Efficient Conversion Method from Arithmetic to Boolean Masking in Constrained Devices

Yoo-Seung Won[1] and Dong-Guk Han[1,2]*

[1]Department of Financial Information Security, Kookmin University, Seoul, Korea
mathwys87@kookmin.ac.kr
[2]Department of Mathematics, Kookmin University, Seoul, Korea
christa@kookmin.ac.kr

Abstract. A common technique employed for preventing a side channel analysis is boolean masking. However, the application of this scheme is not so straightforward when it comes to block ciphers based on Addition-Rotation-Xor structure. In order to address this issue, since 2000, scholars have investigated schemes for converting Arithmetic to Boolean (AtoB) masking and Boolean to Arithmetic (BtoA) masking schemes. However, these solutions have certain limitations. The time performance of the AtoB scheme is extremely unsatisfactory because of the high complexity of $\mathcal{O}(k)$ where k is the size of addition bit. At the FSE 2015, an improved algorithm with time complexity $\mathcal{O}(\log k)$ based on the Kogge-Stone carry look-ahead adder was suggested. Despite its efficiency, this algorithm cannot consider for constrained environments. Although the original algorithm naturally extends to low-resource devices, there is no advantage in time performance; we call this variant as the generic variant. In this study, we suggest an enhanced variant algorithm to apply to constrained devices. Our solution is based on the principle of the Kogge-Stone carry look-ahead adder, and it uses a divide and conquer approach. In addition, we prove the security of our new algorithm against first-order attack. In implementation results, when $k = 64$ and the register bit size of a chip is 8, 16 or 32, we obtain 58%, 72%, or 68% improvement, respectively, over the results obtained using the generic variant. When applying those algorithms to first-order SPECK, we also achieve about 40% improvement. Moreover, our proposal extends to higher-order countermeasures as previous study.

Keywords: Arithmetic to Boolean masking, Kogge-Stone carry look-ahead adder, ARX-based cryptographic algorithm

1 Introduction

Side channel analysis, which has been in the spotlight over the past decade, belongs to the genre of software and hardware implementation attacks. With respect to the properties of the cryptographic algorithm and physical information,

* The corresponding author.

the attack schemes of adversaries are diverse and they involve, for example, simple power analysis and differential power analysis attacks [2]. To counteract these attacks, various countermeasures have been proposed in the literature. However, among those proposals, countermeasures such as boolean masking, hiding, and threshold implementation have outlasted other methods and are usually recommended. Especially, in terms of software implementation, first-order attacks cannot destroy the (first-order) boolean masking scheme regardless of the number of traces.

In another context, the advent of the Internet of Things (IoT) has encouraged cryptographic algorithms, in some instances, to play an important role in satisfying the needs of the IoT environment. In designing such algorithms, external conditions including time performance and gate size have become increasingly significant. Recently, cryptographic algorithms that satisfy these conditions have been published. In the case of block ciphers, most of their structures utilize S-boxes of nibble units or use addition to introduce a confusion factor. Moreover, Addition-Rotation-Xor(ARX)-based structure can enhance time performance and mathematical security through software implementation. For these reasons, several lightweight cryptographic algorithms such as SPECK [13], LEA [15], and LSH [14] have emerged.

From the side channel countermeasure perspective, it is quite complicated to apply a boolean masking scheme to a block cipher with an ARX-based structure. To overcome this challenge, other approaches are required. Initially, algorithms were developed for performing conversions between boolean and arithmetic masking schemes. Goubin, in particular, described a very elegant algorithm for converting from boolean to arithmetic (BtoA) masking [3], using only a constant number of operations, independent of the size of the addition bit. This allows for the easy exploitation of the BtoA masking algorithm at low cost. On the other hand, there also have been several changes to the algorithm that convert from arithmetic to boolean (AtoB) masking, in order to improve time performance and/or reduce memory consumption. First, Goubin reported an AtoB masking algorithm [3] which has $\mathcal{O}(k)$ complexity where k is the addition bit size. Later, at CHES 2003, an AtoB masking algorithm [4] with a precomputed table was introduced to reduce the time performance, rather than increasing the memory consumption. This algorithm can also be easily modified to suit the IoT environment due to the fact that the size of the precomputed table can be defined beforehand. Shortly thereafter, Neiße *et al.* in [6] suggested that their algorithm could result in lower memory consumption than the previous algorithm. Within the decade, Debraize proposed a new high performance algorithm as well as two modified algorithms with precomputed tables [11]. Despite these efforts, the problem of time complexity has endured at about $\mathcal{O}(k)$.

In recent years, the rising demand for higher-order masking countermeasures has led to the introduction of second-order masking countermeasures. At SPACE 2013, Vadnala *et al.* proposed two secure algorithms for converting between boolean and arithmetic masking schemes against second-order attacks [12]. These algorithms apply the generic second-order secure countermeasure devel-

oped by Rivain *et al.* [9], but they were difficult to expand for a larger addition bit size becuase 2^k or $2^k/k$ bytes of RAM is required. To overcome this obstacle, handling of the carry bit was required and Vadnala *et al.* in [18] solved this problem without difficulty by utilizing a precomputed table to handle carry bit,

At about the same time, general approaches were achieved also for higher-order BtoA and AtoB schemes. The initial proposal [16] based on Goubin's conversion method [3] was described at the CHES 2014, although it has high time complexity of $\mathcal{O}(n^2 \cdot k)$ for $n = 2t + 1$ shares.

Another approach is to execute an arithmetic operation without converting the masking form. At the COSADE 2014, Karroumi *et al.* in [17] demonstrated that it was possible to use addition/subtraction for two boolean masked inputs. As a result, an efficient algorithm was developed to satisfy the IoT environment, using lookup tables. Despite all these efforts, however, the arithmetic operation of algorithms without conversion continues to be characterized by a time complexity of about $\mathcal{O}(k)$.

Unlike the previous algorithms that were based on the classical ripple carry adder, algorithms based on the Kogge-Stone carry look-ahead adder with the logarithmic complexity $\mathcal{O}(\log k)$ [19] have recently been proposed. These algorithms not only easily expand to a higher-order masking scheme but can also be applied to algorithms for AtoB masking and arithmetic operation without conversion. To the best of our knowledge, these algorithms achieve an outstanding performance. Although the algorithms proposed by Coron *et al.* [19] require a low complexity of $\mathcal{O}(\log k)$, they become infeasible for implementation on low-resource devices such as the smart card. Practically, these algorithms cannot be applied to the constrained devices without any modifications if the register bit size of a chip is l bit is less than that of addition k bit. However, by using an array concept, the original algorithm naturally extends to low-resource devices although there is no associated advantage in time performance. In this paper, we call this algorithm using an array concept as the generic variant. As a result, the generic variant algorithm may lose their merit when this algorithm is implemented IoT devices.

Our Contributions. The addition size of the most cryptographic algorithms with ARX-based structure does not correspond to the register size of IoT devices. Although the generic variant algorithm from [19] can directly be applied to a chip with the register l bit, we should endure a high cost. In response to this limitation, we suggest that the enhanced variant algorithms are significantly faster than the generic algorithm. Our solution follows the basic concept of the Kogge-Stone carry look-ahead adder, but uses a divide and conquer approach to prevent the time complexity from becoming too great. The currently proposed core concept involves a means of handling the carry occurrence of the Kogge-Stone carry look-ahead adder. Part of this procedure is to control the carries propagating from less to more significant words, which must also be protected by masking to prevent any first leakage. More precisely, after the carry value is generated from the previous block, it fills up the least significant bit in the next block. When the register size is $k/2$, we also provide the advanced algorithm more

than the enhanced variant algorithm. We demonstrate that this can be achieved in an efficient and secure fashion by using the principle of the Kogge-Stone carry look-ahead adder for the carries. We also verify the correctness and prove the security of our algorithms. In implementation results, the enhanced variant algorithms have a 58% $\sim$ 72% advantage over the generic variant algorithms in time performance. In the case of $m = 2$, we also acquire more 27% improvement compared to our general solution algorithm is an enhanced variant algorithm. Moreover, we apply the generic variant and enhanced variant algorithms for first-order masked SPECK. As a consequent, we obtain improvement of about 40% over the generic algorithm results.

Organization of the paper. This paper is organized as follow: We describe the Kogge-Stone carry look-ahead adder used for our variant algorithms in Section 2. Section 3 proposes the enhanced variant algorithms used for our core idea. Also, we present the result of the simulated implementation for our variant algorithms and first-order block cipher SPECK in Section 4. Finally, Section 5 concludes the paper.

2 Kogge-Stone Carry Look-Ahead Adder and Its Countermeasure

2.1 Notation

Before representing the detailed description, refer to Table 1. provided for a better understanding. The notations maintain the consistency for the representation of all variables until the end of the paper.

For example, if the register bit size is 16 and the size of the addition is 64, then the number of data blocks is 4. That is, $x = \left(x_{(3)}||x_{(2)}||x_{(1)}||x_{(0)}\right)$.

2.2 Kogge-Stone Carry Look-Ahead Adder

In this section, we first recall the Kogge-Stone carry look-ahead adder [1], [19] which is based on recurrence equations, written $c(i) = c^{(i)}$, as follows:

$$\begin{cases} c(0) = 0 \\ \forall i \geq 1,\ c(i) = \{a(i-1) \wedge c(i-1)\} \oplus b(i-1) \end{cases} \tag{1}$$

where $x^{(i)}$ is a low significant ith bit, $c^{(i)}$ is generated from the carry bit of $\sum_{j=0}^{i-1} 2^j x^{(j)} + \sum_{j=0}^{i-1} 2^j y^{(j)}$, and $c(i)$, $a(i)$, $b(i)$ respectively represent $c^{(i)}$, $x^{(i)} \oplus y^{(i)}$, $x^{(i)} \wedge y^{(i)}$. We can therefore compute the carry bit. Based on the recurrence equation, we can re-construct the k bit addition as shown below.

$$x + y = \sum_{i=0}^{k-1} 2^i x^i + \sum_{i=0}^{k-1} 2^i y^i = \sum_{i=0}^{k-1} (x^{(i)} \oplus y^{(i)} \oplus c^{(i)}) \tag{2}$$

Notation	Description
x	k bit value
l	Register bit size of chip
m	Number of data blocks $m = k/l$ $(0 < m \leq l)$
$x^{(i)}$	Least significant i th bit
$x_{(i)}$	i th data block of x $\left(x_{(i)} = 2^{i \times l} \sum_{j=0}^{l-1} (2^j \times x_{i \times l + j})\right)$
$X_{(i)}$	Modified i th data block that includes the carry value generated from the previously modified data $\begin{cases} 2^{i(l-1)} \left(c^{(i(l-1))} + \sum_{j=1}^{l-1} 2^j x^{(i(l-1)+j)}\right) (1 \leq i < m) \\ 2^{i(l-1)} \left(c^{(i(l-1))} + \sum_{j=1}^{m-1} 2^j x^{(i(l-1)+j)}\right) (i = m) \\ x_{(0)} \quad (i = 0) \end{cases}$
$X'_{(i)}$	Modified i th data block that doesn't include the carry value generated from the previously modified data $\begin{cases} 2^{i(l-1)} \sum_{j=1}^{l-1} 2^j x^{(i(l-1)+j)} (1 \leq i < m) \\ 2^{i(l-1)} \sum_{j=1}^{m-1} 2^j x^{(i(l-1)+j)} (i = m) \\ x_{(0)} \quad (i = 0) \end{cases}$

Table 1: Notations used

According to Lemma 1 of [19], Eqn. (1) can be converted to a Kogge-Stone recursive equation:

$$\begin{cases} P_{i,j} = P_{i-1,j} \\ G_{i,j} = G_{i-1,j} \quad (0 \leq j < 2^{i-1}) \end{cases}$$
$$\begin{cases} P_{i,j} = P_{i-1,j} \wedge P_{i-1,j-2^{i-1}} \\ G_{i,j} = (P_{i-1,j} \wedge G_{i-1,j-2^{i-1}}) \oplus G_{i-1,j} \quad (2^{i-1} \leq j < k) \end{cases} \tag{3}$$
$$\Longrightarrow \begin{cases} c(0) = 0, \quad c(1) = G_{0,0} \\ c(j+1) = G_{i,j} \quad (2^{i-1} \leq j < 2^i) \end{cases}$$

When i corresponds to $\lceil \log (k-1) \rceil$, we obtain the most significant carry bit $c^{(k-1)}$.

Eqn. (3) naturally extends to the recurrence equation of the k bit addition, and time complexity with $\mathcal{O}(\log k)$ is still required. By Theorem 3 of [19], the recurrence equation of the k bit addition is as shown below.

$$\begin{cases} P_i = P_{i-1} \wedge (P_{i-1} \ll 2^{i-1}) \quad (1 \leq i < n) \\ G_i = (P_{i-1} \wedge (G_{i-1} \ll 2^{i-1})) \oplus G_{i-1} \end{cases} \Longrightarrow x + y = x \oplus y \oplus (2G_n) \quad (4)$$

where $P_i = \sum_{j=2^i-1}^{k-1} 2^j P_{i,j}$ with $P_0 = x \oplus y$, $G_i = \sum_{j=0}^{k-1} 2^j G_{i,j}$ with $G_0 = x \wedge y$, and n represents $\lceil \log(k-1) \rceil$. Based on Eqn. (4), the sequence can be computed using Algorithm 1.

Algorithm 1 Kogge-Stone Adder

Input: $x, y \in \{0,1\}^k$, $n = \max(\lceil \log(k-1) \rceil, 1)$
Output: $z = x + y \mod 2^k$

1: $P \leftarrow x \oplus y$
2: $G \leftarrow x \wedge y$
3: **for** $i := 1$ **to** $n-1$ **do**
4: $\quad G \leftarrow (P \wedge (G \ll 2^{i-1})) \oplus G$
5: $\quad P \leftarrow P \wedge (P \ll 2^{i-1})$
6: **end for**
7: $G \leftarrow (P \wedge (G \ll 2^{n-1})) \oplus G$
8: **return** $x \oplus y \oplus (2G)$

2.3 Generic variant for Kogge-Stone Adder and AtoB Masking

In this section, we introduce the k bit addition when the register bit size corresponds to the l bit, where is less than k. Basically, a generic variant algorithm using the array concept is a direct application of the Kogge-Stone adder and we can convert a generic variant algorithm into a first-order algorithm. Refer to the generic variant for Kogge-Stone adder and AtoB masking in Appendix A.

3 Enhanced Variant for AtoB Masking based on Kogge-Stone Adder

3.1 Main Idea Sketch

In this section, we introduce our main idea for providing better comprehension. Assuming k and l correspond to 8 and 4, respectively, we have to divide into 2 array from original data for the generic variant Kogge-Stone adder because the size of the register is smaller than that of addition; refer to Array1 and Array0 in Fig.1. Therefore, the intermediate calculations of generic variant algorithm

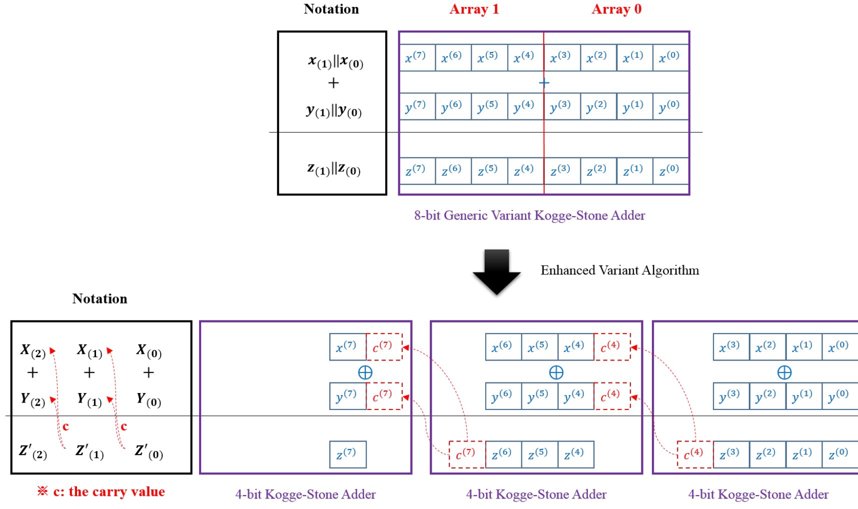

Fig. 1: Main Idea Sketch for Enhanced Variant Algorithm

consume more cost than those of original algorithm because of the difference between original Kogge-Stone adder and generic variant algorithm.

However, for applying our main idea, we have to split into 3 array before calculating the addition. As seen Fig.1, considering the register size of a chip, we utilize the original Kogge-Stone adder without any modifications. To acquire the most significant bit of each block, we review the correction of Theorem 3, as addressed in [19], for our new design. $2G_n$ is not exactly equal to $\sum_{j=0}^{k-1} 2^j c_j$, but rather $\sum_{j=0}^{k} 2^j c_j$. In other words, the $2G_n$ value that leaks the most significant k-th bit is the basis of our enhanced variant algorithm. That is, after the carry value is generated from the previous block, it fills up the least significant bit in the next block.

3.2 Enhanced Variant for Kogge-Stone Adder

Our new conversion algorithm is based on the principle of the Kogge-Stone adder. As mentioned earlier, due to the missing value, we can build an enhanced variant of the Kogge-Stone adder and AtoB masking. The correctness of this technique is demonstrated, as follows:

Theorem 1. *Let* $x = (x_{(m-1)}||\cdots||x_{(0)}), y = (y_{(m-1)}||\cdots||y_{(0)}), z = x + y$ *be elements of* $\{0,1\}^k$*. Then* $\sum_{i=0}^{m-1} z_{(i)} = \sum_{i=0}^{m} Z'_{(i)} \left(= \sum_{i=0}^{m-1} \left(x_{(i)} + y_{(i)}\right)\right)$

Proof. By notation, we can rewrite the equation as follows:

$$\sum_{i=0}^{m-1} z_{(i)} \underset{\langle 1 \rangle}{=} \sum_{i=0}^{m} Z'_{(i)} \underset{\langle 2 \rangle}{=} \sum_{i=0}^{m} \left(X_{(i)} + Y_{(i)} \mod 2^{(i+1)\times l}\right)$$

For ⟨1⟩, we refer to Lemma 3 in Appendix B. Thus, we prove ⟨2⟩ in this theorem. Also, by proving Eqn. (5), the above equation can be naturally proven. Therefore, we provide only the proof of Eqn. (5).

$$Z'_{(i)}/2^{i(l-1)} = \{X_{(i)} + Y_{(i)}\}/2^{i(l-1)} \mod 2^l \ (0 \leq i \leq m) \quad (5)$$

$\{X_{(i)} + Y_{(i)}\}/2^{i(l-1)} \mod 2^l \ (0 < i < m)$

$$= \left(c^{(i(l-1))} + \sum_{j=1}^{l-1} 2^j x^{(i(l-1)+(j-1))} + c^{(i(l-1))} + \sum_{j=1}^{l-1} 2^j y^{(i(l-1)+(j-1))} \right)$$

For comfortable expression, we re-index some variables as below.

($c^{(i(l-1))}$ is $a^{(0)}$ or $b^{(0)}, x^{(i(l-1)+j)} = a^{(j+1)}, y^{(i(l-1)+j)} = b^{(j+1)}$)

$$= \left(\sum_{j=0}^{l-1} 2^j a^{(j)} + \sum_{j=0}^{l-1} 2^j b^{(j)} \right) \mod 2^l$$

$$= \bigoplus_{j=0}^{l-1} 2^j \left(a^{(j)} \oplus b^{(j)} \oplus d^{(j)} \right)$$

($d^{(j)}$which is generated from $\sum_{p=0}^{j-1} a^{(p)} + \sum_{p=0}^{j-1} b^{(p)}$ is the carry value of most significant bit)

$$= c^{(i(l-1))} \oplus c^{(i(l-1))} \oplus \bigoplus_{j=1}^{l-1} 2^j \left(x^{(i(l-1)+j-1)} \oplus y^{(i(l-1)+j-1)} \oplus c^{(i(l-1)+j-1)} \right)$$

(By Theorem 4 in Appendix B)

$$= \bigoplus_{j=1}^{l-1} 2^j \left(z^{(i(l-1)+j-1)} \right) = Z'_{(i)}/2^{i(l-1)}$$

In the case of $i = 0$ and m, the proof is straightforward so it is omitted here. □

Algorithm 2 Enhanced Variant for Kogge-Stone Adder

Input: $x = \left(x_{(m-1)} || \cdots || x_{(0)}\right), y = \left(y_{(m-1)} || \cdots || y_{(0)}\right) \in \{0,1\}^k$
$n = \max(\lceil \log (l-1) \rceil, 1)$
Output: $z = \left(z_{(m-1)} || \cdots || z_{(0)}\right) = x + y \mod 2^k$

1: $X_{(0)} \leftarrow \left(x^{(l-1)} || \cdots || x^{(0)}\right)$
2: $Y_{(0)} \leftarrow \left(y^{(l-1)} || \cdots || y^{(0)}\right)$
3: $C \leftarrow 0$
4: **for** $i := 1$ **to** m **do**
5: $\quad X_{(i)} \leftarrow \left(x^{(i(l-1)+l-2)} || \cdots || x^{(i(l-1)+0)} || 0\right)$
6: $\quad Y_{(i)} \leftarrow \left(y^{(i(l-1)+l-2)} || \cdots || y^{(i(l-1)+0)} || 0\right)$
7: **end for**
8: **for** $j := 0$ **to** m **do**
9: $\quad P \leftarrow X_{(j)} \oplus Y_{(j)}$
10: $\quad G \leftarrow \left(X_{(j)} \wedge Y_{(j)}\right) \oplus C$
11: $\quad$ **for** $i := 1$ **to** $n-1$ **do**
12: $\quad\quad G \leftarrow \left(P \wedge \left(G \ll 2^{i-1}\right)\right) \oplus G$
13: $\quad\quad P \leftarrow P \wedge \left(P \ll 2^{i-1}\right)$
14: $\quad$ **end for**
15: $\quad G \leftarrow \left(P \wedge \left(G \ll 2^{n-1}\right)\right) \oplus G$
16: $\quad Z'_{(j)} \leftarrow X_{(j)} \oplus Y_{(j)} \oplus (2G)$
17: $\quad$ **if** $j \neq 0$ **then**
18: $\quad\quad Z'_{(j)} \leftarrow Z'_{(j)} \gg 1$
19: $\quad$ **end if**
20: $\quad C \leftarrow G \gg (l-1)$
21: **end for**
22: $\left(z_{(m-1)} || \cdots || z_{(0)}\right) \leftarrow \left(Z'_{(m)} || \cdots || Z'_{(0)}\right) \mod 2^k$
23: **return** $z = \left(z_{(m-1)} || \cdots || z_{(0)}\right)$

Based on Theorem 1, we can build an enhanced variant algorithm; see Algorithm 2. There is no need to calculate the carry value in Step 9; $(X_{(j)} \oplus C) \oplus (Y_{(j)} \oplus C) = X_{(j)} \oplus Y_{(j)}$. Also, in Step 10, the result of the addition can obviously be acquired for only one operation with a carry value; $(X_{(j)} \oplus C) \wedge (Y_{(j)} \oplus C) = (X_{(j)} \wedge Y_{(j)}) \oplus C$. Moreover, the important point is that the number of inner-loops is $\lceil \log (l-1) \rceil$ and the unit of operation is the l bit although the number of outer-loops is $m+1$.

3.3 Enhanced Variant for AtoB Masking

Similarly, here we demonstrate how to secure the AtoB masking algorithm. We can convert Algorithm 2 into a first-order secure algorithm by protecting all intermediate variables utilizing Algorithm 5; see Algorithm 3.

In Algorithm 3, some of the secure algorithms adopted, including $\mathsf{SecShift}_l$, SecAnd_l, and SecXor_l are based on the l bit unit using only one array, unlike the secure algorithms in Algorithm 5.

Algorithm 3 Enhanced Variant for AtoB Masking

Input: $a = \left(a_{(m-1)}||\cdots||a_{(0)}\right), r = \left(r_{(m-1)}||\cdots||r_{(0)}\right) \in \{0,1\}^k$
$n = \max(\lceil \log(l-1) \rceil, 1)$
Output: $x = \left(x_{(m-1)}||\cdots||x_{(0)}\right)$ such that $x \oplus r = a + r \mod 2^k$

1: $s \leftarrow \{0,1\}^l$, $t \leftarrow \{0,1\}^l$, $u \leftarrow \{0,1\}^l$, $\delta \leftarrow \{0,1\}^l$
2: $a_{(0)} \leftarrow \left(a^{(l-1)}||\cdots||a^{(0)}\right)$, $r_{(0)} \leftarrow \left(r^{(l-1)}||\cdots||r^{(0)}\right)$, $C \leftarrow \delta$
3: **for** $i := 1$ **to** m **do**
4: $a_{(i)} \leftarrow \left(a^{(i(l-1)+l-2)}||\cdots||a^{(i(l-1)+0)}||0\right)$
5: $r_{(i)} \leftarrow \left(r^{(i(l-1)+l-2)}||\cdots||r^{(i(l-1)+0)}||0\right)$
6: **end for**
7: **for** $j := 0$ **to** m **do**
8: $P \leftarrow a_{(j)} \oplus s$
9: $P \leftarrow P \oplus r_{(j)}$
10: $G \leftarrow s \oplus \left(\left(a_{(j)} \oplus t\right) \wedge r_{(j)}\right) \oplus C$
11: $G \leftarrow G \oplus \left(t \wedge r_{(j)}\right) \oplus \delta$
12: **for** $i := 1$ **to** $n-1$ **do**
13: $H \leftarrow \mathsf{SecShift}_l[G, s, t, 2^{i-1}]$
14: $W \leftarrow \mathsf{SecAnd}_l[P, H, s, t, u]$
15: $G \leftarrow \mathsf{SecXor}_l[G, W, u]$
16: $H \leftarrow \mathsf{SecShift}_l[P, s, t, 2^{i-1}]$
17: $P \leftarrow \mathsf{SecAnd}_l[P, H, s, t, u]$
18: $P \leftarrow P \oplus s$
19: $P \leftarrow P \oplus u$
20: **end for**
21: $H \leftarrow \mathsf{SecShift}_l[G, s, t, 2^{n-1}]$
22: $W \leftarrow \mathsf{SecAnd}_l[P, H, s, t, u]$
23: $G \leftarrow \mathsf{SecXor}_l[G, W, u]$
24: $X'_{(j)} \leftarrow a_{(j)} \oplus (2G)$
25: $X'_{(j)} \leftarrow X'_{(j)} \oplus (2s)$
26: **if** $j \neq 0$ **then** $X'_{(j)} \leftarrow X'_{(j)} \gg 1$
27: **end if**
28: $C \leftarrow [\{G \gg (l-1)\} \oplus \delta] \oplus \{s \gg (l-1)\}$
29: **end for**
30: $\left(x_{(m-1)}||\cdots||x_{(0)}\right) \leftarrow \left(X'_{(m)}||\cdots||X'_{(0)}\right) \mod 2^k$
31: **return** $x = \left(x_{(m-1)}||\cdots||x_{(0)}\right)$

In addition, there are some differences between Algorithms 2 and 3, because the δ value is required in order to prevent the leakage of the carry value. Thus, the carry value is masked by the δ value in Step 28. As such, in Step 11, we must eliminate the masked value δ because the G value had already been masked by s in Step 10. The following Lemma proves the security of our new algorithm against first-order attack.

Lemma 1. *When r is uniformly distributed in $\mathbb{F}_{2^k}$, any intermediate variable in Algorithm 3 has a distribution independent of $x = A + r \mod 2^k$.*

Proof. The proof is based on Lemma 5 of the previous paper [19], and also on the fact that all intermediate variables from Steps 10, 11, and 28 have a distribution independent of x.

Case 1: All variables from Step 28. have a distribution independent of x
G, $G \gg (l-1)$, $\{G \gg (l-1)\} \oplus \delta$, and $[\{G \gg (l-1)\} \oplus \delta] \oplus \{s \gg (l-1)\}$ $\left(= c^{(j(l-1)+(i-2))} \oplus s\right)$ have a distribution independent of x, since $G \oplus s$ is $\left(c^{(j(l-1)+(i-2))}||\cdots||c^{(j(l-1)+0)}||0\right)$.

Case 2: All variables from Step 10. have a distribution independent of x
C has a distribution independent of x because of the masked value δ. Therefore, other intermediate variables have uniform distribution.

Case 3: All variables from Step 11. have a distribution independent of x
Since G corresponds to $s \oplus \left(\left(a_{(j)} \oplus t\right) \wedge r_{(j)}\right) \oplus C \oplus \left(t \wedge r_{(j)}\right) \oplus \delta$ $(= (a_{(j)} \wedge r_{(j)}) \oplus c^{(j(l-1)+(i-2))} \oplus s)$, G has a distribution independent of x, and other intermediate variables have uniform distribution. □

Algorithm 3 is the generalization of the l and k variables. However, in the case $m = 2$, we obtain more improvement time performance by modifying the operation for the most significant block. In other words, the outer-loop can be reduced $m-1$ times, since there remains just a single bit in $X_{(m)}$ and $Y_{(m)}$ excluding the carry value. The following Lemma provides more detail. Note also that the algorithm in Appendix C could be used as an enhanced variant of AtoB masking for $m = 2$.

Lemma 2. *When m is 2, $X_{(m)} + Y_{(m)}$ is identical to $2^{m(l-1)+1}z^{(m(l-1)+1)}$. Moreover, the total operation requires only two XOR operations.*

Proof.

$$
\begin{aligned}
& X_{(m)} + Y_{(m)} \\
&= 2^{m(l-1)}\left(c^{(m(l-1))} + 2 \times x^{(m(l-1)+1)}\right) + 2^{m(l-1)}\left(c^{(m(l-1))} + 2 \times y^{(m(l-1)+1)}\right) \\
&= 2^{m(l-1)+1}\left(c^{(m(l-1))} + x^{(m(l-1)+1)} + y^{(m(l-1)+1)}\right) \\
&= 2^{m(l-1)+1}\left(c^{(m(l-1))} \oplus x^{(m(l-1)+1)} \oplus y^{(m(l-1)+1)}\right) \\
&= 2^{m(l-1)+1}z^{(m(l-1)+1)}
\end{aligned}
$$

4 Implementation Results

In this section, we give the simulation results of our new algorithms and first-order masked lightweight block cipher SPECK [13] based on the ARX structure.

4.1 Simulation Results for our Variant Algorithms

We implemented our variant algorithms along with the Kogge-Stone adder. Considering low-resource devices such as the smart card, we implemented three possibilities partitioning 64 bit additions, with the register length $l = 8$, $l = 16$ and $l = 32$. To compare time performance, we used the AVR Studio 6.2 simulation for $l = 8$, the IAR Embedded Workbench Evaluation simulation for $l = 16$, and the CodeWarrior for ARM Developer Suite v1.2 for $l = 32$. In addition, to ensure a fairness of the experiment, we do not consider the time performance for the generation of random numbers.

For $l = 8$, we obtained a 58% improvement over the generic variant algorithm, for $l = 16$, we obtained a considerable 72% improvement, and for $l = 32$ we obtained a 68% improvement. Especially, we have more 27% improvement over a general enhanced variant algorithm compared to between Algorithm 3 and Algorithm 6.

Algorithm	l	k	Clock Cycle (CC)	Penalty Factor
Algorithm 5	8	64	2864	1.00
Algorithm 3	8	64	1217	0.42
Algorithm 5	16	64	2705	1.00
Algorithm 3	16	64	765	0.28
Algorithm 5	32	64	1196	1.00
Algorithm 3	32	64	526	0.44
Algorithm 6	32	64	384	0.32

Table 2: Simulation Results for our Variant Algorithms

4.2 Simulation Results for first-order SPECK

In this section, we apply our countermeasure to realize the first-order secure implementation of SPECK-256/256. Since the structure of SPECK has ARX operations, we must convert between boolean and arithmetic masking scheme. For BtoA masking scheme, we used Goubin's algorithm in [3]. Considering the SPECK structure, we must perform two BtoA maskings and one AtoB masking per round, totaling 64 BtoA maskings and 32 AtoB maskings overall.

The results for computing the SPECK-256/256 of a single message block are summarized in Table 3. Similar to previous simulation results, we achieved about 36%, 43%, and 37% improvements over the masked SPECK for $l = 8$, $l = 16$ and $l = 32$, respectively.

Algorithm	l	k	Clock Cycle (CC)	Penalty Factor
Non-Masked SPECK	8	64	24,360	1.00
Masked SPECK with Algorithm 5	8	64	177,303	7.27
Masked SPECK with Algorithm 3	8	64	112,951	4.64
Non-Masked SPECK	16	64	21,446	1.00
Masked SPECK with Algorithm 5	16	64	143,642	6.70
Masked SPECK with Algorithm 3	16	64	81,562	3.80
Non-Masked SPECK	32	64	10,279	1.00
Masked SPECK with Algorithm 5	32	64	71,006	6.91
Masked SPECK with Algorithm 3	32	64	49,470	4.81
Masked SPECK with Algorithm 6	32	64	44,926	4.37

Table 3: Simulation Results for Non-Masked and Masked SPECK

5 Conclusion

For low-resource devices, we proposed two enhanced variant algorithms that are based on the principle of the Kogge-Stone carry look-ahead adder. One is a generalized algorithm for the register size l and the other, in the case $l = k/2$, has more improvement over a general enhanced variant algorithm in time performance. As such, all variant algorithms not only keep the logarithmic complexity $\mathcal{O}(\log k)$ but also outperform the generic variant algorithm of the Kogge-Stone adder. We also proved our algorithms to be secure against first-order attacks, and that they naturally extend to higher-order AtoB masking scheme and arithmetic operation without conversion. In the implementations, we obtained improvements of about $58 \sim 72\%$ over the generic variant algorithm results. Moreover, when applied to lightweight block cipher SPECK, we obtained improvements of about $36 \sim 43\%$ compared to the generic variant algorithm from [19].

References

1. Kogge, P.M. and Stone, H.S.: A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. Computers, IEEE Transactions on, 100(8), pp. 786-793. (1973)
2. Kocher, P., Jaffe, J., and Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999, LNCS, vol. 1666, pp. 388-397. Springer, Heidelberg (1999)
3. 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)
4. Coron, J.-S. and 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)
5. Ishai, Y., Sahai, A., and Wagner, D.: Private Circuits: Securing Hardware against Probing Attacks. In: Boneh, D. (ed.) CRYPTO 2013, LNCS, vol. 2729, pp. 463-481. Springer, Heidelberg (2003)

6. Neiße, O. and 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)
7. Y.Beak and M.-J. Noh: Differential power attack and masking method. Trends in Mathematics 8, pp.1-15. (2005)
8. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks - Revealing the Secrets of Smart Cards. Springer (2007)
9. Rivain, M., Dottax, E., and 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)
10. Kim, H., Cho, Y.I., Choi, D., Han, D.-G., and Hong, S.: Efficient Masked Implementation for SEED Based on Combined Masking. ETRI Journal 33(2), pp. 267-274. (2011)
11. 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)
12. Vadnala, P.K. and Großschädl, J.: Algorithms for Switching between Boolean and Arithmetic Masking of Second Order. In: Gierlichs, B., Guilley, S., Mukhopadhyay, D. (eds.) SPACE 2013, LNCS, vol. 8204, pp. 95-110. Springer, Heidelberg (2013)
13. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., and Wingers, L.: The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404 (2013)
14. Kim, D.-C., Hong, D., Lee, J.-K., Kim, W.-H., and Kwon, D.: LSH: A New Fast Secure Hash Function Family. In: Lee, J., Kim., J. (eds.) ICISC 2014, LNCS, vol. 8949, pp. 286-313. Springer, Heidelberg (2014)
15. Hong, D., Lee, J.-K., Kim, D.-C., Kwon, D., Ryu, K.H., and Lee D.-G.: LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors. In: Kim, Y., Lee, H., Perrig, A. (eds.) WISA 2014, LNCS, vol. 8267, pp. 3-27. Springer, Heidelberg (2014)
16. Coron, J.-S., Großschädl, J., and Vadnala, P.K.: Secure Conversion between Boolean and Arithmetic Masking of Any Order. In: Batina, L., Robshaw, M. (eds.) CHES 2014, LNCS, vol. 8731, pp. 188-205. Springer, Heidelberg (2014)
17. Karroumi, M., Richard, B., and Joye, M.: Addition with Blinded Operands. In: Prouff, E. (ed.) COSADE 2014, LNCS, vol. 8622, pp. 41-55. Springer, Heidelberg (2014)
18. Vadnala, P.K. and Großschädl, J.: Faster Mask Conversion with Lookup Tables. In: Mangard, S., Poschmann, A.Y. (eds.) COSADE 2015, LNCS, vol. 9064, pp. 207-221. Springer, Heidelberg (2015)
19. Coron, J.-S., Großschädl, J., Tibouchi, M., and Vadnala, P.K.: Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In: Leander, G. (ed.) FSE 2015, LNCS, vol. 9054, pp. 130-149. Springer, Heidelberg (2015)

A Generic Variant for Kogge-Stone Adder and AtoB Masking

A.1 Generic Variant for Kogge-Stone Adder

A Shift algorithm indicates a left shift operation after it has split variable x into some data blocks. Thus, the Shift algorithm has greater associated cost than the general left shift operation.

Algorithm 4 Generic Variant for Kogge-Stone Adder

Input: $x = \left(x_{(m-1)}||\cdots||x_{(0)}\right), y = \left(y_{(m-1)}||\cdots||y_{(0)}\right)$
$n = \max(\lceil \log{(k-1)} \rceil, 1)$
Output: $z = \left(z_{(m-1)}||\cdots||z_{(0)}\right) = x + y \mod 2^k$
1: $\left(p_{(m-1)}||\cdots||p_{(0)}\right) \leftarrow \left(x_{(m-1)}||\cdots||x_{(0)}\right) \oplus \left(y_{(m-1)}||\cdots||y_{(0)}\right)$
2: $\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \left(x_{(m-1)}||\cdots||x_{(0)}\right) \wedge \left(y_{(m-1)}||\cdots||y_{(0)}\right)$
3: **for** $i := 1$ **to** $n-1$ **do**
4: $\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \mathsf{Shift}[g, 2^{i-1}]$
5: $\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \left(p_{(m-1)}||\cdots||p_{(0)}\right) \wedge \left(h_{(m-1)}||\cdots||h_{(0)}\right)$
6: $\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \left(h_{(m-1)}||\cdots||h_{(0)}\right) \oplus \left(g_{(m-1)}||\cdots||g_{(0)}\right)$
7: $\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \mathsf{Shift}[p, 2^{i-1}]$
8: $\left(p_{(m-1)}||\cdots||p_{(0)}\right) \leftarrow \left(p_{(m-1)}||\cdots||p_{(0)}\right) \wedge \left(h_{(m-1)}||\cdots||h_{(0)}\right)$
9: **end for**
10: $\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \mathsf{Shift}[g, 2^{n-1}]$
11: $\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \left(p_{(m-1)}||\cdots||p_{(0)}\right) \wedge \left(h_{(m-1)}||\cdots||h_{(0)}\right)$
12: $\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \left(h_{(m-1)}||\cdots||h_{(0)}\right) \oplus \left(g_{(m-1)}||\cdots||g_{(0)}\right)$
13: $\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \mathsf{Shift}[p, 2^{n-1}]$
14: **return** $\left(x_{(m-1)} \oplus y_{(m-1)} \oplus h_{(m-1)}||\cdots||x_{(0)} \oplus y_{(0)} \oplus h_{(0)}\right)$

A.2 Generic Variant for AtoB Masking based on Kogge-Stone Adder

The generic variant for AtoB masking is a direct application of the previous paper [19]. To compare the enhanced variant version, we use this variant. As before, we are given as input two arithmetic shares A, r of $x = A + r \mod 2^k$, and we must obtain the result x' such that $x = x' \oplus r$, without leaking sensitive variable with respect to x. To protect the leakage of all the intermediate variables, all operations must be converted into secure versions.

Fortunately, Kogge-Stone adder involves uncomplicated operations;, *i.e.*, And, Xor, and Shift. Since it is well known, we use a secure And operation, in particular, as the field multiplication of the S-box operation. It is easy to protect against first-order attack for the other operations. For further details, refer to [19]. In this paper, we denote the secure operations as SecAnd, SecXor, and SecShift, respectively. Finally we can convert Algorithm 4 into a first-order secure algorithm, as in the previous paper, using Algorithm 5.

Algorithm 5 Generic Variant for AtoB Masking

Input: $a = \left(a_{(m-1)}||\cdots||a_{(0)}\right), r = \left(r_{(m-1)}||\cdots||r_{(0)}\right)$
$n = \max(\lceil \log{(k-1)} \rceil, 1)$
Output: $x = \left(x_{(m-1)}||\cdots||x_{(0)}\right)$ such that $x \oplus r = a + r \mod 2^k$
1: $\left(s_{(m-1)}||\cdots||s_{(0)}\right) \leftarrow \{0,1\}^k$
2: $\left(t_{(m-1)}||\cdots||t_{(0)}\right) \leftarrow \{0,1\}^k$
3: $\left(u_{(m-1)}||\cdots||u_{(0)}\right) \leftarrow \{0,1\}^k$

4: $\left(p_{(m-1)}||\cdots||p_{(0)}\right) \leftarrow \left(a_{(m-1)}||\cdots||a_{(0)}\right) \oplus \left(s_{(m-1)}||\cdots||s_{(0)}\right)$
5: $\left(p_{(m-1)}||\cdots||p_{(0)}\right) \leftarrow \left(p_{(m-1)}||\cdots||p_{(0)}\right) \oplus \left(r_{(m-1)}||\cdots||r_{(0)}\right)$
6: $\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \left(a_{(m-1)}||\cdots||a_{(0)}\right) \oplus \left(t_{(m-1)}||\cdots||t_{(0)}\right)$
7: $\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \left(g_{(m-1)}||\cdots||g_{(0)}\right) \wedge \left(r_{(m-1)}||\cdots||r_{(0)}\right)$
8: $\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \left(g_{(m-1)}||\cdots||g_{(0)}\right) \oplus \left(s_{(m-1)}||\cdots||s_{(0)}\right)$
9: $\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \left(t_{(m-1)}||\cdots||t_{(0)}\right) \wedge \left(r_{(m-1)}||\cdots||r_{(0)}\right)$
10: $\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \left(g_{(m-1)}||\cdots||g_{(0)}\right) \oplus \left(h_{(m-1)}||\cdots||h_{(0)}\right)$
11: **for** $i := 1$ **to** $n-1$ **do**
12: $\quad\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \mathsf{SecShift}[g, s, t, 2^{i-1}]$
13: $\quad\left(w_{(m-1)}||\cdots||w_{(0)}\right) \leftarrow \mathsf{SecAnd}[p, h, s, t, u]$
14: $\quad\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \mathsf{SecXor}[g, w, u]$
15: $\quad\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \mathsf{SecShift}[p, s, t, 2^{i-1}]$
16: $\quad\left(p_{(m-1)}||\cdots||p_{(0)}\right) \leftarrow \mathsf{SecAnd}[p, h, s, t, u]$
17: $\quad\left(p_{(m-1)}||\cdots||p_{(0)}\right) \leftarrow \left(p_{(m-1)}||\cdots||p_{(0)}\right) \oplus \left(s_{(m-1)}||\cdots||s_{(0)}\right)$
18: $\quad\left(p_{(m-1)}||\cdots||p_{(0)}\right) \leftarrow \left(p_{(m-1)}||\cdots||p_{(0)}\right) \oplus \left(u_{(m-1)}||\cdots||u_{(0)}\right)$
19: **end for**
20: $\left(h_{(m-1)}||\cdots||h_{(0)}\right) \leftarrow \mathsf{SecShift}[g, s, t, 2^{n-1}]$
21: $\left(w_{(m-1)}||\cdots||w_{(0)}\right) \leftarrow \mathsf{SecAnd}[p, h, s, t, u]$
22: $\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \mathsf{SecXor}[g, w, u]$
23: $\left(g_{(m-1)}||\cdots||g_{(0)}\right) \leftarrow \mathsf{Shift}[g, 1]$
24: $\left(x_{(m-1)}||\cdots||x_{(0)}\right) \leftarrow \left(a_{(m-1)}||\cdots||a_{(0)}\right) \oplus \left(g_{(m-1)}||\cdots||g_{(0)}\right)$
25: $\left(s_{(m-1)}||\cdots||s_{(0)}\right) \leftarrow \mathsf{Shift}[s, 1]$
26: $\left(x_{(m-1)}||\cdots||x_{(0)}\right) \leftarrow \left(x_{(m-1)}||\cdots||x_{(0)}\right) \oplus \left(s_{(m-1)}||\cdots||s_{(0)}\right)$
27: **return** $\left(x_{(m-1)}||\cdots||x_{(0)}\right)$

B Remaining proof of Theorem 1

Here, we provide the remaining proof of Theorem 1, and thereby prove the following Lemma.

Lemma 3. *$\sum_{i=0}^{m-1} x_{(i)}$ is equal to $\sum_{i=0}^{m} X'_{(i)}$.*

Proof.

$$\begin{aligned}
&\sum_{i=0}^{m-1} x_{(i)} \\
&= x_{(0)} + x_{(1)} + \cdots + x_{(m-2)} + x_{(m-1)} \\
&= 2^0 \left(2^0 x^{(0)} + \cdots + 2^{l-1} x^{(l-1)}\right) \\
&\quad + 2^l \left(2^0 x^{(l+0)} + \cdots + 2^{l-1} x^{(l+l-1)}\right) \\
&\quad + \cdots
\end{aligned}$$

$$
\begin{aligned}
&+ 2^{l(m-2)} \left(2^0 x^{(l(m-2)+0)} + \cdots + 2^{l-1} x^{(l(m-2)+l-1)}\right) \\
&+ 2^{l(m-1)} \left(2^0 x^{(l(m-1)+0)} + \cdots + 2^{l-1} x^{(l(m-1)+l-1)}\right) \\
= &\, 2^0 \left(2^0 x^{(0)} + \cdots + 2^{l-1} x^{(l-1)}\right) \\
&+ 2^{l-1} \left(2^1 x^{((l-1)+1)} + \cdots + 2^{l-1} x^{((l-1)+l-1)}\right) \\
&+ \cdots \\
&+ 2^{(l-1)(m-1)} \left(2^1 x^{((l-1)(m-1)+1)} + \cdots + 2^{l-1} x^{((l-1)(m-1)+l-1)}\right) \\
&+ 2^{(l-1)m} \left(2^1 x^{((l-1)m+1)} + \cdots + 2^{l-1} x^{((l-1)m+(m-1))}\right) \\
= &\, X'_{(0)} + \cdots + X'_{(m-1)} + X'_{(m)} = \sum_{i=0}^{m} X'_{(i)}
\end{aligned}
$$

□

Lemma 4. *$d^{(j)}$ is equal to $c^{(i(l-1)+j-1)}$.*

Proof. We proceed to this Lemma by mathematical induction.

Show that the equation of Lemma 4 holds for $j = 0$.

$$
\begin{aligned}
d^{(1)} &= \{(a^{(0)} \oplus b^{(0)}) \wedge d^{(0)}\} \oplus (a^{(0)} \wedge b^{(0)}) \\
&= \{(c^{(i(l-1))} \oplus c^{(i(l-1))}) \wedge d^{(0)}\} \oplus (c^{(i(l-1))} \wedge c^{(i(l-1))}) \\
&= (0 \wedge d^{(0)}) \oplus (c^{(i(l-1))} \wedge c^{(i(l-1))}) = c^{(i(l-1))}
\end{aligned}
$$

Show that if the equation of Lemma 4 holds for $j = k$, then also the equation holds for $j = k + 1$. (That is, $d^{(k)} = c^{(i(l-1)+k-1)}$)

$$
\begin{aligned}
&d^{(k+1)} \\
&= \{(a^{(k)} \oplus b^{(k)}) \wedge d^{(k)}\} \oplus (x^{(i(l-1)+k-1)} \wedge y^{(i(l-1)+k-1)}) \\
&= \{(x^{(i(l-1)+k-1)} \wedge y^{(i(l-1)+k-1)}) \wedge c^{(i(l-1)+k-1)}\} \oplus (x^{(i(l-1)+k-1)} \wedge y^{(i(l-1)+k-1)}) \\
&= c^{(i(l-1)+k)}
\end{aligned}
$$

□

C Enhanced Variant for AtoB Masking ($m = 2$)

Algorithm 6 Enhanced Variant for AtoB Masking ($m = 2$)

Input: $a = \left(a_{(m-1)}||\cdots||a_{(0)}\right), r = \left(r_{(m-1)}||\cdots||r_{(0)}\right) \in \{0,1\}^k$
$n = \max(\lceil \log (l-1) \rceil, 1)$
Output: $x = \left(x_{(m-1)}||\cdots||x_{(0)}\right)$ such that $x \oplus r = a + r \mod 2^k$
1: $s \leftarrow \{0,1\}^l$, $t \leftarrow \{0,1\}^l$, $u \leftarrow \{0,1\}^l$, $\delta \leftarrow \{0,1\}^l$
2: $a_{(0)} \leftarrow \left(a^{(l-1)}||\cdots||a^{(0)}\right)$, $r_{(0)} \leftarrow \left(r^{(l-1)}||\cdots||r^{(0)}\right)$, $C \leftarrow \delta$
3: **for** $i := 1$ **to** m **do**
4: $\quad a_{(i)} \leftarrow \left(a^{(i(l-1)+l-2)}||\cdots||a^{(i(l-1)+0)}||0\right)$
5: $\quad r_{(i)} \leftarrow \left(r^{(i(l-1)+l-2)}||\cdots||r^{(i(l-1)+0)}||0\right)$
6: **end for**
7: **for** $j := 0$ **to** $m-1$ **do**
8: $\quad P \leftarrow a_{(j)} \oplus s$
9: $\quad P \leftarrow P \oplus r_{(j)}$
10: $\quad G \leftarrow s \oplus \left(\left(a_{(j)} \oplus t\right) \wedge r_{(j)}\right) \oplus C$
11: $\quad G \leftarrow G \oplus \left(t \wedge r_{(j)}\right) \oplus \delta$
12: $\quad$ **for** $i := 1$ **to** $n-1$ **do**
13: $\quad\quad H \leftarrow \mathsf{SecShift}_l[G, s, t, 2^{i-1}]$
14: $\quad\quad W \leftarrow \mathsf{SecAnd}_l[P, H, s, t, u]$
15: $\quad\quad G \leftarrow \mathsf{SecXor}_l[G, W, u]$
16: $\quad\quad H \leftarrow \mathsf{SecShift}_l[P, s, t, 2^{i-1}]$
17: $\quad\quad P \leftarrow \mathsf{SecAnd}_l[P, H, s, t, u]$
18: $\quad\quad P \leftarrow P \oplus s$
19: $\quad\quad P \leftarrow P \oplus u$
20: $\quad$ **end for**
21: $\quad H \leftarrow \mathsf{SecShift}_l[G, s, t, 2^{n-1}]$
22: $\quad W \leftarrow \mathsf{SecAnd}_l[P, H, s, t, u]$
23: $\quad G \leftarrow \mathsf{SecXor}_l[G, W, u]$
24: $\quad X'_{(j)} \leftarrow a_{(j)} \oplus (2G)$
25: $\quad X'_{(j)} \leftarrow X'_{(j)} \oplus (2s)$
26: $\quad$ **if** $j \neq 0$ **then**
27: $\quad\quad X'_{(j)} \leftarrow X'_{(j)} \gg 1$
28: $\quad$ **end if**
29: $\quad C \leftarrow [\{G \gg (l-1)\} \oplus \delta] \oplus \{s \gg (l-1)\}$
30: **end for**
31: $x_{(m)} \leftarrow \left(a_{(m)} \oplus C\right) \oplus \delta$
32: $\left(x_{(m-1)}||\cdots||x_{(0)}\right) \leftarrow \left(X'_{(m)}||\cdots||X'_{(0)}\right) \mod 2^k$
33: **return** $x = \left(x_{(m-1)}||\cdots||x_{(0)}\right)$