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 becomes necessary to convert masks from one form to the other. There exist a few algorithms for mask conversion that are secure against first-order DPA attacks, but they can not be generalized to higher orders. At CHES 2014, Coron-Großchädl-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 this paper, we improve the algorithms for second-order conversion using lookup tables so that only three shares instead of five are needed, which is the minimal number for second-order security. Furthermore, we also improve the first-order secure addition algorithm due to Karroumi-Richard-Joye, again using lookup tables. We prove the security of all proposed algorithms on the basis of well established assumptions and models. Finally, we provide experimental evidence of our improved mask conversion applied to HMAC-SHA-1. Our results show that the proposed algorithms improve the execution time by 85% and do so with negligible 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 '90s, there has been a massive body of research on finding effective countermeasures to thwart these attacks, in particular the highly effective Differential Power Analysis (DPA) attacks [9,10]. 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" countermeasures such as the insertion of random delays or the shuffling

of operations. On the other hand, the most important 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]. 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 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. Indeed, a straightforward DPA attack may yield x_1 or x_2 (both of which appear as random numbers to the attacker), but knowledge of x_1 alone or x_2 alone should not reveal any information about the sensitive variable x.

One of the main challenges when applying masking to a block cipher is to implement the round functions in 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 may introduce significant overheads for the non-linear parts of a cipher, i.e. the S-boxes. In addition, all round functions need to be executed twice (namely for x_1 and x_2, where $x = x_1 \oplus x_2$), which entails a further performance penalty. Moreover a basic masking scheme as described above is vulnerable to a so-called second-order DPA attack, in which an attacker combines information from two leakage points. Namely, he exploits the side-channel leakage originating from x_1 and x_2 simultaneously [12]. 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 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 processed independently. In this way, it is guaranteed that the joint distribution 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 a sufficient amount of noise, the effort for attacking a higher-order masked implementation increases exponentially with d [2].

Depending on the algorithmic properties of a block cipher, a masking scheme may 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 (resp. plaintext). Examples of symmetric algorithms that involve arithmetic as well as Boolean operations include the hash functions SHA-1, SHA-2, Blake and Skein, various ARX-based block ciphers (e.g. XTEA, Threefish) and all four finalists of the eStream software portfolio. Given the widespread deployment of these cryptosystems in various kinds of applications (including ones that require sophisticated countermeasures against DPA), it is important to develop efficient techniques for conversion between Boolean and arithmetic masks. However, almost all conversion methods reported in the literature are only applicable to first-order masking [4–6, 8, 11]. Two exceptions are the second-order conversion scheme of Vadnala and Großschädl [14] and recent

higher-order conversion scheme by Coron-Großschädl-Vadnala [3]. We briefly recall these schemes below.

Vadnala-Großschädl Scheme [14]. The foundation of this scheme is the generic second-order countermeasure proposed by Rivain, Dottax and Prouff [13]. The algorithm to compute a second-order secure masked S-box output from a second-order secure masked input due to Rivain et al. is recalled below.

Algorithm 1 Sec2O-masking

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: $\quad a' \leftarrow a \oplus r'$
5: $\quad T[a'] \leftarrow ((S(x_1 \oplus a) \oplus y_1) \oplus y_2)$
6: **end for**
7: **return** $T[r]$

In Algorithm 1, a lookup table is created 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 will not be able to guess the point in time at which the actual value of x is being leaked. In [13], the authors proved the security of the algorithm by proving that no pair of intermediate variables leaks any sensitive value.

In second-order Boolean to arithmetic conversion, the goal is to solve the problem of computing arithmetic shares from a given set of Boolean shares without introducing any second or first-order leakage. To achieve second-order DPA resistance, we need three Boolean shares x_1, x_2, x_3 so that the sensitive variable x is given by $x = x_1 \oplus x_2 \oplus x_3$. 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 modifies the masked lookup table in Algorithm 1 to store the value $((x_1 \oplus a) - A_2) - A_3$ instead of masked S-box output. The rest of the algorithm is similar to the original. They followed the same approach in the case of arithmetic to Boolean conversion.

Coron-Großschädl-Vadnala Scheme [3]. At CHES 2014, Coron, Großschädl and Vadnala proposed conversion algorithms 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 as a subroutine to derive algorithms for converting between Boolean and arithmetic masking again with the complexity $\mathcal{O}(d^2 \cdot n)$.

Our Contributions. The generic solution proposed by Coron-Großschädl-Vadnala requires 5 shares to protect against second-order attacks, which entails a significant overhead in terms of the required amount of random numbers and execution time. Though the algorithms proposed by Vadnala and Großschädl secure against second-order attacks require only 3 shares, they become infeasible for implementation on low-resource devices like smart cards for $n > 10$ (the additions are performed modulo 2^n), as we require lookup table of size 2^n.

In this paper, we propose second-order secure conversion algorithms, which overcome the above limitations and can be easily applied to cryptographic constructions with arbitrary n, *e.g.* HMAC-SHA-1 with $n = 32$. The proposed algorithms use only 3 shares and are significantly faster than the state-of-the-art. Our solution follows the basic concept of Vadnala and Großschädl (which, in turn, is based on the work of Rivain and Prouff), but uses a divide and conquer approach so as to prevent the lookup tables becoming prohibitively large. In the case of Boolean to arithmetic conversion, we divide the Boolean shares into words of $l \leq 8$ bits and compute the words of the corresponding arithmetic shares independently in a word-by-word fashion. Part of this procedure is to handle 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 demonstrate that this can be achieved in an efficient and secure fashion by using separate look-up tables for the carries. Furthermore, we prove the security of our conversion schemes in the same model as in [13]. Using the similar techniques, we show that the efficiency of first-order secure masked addition due to Karroumi, Richard and Joye [8] can also be improved.

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 here is that we divide 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

The problem here is, we are given three Boolean shares x_1, x_2, x_3 so that the sensitive variable x is obtained by $x = x_1 \oplus x_2 \oplus x_3$. 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. 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 [14] using the approach followed by Rivain et al. in [13]. But as mentioned earlier, their scheme becomes infeasible to be used in practice when $n > 10$, as it requires a lookup table of size 2^n. To obtain a solution for $n > 10$, we use divide and conquer approach. That is, we divide each share into p words of l bits each, and compute $(A_1^i)_{(0 \leq i \leq p-1)}$ independently, where $A_1 = A_1^{p-1}|| \cdots ||A_1^0$. In this case, we also need to handle the carries from word i to word $i+1$. These carries in turn also need to be protected by masking, which can leak information about the sensitive variable otherwise. In the following, we present 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 in computing A_1^i and output carries i.e., carries raised while computing A_1^i. As computing A_1^i involves two subtractions, there will be two output carries from each word i, which become input carries for the word $i+1$. For the first word, input carries are initialized to 0, i.e., $c_1^0 = 0, c_2^0 = 0$. We compute A_1^i from the input x^i and carries c_1^i, c_2^i as follows:

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

Here the operation $a -_l b$ represents $(a-b) \mod 2^l$. 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 operation $(a-b)$. Note that each of the carry computation involves two subtractions: one with the input carry and the other with one of the random shares i.e., A_2^i or A_3^i. In the simplest case, a subtraction $a - b$ produces a carry if $a < b$. However, in our case, we have operations of the form $(a -_l c) -_l b$ where both 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 $a < b$. But when $c = 1$, we have to consider another case, namely $a < c$, which can only happen if $a = 0$ and $c = 1$. In this special case, the difference $a -_l c$ becomes $2^l - 1$, thereby producing a carry that needs to be handled as well. However, there won't be a carry from the second subtraction as $b \leq 2^{l-1}$. Namely, the carries from these two cases are mutually exclusive; hence the output carry is set to one when either of them produces a carry as shown in (1) and (2). For simplicity, we define functions $F_1 : \{0,1\}^{l+1} \rightarrow \{0,1\}^{l+1}$, $F_2 : \{0,1\}^{2l} \rightarrow \{0,1\}^{l+1}$ 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 the word 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 as follows:

$$(B_1^i||d_1^i) = F_1(x^i, c_1^i)$$

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

where $A_1^i = B_4^i$ and $c_1^{i+1} = d_1^i \oplus d_2^i, c_2^{i+1} = d_3^i \oplus d_4^i$. According to [13], the S-box must be balanced for their scheme to be secure [1]. In our case, the function F_1 plays the same role and is balanced. Hence, the security guarantee is preserved. We first present non-randomized version of our solution below for simplicity.

Algorithm 2 Insecure 20B→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 now is to obtain the same result without leaking 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 fist give the algorithm to compute the result for one word i.e. A_1^i securely; then we use this as a subroutine to compute A_1. Our solution given in Algorithm 3 uses the similar technique used by Rivain et al in [13] (Recalled in Algorithm 1) in combination with Algorithm 2. Algorithm 3 takes 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 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 for storing A_1^i and two bits for the two carries correspondingly. As we can see, the rest of the algorithm is similar to the original algorithm except for handling 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 for storing the value and the carries so that the security proof can be easily obtained as in [13].

Algorithm 3 Sec20B→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: $\quad a_1' \leftarrow a_1 \oplus r_1'$; $a_2' \leftarrow a_2 \oplus r_2'$; $a_3' \leftarrow a_3 \oplus r_3'$
5: $\quad (B_1^i, d_1^i) \leftarrow F_1((x_1^i \oplus a_1), (g_1^i \oplus a_2))$
6: $\quad (B_2^i, d_2^i) \leftarrow F_2(B_1^i, A_2^i)$
7: $\quad (B_3^i, d_3^i) \leftarrow F_1(B_2^i, (g_4^i \oplus a_3))$
8: $\quad (B_4^i, d_4^i) \leftarrow F_2(B_3^i, A_3^i)$
9: $\quad e_1^i \leftarrow ((d_1^i \oplus h_1^i) \oplus d_2^i) \oplus h_2^i$
10: $\quad e_2^i \leftarrow ((d_3^i \oplus h_3^i) \oplus d_4^i) \oplus h_4^i$
11: $\quad (T_1[a_1'||a_2'||a_3'], T_2[a_1'||a_2'||a_3'], T_3[a_1'||a_2'||a_3']) \leftarrow (B_4^i, e_1^i, e_2^i)$
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 method 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. Hence, the three shares for both the carries are set to zero (Step 1). Here, $g_1^0 = g_2^0 = g_3^0 = c_1^0 = 0$ 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 Sec20B→A_Word (Algorithm 3) [3]. Note here 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 required is $(2^{l+2} \cdot (l+2))$ bits.

Algorithm 4 Sec20B→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: $\quad A_2^i, A_3^i \leftarrow \mathsf{Rand}(l)$ ▷ Generate output masks randomly
4: $\quad h_1^i, h_2^i, h_3^i, h_4^i \leftarrow \mathsf{Rand}(1)$
5: $\quad (A_1^i, g_1^{i+1}, g_4^{i+1}) \leftarrow$ Sec20B→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: $\quad g_2^{i+1}, g_3^{i+1}, g_5^{i+1}, g_6^{i+1} \leftarrow h_1^i, h_2^i, h_3^i, h_4^i$
7: **end for**
8: **return** A_1, A_2, A_3

[3] Every call to the function Sec20B→A_Word creates a new table and used for that particular word only. Hence unlike the original method in [13], we don't reuse the table.

2.2 Security Analysis.

For an algorithm to be secure against second-order attacks, no pair of the intermediate variables appearing in the algorithm should jointly leak the sensitive variable. In [13] 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 similar method to prove the security of Algorithm 3. We then prove the security of Algorithm 4 using induction.

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

Proof. We list all the intermediate variables used in Algorithm 1 and Algorithm 3 in Table 1. The intermediate variables computed using 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 case of Algorithm 1.

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

Table 1. Comparison between Intermediate variables used in Algorithm 1 and Algorithm 3

□

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

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

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 secure from Lemma 1. Now assume that the algorithm is secure for $p = n$. Let E_i be the set that represents the collection of all the intermediate variables corresponding to the word i. Then, according to the induction hypothesis, $\{E_1, \cdots E_n\} \times \{E_1, \cdots E_n\}$ 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: $\{E_{n+1} \times E_{n+1}\}$, $\{E_1, \cdots E_n\} \times \{E_1, \cdots E_n\}$, $\{E_{n+1}\} \times \{E_1, \cdots E_n\}$. 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. All the intermediate variables in E_{n+1} fall into two categories: the variables that are generated randomly and are independent of any variables in $\{E_1, \cdots E_n\}$; and the variables which are a function of one or more of the following: (x^{n+1}), (c_1^{n+1}) and (c_2^{n+1}). Any pair of the intermediate variables involving the first category are independent of the sensitive variables by definition and the first-order resistance of $\{E_1, \cdots E_n\}$. The carry shares for the word $n+1$: $(c_i^{n+1})_{1 \leq i \leq 2}$ are computed from the word n. Hence the security of $(c_i^{n+1})_{1 \leq i \leq 3} \times \{E_1, \cdots E_n\}$ is already established in $\{E_n\} \times \{E_1, \cdots E_n\}$. It is easy to 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, 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 solve this problem, we follow the same strategy as in Section 2.1. We 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 to first or second-order DPA. We use the following approach: we first obtain a method to convert a single arithmetic share word; then we apply this procedure recursively to all the words. For each word, we have to deal with two carries corresponding to the two additions, i.e., the carry from the addition of the 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 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: $\quad a_1' \leftarrow a_1 -_l r_1'$ ▷ $a_1' = r_1 \implies a = A_2^i + A_3^i$
6: $\quad$ **for** $a_2 := 0$ to 1, $a_3 := 0$ to 1 **do**
7: $\quad\quad a_2' \leftarrow a_2 \oplus r_2'$; $a_3' \leftarrow a_3 \oplus r_3'$
8: $\quad\quad (B_1^i || d_2^i) \leftarrow F_3(A_1^i + a_3 + (a_2 +_l a_1))$
9: $\quad\quad d_1^i \leftarrow \mathsf{Carry}(a_1, r_1') \oplus \mathsf{Carry}(a_1, -a_2)$
10: $\quad\quad x_1^i \leftarrow (B_1 \oplus x_2^i) \oplus x_3^i$ ▷ Apply Boolean masking to the result
11: $\quad\quad e_1^i \leftarrow (d_1^i \oplus h_1^i) \oplus h_2^i$ ▷ Apply masking to the carries
12: $\quad\quad e_2^i \leftarrow (d_2^i \oplus h_3^i) \oplus h_4^i$
13: $\quad\quad T_1[a_1'||a_2'||a_3'], T_2[a_1'||a_2'||a_3'], T_3[a_1'||a_2'||a_3'] \leftarrow (x_1^i, e_1^i, e_2^i)$
14: $\quad$ **end for**
15: **end for**
16: **return** $T_1[r_1||r_2||r_3], T_2[r_1||r_2||r_3], T_3[r_1||r_2||r_3]$

Algorithm 5 gives the solution for converting one word of Boolean shares to corresponding arithmetic shares. We again use the technique from Algorithm 1 as in Algorithm 3. As 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' (Step 2) and a_1' (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 l bits are used for x_1^i and the extra two bits for the 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 while computing x_1^i. Hence, we compute the value of a_1 as: $a_1 = (a_1' +_l r_1' +_l a_2)$, where 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 out 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 function $F_3 : \{0,1\}^{l+1} \rightarrow \{0,1\}^{l+1}$, which is defined as:

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

We then 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, and the second part

corresponds to the second carry which becomes a_3 for the next word [5]. Namely, when $a'_1 = r_1$,

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

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 (Steps 10, 11 and 12).

Finally we give the full algorithm to convert 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 Sec20A→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: $\quad x_2^i, x_3^i \leftarrow \mathsf{Rand}(l)$ ▷ Generate output masks randomly
4: $\quad h_1^i, h_2^i, h_3^i, h_4^i \leftarrow \mathsf{Rand}(1)$
5: $\quad (x_1^i, g_1^{i+1}, g_4^{i+1}) \leftarrow$ Sec20A→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: $\quad g_2^{i+1}, g_3^{i+1}, g_5^{i+1}, g_6^{i+1} \leftarrow h_1^i, h_2^i, h_3^i, h_4^i$
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

As already established, this paper focuses on the problem of dealing with arithmetic operations on Boolean masks. Till now, we solved this problem by converting the Boolean masks to arithmetic masks. The idea is that once we have the arithmetic masks, we can perform arithmetic operations directly and then convert the result back to Boolean masks. But there also exist an alternative solution to the original problem i.e., devising a solution to perform addition directly on the Boolean masks. This idea was first studied with respect to first-order masking in [1] and precised in [8]. In this section, we provide a more efficient method using lookup tables based on the conversion method proposed by Debraize [5].

[5] Note here that even though x^i and the carries are computed in clear, they are hidden among $2^{l+2} - 1$ dummy computations, which is the main basis for Rivain et al's original algorithm.

The problem here is: 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. We follow the similar divide and conquer approach 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. Moreover, our method also masks the carry from word i to word $i + 1$. The addition of each word is performed using a lookup table, which can be reused for all the words. [6]

Our algorithm to generate the lookup table is given in Algorithm 7. It creates a table of 2^{2l+1} entries, where each entry requires $l+1$ bit 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 here that the inputs masks are t_1, t_2 and ρ (carry); the out masks are t_1 and ρ (carry).

Algorithm 7 GenTable

Input:
Output: Table T, t_1, t_2, ρ
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, ρ

The full algorithm 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 differentiate between carry and no carry cases as follows: if $\beta = \rho$ 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 $z_1 = z_1^{p-1}|| \cdots ||z_1^0 = (x + y) \oplus r \oplus s$.

[6] In case of second-order masking, we use different tables. But we can reuse the table for first-order masking.

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: $\quad x_1^i \leftarrow x_1^i \oplus t_1 \oplus r^i$
4: $\quad y_1^i \leftarrow y_1^i \oplus t_2 \oplus s^i$
5: $\quad (\beta||z_1^i) \leftarrow T[\beta||x_1^i||y_1^i]$
6: $\quad z_1^i \leftarrow (z_1^i \oplus r^i \oplus s^i) \oplus (t_1)$
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. Hence the proof is straightforward. □

5 Implementation Results

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 $A \rightarrow B$ [3]	-	54060	-	484
CGV $B \rightarrow A$ [3]	-	81005	-	822
first-order addition				
KRJ addition [8]	-	371	-	1
Algorithm 8	4	294	512	3

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

We implemented all the proposed algorithms on a 32-bit ARM microcontroller. The results are summarized in Table 2. We used three different word sizes ($\ell = 1, 2, 4$) for second-order conversion algorithms and word size $\ell = 4$

for first-order masked addition.[7] To compare our results with the existing techniques, we also implemented CGV method [3] for second-order conversion and KRJ method [8] for first-order secure addition. As expected, the improvement in case of second-order conversion algorithms is significant due to the decrease in the number of shares from five to three. We can see that the conversion algorithms give best results for $\ell = 2$. Our Boolean to arithmetic conversion algorithm with negligible memory requirements (around 8 to 64 bytes) is roughly 86% faster than the CGV algorithm. Similarly, our arithmetic to Boolean conversion algorithm improve the running time by 83%, with equivalent memory requirements. On the other hand, we improve the performance of first-order algorithms by roughly 20%.

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 [8]	-	328	3.1
Algorithm 8	4	308	2.9

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

To study the implications of our new algorithms on practical implementations, we applied these techniques to HMAC-SHA-1. The corresponding 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 algorithms. In case of first-order masking, the improvement is around 6% including the precomputation time required to create the table.

6 Conclusion

In this paper, we proposed time-memory trade-off solutions for conversion between Boolean and arithmetic masking for first and second-order. For second-order conversion, we improved the number of shares required from 5 to 3 when compared to CGV method. We have shown that with negligible memory overhead (around 16 bytes), we can improve the performance of the existing algorithms up to 85%.

One open issue is to find a way to perform addition on Boolean shares directly, which is secure against attacks of second-order. We can not apply the generic

[7] We observed that for $\ell < 4$ *KRJ* algorithm perform better than ours.

method of [13] in this case because the S-box is not balanced. Such an S-box would require input of size $2l + 1$-bit (l-bit for each of the two arguments to addition 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 true and hence we can mount a second-order attack. Finding a solution to this problem could further improve the performance of second-order masking.

References

1. Y. Beak and M.-J. Noh. Differetial power attack and masking method. *Trends in Mathematics*, 8:1–15, 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. Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003.
8. M. Karroumi, B. Richard, and M. Joye. Addition with blinded operands. In *COSADE*, 2014.
9. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.
10. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.
11. O. Neiße and J. Pulkus. Switching blindings with a view towards IDEA. In *CHES*, pages 230–239, 2004.
12. 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.
13. M. Rivain, E. Dottax, and E. Prouff. Block ciphers implementations provably secure against second order side channel analysis. In *FSE*, pages 127–143, 2008.
14. P. K. Vadnala and J. Großschädl. Algorithms for switching between boolean and arithmetic masking of second order. In *SPACE*, pages 95–110, 2013.