pages 89–97, 2003.

5. B. Debraize. Efficient and provably secure methods for switching from arithmetic to Boolean masking. In *CHES*, pages 107–121, 2012.
6. L. Goubin. A sound method for switching between Boolean and arithmetic masking. In *CHES*, pages 3–15, 2001.
7. M. Karroumi, B. Richard, and M. Joye. Addition with blinded operands. In *COSADE*, 2014.
8. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.
9. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.
10. O. Neiße and J. Pulkus. Switching blindings with a view towards IDEA. In *CHES*, pages 230–239, 2004.
11. E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical second-order DPA attacks for masked smart card implementations of block ciphers. In *CT-RSA*, pages 192–207, 2006.
12. M. Rivain, E. Dottax, and E. Prouff. Block ciphers implementations provably secure against second order side channel analysis. In *FSE*, pages 127–143, 2008.
13. P. K. Vadnala and J. Großschädl. Algorithms for switching between boolean and arithmetic masking of second order. In *SPACE*, pages 95–110, 2013.