Algorithms for Switching between Boolean and Arithmetic Masking of Second Order

Praveen Kumar Vadnala and Johann Großschädl

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

Abstract. Masking is a widely-used countermeasure to thwart Differential Power Analysis (DPA) attacks, which, depending on the involved operations, can be either Boolean, arithmetic, or multiplicative. When used to protect a cryptographic algorithm that performs both Boolean and arithmetic operations, it is necessary to change the masks from one form to the other in order to be able to unmask the secret value at the end of the algorithm. To date, known techniques for conversion between Boolean and arithmetic masking can only resist first-order DPA. This paper presents the first solution to the problem of converting between Boolean and arithmetic masking of second order. To set the context, we show that a straightforward extension of first-order conversion schemes to second order is not possible. Then, we introduce two algorithms to convert from Boolean to arithmetic masking based on the second-order provably secure S-box output computation method proposed by Rivain et al (FSE 2008). The same can be used to obtain second-order secure arithmetic to Boolean masking. We prove the security of our conversion algorithms using similar arguments as Rivain et al. Finally, we provide implementation results of the algorithms on three different platforms.

Keywords: Differential power analysis, Second-order DPA, Arithmetic masking, Boolean Masking, Provably secure masking.

1 Introduction

Side-channel cryptanalysis exploits information leakage from the execution of a concrete implementation of a cryptographic algorithm [12]. Therefore, this kind of attack is methodically very different from "traditional" cryptanalysis, which essentially focuses on finding secret keys in a black box model given only pairs of plaintexts and ciphertexts. The first form of side-channel attacks discussed in the literature are timing attacks, i.e. attacks exploiting measurable differences in the execution time of a cryptographic algorithm or a specific operation it is based upon [13,11]. A more sophisticated class of attacks are power analysis attacks, which aim to deduce information about the secret key from the power consumption of the device while a certain operation is carried out [15]. A third

B. Gierlichs, S. Guilley, and D. Mukhopadhyay (Eds.): SPACE 2013, LNCS 8204, pp. 95–110, 2013.

class are electromagnetic (EM) attacks, which exploit the relationship between secret data and EM emanations produced by the device [1].

Power analysis attacks received extensive attention from the cryptographic community ever since Kocher and his team published them for the first time in their seminal paper *Differential Power Analysis* [14]. These attacks allow one to recover the full secret key with relatively few measurements and it is close to impossible to totally circumvent them with current semiconductor technologies [15]. While Simple Power Analysis (SPA) attacks try to recover a secret value by directly "comparing" the power measurements with the corresponding operations, Differential Power Analysis (DPA) attacks are much more sophisticated and aim to reveal a secret value by applying statistical techniques on multiple measurements of the same operation. Masking is a widely-used countermeasure to thwart DPA attacks, which involves using random variables, called masks, to reduce the correlation between the secret value and the obtained leakage [2]. In order to circumvent first-order DPA attacks [15] that involve a single operation using masking, we divide the secret value into two shares: a mask generated randomly and the masked value of the secret. However, this approach can still be attacked via a second-order DPA involving two operations corresponding to the two shares of the secret [16,19]. In general, a d-th order masking scheme is vulnerable to a $(d+1)$-th order DPA attack involving all $d+1$ shares of the secret. These attacks are called Higher-Order DPA attacks (HODPA).

Depending on the operation to be protected, a masking scheme can either be Boolean (using logical XOR), arithmetic (using modular addition/subtraction) or multiplicative (using modular multiplication). To successfully "unmask" the variable at the end of the algorithm, one has to track the change of the masked secret value during the execution of the algorithm. If an algorithm contains two of the three afore-mentioned operations (i.e. XOR, modular addition/subtraction, modular multiplication), the masks have to be converted from one form to the other, keeping this conversion free from any leakage. Goubin introduced secure methods to convert between first-order Boolean and arithmetic masks in [10]. Coron and Tchulkine improved the method for switching from arithmetic to Boolean masking in [5], which was recently further improved by Debraize in [6]. While solutions exist for converting between arithmetic and multiplicative masking of higher order [7,8,9], the conversion between Boolean and arithmetic masking is currently limited to first order. We aim to fill this gap by presenting algorithms to switch between Boolean and arithmetic masks of second order.

In the context of second-order masking, a sensitive variable x is represented by three shares; these are $x_1 = x \oplus x_2 \oplus x_3$, x_2 and x_3 in the case of Boolean masking, and $A_1 = x - A_2 - A_3$, A_2 and A_3 for arithmetic masking [15]. The problem is to convert between Boolean and arithmetic masking without introducing any first-order or second-order leakage. Unfortunately, it is not possible to extend the existing first-order secure conversion schemes to second order as we will show later. Therefore, we employ techniques proposed by Rivain et al in [21] to arrive at the first solution for converting second-order masks from one form to the other. In [21], the authors describe two provably secure methods to

compute S-box outputs without first or second-order leakage. By applying these methods, we present a total of four algorithms, two for each conversion type.

The rest of the paper is organized as follows. We review some of the existing solutions for a first-order conversion as well as the paper of Rivain et al in Section 2. Then, we show that a straightforward application of Goubin's method is insecure for second-order conversion in Section 3. We introduce our algorithms for Boolean to arithmetic conversion in Section 4. The algorithms to convert in the opposite direction (i.e. arithmetic to Boolean) can be derived similarly and are described in Section 5. We prove the security of our algorithms in Section 6 for a device leaking in the Hamming weight model. Section 7 summarizes some implementation results and, finally, Section 8 concludes the paper.

2 Previous Work

This section provides an overview of recent results that will be used later in the paper. We first summarize existing methods for switching between arithmetic and Boolean masking of first order. Then, we describe two techniques proposed by Rivain et al at FSE 2008 to compute S-box outputs secure against second-order DPA attacks [21].

2.1 Securing Conversions against First-Order DPA

The first solution to the problem of Boolean to arithmetic mask conversion was presented by Messerges in [17], which was later proven to be insecure by Coron and Goubin in [4]. At CHES 2001, Goubin proposed an algorithm for switching between Boolean and arithmetic masks secure against first-order DPA attacks [10]. His algorithm to convert from Boolean to arithmetic masking is based on the following fact: for $I = \{0, 1, \ldots, 2^n - 1\}$ with $n \geq 1$ and $x' \in I$, the function $\phi_{x'}(r) : I \to I$ defined as $\phi_{x'}(r) = (x' \oplus r) - r \bmod 2^n$ is affine over the field GF(2). Therefore, the function $\psi_{x'} = \phi_{x'} \oplus \phi_{x'}(0)$ is linear over GF(2) and, as a result, $x = x' \oplus r$ with Boolean shares (x', r) can be converted to the equivalent arithmetic shares (A, r) for any random γ via the following relation:

$$\begin{aligned} A &= \phi_{x'}(r) = \psi_{x'}(r) \oplus x' \\ &= \psi_{x'}(\gamma) \oplus \psi_{x'}(r \oplus \gamma) \oplus x' \\ &= [(x' \oplus \gamma) - \gamma] \oplus x' \oplus [(x' \oplus (r \oplus \gamma)) - (r \oplus \gamma)] \end{aligned}$$

This method is highly efficient, requiring only a constant number of elementary operations. Goubin's arithmetic to Boolean conversion is based on the following fact: $x' = (A + r) \oplus r$ is equivalent to $x' = A \oplus u_{n-1}$, where

$$\begin{cases} u_0 = 0 \\ u_{i+1} = 2[u_i \wedge (A \oplus r) \oplus (A \wedge r)] \ \forall\ i \geq 0 \end{cases}$$

Unfortunately, this method is far less efficient since the number of operations is proportional to the size of the registers.

Algorithm 1. Secure second-order S-box output computation: First variant

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)-bit S-box lookup function S
Output: Masked S-box output: $S(x) \oplus y_1 \oplus y_2$
1: Randomly generate n-bit number r
2: $r' \leftarrow (r \oplus x_2) \oplus x_3$
3: **for** a **from** 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 2003, Coron and Tchulkine proposed a new algorithm for conversion from arithmetic to Boolean masking [5]. To convert two n-bit ($n = p \cdot k$) arithmetic shares A and R with $x = A + R \bmod n$ into two Boolean shares x' and R such that $x = x' \oplus R$, the algorithm works on each k-bit word independently. Therefore, A and R are divided into p words of k bits: $A = A_1 \,||\, A_2 \,||\, \ldots \,||\, A_{p-1}$ and $R = R_1 \,||\, R_2 \,||\, \ldots \,||\, R_{p-1}$. Now, the Boolean share equivalent to the i-th word of the arithmetic share A_i is computed as $x'_i = (A_i + R_i + c_{i+1}) \oplus R_i$, where c_{i+1} is the carry bit produced from the previous k-bit word. The algorithm precomputes two small tables of 2^k entries each, and reuses them several times in the course of the conversion. The first table serves to convert each arithmetic-share word independently to the equivalent Boolean-share word. This table contains the entries $(z + r) \oplus r$ for all possible values of $z \in [0, 2^k - 1]$ and the random value $r \in [0, 2^k - 1]$. The correct value for the i-th word can be obtained when $z = (A_i - r + R_i) + c_{i+1}$. The second table is used to mask the carry that needs to be passed from one word to the next-higher. Even though this increases the memory requirements, the conversion time is reduced significantly. Neiße and Pulkus [18] modified the algorithm so as to reduce the memory needed to store the tables. At CHES 2012, Debraize discovered a bug in the Coron-Tchulkine algorithm and devised a new variant that is also more efficient [6].

2.2 Generic Countermeasure against Second-Order DPA

At FSE 2008, Rivain et al proposed two algorithms to protect the computation of S-box outputs against second-order attacks [21]. Given three input shares of a secret value x, namely $x_1 = x \oplus x_2 \oplus x_3$, x_2, and x_3 (which are all in $\mathbb{F}_{2^n}$) and two output shares $y_1, y_2 \in \mathbb{F}_{2^m}$ along with an (n, m) S-box lookup function S, they compute the third share y_3 such that $y_1 \oplus y_2 \oplus y_3 = S(x)$. Hence, we have $y_3 = S(x) \oplus y_1 \oplus y_2$. The algorithms compute $(S(x_1 \oplus a) \oplus y_1) \oplus y_2$ for all possible values of a (i.e. $0 \leq a \leq 2^{n-1}$), among which the correct value can be obtained when $a = x_2 \oplus x_3$. We recall these algorithms below.

Algorithm 1 uses a table of 2^n entries to store $(S(x_1 \oplus a) \oplus y_1) \oplus y_2$ for all possible values of a. Here, the value $(x_2 \oplus x_3)$ is masked via a random variable r, the result of which is assigned to r'. Thereafter, the entry corresponding to

Algorithm 2. Secure second-order S-box output computation: Second variant

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)-bit S-box lookup function S
Output: Masked S-box output: $S(x) \oplus y_1 \oplus y_2$
1: Randomly generate one bit b
2: **for** a **from** 0 **to** $2^n - 1$ **do**
3: $cmp \leftarrow compare_b(x_2 \oplus a, x_3)$
4: $R_{cmp} \leftarrow ((S(x_1 \oplus a) \oplus y_1) \oplus y_2)$
5: **end for**
6: **return** R_b

$(S(x_1 \oplus a) \oplus y_1) \oplus y_2$ will be stored at location $a' = a \oplus r'$. The correct value of the third share y_3 can be retrieved by accessing the value stored in the table at location $T[r]$. As $r = a'$, the value of a becomes $a = r \oplus r' = x_2 \oplus x_3$, thus yielding the desired result.

The security of Algorithm 1 can be proven by showing that it is impossible to recover x by combining any pair of intermediate variables computed by the algorithm. We refer the interested reader to Section 3.1 in [21] for the complete proof. In Section 6, we will use the same approach to prove the security of our conversion techniques. Algorithm 1 requires a table of 2^n words (each having a length of m bits) in RAM, and is, therefore, not suitable for low-cost devices. To overcome this issue, Rivain et al introduced another algorithm consuming less memory at the expense of executing more operations.

Algorithm 2 specifies the second solution proposed by Rivain et al in [21] to securely compute an S-box output. In this variant, they use a function called $compare_b(x, y)$, which returns b if $x = y$ and $\bar{b}$ otherwise. A first-order secure implementation of $compare_b$ is necessary to guarantee the security of the algorithm. To this end, Rivain et al [21] presented a method for implementing the $compare_b$ function, shown in Algorithm 3. The secure S-box computation works as follows: First, a random bit b is generated, which is one of the inputs to the $compare_b$ function. Then, for each possible value of a, the algorithm computes $(S(x_1 \oplus a) \oplus y_1) \oplus y_2$, which will be written to either R_b or $R_{\bar{b}}$, depending on the actual output of the $compare_b$ function. The inputs to the $compare_b$ function are $x_2 \oplus a$ and x_3. When $a = x_2 \oplus x_3$, $compare_b(x_2 \oplus a, x_3)$ returns b, thus the result is stored in R_b. In all other cases, the returned value is $\bar{b}$, so the result is stored in the register $R_{\bar{b}}$. At the end of the algorithm, the value stored in R_b is $S(x) \oplus y_1 \oplus y_2$, which is exactly what we wanted to achieve.

Note that Algorithm 2 needs only 2^n bits in RAM, namely for the function $compare_b$. Thus, it requires m times less memory than Algorithm 1, though the execution time is longer due to multiple calls to the $compare_b$ function.

3 Applying Goubin's Conversion to Second Order

In this section, we demonstrate that a straightforward application of Goubin's conversion technique [10] to the second order does not work. Assume we have

Algorithm 3. Computation of the $compare_b$ function

Input: x, y, b, n
Output: b if $x = y$, $\bar{b}$ otherwise
1: $r_3 \leftarrow rand(n)$
2: $T[0 : 2^n - 1] \leftarrow \bar{b}, \bar{b}, \ldots, \bar{b}$
3: $T[r_3] \leftarrow b$
4: **return** $T[(x \oplus r_3) \oplus y]$

three Boolean shares x_1, x_2, x_3 whereby $x = x_1 \oplus x_2 \oplus x_3$. We need to find the arithmetic shares A_1, A_2, and A_3 such that $x = A_1 + A_2 + A_3 \bmod 2^n$. To do so, we can iteratively compute A_1, A_2, A_3 as follows:

$$\begin{aligned} x &= A_1 + (x_2 \oplus x_3) \\ x &= A_1 + A_2 + x_3 \\ A_1 &= x - (x_2 \oplus x_3) \\ A_2 &= (x_2 \oplus x_3) - x_3 \\ A_3 &= x_3 \end{aligned}$$

Based on the above, we can compute A_1 in the following way:

$$\begin{aligned} A_1 &= x_1 \oplus (x_2 \oplus x_3) - (x_2 \oplus x_3) = \phi_{x_1}(x_2 \oplus x_3) \\ &= \phi_{x_1}(x_2) \oplus \phi_{x_1}(x_3) \oplus x_1. \end{aligned}$$

One could try to securely compute $\phi_{x_1}(x_2)$ and $\phi_{x_1}(x_3)$ as follows:

$$\begin{aligned} \phi_{x_1}(x_2) &= \phi_{x_1}(x_2 \oplus r) \oplus \phi_{x_1}(r) \oplus x_1 \\ \phi_{x_1}(x_3) &= \phi_{x_1}(x_3 \oplus r) \oplus \phi_{x_1}(r) \oplus x_1. \end{aligned}$$

This means,

$$\begin{aligned} A_1 &= \phi_{x_1}(x_2 \oplus r) \oplus \phi_{x_1}(r) \oplus \phi_{x_1}(x_3 \oplus r) \oplus \phi_{x_1}(r) \oplus x_1 \\ &= \phi_{x_1}(x_2 \oplus r) \oplus \phi_{x_1}(x_3 \oplus r) \oplus x_1 \\ &= ((x_1 \oplus x_2 \oplus r) - (x_2 \oplus r)) \oplus ((x_1 \oplus x_3 \oplus r) - (x_3 \oplus r)) \oplus x_1. \end{aligned}$$

But we can combine the leakages from $x_1 \oplus x_2 \oplus r$ and $x_3 \oplus r$ to get $x_1 \oplus x_2 \oplus x_3 = x$, inducing a second order attack. Similarly, we can combine the leakages from $x_1 \oplus x_3 \oplus r$ and $x_2 \oplus r$ to get $x_1 \oplus x_2 \oplus x_3 = x$. Now, let us consider the case where we use a different random r_i for computing each $\phi_{x_i}(x_j)$, i.e.

$$\begin{aligned} \phi_{x_1}(x_2) &= \phi_{x_1}(x_2 \oplus r_1) \oplus \phi_{x_1}(r_1) \oplus x_1 \\ \phi_{x_1}(x_3) &= \phi_{x_1}(x_3 \oplus r_2) \oplus \phi_{x_1}(r_2) \oplus x_1. \end{aligned}$$

This means,

$$A_1 = \phi_{x_1}(x_2 \oplus r_1) \oplus \phi_{x_1}(r_1) \oplus \phi_{x_1}(x_3 \oplus r_2) \oplus \phi_{x_1}(r_2) \oplus x_1.$$

Now, when computing A_1, regardless of what sequence we choose, we would be leaking the secret x while combining the results. For example, assume that we calculate according to the following sequence:

$$\begin{aligned}\phi_{x_1}(r_1) \oplus \phi_{x_1}(x_3 \oplus r_2) \oplus \phi_{x_1}(r_2) &= \phi_{x_1}(x_3 \oplus r_1) \\ &= ((x_1 \oplus x_3 \oplus r_1) - (x_3 \oplus r_1))\end{aligned}$$

Let us further assume that $\phi_{x_1}(x_2 \oplus r_1)$ is calculated as follows:

$$\phi_{x_1}(x_2 \oplus r_1) = ((x_1 \oplus x_2 \oplus r_1) - (x_2 \oplus r_1))$$

Then, we can combine the leakages from $x_1 \oplus x_2 \oplus r_1$ and $x_3 \oplus r_1$ to find the value of x. From this, we conclude that the straightforward application of the method of Goubin does not work for second order.

4 Second-Order Boolean to Arithmetic Masking

This section addresses the problem of securely converting second-order Boolean shares to the corresponding arithmetic shares without any second-order or first-order leakage. To start with, we are given three Boolean shares x_1, x_2, x_3 such that $x = x_1 \oplus x_2 \oplus x_3$ where x is a sensitive variable. 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 information exploitable in a first or second-order DPA attack. We propose two algorithms to achieve this goal; one is based on Algorithm 1 and the second on Algorithm 2. Both of our algorithms use the secure S-box output computation of Rivain et al [21], which simplifies the security proofs.

The first of our variants is given in Algorithm 4; we devised this conversion by modifying Algorithm 1 appropriately. The algorithm generates two shares A_2, A_3 randomly from $[0, 2^n - 1]$ and computes the third share via the relation $A_1 = (x - A_2) - A_3$. The aim of Algorithm 1 was to output $S(x) \oplus y_1 \oplus y_2$ as result. Hence, it computed $(S(x_1 \oplus a) \oplus y_1) \oplus y_2$ for every possible value of the variable a from 0 to $2^n - 1$, and then obtained the correct value for the case $a = x_2 \oplus x_3$. But here, our aim is to compute $(x - A_2) - A_3$, which requires us to modify the table entries to $((x_1 \oplus a) - A_2) - A_3$ so that we can obtain the correct value when $a = x_2 \oplus x_3$. Note that the subtractions are modulo 2^n.

Correctness: When $a' = r$, a becomes $r \oplus r' = x_2 \oplus x_3$. Thus, $T[a'] = T[r] = ((((x_1 \oplus x_2) \oplus x_3) - A_2) - A_3) = (x - A_2) - A_3$, from which follows that $A_1 = (x - A_2) - A_3$ and finally $x = A_1 + A_2 + A_3$.

We devised Algorithm 5 by appropriately adapting Algorithm 2. Again, we first compute the value of $((x_1 \oplus a) - A_2) - A_3$ for all possible values of a and store the result in R_b or $R_{\bar{b}}$, depending on the return value of $compare_b$. When $a = x_2 \oplus x_3$, the value of $x_2 \oplus a$ and x_3 become equal, hence $compare_b$ returns b. Consequently, the correct value of $A_1 = (x - A_2) - A_3$ is stored in R_b. In all other cases (i.e. $a \neq x_2 \oplus x_3$), the value $((x_1 \oplus a) - A_2) - A_3$ is stored in $R_{\bar{b}}$.

Algorithm 4. Boolean to arithmetic conversion of 2nd order: First variant

Input: Boolean shares: $x_1 = x \oplus x_2 \oplus x_3$, x_2, x_3
Output: Arithmetic shares: $A_1 = (x - A_2) - A_3$, A_2, A_3
1: Randomly generate n-bit numbers r, A_2, A_3
2: $r' \leftarrow (r \oplus x_2) \oplus x_3$
3: **for** a **from** 0 **to** $2^n - 1$ **do**
4: $\quad a' \leftarrow a \oplus r'$
5: $\quad T[a'] \leftarrow ((x_1 \oplus a) - A_2) - A_3$
6: **end for**
7: $A_1 = T[r]$
8: **return** A_1, A_2, A_3

Algorithm 5. Boolean to arithmetic conversion of 2nd order: Second variant

Input: Boolean shares: $x_1 = x \oplus x_2 \oplus x_3$, x_2, x_3
Output: Arithmetic shares: $A_1 = (x - A_2) - A_3$, A_2, A_3
1: Randomly generate n-bit numbers A_2, A_3
2: Randomly generate one bit b
3: **for** a **from** 0 **to** $2^n - 1$ **do**
4: $\quad cmp \leftarrow compare_b(x_2 \oplus a, x_3)$
5: $\quad R_{cmp} \leftarrow ((x_1 \oplus a) - A_2) - A_3$
6: **end for**
7: $A_1 = R_b$
8: **return** A_1, A_2, A_3

5 Second-Order Arithmetic to Boolean Masking

In this section, we briefly introduce two algorithms to securely convert second-order arithmetic shares into the "corresponding" Boolean shares, whereby the conversion does not introduce any second-order (or first-order) leakage. More precisely, given three arithmetic shares A_1, A_2, A_3 of a sensitive variable x such that $x = A_1 + A_2 + A_3$, both of these algorithms compute the Boolean shares x_1, x_2, x_3 satisfying $x = x_1 \oplus x_2 \oplus x_3$ without second or first-order leakage.

Algorithm 6 employs a lookup table similar to Algorithm 4. Here, the value of r' is $(A_2 - r) + A_3$, where r is a random value in the range $[0, 2^n - 1]$. The table entries corresponding to $a' = a - r'$ are now $((A1 + a) \oplus x_2) \oplus x_3$ instead of $((x_1 \oplus a) - A_2) - A_3$. Similar to Algorithm 4, the two shares x_2 and x_3 are generated randomly from $[0, 2^n - 1]$, while the third share x_1 is $T[r]$.

Correctness: When $a' = r$, a becomes $r + r' = A_2 + A_3$. Thus, $T[a'] = T[r] = ((((A_1 + A_2) + A_3) \oplus x_2) \oplus x_3) = (x \oplus x_2) \oplus x_3$, from which follows that $x_1 = (x \oplus x_2) \oplus x_3$ and finally $x = x_1 \oplus x_2 \oplus x_3$.

Algorithm 7 shows the other method to convert arithmetic shares of second order to "equivalent" Boolean shares. Among the three Boolean shares, x_2 and x_3 are generated randomly within the range $[0, 2^{n-1}]$. One of the two registers R_0, R_1 serves to store the correct value of x_1 and the other is used for storing

Algorithm 6. Arithmetic to Boolean conversion of 2nd order: First variant

Input: Arithmetic shares: $A_1 = (x - A_2) - A_3,\ A_2,\ A_3$
Output: Boolean shares: $x_1 = x \oplus x_2 \oplus x_3,\ x_2,\ x_3$

Randomly generate n-bit numbers $r,\ x_2,\ x_3$
$r' \leftarrow (A_2 - r) + A_3$
for a **from** 0 **to** $2^n - 1$ **do**
    $a' \leftarrow a - r'$
    $T[a'] \leftarrow ((A_1 + a) \oplus x_2) \oplus x_3$
end for
$x_1 = T[r]$
return x_1, x_2, x_3

Algorithm 7. Arithmetic to Boolean conversion of 2nd order: Second variant

Input: Arithmetic shares: $A_1 = (x - A_2) - A_3,\ A_2,\ A_3$
Output: Boolean shares: $x_1 = x \oplus x_2 \oplus x_3,\ x_2,\ x_3$

Randomly generate n-bit numbers $x_2,\ x_3$
Randomly generate one bit b
for a **from** 0 **to** $2^n - 1$ **do**
    $cmp \leftarrow compare_b(a - A_2, A_3)$
    $R_{cmp} \leftarrow ((A_1 + a) \oplus x_2) \oplus x_3$
end for
$x_1 = R_b$
return x_1, x_2, x_3

the incorrect value. The compare instruction compares $(a - A_2)$ with $A3$; when they are equal, $compare_b$ returns b and, thus, the result is stored in R_b. In this case, the result is the correct value of x_1, which means $((A_1 + A_2 + A_3) \oplus x_2) \oplus x_3 = (x \oplus x_2) \oplus x_3$. Otherwise, the result is incorrect and stored in R'_b.

6 Security Analysis

We first review the security model introduced in [21]. Then, based on the same model, we present the security proofs of all our four algorithms against second-order attacks. We assume that the device leaks in the Hamming weight model (i.e. the leakage is proportional to the Hamming weight of the data processed on the device). Below we summarize some basic definitions and results that are used in the proofs for quick reference (partly taken from [21]).

- *Sensitive variable:* An intermediate variable obtained by applying a certain function on a known value (e.g. plaintext) and the secret key.
- *Primitive random variable:* An intermediate variable generated by a random number generator with uniform distribution.
- *Functional dependence:* If an intermediate variable is obtained by applying a discrete function on some other variable X, then it is said to be functionally dependent on X. Otherwise, it is functionally independent.

- *Statistical dependence:* If the statistical distribution of an intermediate variable varies according to some other variable X, then it is said to be statistically dependent on X. Otherwise, it is statistically independent.
- Functional independence implies statistical independence, but the converse is false.
- In second-order DPA, leakages from at most two intermediate variables are allowed to be exploited simultaneously. So, for a cryptographic algorithm to be called second-order secure, it is important that every pair of intermediate variables is statistically independent of any sensitive variable.
- A set of intermediate variables is statistically independent from a variable X if, and only if, all intermediate variables belonging to the set are statistically independent of X.
- Given two sets A and B, $A \times B$ is statistically independent from a variable X if, and only if, all pairs in $A \times B$ are statistically independent of X.
- **Lemma 1.** *For statistically independent random variables X and Y, it holds that for every measurable function f, $f(X)$ is statistically independent of the variable Y.*
- **Lemma 2.** *Let X and Y be statistically independent random variables where Y is uniformly distributed, and Z a variable that is statistically independent of Y and functionally independent of X. In this case, the pair $(Z, X \oplus Y)$ is statistically independent of X.*

Limitations of the Security Proofs: The algorithms in [21], though proven secure against "standard" DPA attacks, suffer from two problems. Firstly, the algorithm not using table computations, i.e. Algorithm 2, is only secure in the Hamming weight model. At COSADE 2012, Coron et al have shown that this algorithm is *not* secure when the device leaks in the Hamming distance model [3]. They also demonstrated that a straightforward conversion of a proof from the Hamming weight model to Hamming distance model by initializing the bus (resp. register) with 0 before every write operation has a second-order flaw. As a consequence, the proof of Algorithm 2 from [21] is not valid anymore in the Hamming distance model. The conversion of a security proof from one leakage model to another is still an open issue. Since Algorithm 5 and Algorithm 7 are similar to Algorithm 2, they suffer from said limitation too. However, a solution to the conversion problem for Rivain et al's generic countermeasure for secure S-box computation would, of course, also apply to our algorithms.

Secondly, some current developments in side-channel cryptanalysis indicate that masking might succumb to a so-called *horizontal side-channel attack* (see e.g. [20,22]). By targeting the table generation phase of a masking scheme, an attacker may succeed to recover the secret key when the signal-to-noise ratio is low. However, these attacks are generic in the sense that they are applicable to essentially any practical masking scheme; our algorithms are no exception. The problem of securely generating the masked table is still an open challenge and requires further attention. Any solution to this problem can be readily applied to our algorithms as well to help them resist horizontal attacks. Hence, despite these limitations, our algorithms are still practically relevant.

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

Proof. We follow the notation of [21] for the sake of simplicity. Each intermediate variable of the algorithm can be seen as a result of applying the function I_j on the loop index a. Assume that $I_{index} = I_{index}(a)$ for $0 \leq a \leq 2^n - 1$ and $I = \bigcup_{index=0}^{num} I_{index}$, whereby *num* specifies the total number of intermediate variables. We list all intermediate variables used in Algorithm 4 in Table 1.

Table 1. Intermediate variables used in Algorithm 4

index	I_{index}
1	x_2
2	x_3
3	A_2
4	A_3
5	r
6	$r \oplus x_2$
7	$r \oplus x_2 \oplus x_3$
8	a
9	$a \oplus r \oplus x_2 \oplus x_3$
10	$x \oplus x_2 \oplus x_3$
11	$x \oplus x_2 \oplus x_3 \oplus a$
12	$(x \oplus x_2 \oplus x_3 \oplus a) - A_2$
13	$((x \oplus x_2 \oplus x_3 \oplus a) - A_2) - A_3$
14	$(x - A_2) - A_3$

Table 2. Intermediate variables used in Algorithm 5

index	I_{index}
1	x_2
2	x_3
3	A_2
4	A_3
5	b
6	a
7	$x_2 \oplus a$
8	$\delta_0(x_2 \oplus a \oplus x_3) \oplus b$
9	$x \oplus x_2 \oplus x_3$
10	$x \oplus x_2 \oplus x_3 \oplus a$
11	$(x \oplus x_2 \oplus x_3 \oplus a) - A_2$
12	$((x \oplus x_2 \oplus x_3 \oplus a) - A_2) - A_3$
13	$(x - A_2) - A_3$

We recall that, for an algorithm to be secure against second-order DPA, no pair of intermediate variables should be statistically dependent on a sensitive variable. Consequently, we need to prove that $I \times I$ is statistically independent of x. To simplify matters, we divide the set of intermediate variables into three subsets: $E_1 = I_1 \cup I_2 \cup \ldots \cup I_9$, $E_2 = I_{10} \cup I_{11} \cup \ldots \cup I_{13}$, and $E_3 = I_{14}$. The objective now is to prove that all possible combinations of these three sets are statistically independent of x.

1. $\mathbf{E_1 \times E_1}$**:** All the intermediate variables in E_1 are functionally independent of x. Hence, $E_1 \times E_1$ is statistically independent of x.
2. $\mathbf{E_2 \times E_2}$**:** It can be seen that $I_{10} = x \oplus x_2 \oplus x_3$ is statistically independent of x. As all elements in $E_2 \times E_2$ are functions of I_{10}, it can be inferred that $E_2 \times E_2$ is statistically independent of x by applying Lemma 2.
3. $\mathbf{E_3 \times E_3}$**:** It is also straightforward to see that $E_3 \times E_3$ is statistically independent of x since $x - (A_2 - A_3)$ is statistically independent of x.
4. $\mathbf{E_1 \times E_2}$**:** E_1 is statistically independent of $(x_2 \oplus x_3)$ and functionally independent of x. According to Lemma 2, $E_1 \times \{x \oplus x_2 \oplus x_3\}$ is statistically independent of x. Hence, according to Lemma 1, $E_1 \times E_2$ is statistically independent of x.

5. **$\mathbf{E_1} \times \mathbf{E_3}$:** Since E_1 is statistically independent of $A_2 - A_3$, the combination $E_1 \times \{x - (A_2 - A_3)\}$ (i.e. $E_1 \times E_3$) is statistically independent of x. As the pair $(x \oplus x_2 \oplus x_3, (x - A_2) - A_3)$ is statistically independent of x, it holds that $(I_{10} \cup I_{11} \cup I_{12}) \times E_3$ is statistically independent of x because all these can be expressed as a function of $(x \oplus x_2 \oplus x_3, (x - A_2) - A_3)$.
6. **$\mathbf{E_2} \times \mathbf{E_3}$:** We need to prove that $I_{13} \times E_3$ is statistically independent of x to establish that $E_2 \times E_3$ is statistically independent of x. Suppose that $v_1 = (x - A_2) - A_3$ and $v_2 = (x \oplus x_2 \oplus x_3 \oplus a)$. It can be seen that v_1 and v_2 are statistically independent of each other as well as of variable x. We can write $I_{13} \times E_3$ as $\{v_1 + v_2 - x\} \times v_1$, which is statistically independent of x.

From all this it can be concluded that Proposition 1 holds. □

Proposition 2. *Algorithm 5 is secure against second-order DPA.*

Proof. Assume that the Boolean function $\delta_0(x) = 0$ only when $x = 0$. So, the function $compare_b(x, y)$ can be represented as $\delta_0(x \oplus y) \oplus b$. For Algorithm 5 to be secure, it is important that the $compare_b$ function is implemented in a way which prevents any first-order leakage on $compare(x, y)$. One such method is recalled in Algorithm 3 (originally proposed in [21]) and we can construct the proof on this method. The intermediate variables appearing in Algorithm 5 are given in Table 2. It can be easily seen that nearly all intermediate variables are identical to those in Algorithm 4. Thus, we can prove the security of Algorithm 5 by using the same arguments as given in the proof of Proposition 1. □

Table 3. Intermediate variables used in Algorithm 6

index	I_{index}
1	x_2
2	x_3
3	A_2
4	A_3
5	r
6	$A_2 - r$
7	$A_2 - r + A_3$
8	a
9	$a - A_2 + r - A_3$
10	$x - A_2 - A_3$
11	$x - A_2 - A_3 + a$
12	$(x - A_2 - A_3 + a) \oplus x_2$
13	$(x - A_2 - A_3 + a) \oplus x_2 \oplus x_3$
14	$x \oplus x_2 \oplus x_3$

Table 4. Intermediate variables used in Algorithm 7

index	I_{index}
1	x_2
2	x_3
3	A_2
4	A_3
5	b
6	a
7	$a - A_2$
8	$\delta_0((a - A_2) \oplus A_3) \oplus b$
9	$x - A_2 - A_3$
10	$x - A_2 - A_3 + a$
11	$(x - A_2 - A_3 + a) \oplus x_2$
12	$(x - A_2 - A_3 + a) \oplus x_2 \oplus x_3$
13	$x \oplus x_2 \oplus x_3$

Proposition 3. *Algorithm 6 is secure against second-order DPA.*

Proof. Table 3 lists all intermediate variables appearing in Algorithm 6. We can use similar arguments as in Proposition 1 to prove that no pair of intermediate variables is statistically dependent on x. □

Proposition 4. *Algorithm 7 is secure against second-order DPA.*

Proof. We show all intermediate variables that appear in Algorithm 7 in Table 4. Again, the security proof can be developed similar to the one of Proposition 1, namely by showing that no pair of the intermediate variables is statistically dependent on x. □

7 Implementation Results

We implemented all algorithms from Section 4 and Section 5 in Matlab and in ANSI C, whereby we only considered the simplest case of converting between 8-bit masks. The Matlab implementation served as a reference for the C implementation so that we could easily verify the correctness of the latter. We tested our four algorithm individually using 100,000 pseudo-random inputs and found that all of them produce the correct result in all cases. The C implementation generates the random numbers with help of the `rand` function of the standard C library[1]. Although this is sufficient for testing, a real-world implementation would need pseudo-random numbers of better quality. Furthermore, it should be noted that we developed all our implementations primarily for the purpose of having a proof of concept rather than achieving high performance. The implementations can be further optimized, which means the results we report in this section should be seen as upper bounds of the execution time. Also, if the conversions are used in real-world applications, one needs to take care that the compilation process respects the flow of intermediate variables assumed in the security proofs given in Section 6. If this is not the case, it becomes necessary to develop an assembly language implementation.

The implementation of Algorithm 4 and Algorithm 6 is straightforward; we create a table of 256 bytes and, for each of the 256 possible values of a, store the corresponding entry in a byte. The indexing of the table is done via a' and the correct value of the third share is retrieved by accessing the table entry corresponding to r. An optimized implementation of Algorithm 5 and Algorithm 7 has to perform the $compare_b$ function as efficiently as possible. We used the following approach to implement this function. First, we create an array of 32 bytes and initialize all the bits to $\bar{b}$. We treat the array as a collection of 256 bits, all initialized to $\bar{b}$. Then, for a random value r_3, we set the corresponding bit position in the array to b. Each call to $compare_b$ is now simply replaced by a single look-up into the array. To give an example, $compare_b(x, y)$ is obtained by retrieving the value at the bit position $(x \oplus r_3) \oplus y$. The index of the byte containing the bit can be obtained through a logical right-shift operation. The bit itself can be extracted from the byte via a shift operation too.

[1] On an 8-bit AVR processor, e.g. ATmega128, calling the `rand` function takes around 800 clock cycles when using the `avr-libc` library of the WinAVR tool suite.

Table 5. Implementation results on an 8, 16 and 32-bit platform

Algorithm	Cycles	RAM (bytes)
8-bit architecture (AVR)		
Algorithm 4	5769	256
Algorithm 5	6742	32
Algorithm 6	5769	256
Algorithm 7	6742	32
16-bit architecture (MSP)		
Algorithm 4	4983	256
Algorithm 5	16706	32
Algorithm 6	4983	256
Algorithm 7	16706	32
32-bit architecture (ARM)		
Algorithm 4	793	256
Algorithm 5	1087	32
Algorithm 6	793	256
Algorithm 7	1087	32

In order to assess the execution time of the algorithms, we compiled them for the 32-bit ARM platform as well as the 8-bit AVR platform and performed simulations with AVR Studio. In addition, we evaluated our four algorithms on a low-power 16-bit micro-controller, the TI MSP430, with the help of a cycle-accurate instruction-set simulator. Table 5 illustrates the simulation results we obtained on these three platforms. The second column of Table 5 specifies the execution time (in clock cycles) needed to convert an 8-bit mask from one form to the other. The third column gives the memory (RAM) requirements of the algorithms in number of bytes. As we can see, the two algorithms using table look-ups, i.e. Algorithm 4 and Algorithm 6, are faster than the ones which do not use tables. This is because of the additional time required to evaluate the $compare_b$ function in the case of Algorithm 5 and Algorithm 7. However, both algorithms based on the table computation method require exactly eight times more memory than their counterparts.

Note that the execution times of Algorithm 5 and Algorithm 7 obtained on the 16-bit platform are somewhat misleading. As can be seen from Table 5, the execution time on the TI MSP430 is by a factor of roughly 2.5 slower than the time on the 8-bit AVR. This can be explained through the fact that the used MSP430 processor does not have a barrel shifter, which means a shift operation by n bit positions takes n clock cycles. On the other hand, the AVR features a fast barrel shifter able to execute all shift operations in one cycle, irrespective of the shift distance.

8 Conclusions

In this paper, we addressed the practical problem of converting between second-order Boolean and arithmetic masking. We introduced two algorithms secure

against second-order attacks for each direction by applying the generic second-order secure countermeasure proposed by Rivain et al at FSE 2008. The time complexity of these algorithms is $O(2^n)$, where n is the size of the data to be converted. All algorithms are proven to be secure when the device leaks in the Hamming weight model. Our implementation results show that the algorithms without tables require eight times less memory (i.e. RAM) than the table-based algorithms, but the saving in RAM footprint comes at the expense of increased execution time. The proposed algorithms become costly when the length of the data to be converted exceeds 16 bits. As part of our future research, we aim to improve the efficiency of the conversion methods by devising algorithms with a better time-memory trade-off.

References

1. Agrawal, D., Archambeault, B., Rao, J.R., Rohatgi, P.: The EM side-channel(s). In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 29–45. Springer, Heidelberg (2003)
2. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)
3. Coron, J.-S., Giraud, C., Prouff, E., Renner, S., Rivain, M., Vadnala, P.K.: Conversion of security proofs from one leakage model to another: A new issue. In: Schindler, W., Huss, S.A. (eds.) COSADE 2012. LNCS, vol. 7275, pp. 69–81. Springer, Heidelberg (2012)
4. Coron, J.-S., Goubin, L.: On boolean and arithmetic masking against differential power analysis. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 231–237. Springer, Heidelberg (2000)
5. Coron, J.-S., Tchulkine, A.: A new algorithm for switching from arithmetic to boolean masking. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 89–97. Springer, Heidelberg (2003)
6. Debraize, B.: Efficient and provably secure methods for switching from arithmetic to boolean masking. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 107–121. Springer, Heidelberg (2012)
7. Genelle, L., Prouff, E., Quisquater, M.: Secure multiplicative masking of power functions. In: Zhou, J., Yung, M. (eds.) ACNS 2010. LNCS, vol. 6123, pp. 200–217. Springer, Heidelberg (2010)
8. Genelle, L., Prouff, E., Quisquater, M.: Montgomery's trick and fast implementation of masked AES. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 2011. LNCS, vol. 6737, pp. 153–169. Springer, Heidelberg (2011)
9. Genelle, L., Prouff, E., Quisquater, M.: Thwarting higher-order side channel analysis with additive and multiplicative maskings. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 240–255. Springer, Heidelberg (2011)
10. Goubin, L.: A sound method for switching between boolean and arithmetic masking. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 3–15. Springer, Heidelberg (2001)
11. Handschuh, H., Heys, H.M.: A timing attack on RC5. In: Tavares, S., Meijer, H. (eds.) SAC 1998. LNCS, vol. 1556, pp. 306–318. Springer, Heidelberg (1999)

12. Kelsey, J., Schneier, B., Wagner, D., Hall, C.: Side channel cryptanalysis of product ciphers. In: Quisquater, J.-J., Deswarte, Y., Meadows, C., Gollmann, D. (eds.) ESORICS 1998. LNCS, vol. 1485, pp. 97–110. Springer, Heidelberg (1998)
13. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
14. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
15. Mangard, S., Oswald, M.E., Popp, T.: Power Analysis Attacks - Revealing the Secrets of Smart Cards, vol. 54, pp. 1–337. Springer (2007)
16. Messerges, T.S.: Using second-order power analysis to attack DPA resistant software. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 238–251. Springer, Heidelberg (2000)
17. Messerges, T.S.: Securing the AES finalists against power analysis attacks. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 150–164. Springer, Heidelberg (2001)
18. Neiße, O., Pulkus, J.: Switching blindings with a view towards IDEA. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 230–239. Springer, Heidelberg (2004)
19. Oswald, E., Mangard, S., Herbst, C., Tillich, S.: Practical second-order DPA attacks for masked smart card implementations of block ciphers. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 192–207. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11605805_13
20. Pan, J., Hartog, J.I., Lu, J.: You cannot hide behind the mask: Power analysis on a provably secure s-box implementation. In: Youm, H.Y., Yung, M. (eds.) WISA 2009. LNCS, vol. 5932, pp. 178–192. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-10838-9_14
21. Rivain, M., Dottax, E., Prouff, E.: Block ciphers implementations provably secure against second order side channel analysis. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 127–143. Springer, Heidelberg (2008)
22. Tunstall, M., Whitnall, C., Oswald, E.: Masking tables—an underestimated security risk. In: Moriai, S. (ed.) Fast Software Encryption, 20th International Workshop, FSE 2013, Singapore, March 10-13. LNCS, Springer (2013) (Revised Selected Papers)