applied sciences

MDPI

Article

Lightweight Conversion from Arithmetic to Boolean Masking for Embedded IoT Processor

HanBit Kim [1], Seokhie Hong [1] and HeeSeok Kim [2,*]

[1] Graduate School of Information Security and Institute of Cyber Security & Privacy (ICSP), Korea University, Seoul 02841, Korea; luzdamoon@korea.ac.kr (H.K.); shhong@korea.ac.kr (S.H.)

[2] Department of Cyber Security, College of Science and Technology, Korea University, 2511 Sejong-Ro, Sejong 30019, Korea

* Correspondence: 80khs@korea.ac.kr

Received: 8 March 2019; Accepted: 29 March 2019; Published: 5 April 2019

Abstract: A masking method is a widely known countermeasure against side-channel attacks. To apply a masking method to cryptosystems consisting of Boolean and arithmetic operations, such as ARX (Addition, Rotation, XOR) block ciphers, a masking conversion algorithm should be used. Masking conversion algorithms can be classified into two categories: "Boolean to Arithmetic (B2A)" and "Arithmetic to Boolean (A2B)". The A2B algorithm generally requires more execution time than the B2A algorithm. Using pre-computation tables, the A2B algorithm substantially reduces its execution time, although it requires additional space in RAM. In CHES2012, B. Debraize proposed a conversion algorithm that somewhat reduced the memory cost of using pre-computation tables. However, they still require $(2^{(k+1)})$ entries of length $(k+1)$-bit where k denotes the size of the processed data. In this paper, we propose a low-memory algorithm to convert A2B masking that requires only $(2^k)(k)$-bit. Our contributions are three-fold. First, we specifically show how to reduce the pre-computation table from $(k+1)$-bit to (k)-bit, as a result, the memory use for the pre-computation table is reduced from $(2^{(k+1)})(k+1)$-bit to $(2^k)(k)$-bit. Second, we optimize the execution times of the pre-computation phase and the conversion phase, and determine that our pre-computation algorithm requires approximately half of the operations than Debraize's algorithm. The results of the 8/16/32-bit simulation show improved speed in the pre-computation phase and the conversion phase as compared to Debraize's results. Finally, we verify the security of the algorithm against side-channel attacks as well as the soundness of the proposed algorithm.

Keywords: ARX block ciphers; Arithmetic to Boolean masking; side-channel attacks

1. Introduction

Side-channel attacks exploit various types of physical leakage—including power consumption, electromagnetic radiation, running time, etc.—during the execution of a cryptographic algorithm on a real device [1–3]. Differential Power Analysis (DPA), which was introduced in 1999 by P. Kocher, is a statistical method that retrieves secret keys using information leakage from the power consumption of the device [4]. Since the introduction of DPA, the importance of implementing countermeasures against side-channel attacks has only grown.

The masking method was suggested by T. Messerges to provide theoretical security against the DPA attack [5–7]. The masking method breaks the relation between the algorithmic value (the value specified by

the algorithm) and the processed value (the value that is actually processed by the device) by using random numbers. The typical types of masking methods include Boolean masking and arithmetic masking [8]. Boolean masking uses an XOR (exclusive or) to blind values such as $x' = x \oplus r$, and arithmetic masking uses an algebraic operation such as $A = (x - r) \bmod 2^k$.

These two types of masking should be selectively used for cryptographic algorithms that consist of Boolean and arithmetic operations such as ARX (Addition, Rotation, XOR) block ciphers [9–11], cryptographic hash functions [12,13], and stream ciphers [14]. In general, Boolean operations (AND, XOR, SHIFT, etc.) and arithmetic operations (Addition, Subtraction, Multiplication, etc.) can be efficiently computed using Boolean masking and arithmetic masking, respectively; however, it is very difficult to execute arithmetic operations in Boolean masking and to execute Boolean operations in arithmetic masking. This problem can be easily solved using a masking conversion algorithm between Boolean and arithmetic masking.

1.1. Related Work

The first masking conversion algorithm to counteract first-order DPA was proposed by L. Goubin in 2001 [15]. This conversion algorithm from Boolean to arithmetic masking (B2A) has been elaborately implemented without any improvements made upon it. In contrast, the conversion algorithm from arithmetic to Boolean masking (A2B) is quite sluggish and could be improved in terms of the execution time. More specifically, the B2A algorithm has a constant execution time within the arbitrary bit size of the processed data, while the A2B algorithm requires substantially more execution time as the bit size increases.

Coron et al. tried to use memory to address this limitation [16]. They described the use of pre-computation tables to obtain large benefits in terms of execution time, although it required additional space in RAM. There have also been numerous attempts to optimize the memory use [17–21].

Recently B. Debraize proposed a conversion algorithm that offered a substantial reduction in memory for the use of pre-computation tables [22]. Briefly, this proposal iterates the conversion phase with a k-bit value using a $(k + 1)$-bit table.

1.2. Our Contribution

Masking conversion algorithms are mainly used to mask ARX ciphers. Most of the ARX ciphers, which is used in the IoT environments, are consisted of 32-bit unit operations. These ciphers aim to achieve fast encryption and a small code size. However, even if the ARX cipher is masked by Debraize's method, memory usage is still of concern. To overcome this problem, we propose an extremely low-memory algorithm that converts from arithmetic masking to Boolean masking.

The contributions of this paper are as follows:

- Reducing the memory usage: We further reduce the memory usage of the pre-computation table from the $(2^{(k+1)})(k + 1)$-bit achieved by Debraize's method [22] to $(2^k)(k)$-bit. The main idea is that the pre-computation table can be reduced by one bit based on the fact that the XOR operation is the same as the subtraction on $\mathbb{F}_2$; this is the so-called *LSB trick*. The LSB trick has been mentioned in previous papers, but we apply this trick specifically to the A2B algorithms. Furthermore, we validate this intuitive fact using mathematical induction. As a result, our proposal can save one bit for each table. Our proposal also allows for compact, optimized memory usage in the real world. For example, if k is 8-bit, our algorithm can be constructed using a `char` data type.
- Reducing the execution time: We reduce the execution time of the pre-computation phase to approximately half of that achieved by Debraize's method. When measuring the execution time of the algorithms using a pre-computation table, some researchers have focused only on the encryption parts.

However, in the real world, the execution time of the pre-computation phase cannot be disregarded. We design a new pre-computation algorithm to minimize the number of **for** loops and the number of operators in each **for** loop. Our evaluation shows that our proposal requires approximately half of the operations of Debraize's pre-computation algorithm. In additional, we simulate 8-bit, 16-bit, and 32-bit environments. On average, the results of the simulation show improvements in speed of 87% in the pre-computation phase and of 23% in the conversion, compared to that in [22].

- Verifying the security and the soundness: We verify the security against side-channel attacks and the soundness of the proposed algorithm using mathematical induction. That is, we show that all intermediate values of the proposed algorithm are random, and the soundness is reflected by the fact that the proposed algorithm always returns the correct output for an arbitrary input.

1.3. Outline of the Paper

The rest of this paper is organized as follows. In Section 2, we introduce the conversion algorithms currently used to convert between Boolean masking and arithmetic masking. Section 3 is the core of the paper, in which we present a new conversion algorithm to convert from arithmetic masking to Boolean masking. In Section 4, we analyze the security against side-channel attacks as well as the soundness of the proposed algorithm. In Section 5, we present performance metrics for our method and Debraize's method. Finally, we conclude in Section 6.

2. Background

2.1. Masking Method

Masking methods use random numbers to break the relationship between the power consumption of a crypto-device and sensitive values in a crypto-algorithm. Numerous articles have been published on a variety of masking types, such as Boolean masking [23], arithmetic masking [8], polynomial masking [24], and inner product masking [25]. Among these, Boolean masking and arithmetic masking are the most widely known masking types.

Boolean masking and arithmetic masking use the following respective formulae:

- Boolean Masking: $x' = x \oplus r_x$
- Arithmetic Masking: $A = x - r_x \bmod 2^k$

Given k-bit values; x is a sensitive value depending on the key; and x', A, and r_x are Boolean masked values, arithmetic masked values, and masking values, respectively.

Boolean and arithmetic masking are used for algorithms that consist of Boolean and arithmetic operations, such as ARX block ciphers [9–11], cryptographic hash functions [13], and stream ciphers [14]. More specifically, additions are computed on the arithmetic masking, and SHIFT, XOR, and AND are computed on Boolean masking. In this case, a conversion algorithm between arithmetic and Boolean masking is necessary, and this algorithm must be secure against DPA attacks.

The first masking conversion algorithm was introduced by T. Messerges [26], but vulnerabilities were discovered in it against DPA attacks. In CHES 2001, L. Goubin proposed an improved masking conversion algorithm that guaranteed security against DPA attacks [15]. That algorithm can convert from Boolean to arithmetic masking with only 5 XORs and 2 subtractions. This algorithm has recently been extended to a higher-order masking scheme [20,27]. According to [20,27], these countermeasures have time complexity $O(n^2 \cdot k)$ and $O(2^n)$ for n shares on the security against t-th order DPA attacks, respectively.

In contrast, conversion algorithms from arithmetic to Boolean masking have been improved through various efforts.

2.2. Arithmetic to Boolean (A2B) Masking

Conversion algorithms from arithmetic to Boolean masking can be broadly classified into two types based on the concept of hardware adders: carry look-ahead-based algorithms and ripple-carry-based algorithms. The first category operates with two values, i.e., *Generation* and *Propagation*, whether carry values are either propagated (at least one input is 1) or generated (both inputs are 1). Carry look-ahead-based algorithms can be implemented with Boolean operators such as SHIFT, AND, and XOR. A conversion algorithm of the first category, presented in FSE 2015 by Coron et al., reduces the complexity to a logarithmic scale [21]. However, it is hard to speed up this algorithm using memory storage, such as with pre-computation tables. On the other hand, algorithms in the second category iterate the task of adding the carry value generated in the lower stages to the upper stages. Ripple-carry-based algorithms are easy to optimize in terms of execution time by using memory. There have been various comparisons between the two categories. However, the performance of each algorithm and type of algorithm depends on the implementation environment, such as memory size, CPU architecture, and cryptographic algorithms.

In this paper, we propose an A2B algorithm to optimize memory storage based on the second type of algorithm.

2.3. The A2B Algorithm Based on the Ripple-Carry Adder

Ripple-carry-based A2B algorithms were first proposed by Goubin then improved by Coron, Neiße, and Debraize by using a pre-computation table. This paper briefly summarizes the point of each algorithm and explains their working principles in the following sections.

2.3.1. Goubin's A2B Algorithm

The Algorithm 1 was proposed by L. Goubin in CHES2001. We classify it as a ripple-carry-based algorithm because it computes the carry bit generated from the lower bits to the upper bits in Lines 11–16, such as the concept of the ripple-carry-based algorithms. Goubin's A2B algorithm iterates K times with the K-bit input, meaning that this algorithm is inefficient for large inputs. The algorithm is based on the following recursion formula:

Theorem 1 (Goubin's recursion formula [15])**.** *If we denote* $x' = (A + r) \oplus r$*, we also have* $x' = A \oplus u_{K-1}$*, where* u_{K-1} *is obtained from the following recursion formula:*

$$\begin{cases} u_0 = 0 & (1) \\ \forall K \geq 0, u_{K+1} = 2[u_K \wedge (A \oplus r) \oplus (A \wedge r)]. & (2) \end{cases}$$

Algorithm 1 Goubin's A2B Algorithm [15]

Require: (A, r) /* $A = x - r$ */
Ensure: (x', r) /* $x' = x \oplus r$ */
1: $\mathrm{Y} \leftarrow rand()$
2: $T \leftarrow 2\mathrm{Y}$
3: $x' \leftarrow \mathrm{Y} \oplus r$
4: $\Omega \leftarrow \mathrm{Y} \wedge x'$
5: $x' \leftarrow T \oplus A$
6: $\mathrm{Y} \leftarrow \mathrm{Y} \oplus x'$
7: $\mathrm{Y} \leftarrow \mathrm{Y} \wedge r$
8: $\Omega \leftarrow \Omega \oplus \mathrm{Y}$
9: $\mathrm{Y} \leftarrow T \wedge A$
10: $\Omega \leftarrow \Omega \oplus \mathrm{Y}$
11: **for** $i = 1$ to $K - 1$ **do**
12: $\quad \mathrm{Y} \leftarrow T \wedge r$
13: $\quad \mathrm{Y} \leftarrow \mathrm{Y} \oplus \Omega$
14: $\quad T \leftarrow T \wedge A$
15: $\quad \mathrm{Y} \leftarrow \mathrm{Y} \oplus T$
16: $\quad T \leftarrow 2\mathrm{Y}$
17: **end for**
18: $x' \leftarrow x' \oplus T$
19: **return** x', r

2.3.2. Coron's A2B Algorithm

As Goubin's A2B algorithm requires several operations that are linear in terms of the sizes of bits, it can serve as a bottleneck when implemented. Coron et al. improved the A2B algorithm by using pre-computation look-up tables. They used the pre-computation table $T[.]$ of $(k \cdot n)$-bit size to reduce the number of iterations from K to K/k. The Algorithms 2 and 3 are improved versions from [22]. Although Coron's algorithm takes time in the pre-computation phase, the execution time of its conversion phase can be reduced in comparison to that of Goubin's algorithm. In summary, Coron's algorithm has been improved in terms of execution time. However, this algorithm still has a disadvantage in terms of its memory usage (i.e., the table size is too large). Table 1 shows the intermediate value A of Algorithm 3 when loop $i = 0$. 0_k indicates 0 filled with k-bits. c denotes the carry bit at the pre-computation table $T[A]$ in Algorithm 2. As can be verified in Table 1, the initial and final forms of the **for** loop are the same, i.e., it is intuitively confirmed that the algorithm operates recursively and correctly. The main idea for handling the carry bit is to blind the carry bit with a large random number γ. To omit the step of removing the masking value γ in the conversion phase, these algorithms calculate Γ in the pre-computation phase and subtract it at the beginning of the conversion phase.

Algorithm 2 Improved Coron's Table T Generation [22]

Require: *None*
Ensure: $T[.], r, \Gamma$
1: Generate a random k-bit r and a random $((n-1) \cdot k)$-bit γ
2: $\Gamma \leftarrow \sum_{i=1}^{n-1} 2^{i \cdot k} \cdot \gamma \bmod 2^{n \cdot k}$
3: $\gamma' \leftarrow \gamma || r$
4: **for** $A = 0$ to $2^k - 1$ **do**
5: $\quad T[A] \leftarrow (A + \gamma' \bmod 2^{k \cdot n}) \oplus r$
6: **end for**
7: **return** $T[.], r, \Gamma$

Algorithm 3 Improved Coron's A2B Algorithm [22]

Require: $(A, R), T[.], r, \Gamma$ /* $x = A + R \bmod 2^{k \cdot n}$ */

/* $T[.], r, \Gamma$ generated during pre-computation phase */

Ensure: (x', R) /* $x = x' \oplus R$ */

1: $A \leftarrow A - (r||\ldots||r) - \Gamma \bmod 2^{k \cdot n}$
2: **for** $i = 0$ to $n - 1$ **do**
3: Split A into $A_h||A_l$ and R into $R_h||R_l$
(such that A_l and R_l have size k)
4: $A \leftarrow A + R_l \bmod 2^{(n-i) \cdot k}$
5: $A \leftarrow (A_h||0_k) + T[A_l] \bmod 2^{k \cdot n}$
6: $x'_i \leftarrow A_l \oplus R_l$
7: $A \leftarrow A_h$ and $R \leftarrow R_h$
8: **end for**
9: **return** $(x'_{n-1}||\ldots||x'_0) \oplus (r||\ldots||r)$

Table 1. Intermediate Values of Algorithm 3.

Line	Intermediate Value of $A(i = 0)$
1	$A - (r||\ldots||r) - \Gamma \bmod 2^{k \cdot n}$ $= x - R - (r||\ldots||r) - \sum_{i=1}^{n-1} 2^{i \cdot k} \cdot \gamma \bmod 2^{k \cdot n}$
3	$(x_h||0_k) - (R_h||0_k) - (r||\ldots||0_k) - \sum_{i=1}^{n-1} 2^{i \cdot k} \cdot \gamma$ $+x_l - R_l - r \bmod 2^{k \cdot n}$
4	$(x_h||0_k) - (R_h||0_k) - (r||\ldots||0_k) - \sum_{i=1}^{n-1} 2^{i \cdot k} \cdot \gamma$ $+x_l - r \bmod 2^{k \cdot n}$
5	$(x_h||0_k) - (R_h||0_k) - (r||\ldots||0_k) - \sum_{i=1}^{n-1} 2^{i \cdot k} \cdot \gamma$ $+T[x_l - r] \bmod 2^{k \cdot n}$ $= (x_h||0_k) - (R_h||0_k) - (r||\ldots||0_k) - \sum_{i=\mathbf{2}}^{n-1} 2^{i \cdot k} \cdot \gamma$ $+(c||0_k) + (x_l \oplus r) \bmod 2^{k \cdot n}$ $= ((x_h + c)||0_k) - (R_h||0_k) - (r||\ldots||0_k) - \sum_{i=\mathbf{2}}^{n-1} 2^{i \cdot k} \cdot \gamma$ $+(x_l \oplus r) \bmod 2^{k \cdot n}$
7	$= x_h - R_h - (r||\ldots||r) - \sum_{i=1}^{n-2} 2^{i \cdot k} \cdot \gamma \bmod 2^{(n-1) \cdot k}$

2.3.3. Neiße's A2B Algorithm

Neiße et al. suggested a new method to handle carry bits using complemented values in the conversion phase. This method was able to reduce the size of the pre-computation table. An adapted version of the Neiße-Pulkus method was proposed in [22]; however, the author claimed that this A2B algorithm is vulnerable to combination attacks that distinguish 00...00 and 11...11(-1) using the Simple Power Analysis (SPA) attack and recovers the secret values using the DPA attack. For example, let us suppose that the value z at the conversion phase in [22] is extracted as 0 by the SPA attack. In this case, the most significant bit (i.e., the carry bit) is biased, and this biased bit allows for side-channel attacks. However, this vulnerability can easily be made secure. In addition, Neiße's A2B algorithm in [22] does not work. Algorithms 4 and 5 are versions of this algorithm that are correct and secure against combination attacks due to their use of the LSB (Least Significant Bit) trick. The LSB trick is a technique that reduces pre-computation tables by one bit based on the equivalence that the Boolean masked value B_i is the same as the arithmetic masked value A_i on $\mathbb{F}_2$. The LSB trick was mentioned in Neiße's paper, and we applied this trick specifically to the A2B algorithms. However, it is not perfectly safe to change $(0, -1)$ to $(0, 1)$ against SPA. It has been

published that binary classification of "0" and "1" is possible using side-channel information [28]. Neiße's A2B algorithm could be potentially exploited via side-channel vulnerabilities.

The working principle of Neiße's algorithm is that the converted data from arithmetic to Boolean masking are either (x, R) or $(\bar{x}, \bar{R})$. Their complementary values are determined by z. Although this masking scheme does not mask the carry bit with a probability of 1/2, the carry bit is 0 or 1 with probabilities of $\frac{(2^k-1)}{2^{k+1}}$ and $\frac{(2^k+1)}{2^{k+1}}$ respectively for arbitrary z and r. In other words, the probability that the carry bit is generated for any sensitive value x is always equal to the same distribution, and so it is safe against DPA attacks. Table 2 shows the intermediate value A of Algorithm 5 when loop $i = 0$. For a value w, let $\widetilde{w}$ denote w if $z = 0$, and $\overline{w}$ if $z = 1$. These computations are based on following equations.

$$\begin{cases} \widetilde{(p+q)} = \widetilde{p} + \widetilde{q} + z & \text{(3a)} \\ \widetilde{(p-q)} = \widetilde{p} - \widetilde{q} - z & \text{(3b)} \end{cases}$$

As we can verify in Table 2, the initial and final forms of the **for** loop are the same. The algorithm operates recursively.

Algorithm 4 Neiße's Table T Generation [17]

Require: *None*
Ensure: $T[.], r, \gamma, z, Z[.]$
1: Generate a random bit z, a random k-bit r, and a random $(n \cdot k)$-bit γ
2: **for** $A = 0$ to $2^k - 1$ **do**
3: $\quad T[A] \leftarrow \{(A + r) \oplus r\} \gg 1$
4: **end for**
5: $Z[0] \leftarrow \gamma$
6: $Z[1] \leftarrow \gamma \oplus (-1 \bmod 2^{k \cdot n})$
7: **return** $T[.], r, \gamma, z, Z[.]$

Algorithm 5 Neiße's A2B Algorithm with LSB trick [17]

Require: $(A, R), T[.], r, \gamma, z, Z[.]$ /* $x = A + R \bmod 2^{k \cdot n}$ */
/* $T[.], r, \gamma, z, Z[.]$ generated during pre-computation phase */
Ensure: (x', R) /* $x = x' \oplus R$ */
1: $A \leftarrow ((A \oplus \gamma) \oplus Z[z]) - (r|| \ldots ||r) + z \bmod 2^{k \cdot n}$
2: $R \leftarrow (R \oplus \gamma) \oplus Z[z]$
3: **for** $i = 0$ to $n - 1$ **do**
4: $\quad$ Split A into $A_h||A_l$ and R into $R_h||R_l$
$\quad$ (such that A_l and R_l have size k)
5: $\quad A \leftarrow A + R_l \bmod 2^{(n-i) \cdot k}$
6: $\quad A \leftarrow (A_h||0_k) + (T[A_l]||\mathtt{LSB}_{A_l}) \bmod 2^{(n-i) \cdot k}$
7: $\quad x'_i \leftarrow A_l \oplus R_l$
8: $\quad A \leftarrow A_h$ and $R \leftarrow R_h$
9: **end for**
10: **return** $(x'_{n-1}|| \ldots ||x'_0) \oplus (r|| \ldots ||r)$

Table 2. Intermediate Values of Algorithm 5.

Line	Intermediate Value of $A(i=0)$
1	$\widetilde{A}+z-(r\|\|\ldots\|\|r) \bmod 2^{k\cdot n}$ $=\widetilde{(x-R)}+z-(r\|\|\ldots\|\|r) \bmod 2^{k\cdot n}$ $=\widetilde{x}-\widetilde{R}-(r\|\|\ldots\|\|r) \bmod 2^{k\cdot n}$ $(\because \widetilde{(a-b)}=\widetilde{a}-\widetilde{b}-z)$
4	$(\widetilde{x_h}\|\|0_k)-(\widetilde{R_h}\|\|0_k)-(r\|\|\ldots\|\|0_k)$ $+\widetilde{x_l}-\widetilde{R_l}-r \bmod 2^{k\cdot n}$
5	$(\widetilde{x_h}\|\|0_k)-(\widetilde{R_h}\|\|0_k)-(r\|\|\ldots\|\|0_k)$ $+\widetilde{x_l}-r \bmod 2^{k\cdot n}$
6	$(\widetilde{x_h}\|\|0_k)-(\widetilde{R_h}\|\|0_k)-(r\|\|\ldots\|\|0_k)$ $+T[\widetilde{x_l}-r] \bmod 2^{k\cdot n}$ $=(\widetilde{x_h}\|\|0_k)-(\widetilde{R_h}\|\|0_k)-(r\|\|\ldots\|\|0_k)+(c'\|\|0_k)$ $+(\widetilde{x_l}\oplus r) \bmod 2^{k\cdot n}$ $=(\widetilde{(x_h+c)}\|\|0_k)-(\widetilde{R_h}\|\|0_k)-(r\|\|\ldots\|\|0_k)$ $+(\widetilde{x_l}\oplus r) \bmod 2^{k\cdot n}$
8	$=\widetilde{x_h}-\widetilde{R_h}-(r\|\|\ldots\|\|r) \bmod 2^{(n-1)\cdot k}$

2.3.4. Debraize's A2B Algorithm

B. Debraize proposed an A2B algorithm which was quite optimized in terms of the memory usage of the pre-computation table as well as the execution time with security against DPA attacks. Debraize's algorithms reduce the complexity of the conversion phase by using the masked carry bit as the input to the table. This means that the algorithm does not require for extra costs to handle the carry bit. However, the LSB trick briefly mentioned previously in this paper is not easy to directly apply to Debraize's algorithms. Algorithms 6 and 7 are versions of Debraize's algorithms adjusted to include the LSB trick. As shown at Line 6 in Algorithm 7, it is required for the process to update the carry bit of the previous **for** loop on the LSB. Table 3 provides an understanding of Debraize's conversion process. Although the original algorithms are well designed in terms of execution time and memory usage, applying the LSB trick incurs additional costs.

Algorithm 6 Debraize's Table T Generation [22]

Require: *None*
Ensure: $T[.], r, \rho$
1: Generate a random k-bit r and a random bit ρ
2: **for** $A = 0$ to $2^k - 1$ **do**
3: $\quad T[\rho||A] \leftarrow \{(A+r) \oplus (\rho||r)\} \gg 1$
4: $\quad T[\rho \oplus 1||A] \leftarrow \{(A+r+1) \oplus (\rho||r)\} \gg 1$
5: **end for**
6: **return** $T[.], r, \rho$

Algorithm 7 Debraize's A2B Algorithm with LSB trick [22]

Require: $(A, R), T[.], r, \rho$ /* $x = A + R \bmod 2^{k \cdot n}$ */

/* $T[.], r, \rho$ generated during pre-computation phase */

Ensure: (x', R) /* $x = x' \oplus R$ */

1: $A \leftarrow A - (r||\ldots||r) \bmod 2^{n \cdot k}$
2: $\beta \leftarrow \rho$
3: **for** $i = 0$ to $n - 1$ **do**
4: Split A into $A_h||A_l$ and R into $R_h||R_l$
(such that A_l and R_l have size k)
5: $A \leftarrow A + R_l \bmod 2^{(n-i) \cdot k}$
6: $\beta||x'_i \leftarrow ((T[\beta||A_l]||\mathtt{LSB}_{A_l}) \oplus \beta) \oplus \rho$
7: $x'_i \leftarrow x'_i \oplus R_l$
8: $A \leftarrow A_h$ and $R \leftarrow R_h$
9: **end for**
10: **return** $(x'_{n-1}||\ldots||x'_0) \oplus (r||\ldots||r)$

Table 3. Intermediate Values of Algorithm 7.

Line	Intermediate Value of A and β
1	$\mathrm{A} : A - (r||\ldots||r) \bmod 2^{k \cdot n}$ $= x - R - (r||\ldots||r) \bmod 2^{k \cdot n}$
2	$\beta : \rho$
4 ($i = 0$)	$\mathrm{A} : (x_h||0_k) - (R_h||0_k) - (r||\ldots||0_k)$ $+x_l - R_l - r \bmod 2^{k \cdot n}$
5 ($i = 0$)	$\mathrm{A} : (x_h||0_k) - (R_h||0_k) - (r||\ldots||0_k)$ $+x_l - r \bmod 2^{k \cdot n}$
6 ($i = 0$)	$\beta||x'_0 : ((T[\beta||A_l]||\mathtt{LSB}_{A_l}) \oplus \beta) \oplus \rho$ $= ((T[(\rho \oplus 0)||(x_l - r)]||\mathtt{LSB}_{(x_l - r)}) \oplus \rho) \oplus \rho$ $= (c \oplus \rho)||(x_l \oplus r)||\mathtt{LSB}_{(x_l - r)}$ $= (c \oplus \rho)||(x_l \oplus r)$ $\therefore \beta : c \oplus \rho$ $x'_0 : x_l \oplus r$
8 ($i = 0$)	$\mathrm{A} : x_h - R_h - (r||\ldots||r) \bmod 2^{(n-1) \cdot k}$ $= x - R - (r||\ldots||r) \bmod 2^{(n-1) \cdot k}$ (Carry c is not yet updated in the register A)
4 ($i = 1$)	$\mathrm{A} : (x_h||0_k) - (R_h||0_k) - (r||\ldots||0_k)$ $+x_l - R_l - r \bmod 2^{(n-1) \cdot k}$
5 ($i = 1$)	$\mathrm{A} : (x_h||0_k) - (R_h||0_k) - (r||\ldots||0_k)$ $+x_l - r \bmod 2^{(n-1) \cdot k}$
6 ($i = 1$)	$\beta||x'_1 : ((T[\beta||A_l]||\mathtt{LSB}_{A_l}) \oplus \beta) \oplus \rho$ $= ((T[(\rho \oplus c)||(x_l - r)]||\mathtt{LSB}_{(x_l - r)}) \oplus (c \oplus \rho)) \oplus \rho$ $= ((c_{\mathtt{new}} \oplus \rho)||((x_l + c) \oplus r)||\mathtt{LSB}_{(x_l - r)}) \oplus c$ $= (c_{\mathtt{new}} \oplus \rho)||((x_l + c) \oplus r)||(\mathtt{LSB}_{(x_l - r)} \oplus c)$ $= (c_{\mathtt{new}} \oplus \rho)||((x_l + c) \oplus r)||\mathtt{LSB}_{(x_l + c - r)}$ $= (c_{\mathtt{new}} \oplus \rho)||((x_l + c) \oplus r)$ $\therefore \beta : c_{\mathtt{new}} \oplus \rho$ $x'_i : x_l \oplus r$
8 ($i = 1$)	$\mathrm{A} : x_h - R_h - (r||\ldots||r) \bmod 2^{(n-2) \cdot k}$ $= x - R - (r||\ldots||r) \bmod 2^{(n-2) \cdot k}$ (Carry $c_{\mathtt{new}}$ is not yet updated in the register A)

3. Our Proposal

In this section, we propose an A2B algorithm that is designed for extremely low memory usage while preserving execution time. Our A2B algorithm consists of the use of one table only (like [17]), combined with the stronger secure management of the carry bit such that the carry bit is masked with the same probability (like [22]). The key idea is to construct a pre-computation table of the same size as the input bits. To handle the carry bit with stronger security, we designed a memory of size two which was constructed to minimize the additional costs in the A2B conversion phase.

3.1. Pre-Computation Phase

Algorithm 8 is a new algorithm that generates the pre-computation tables of our A2B algorithm. Table G is stored, except for the LSB of $\{(A+r) \oplus (\gamma||r)\} \in \mathbb{F}_{2^{k+1}}$ at Line 3. The A2B algorithm works correctly even if the pre-computation table does not store the LSB; this is the so-called *LSB trick*. The reason for this is that the subtraction in the arithmetic masking and the XOR in the Boolean masking are equal on $\mathbb{F}_2$. This means that the information regarding the LSB can be handled by a trick in the A2B algorithm without needing to be stored in the pre-computation table. The conversion process is described in further detail in Section 4. In summary, our pre-computation table G is constructed with k-bit memories, and it can be hugely advantageous in real devices that use the `char` type. In addition, table C is used to handle carry bits and Γ is used to guarantee security against DPA attacks. For a carry bit c, table C is based on the following equation, with α and k denoting a random value and the size of converted bits, respectively.

$$C[c \oplus \gamma] = (c + \alpha) \cdot 2^k$$

The bit sizes of table C and Γ should make up the total bit size of $(k \cdot n)$-bit, and the reason for this is discussed in further detail in Section 4. Algorithm 8 is also remarkable not only in terms of memory usage but also in terms of execution time. Debraize's Algorithm 6 has two steps (Lines 3, 4) inside the **for** loops. This means that the number of generated tables is 2^{k+1}. Quantitatively, the number of **for** loops of Algorithm 6 can be considered to be 2^{k+1}, whereas ours has only 2^k loops, like [17]. Therefore, our algorithm will take approximately half of the execution time required by Debraize's algorithm. In terms of the security against side-channel attacks of a carry bit, our A2B masking scheme masks the carry bit with a probability of $1/2$, like [22], and the outputs of table C are uniformly distributed in $[0, 2^{k\cdot(n-1)})||0_k$. We designed Γ to eliminate the extra step of removing α in the conversion phase.

Algorithm 8 Table G and C generation

/ G is used of k-bit register */*

/ C and* Γ *are used of* $(k \cdot n)$*-bit register */*

Require: *None*
Ensure: $G[.], C[.], \Gamma, r$
1: Generate a random $((k-1)\cdot n)$-bit α, a random k-bit r and a random bit γ
2: **for** $A = 0$ to $2^k - 1$ **do**
3: $\quad G[A] \leftarrow \{(A+r) \oplus (\gamma||r)\} \gg 1$
4: **end for**
5: $C[\gamma] \leftarrow \alpha \cdot 2^k$
6: $C[\gamma \oplus 1] \leftarrow (\alpha + 1) \cdot 2^k$
7: $\Gamma \leftarrow \sum_{l=1}^{n-1} (\alpha \cdot 2^{l \cdot k})$
8: **return** $G[.], C[.], \Gamma, r$

3.2. Conversion Phase

Algorithm 9 is the proposed A2B algorithm using the pre-computation table. This algorithm divides the arithmetic masked value A into a k-bit, converts it to a Boolean masked value B_i, and handles carry bit using the table C. At Line 4, the masked LSB can be computed correctly because the Boolean masked value B_i is the same as the arithmetic masked value A_i on $\mathbb{F}_2$. At Line 6, the carry bit t, which is masked by γ in Algorithm 8 (Line 3), is handled by table $C[t]$ by adding α or $\alpha + 1$ when the carry bit is 0 or 1, respectively. As we designed the algorithm to subtract Γ at Line 1, α of table C is removed without the need for any extra steps. It is also worth noting that our algorithm does not require modulus operations. In real devices, data that exceeds the memory size of the register is automatically deleted. Based on this, our algorithm proceeds with the A2B conversion from the least significant word to the most significant word without any modulus operations.

Algorithm 9 Conversion of a $(k \cdot n)$-bit variable

/* **Notation**: Split A into $A_{n-1}||\ldots||A_i||\ldots||A_0$ and

R into $R_{n-1}||\ldots||R_i||\ldots||R_0$, such that A_i and R_i have size k */

/* *r and B_i is used of k-bit register* */

/* *A is used of* $(k * n)$*-bit register* */

Require: $(A, R), G[.], C[.], \Gamma, r$ /* $x = A + R \bmod 2^{k \cdot n}$ */

/* $G[.], C[.], \Gamma, r$ generated during pre-computation phase */

Ensure: (x', R) /* $x = x' \oplus R$ */

1: $A \leftarrow A - (r||\ldots||r) - \Gamma$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad A \leftarrow A + (R_i \cdot 2^{i \cdot k})$
4: $\quad t||B_i \leftarrow G[A_i]||\mathrm{LSB}_{A_i}$
5: $\quad B_i \leftarrow B_i \oplus R_i$
6: $\quad A \leftarrow A + \underline{(C[t] \cdot 2^{i \cdot k})}$
7: $\quad A \leftarrow A\&(2^{(i+1) \cdot k} - 1)$
8: **end for**
9: **return** $(B_{n-1}||\ldots||B_0) \oplus (r||\ldots||r)$

4. Security Analysis and Soundness of Algorithm

When proposing a new countermeasure, two crucial points are the security analysis and soundness. First, to achieve security against first-order DPA attacks, we prove that all intermediate values to process our A2B algorithm are masked by random numbers. Namely, if the intermediate values of the algorithm are uniformly distributed random numbers (i.e., masking values), this algorithm can be considered as having achieved security against first-order DPA attacks. Second, the soundness is that an algorithm achieves its goal with arbitrary inputs in any case. We then use mathematical induction to prove the soundness of our countermeasures.

Now, we analyze the security of our A2B algorithm. To achieve this goal, we enumerate all intermediate values of our algorithms (Algorithms 8 and 9), then verify whether these values have any random numbers.

Algorithm 8 is the algorithm that computes pre-computation tables. The mainly handled data is the masking value. This phase is a good target for horizontal correlation attacks [29,30]. However, this attack can effectively cope with shuffling and dummy operations. In terms of sensitive values such as the key, this algorithm is computed without any sensitive values. The only information attackers can gain is the masking value. To recover the key, the pre-computation and conversion phase should be measured twice.

These two probings deviate from the assumption of the first-order DPA. That is, the higher-order DPA should be required to restore the secret key in the pre-computation and conversion algorithms.

Algorithm 9 is the A2B algorithm. We prove the randomness of each intermediate value as shown in Table 4. Table 4 shows that the sensitive value x; the random values r, R, γ, and α are combined in V_i. 0_l indicates 0 filled with l-bits. V_i denotes the intermediate values at A in Algorithm 9, where i is the number of the Line, i.e., V_1, V_3, V_6,and V_7 are Lines 1, 3, 6, and 7, respectively.

- V_1 is constructed with the masking value R, the masking value r of table G, and the masking value α of table C. α does not mask the lower k-bit. V_1 is uniformly distributed by R, so it guarantees the side-channel security.
- V_3 is concatenated with the upper part like V_1; the middle part, which is $(x - r)$, by adding $R_i \cdot 2^{i \cdot k}$; and the lower part, which is always filled with 0s, by Line 7. Since masking values R and r are independent, this distribution is uniformly random.
- V_4, V_5 are values that have been converted from arithmetic to Boolean masking by table G. The carry bit is masked with γ, and the rest are masked with r. These Lines are secure because γ and r are independent and random numbers.
- V_6 is constructed with the upper part like V_1; the upper-middle part, which is $(x + c - R - r)$; the lower-middle part like V_3; and the lower part as 0s. The masking value α of the upper-middle part is removed by adding table C. The security against side-channel attacks is explained by the same principle as that of V_3.
- V_7 is the clearing process to 0s in the lower part, this step is important to achieve the side-channel safety. If the algorithm omits this clearing process, the pattern of the masking values of the lower part is represented as a concatenated form like $r||r$. In this case, the distribution of the lower part is not uniform. As a result, the distribution of the side-channel signals is determined by sensitive data. In other words, the algorithm cannot provide security against side-channel attacks, and therefore, a clearing process must be performed.

Table 4. Intermediate Values.

$V_1 ... V_7$ is intermediate values of Algorithm 9
$V_1 = (x - R) - (r\|\| \ldots \|\|r) - \sum_{l=1}^{n-1}(\alpha \cdot 2^{l \cdot k})$
$V_3 = [(x - R - (r\|\| \ldots \|\|r)) - \sum_{l=i+1}^{n-1}(\alpha \cdot 2^{l \cdot k})]\|\|(x - r)\|\|0_{i \cdot k}$
$V_4 = (c \oplus \gamma)\|\|(x_{withoutLSB} \oplus r_{withoutLSB})\|\|(x_{LSB} - r_{LSB})$ $= (c \oplus \gamma)\|\|(x \oplus r)$
$V_5 = x \oplus r \oplus R_i$
$V_6 = (x - R) - (r\|\| \ldots \|\|r)$ $- \sum_{l=i+2}^{n-1}(\alpha \cdot 2^{l \cdot k})\|\|(x + c - R - r)\|\|(x - r)\|\|0_{i \cdot k}$
$V_7 = (x - R) - (r\|\| \ldots \|\|r)$ $- \sum_{l=i+2}^{n-1}(\alpha \cdot 2^{l \cdot k})\|\|(x + c - R - r)\|\|0_{(i+1) \cdot k}$

We now use mathematical induction to prove the soundness of the proposed algorithm. In detail, we observe changes in the internal state at $i = 0$ (the first loop) through the ***Base case***. At this time, we prove the operating principles of the pre-computation tables by Lemmas 1 and 2. In the ***Inductive hypothesis***, we define the state when loop $i = a$. This can be easily inferred in the form of a ***Base case***. Finally, in the ***Inductive step***, we claim that the proposed A2B algorithm for arbitrary k, n works correctly.

X_i^j denotes the partial bits of X from the i-th to the j-th bits. For example, $(R_k^{n-1}||0_0^{k-1})$ means the concatenated bits of the R from $(n-1)$-th to k-th bits, and other bits which are zeros.

Theorem 2. *The proposed A2B algorithm for arbitrary k, n works correctly.*

Proof.
Base case: The algorithm works correctly in the initial state ($i = 0$), as shown in the following steps.

$$\text{(Line 1)} \quad A = (x - R) - (r||\ldots||r) - \sum_{l=1}^{n-1}(\alpha \cdot 2^{l \cdot k}) \tag{4}$$

$$\text{(Line 3)} \quad A = x - (R_k^{n-1}||0_0^{k-1}) - (r||...||r) - \sum_{l=1}^{n-1}(\alpha \cdot 2^{l \cdot k}) \tag{5}$$

We must verify that the A2B operation at Lines 4 and 5 is correct. Refer to the following Lemma 1:

Lemma 1. *At Lines 4 and 5, the arithmetic masked value A_i is correctly converted from arithmetic masking to Boolean masking with pre-computation table G.*

Proof. The value of table G is as follows.

$$\begin{aligned} G[A_i] &= \{(A_i + r) \oplus (\gamma||r)\} \gg 1 \\ &= \{(c \oplus \gamma)||(x \oplus r)\} \gg 1 \\ &= \{(c \oplus \gamma)||B_i\} \gg 1 \end{aligned}$$

This means that the output of $G[A_i]$ is $\{(c \oplus \gamma)||B_i\}$ without the LSB.

In the LSB, the Boolean masked value ($x' = x \oplus r$) is the same as the arithmetic masked value on $\mathbb{F}_2$ ($A = x - r \bmod 2$), based on the following definition.

Remark 1. *Exclusive or (XOR) is defined in the arithmetic as modulo-2 addition/subtraction.*

In the above definition, LSB of A_i is $A_i \bmod 2$.

$$A_i \bmod 2 = x - r \bmod 2 = (x \oplus r)\&1 = B_i\&1$$

The important implication of this equation is that we can calculate without storing the LSB in the masking conversion process. This is the so-called *LSB trick*. Line 4 is summarized as follows.

$$G[A_i]||(LSB\ of\ A_i) = (c \oplus \gamma)||B_i = t||B_i$$

Therefore, Lines 4 and 5 are valid. □

$$\text{(Line 6)} \begin{cases} \text{when } c = 0, & \\ A = x \quad - \quad (R_k^{n-1}||0_0^{k-1}) & \\ \quad - (r||...||r) - \sum_{l=2}^{n-1}(\alpha \cdot 2^{l \cdot k}) & \text{(6a)} \\ \text{when } c = 1, & \\ A = (x+1) - (R_k^{n-1}||0_0^{k-1}) & \\ \quad - (r||...||r) - \sum_{l=2}^{n-1}(\alpha \cdot 2^{l \cdot k}) & \text{(6b)} \end{cases}$$

Here, for Line 6, we must prove that the carry bit is handled correctly by table C.

Lemma 2. *The carry bit is handled correctly by table C at Line 6.*

Proof. It can be easily proven by the definition of Table C

$$C[\gamma] = \alpha \cdot 2^k, \; C[\gamma \oplus 1] = (\alpha + 1) \cdot 2^k$$

That is, the result of $C[t]$ is $\alpha \cdot 2^k$ when c is 0 and $(\alpha + 1) \cdot 2^k$ when c is 1. When table C is added to A, we designed α of the next converted block to be naturally removed by $-\Gamma(= -\sum_{l=1}^{n-1}(\alpha \cdot 2^{i \cdot k}))$ at Line 1. Consequently, the correct carry bit is added, and thus Line 6 is valid. □

The end of the initial for loop is as follows.

$$\text{(Line 7)} \quad (\dot{x}_k^{n-1}||0_0^{k-1}) - (R_k^{n-1}||0_0^{k-1}) - (r||...||r||0_0^{k-1}) - \sum_{l=2}^{n-1}(\alpha \cdot 2^{l \cdot k}) \tag{7}$$

$\dot{x}$ is a sensitive value computed for the carry bit.

Inductive hypothesis: Suppose the algorithm holds for $i = a$ before the **for** loop.

Then, $A = (x_{a \cdot k}^{n-1}||0_0^{a \cdot k-1}) - (R_{a \cdot k}^{n-1}||0_0^{a \cdot k-1}) - (r||...||r||0_0^{a \cdot k-1}) - \sum_{l=a+1}^{n-1}(\alpha \cdot 2^{l \cdot k})$.

Additionally, suppose the algorithm holds for $i = a + 1$ before the for loop.

Then, $A = (\dot{x}_{(a+1) \cdot k}^{n-1}||0_0^{(a+1) \cdot k-1}) - (R_{(a+1) \cdot k}^{n-1}||0_0^{(a+1) \cdot k-1}) - (r||...||r||0_0^{(a+1) \cdot k-1}) - \sum_{l=a+2}^{n-1}(\alpha \cdot 2^{l \cdot k})$.

Inductive step: Show that if $i = a$ of the **for** loop holds, then $i = a + 1$ of the **for** loop also holds. We proved this step, as shown in Table 5. This table shows that $i = a + 1$ at the end of the **for** loop indeed holds.

Thus, the inductive case holds. Now, by mathematical induction, our A2B algorithm works correctly for an arbitrary k, n Q.E.D. □

Table 5. The Inductive Step.

Line	A
2	$(x_{a\cdot k}^{n-1}\|\|0_0^{a\cdot k-1}) - (R_{a\cdot k}^{n-1}\|\|0_0^{a\cdot k-1})$ $-(r\|\|...\|\|r\|\|0_0^{a\cdot k-1}) - \sum_{l=a+1}^{n-1}(\alpha\cdot 2^{l\cdot k})$
3	$(x_{a\cdot k}^{n-1}\|\|0_0^{a\cdot k-1}) - (R_{(a+1)\cdot k}^{n-1}\|\|0_0^{(a+1)\cdot k-1})$ $-(r\|\|...\|\|r\|\|0_0^{a\cdot k-1}) - \sum_{l=a+1}^{n-1}(\alpha\cdot 2^{l\cdot k})$
4, 5	**Lemma 1**
6	**Lemma 2** when $c=0$, $(x_{a\cdot k}^{n-1}\|\|0_0^{a\cdot k-1}) - (R_{(a+1)\cdot k}^{n-1}\|\|0_0^{(a+1)\cdot k-1})$ $-(r\|\|...\|\|r\|\|0_0^{a\cdot k-1}) - \sum_{l=a+2}^{n-1}(\alpha\cdot 2^{l\cdot k})$ when $c=1$, $((x_{a\cdot k}^{n-1}+2^{(a+1)\cdot k})\|\|0_0^{a\cdot k-1}) - (R_{(a+1)\cdot k}^{n-1}\|\|0_0^{(a+1)\cdot k-1})$ $-(r\|\|...\|\|r\|\|0_0^{a\cdot k-1}) - \sum_{l=a+2}^{n-1}(\alpha\cdot 2^{l\cdot k})$
7	$(\dot{x}_{(a+1)\cdot k}^{n-1}\|\|0_0^{(a+1)\cdot k-1}) - (R_{(a+1)\cdot k}^{n-1}\|\|0_0^{(a+1)\cdot k-1})$ $-(r\|\|...\|\|r\|\|0_0^{(a+1)\cdot k-1}) - \sum_{l=a+2}^{n-1}(\alpha\cdot 2^{l\cdot k})$

5. Performance Analysis

We summarize the performance of the proposed algorithm, Debraize's algorithm ([22]), and Neiße's algorithm ([17]) in Table 6. The proposed algorithms have advantages in terms of the memory usage and the execution time of the pre-computation phase like [17]. The execution time of the A2B conversion phase with the LSB trick also shows good performance in various environments.

To summarize, one of the advantages in terms of memory usage is that it uses approximatively half the memory in the pre-computation phase compared to Debraize's algorithm. Another advantage in terms of memory usage is that the data type of our pre-computation table is compact and reasonable in real devices, because typical devices only support `char`, `short`, and `int`. The advantage in the execution time of the pre-computation phase is that it requires approximately half the time that Debraize's does, as the table only has half as many entries.

In the conversion phase, the operations of the ADD/SUB, XOR/AND and SHIFT are very similar in the three algorithms; however, the proposed algorithm has slightly fewer operators. To count the operations, we counted the number of operators with the following rules.

- In Algorithm 5, the concatenating process at Line 6 (front) as one AND.
- In Algorithm 5, the concatenating process at Line 6 (rear) as one SHIFT, one AND, and one XOR.
- In Algorithm 6, the concatenating process at Line 3 and 4 (front) as one SHIFT and one XOR.
- In Algorithm 6, the concatenating process at Line 3 and 4 (rear) as one SHIFT and one XOR. (Computes only once)
- In Algorithm 7, the splitting process at Line 4 as two SHIFTs and two ANDs.
- In Algorithm 7, the modulus at the Line 5 as one AND.
- In Algorithm 7, the concatenating at the Line 6 (front) as one SHIFT and one AND.
- In Algorithm 7, the concatenating at the Line 6 ($T[\beta||A_l]$) as one SHIFT, one AND, and one XOR.
- In Algorithm 7, the concatenating at the Line 6 (rear) as one SHIFT, one AND, and one XOR.
- In Algorithm 8, the concatenating process at Line 3 as one SHIFT and one XOR. (Computes only once)
- In Algorithm 9, the concatenating process at Line 4 (front) as one SHIFT and one AND.
- In Algorithm 9, the concatenating process at Line 4 (rear) as one SHIFT, one AND, and one XOR.

- In Algorithm 9, the clearing process at Line 7 as one AND.

The results of the simulation show the number of clocks required to execute on the AVR (generic device -v4/8-bit), MSP430 (generic MSP430 device/16-bit) and ARM (Cortex-A17/32-bit) offered by the IAR Embedded Workbench. The parameters k and n of the masking conversion are 8-bit and 32-bit, respectively. The compiler optimization option is not used, because it could create vulnerabilities to side-channel attacks due to unintended optimization. As can be inferred from the number of operators, the numbers of clocks in the pre-computation phase in our simulation and that of [17] are less than half of that in Debraize's algorithm. On the other hand, the results of the conversion phase vary greatly depending on the microcontroller. The reason Debraize's algorithm is relatively fast on the 8-bit-based microcontroller (AVR simulator) is that the 8-bit operators are major parts of the whole operation, whereas the reason our algorithm produced comparable results on the 32-bit-based microcontroller (ARM simulator) is that the main part of ours is mostly composed of 32-bit operations, such as Lines 6, 7 in Algorithm 9. In addition, there are several parts of our algorithm for which it is easy to apply optimization during actual implementation. For example, in a loop unrolling technique, our algorithm can be coded as **for** is to constants, and avoid duplicate operations by using temporary variables. Concretely, in the case of $k = 8$ and $n = 4$, we can code Line 3 of Algorithm 9 like $A + R$`&0xff`, $A + R$`&0xff00`; and Line 7 can also be coded as A`&0xffffff00`, A`&0xffff0000`.

In terms of performance, our results are better than [22] and similar to [17]. However, in terms of side-channel security, [17] has potential vulnerabilities of the binary classification referred to [28]. That is, our proposals provide excellent performance like [17] and stronger secure management of the carry bit like [22]. Due to recent advances in technology, low-priced devices have achieved great improvements, and as a result 32-bit-based microprocessors have gained widespread use. The proposed algorithms are confirmed to have great advantages not only in terms of memory, but also in terms of the execution time and security in a practical environment.

Table 6. Comparison of the total number of operations and table sizes.

	[17]	[22]	**Proposal**
Table size(RAM)	(2^k) k-bit	$(2^{(k+1)})$ k-bit	(2^k) k-bit
[Pre-computation Phase]			
ADD	2^k	$3 \cdot 2^k$	$2^k + n$
XOR	$2^k + 1$	$5 \cdot 2^k + 1$	$2^k + 2$
SHIFT	2^k	$4 \cdot 2^k + 1$	$2^k + n + 2$
[Conversion Phase]			
ADD/SUB	$3 \cdot n + 2$	$n + 1$	$2 \cdot n + 2$
XOR/AND	$8 \cdot n + 6$	$12 \cdot n + 2$	$6 \cdot n + 1$
SHIFT	$3 \cdot n$	$5 \cdot n$	$4 \cdot n$
Look-up Table	$n + 2$	n	$2 \cdot n$
[ATMega Simulator]			
Pre-computation Phase	26,502	56,177	30,113
Conversion Phase	1508	1466	1335
[MSP430 Simulator]			
Pre-computation Phase	11,349	26,681	13,736
Conversion Phase	704	735	557
[ARM Simulator]			
Pre-computation Phase	5147	10,259	5670
Conversion Phase	228	251	197

6. Conclusions and Future Work

In this paper, we proposed an A2B algorithm using a pre-computation table with low memory usage. This algorithm has a comparable overhead in terms of execution time in various environments, and it compactly optimizes the memory in a real environment.

Given that the threat of side-channel attacks is increasing, masking techniques are also advancing. However, the high-order masking methods used by conversion algorithms to convert between arithmetic and Boolean masking using pre-computation tables are still being challenged. Therefore, optimization studies for the high-order masking of conversion algorithms should continue.

Author Contributions: Conceptualization, H.K. (HanBit Kim); methodology, H.K. (HanBit Kim) and H.K. (HeeSeok Kim); software, H.K. (HanBit Kim); validation, H.K. (HanBit Kim) and S.H.; investigation, H.K. (HanBit Kim); writing—original draft preparation, H.K. (HanBit Kim); writing—review and editing, H.K. (HanBit Kim), H.K. (HeeSeok Kim) and S.H.; supervision, H.K. (HeeSeok Kim) and S.H.

Funding: This research received no external funding.

Acknowledgments: This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00520, Development of SCR-Friendly Symmetric Key Cryptosystem and Its Application Modes).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ors, S.B.; Gurkaynak, F.; Oswald, E.; Preneel, B. Power-Analysis Attack on an ASIC AES implementation. In Proceedings of the 2004 International Conference on Information Technology: Coding and Computing (ITCC 2004), Las Vegas, NV, USA, 5–7 April 2004; Volume 2, pp. 546–552.
2. Kocher, P.C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Proceedings of the Annual International Cryptology Conference, Santa Barbara, CA, USA, 18–22 August 1996; pp. 104–113.
3. Gandolfi, K.; Mourtel, C.; Olivier, F. Electromagnetic analysis: Concrete results. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Paris, France, 14–16 May 2001; Springer: Berlin, Germany, 2001; pp. 251–261.
4. Kocher, P.; Jaffe, J.; Jun, B. Differential power analysis. In Proceedings of the Annual International Cryptology Conference, Santa Barbara, CA, USA, 15–19 September 1999; Springer: Berlin, Germany, 1999; pp. 388–397.
5. Oswald, E.; Schramm, K. An efficient masking scheme for AES software implementations. In Proceedings of the International Workshop on Information Security Applications, Jeju Island, Korea, 22–24 August 2005; Springer: Berlin, Germany, 2005; pp. 292–305.
6. Blömer, J.; Guajardo, J.; Krummel, V. Provably secure masking of AES. In Proceedings of the International Workshop on Selected Areas in Cryptography, Waterloo, ON, Canada, 9–10 August 2004; Springer: Berlin, Germany, 2004; pp. 69–83.
7. Messerges, T.S. *Power Analysis Attacks and Countermeasures for Cryptographic Algorithms*; University of Illinois: Chicago, IL, USA, 2000.
8. Coron, J.S.; Goubin, L. On boolean and arithmetic masking against differential power analysis. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Worcester, MA, USA, 17–18 August 2000; Springer: Berlin, Germany, 2000; pp. 231–237.
9. Hong, D.; Sung, J.; Hong, S.; Lim, J.; Lee, S.; Koo, B.S.; Lee, C.; Chang, D.; Lee, J.; Jeong, K.; et al. HIGHT: A new block cipher suitable for low-resource device. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Yokohama, Japan, 10–13 October 2006; Springer: Berlin, Germany, 2006; pp. 46–59.
10. Hong, D.; Lee, J.K.; Kim, D.C.; Kwon, D.; Ryu, K.H.; Lee, D.G. LEA: A 128-bit block cipher for fast encryption on common processors. In Proceedings of the International Workshop on Information Security Applications, Jeju Island, Korea, 19–21 August 2013; Springer: Berlin, Germany, 2013; pp. 3–27.

11. Beaulieu, R.; Treatman-Clark, S.; Shors, D.; Weeks, B.; Smith, J.; Wingers, L. The SIMON and SPECK lightweight block ciphers. In Proceedings of the 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 8–12 June 2015; pp. 1–6.
12. Standard, S.H. *FIPS PUB 180-2*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2002.
13. Kim, D.C.; Hong, D.; Lee, J.K.; Kim, W.H.; Kwon, D. LSH: A new fast secure hash function family. In Proceedings of the International Conference on Information Security and Cryptology, Seoul, Korea, 3–5 December 2014; Springer: Berlin, Germany, 2014; pp. 286–313.
14. Bernstein, D.J. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*; Springer: Berlin, Germany, 2008; pp. 84–97.
15. Goubin, L. A sound method for switching between boolean and arithmetic masking. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Paris, France, 14–16 May 2001; Springer: Berlin, Germany, 2001; pp. 3–15.
16. Coron, J.S.; Tchulkine, A. A new algorithm for switching from arithmetic to boolean masking. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Cologne, Germany, 8–10 September 2003; Springer: Berlin, Germany, 2003; pp. 89–97.
17. Neiße, O.; Pulkus, J. Switching blindings with a view towards IDEA. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Cambridge, MA, USA, 11–13 August 2004; Springer: Berlin, Germany, 2004; pp. 230–239.
18. Karroumi, M.; Richard, B.; Joye, M. Addition with blinded operands. In Proceedings of the International Workshop on Constructive Side-Channel Analysis and Secure Design, Paris, France, 13–15 April 2014; Springer: Berlin, Germany, 2014; pp. 41–55.
19. Vadnala, P.K.; Großschädl, J. Faster Mask Conversion with Lookup Tables. In Proceedings of the International Workshop on Constructive Side-Channel Analysis and Secure Design, Berlin, Germany, 13–14 April 2015; Springer: Berlin, Germany, 2015; pp. 207–221.
20. Coron, J.S.; Großschädl, J.; Vadnala, P.K. Secure conversion between boolean and arithmetic masking of any order. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Busan, Korea, 23–26 September 2014; Springer: Berlin, Germany, 2014; pp. 188–205.
21. Coron, J.S.; Großschädl, J.; Tibouchi, M.; Vadnala, P.K. Conversion from arithmetic to boolean masking with logarithmic complexity. In Proceedings of the International Workshop on Fast Software Encryption, Istanbul, Turkey, 8–11 March 2015; Springer: Berlin, Germany, 2015; pp. 130–149.
22. Debraize, B. Efficient and provably secure methods for switching from arithmetic to boolean masking. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Leuven, Belgium, 9–12 September 2012; Springer: Berlin, Germany, 2012; pp. 107–121.
23. Mangard, S.; Oswald, E.; Popp, T. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*; Springer Science & Business Media: Berlin, Germany, 2008; Volume 31.
24. Goubin, L.; Martinelli, A. Protecting AES with Shamir's secret sharing scheme. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Nara, Japan, 28 September–1 October 2011; Springer: Berlin, Germany, 2011; pp. 79–94.
25. Balasch, J.; Faust, S.; Gierlichs, B. Inner product masking revisited. In Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, 26–30 April 2015; Springer: Berlin, Germany, 2015; pp. 486–510.
26. Messerges, T.S. Securing the AES finalists against power analysis attacks. In Proceedings of the International Workshop on Fast Software Encryption, New York, NY, USA, 10–12 April 2000; Springer: Berlin, Germany, 2000; pp. 150–164.
27. Bettale, L.; Coron, J.S.; Zeitoun, R. Improved High-Order Conversion From Boolean to Arithmetic Masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, *2018*, 22–45.
28. Sim, B.Y.; Kang, J.; Han, D.G. Key Bit-Dependent Side-Channel Attacks on Protected Binary Scalar Multiplication. *Appl. Sci.* **2018**, *8*, 2168. [CrossRef]

29. Tunstall, M.; Whitnall, C.; Oswald, E. Masking tables—An underestimated security risk. In Proceedings of the International Workshop on Fast Software Encryption, Singapore, 11–13 March 2013; Springer: Berlin, Germany, 2013; pp. 425–444.
30. Kim, H.S.; Hong, S. New Type of Collision Attack on First-Order Masked AESs. *ETRI J.* **2016**, *38*, 387–396. [CrossRef]