Published in E. Prouff, Ed., *Constructive Side-Channel Analysis and Secure Design (COSADE 2014)*, vol. 8622 of *Lecture Notes in Computer Science*, pp. 41–55, Springer, 2014.

Addition with Blinded Operands

Mohamed Karroumi[1], Benjamin Richard[1], and Marc Joye[2]

[1] Technicolor
975 avenue des Champs Blancs, 35576 Cesson-Sévigné Cedex, France
{mohamed.karroumi,benjamin.richard}@technicolor.com
[2] Technicolor
175 S San Antonio Rd, Los Altos, CA 94022, USA
marc.joye@technicolor.com

Abstract. The masking countermeasure is an efficient method to protect cryptographic algorithms against Differential Power Analysis (DPA) and similar attacks. For symmetric cryptosystems, two techniques are commonly used: Boolean masking and arithmetic masking. Conversion methods have been proposed for switching from Boolean masking to arithmetic masking, and conversely. The way conversion is applied depends on the combination of arithmetic and Boolean/logical operations executed by the underlying cryptographic algorithm.
This paper focuses on a combination of one addition with one or more Boolean operations. Building on a secure version of a binary addition algorithm (namely, the AND-XOR-and-double method), we show that conversions from Boolean masking to arithmetic masking can be avoided. We present an application of the new algorithm to the XTEA block-cipher.

Keywords: Masking methods, differential power analysis (DPA), side-channel attacks, binary addition, block ciphers, XTEA.

1 Introduction

Differential power analysis DPA and related attacks, introduced by Kocher *et al.* in [13], exploit side-channel leakage to uncover secret information. During the execution of a cryptographic algorithm, the secret key or some related information may be revealed by monitoring the power consumption of the electronic device executing the cryptographic algorithm. DPA-type attacks potentially apply to all cryptosystems, including popular block ciphers like AES (*e.g.*, [19]). Protection against DPA is achieved thanks to randomization techniques. The commonly suggested way to thwart DPA-type attacks for implementations of block ciphers is random masking [2,9,3]. The idea is to blind sensitive data with a random mask at the beginning of the algorithm execution. The algorithm is then executed as usual. Of course, at some step within a round the value of the mask (or a value derived thereof) must be known in order to correct the corresponding output value. This general technique is referred to as the *duplication method* or the *splitting method*. The *transformed masking method* [1] is a specialized technique wherein the *same* mask is used throughout the computation.

More specifically, all intermediate values are XORed with a random mask and the inner operations are modified such that the output of a round is masked by the same mask as that of the input. This was for example applied to DES by modifying its non-linear components (namely, the original S-boxes were replaced with modified S-boxes so as to output the correct masked values), which resulted in an implementation shown to be secure against (first-order) DPA attacks.

Masking and switching methods For block ciphers involving different types of operations, two masking techniques must usually be used: a Boolean masking (generally by applying an XOR) and an arithmetic masking. Both techniques were for instance used for protecting the AES finalists against DPA [16]. Further, as shown in [16], it is useful to have efficient and secure methods for switching from Boolean masking to arithmetic masking, and conversely. The algorithm suggested in [16] was however shown to be vulnerable to a 2-bit DPA attack in [3]. A more secure algorithm was later proposed by Goubin in [8]. The algorithm works in both directions. The secure Arithmetic-to-Boolean (A→B) conversion is however less efficient than the secure Boolean-to-arithmetic (B→A) conversion as its complexity depends on the length of the values being masked. This issue was addressed by Coron and Tchulkine in [4] with a method using Look-Up-Tables (LUTs). In [18], an extension to the table-based algorithm of [4] was proposed, reducing the memory footprint. Another improved version can be found in a recent paper by Debraize [5].

TEA family The TEA block ciphers [21,17,22] are ciphers designed by Needham and Wheeler, featuring a 128-bit key-size and running over (at least) 64 rounds. They are based on a Feistel structure without use of any S-box, nor any key expansion routines. The ciphers make alternate use of XOR, Shift and modular addition, resulting in simple, efficient, and easy to implement algorithms. The *XTEA cipher* [17] was later proposed as an improvement to TEA to counter the attacks of [10]. The TEA family block-ciphers enjoys several salient features making it attractive for light-weight applications: simplicity, minimal key-setup, no look-up tables, and small footprint.

Our contribution The masking problem for (modular) addition can be stated as how to *securely* compute the (modular) addition of k-bit integers x and y from *masked* inputs and the corresponding masks, namely $(\boldsymbol{x}, \boldsymbol{y})$ and (r_x, r_y) where $\boldsymbol{x} = x \oplus r_x$ and $\boldsymbol{y} = y \oplus r_y$, while ensuring that the result, $s = x + y \pmod{2^k}$, is still masked with some Boolean mask r_s — 'securely' here has to be understood as in a way resistant against first-order DPA-type attacks.

A classical solution to this problem is to rely on secure mask-switching methods. Blinded values $\boldsymbol{x} = x \oplus r_x$ and $\boldsymbol{y} = y \oplus r_y$ are first converted into values that are *arithmetically* masked, $\boldsymbol{x'} = x - r_x$ and $\boldsymbol{y'} = y - r_y$, using a secure B→A switching algorithm. Next, the resulting values and their masks are separately added:

$$\boldsymbol{s'} = \boldsymbol{x'} + \boldsymbol{y'} \quad \text{and} \quad r_{s'} = r_x + r_y \ .$$

Noticing that $\boldsymbol{s'} = (x + y) - r_{s'}$, the blinded sum $\boldsymbol{s} = (x + y) \oplus r_{s'}$ is obtained through a secure A→B switching algorithm. This is illustrated in Fig. 1b.

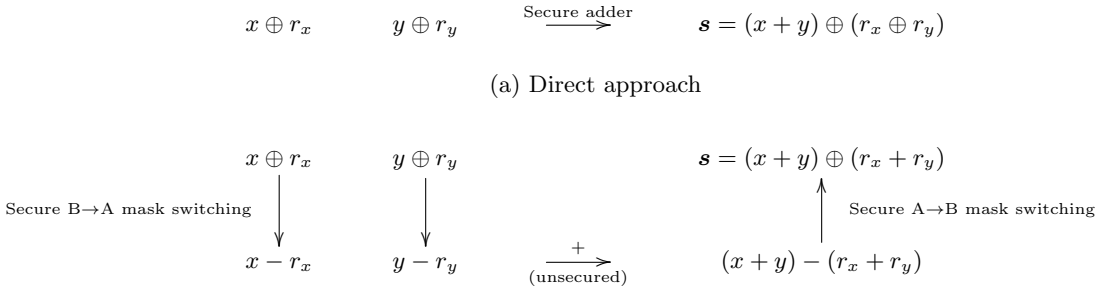

Fig. 1: Solving the masking problem

This paper tackles the masking problem through a more direct approach as described in Fig. 1a. Such an approach was already alluded in [7] where hardware-based solutions using ripple-carry addition methods are presented. These methods are not suited to software implementations and imply dedicated hardware. Dedicated hardware for a specific cryptographic application is in general not available and is expensive to implement. We propose in this paper an example of an algorithm that is *faster* and more *compact* than previous methods —including solutions built on Goubin's method and the table-based methods— for securely adding two blinded operands. The proposed implementation is therefore well adapted to memory-constrained environments. Implementations for securely subtracting two blinded operands and variants thereof are also detailed. We show that the introduced algorithm can be applied to XTEA (and its variants) to protect against DPA-type attacks. The countermeasure advantageously retains the efficiency of the unprotected implementations in terms of memory requirements and speed.

Outline of the paper The rest of this paper is organized as follows. In the next section, we introduce the notation that is used throughout. Section 3 is the core of the paper. We review the ADD-XOR-and-double addition algorithm and then derive therefrom an addition algorithm secure against DPA-type attacks. In Section 4, we analyze the security of the proposed algorithm. In Section 5, we present our approach to thwart DPA on XTEA using various algorithms and evaluate their performance. In Section 6, other applications and extensions of our algorithms are proposed. Finally, we conclude in Section 7.

2 Notation

This section introduces some notation. Following [12, Section 7.1.3], given three integers in their binary notation, namely $x = (\dots x_2 x_1 x_0)_2$, $y = (\dots y_2 y_1 y_0)_2$, and $z = (\dots z_2 z_1 z_0)_2$, we write

$$x \,\&\, y = z \iff z_i = x_i \wedge y_i, \text{ for all } i \geqslant 0\,;$$
$$x \oplus y = z \iff z_i = x_i \oplus y_i, \text{ for all } i \geqslant 0\,;$$

where $\wedge$ and $\oplus$ respectively denote the Boolean operators `and` and `xor` (exclusive or). It is easily verified that the bitwise operators & and $\oplus$ satisfy the following properties:

- [Commutativity] $x \,\&\, y = y \,\&\, x$, $x \oplus y = y \oplus x$;
- [Distributivity] $(x \oplus y) \,\&\, z = (x \,\&\, z) \oplus (y \,\&\, z)$.

We will also make use of the logical left shift operator. For a positive integer t, we write

$$x \ll t = y \iff y_{i+t} = x_i, \text{ for all } i \geqslant 0, \text{ and } y_0, \dots, y_{t-1} = 0 \ .$$

Notice that $x \ll t = 2^t\, x$. Hence we will sometimes write $2x$ instead of $x \ll 1$.

Throughout the paper, unless otherwise indicated, we assume that the involved operands are k-bit integers (typically of 8, 16, 32 or 64 bits) and arithmetic operations are performed modulo 2^k. Modular addition and subtraction are noted "+" and "−", respectively. Likewise, unless otherwise indicated, the shifting operations are performed modulo 2^k. To ease the notation, we sometimes omit writing the congruence operation (*i.e.*, (mod 2^k) is implicit). Finally, we will use boldface symbols to represent masked values; for example, $\boldsymbol{x}$ will denote a masked value for x.

3 Boolean Masking and Addition

3.1 Basic algorithm

Let x and y be two k-bit integers viewed as elements of $\mathbb{Z}_{2^k} = \{0, \dots, 2^k - 1\}$. The goal is to compute their sum $s = x + y \pmod{2^k}$. Letting $x = \sum_{i=0}^{k-1} x_i\, 2^i$ and $y = \sum_{i=0}^{k-1} y_i\, 2^i$ the respective binary expansions of x and y, the pencil-and-paper method to add non-negative integers [11, p. 251] yields $s = \sum_{i=0}^{k-1} s_i\, 2^i$ in a left-to-right fashion as:

$$c_0 = 0 \quad \text{and} \quad \begin{cases} s_i = (x_i + y_i + c_i) \bmod 2 \\ c_{i+1} = (x_i + y_i + c_i) \operatorname{div} 2 \end{cases}$$

for $0 \leqslant i \leqslant k-1$. It is readily seen that the carry-out, c_{i+1}, is equal to 1 if and only if at least two of x_i, y_i and c_i are 1. Hence the previous relation can be rewritten using logical operators as

$$c_0 = 0 \quad \text{and} \quad \begin{cases} s_i = x_i \oplus y_i \oplus c_i \\ c_{i+1} = \mathrm{Maj}\,(x_i, y_i, c_i) \end{cases} \tag{1}$$

using the majority function Maj, given by

$$\begin{aligned} \mathrm{Maj}\,(x_i, y_i, c_i) &:= (x_i \,\&\, y_i) \oplus (x_i \,\&\, c_i) \oplus (y_i \,\&\, c_i) \\ &= (x_i \,\&\, y_i) \oplus \big[(x_i \oplus y_i) \,\&\, c_i\big] = c_i \,\&\, (x_i \oplus y_i) \oplus (x_i \,\&\, y_i) \ . \end{aligned}$$

Summing up, given $x, y \in \mathbb{Z}_{2^k}$, their sum (modulo 2^k) can be obtained as

$$s = x \oplus y \oplus c \quad \text{with } c = \sum_{i=1}^{k-1} c_i \, 2^i \,, \tag{2}$$

where $c_0 = 0$ and $c_i = c_{i-1} \,\&\, (x_{i-1} \oplus y_{i-1}) \oplus (x_{i-1} \,\&\, y_{i-1})$ for $1 \leqslant i \leqslant k-1$.

Since c is defined modulo 2^k, we immediately get from Eqs. (2) and (1)

$$\begin{aligned} c = \sum_{i=1}^{k-1} c_i \, 2^i = \sum_{i=1}^{k} c_i \, 2^i = 2 \sum_{i=0}^{k-1} c_{i+1} \, 2^i = 2 \sum_{i=0}^{k-1} \big[c_i \,\&\, (x_i \oplus y_i) \oplus (x_i \,\&\, y_i)\big] 2^i \\ = 2\big[c \,\&\, (x \oplus y) \oplus (x \,\&\, y)\big] \pmod{2^k} \ . \end{aligned}$$

This suggests to obtain the value of c by iterating the relation

$$c \leftarrow 2\big[c \,\&\, (x \oplus y) \oplus (x \,\&\, y)\big] \tag{3}$$

where c is initialized to 0. This yields the following addition algorithm. See also [14] and [8, Theorem 2].

Algorithm 1 AND-XOR-and-double addition method

Input: $(x, y) \in \mathbb{Z}_{2^k} \times \mathbb{Z}_{2^k}$
Output: $x + y \pmod{2^k}$

1: $\mathsf{A} \leftarrow x$; $\mathsf{B} \leftarrow y$
2: $\mathsf{C} \leftarrow \mathsf{A} \,\&\, \mathsf{B}$; $\mathsf{A} \leftarrow \mathsf{A} \oplus \mathsf{B}$
3: $\mathsf{B} \leftarrow 0$
4: **for** $i = 1$ to $k-1$ **do**
5: $\quad \mathsf{B} \leftarrow \mathsf{B} \,\&\, \mathsf{A}$; $\mathsf{B} \leftarrow \mathsf{B} \oplus \mathsf{C}$
6: $\quad \mathsf{B} \leftarrow \mathsf{B} \ll 1$
7: **end for**
8: $\mathsf{A} \leftarrow \mathsf{A} \oplus \mathsf{B}$
9: **return** A

3.2 DPA-resistant addition

We now consider the case of masked inputs, namely x and y are blinded as

$$\boldsymbol{x} = x \oplus r_x \quad \text{and} \quad \boldsymbol{y} = y \oplus r_y$$

for some Boolean masks $r_x, r_y \in \mathbb{Z}_{2^k}$. The goal is to securely compute $(\boldsymbol{s}, r_s)$ where $\boldsymbol{s} = (x + y) \oplus r_s$ for some mask $r_s \in \mathbb{Z}_{2^k}$, from $(\boldsymbol{x}, r_x)$ and $(\boldsymbol{y}, r_y)$ and without compromising the values of x or of y through DPA.

We rely on Algorithm 1; an application of Equation (2) yields

$$\begin{aligned}\boldsymbol{s} = (x + y) \oplus r_s &= (x \oplus y \oplus c) \oplus r_s = (\boldsymbol{x} \oplus r_x) \oplus (\boldsymbol{y} \oplus r_y) \oplus c \oplus r_s \\ &= \boldsymbol{x} \oplus \boldsymbol{y} \oplus c\end{aligned}$$

by setting $r_s = r_x \oplus r_y$. The carry c in the above formula results from the addition of x and y in the clear! As a consequence, if not carefully done, its evaluation might leak information on x or y by mounting a DPA-type attack. In order to solve this issue, in a way analogous to [8], we initialize the value of c with a Boolean mask γ when evaluating Eq. (3). In more detail, letting $c^{(i)}$ the output value of c in Eq. (3) at iteration i and $\boldsymbol{c}^{(i)} = c^{(i)} \oplus 2\gamma$, we have

$$\begin{cases} \boldsymbol{c}^{(0)} = 2\gamma \\ \boldsymbol{c}^{(i)} = 2\big[\boldsymbol{c}^{(i-1)} \,\&\, (x \oplus y) \oplus \Omega\big], \quad \text{for } 1 \leqslant i \leqslant k-1 \end{cases} \tag{4}$$

where $\Omega = 2\gamma \,\&\, (x \oplus y) \oplus (x \,\&\, y) \oplus \gamma$.

Proof. We have

$$\begin{aligned}\boldsymbol{c^{(i)}} = c^{(i)} \oplus 2\gamma &= 2\big[c^{(i-1)} \,\&\, (x \oplus y) \oplus (x \,\&\, y)\big] \oplus 2\gamma \qquad \text{by Eq. (3)} \\ &= 2\big[(\boldsymbol{c}^{(i-1)} \oplus 2\gamma) \,\&\, (x \oplus y) \oplus (x \,\&\, y)\big] \oplus 2\gamma \\ &= 2\big[(\boldsymbol{c}^{(i-1)} \,\&\, (x \oplus y)) \oplus (2\gamma \,\&\, (x \oplus y)) \oplus (x \,\&\, y)\big] \oplus 2\gamma \\ &= 2\big[\boldsymbol{c}^{(i-1)} \,\&\, (x \oplus y) \oplus \Omega\big]\end{aligned}$$

as expected. □

Given that $\boldsymbol{c}^{(0)} = 2\gamma$ and the definition of Ω, it is interesting to note that the value of $\boldsymbol{c}^{(1)}$ simplifies to

$$\boldsymbol{c}^{(1)} = 2\big[2\gamma \,\&\, (x \oplus y) \oplus \Omega\big] = 2\big[(x \,\&\, y) \oplus \gamma\big] \ .$$

Hence, letting $\Omega_0 = (x \& y) \oplus \gamma$, we can write $\boldsymbol{c}^{(1)} = 2\Omega_0$ and $\Omega = 2\gamma \& (x \oplus y) \oplus \Omega_0$.

Remark 1. We remark that a similar trick applies to Goubin's arithmetic-to-Boolean conversion. Rearranging the operations leads to a reduced cost, from a total of $5k + 5$ operations down to $\underline{5k + 1}$ operations. This optimized variant is detailed in Appendix A.

It remains to express $\boldsymbol{c}^{(i)}$ and Ω as a function of $(\boldsymbol{x}, \boldsymbol{y}, r_x, r_y)$. From Eq. (4), we have

$$\begin{aligned}\boldsymbol{c}^{(i)} &= 2\big[\boldsymbol{c}^{(i-1)} \,\&\, (\boldsymbol{x} \oplus \boldsymbol{y} \oplus r_x \oplus r_y) \oplus \Omega\big] \\ &= 2\big[\big(\boldsymbol{c}^{(i-1)} \,\&\, (\boldsymbol{x} \oplus \boldsymbol{y})\big) \oplus \big(\boldsymbol{c}^{(i-1)} \,\&\, (r_x \oplus r_y)\big) \oplus \Omega\big] \ .\end{aligned}$$

We also have

$$\Omega = 2\gamma \,\&\, (\boldsymbol{x} \oplus \boldsymbol{y} \oplus r_x \oplus r_y) \oplus \Omega_0 = \big[2\gamma \,\&\, (\boldsymbol{x} \oplus \boldsymbol{y})\big] \oplus \big[2\gamma \,\&\, (r_x \oplus r_y)\big] \oplus \Omega_0 \ .$$

We need to introduce a useful theorem from [8]:

Theorem 1 (Goubin). *Using previous notations, for any $\delta \in \mathbb{Z}_{2^k}$, function*

$$\Theta_\delta : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^k}, \gamma \mapsto \big[(2\gamma) \,\&\, \delta\big] \oplus \gamma$$

is bijective. □

Assume that γ is uniformly distributed over $\mathbb{Z}_{2^k}$. The previous theorem implies that $\big[(2\gamma) \,\&\, (x \oplus y)\big] \oplus \gamma$ is uniformly distributed over $\mathbb{Z}_{2^k}$. In turn, this implies that $\Omega = \big[(2\gamma) \,\&\, (x \oplus y)\big] \oplus \gamma \oplus (x \,\&\, y)$ is uniformly distributed over $\mathbb{Z}_{2^k}$. We exploit this observation and evaluate $\boldsymbol{c}^{(i)}$ as

$$\boldsymbol{c}^{(i)} = 2\big[\big(\boldsymbol{c}^{(i-1)} \,\&\, (\boldsymbol{x} \oplus \boldsymbol{y})\big) \oplus \Omega \oplus \big(\boldsymbol{c}^{(i-1)} \,\&\, (r_x \oplus r_y)\big)\big] \ .$$

Algorithmically, we implement this as a for-loop using two accumulators, B and T. At the end of the for-loop, accumulator B contains the value of $\boldsymbol{c} = c \oplus 2\gamma$.

1: $\mathsf{A_0} \leftarrow \boldsymbol{x} \oplus \boldsymbol{y}$; $\mathsf{A_1} \leftarrow r_x \oplus r_y$
2: $\mathsf{B} \leftarrow \boldsymbol{c}^{(1)}$; $\Omega \leftarrow \Omega$
3: **for** $i = 2$ to $k - 1$ **do**
4: $\quad \mathsf{T} \leftarrow \mathsf{B} \,\&\, \mathsf{A_0}$; $\mathsf{B} \leftarrow \mathsf{B} \,\&\, \mathsf{A_1}$
5: $\quad \mathsf{B} \leftarrow \mathsf{B} \oplus \Omega$; $\mathsf{B} \leftarrow \mathsf{B} \oplus \mathsf{T}$
6: $\quad \mathsf{B} \leftarrow \mathsf{B} \ll 1$
7: **end for**

Similarly, noting that if γ is uniformly distributed over $\mathbb{Z}_{2^k}$ then so is $\Omega_0 = \gamma \oplus (x \,\&\, y)$, we implement the calculation of Ω as:

1: $\mathsf{A_0} \leftarrow \boldsymbol{x} \oplus \boldsymbol{y}$; $\mathsf{A_1} \leftarrow r_x \oplus r_y$
2: $\mathsf{C} \leftarrow 2\gamma$; $\Omega \leftarrow \Omega_0$
3: $\mathsf{T} \leftarrow \mathsf{C} \,\&\, \mathsf{A_0}$; $\Omega \leftarrow \Omega \oplus \mathsf{T}$
4: $\mathsf{T} \leftarrow \mathsf{C} \,\&\, \mathsf{A_1}$; $\Omega \leftarrow \Omega \oplus \mathsf{T}$

The last step is the secure evaluation of Ω_0 (*i.e.*, $\gamma \oplus (x \& y)$) from $(\boldsymbol{x}, \boldsymbol{y}, r_x, r_y)$. We make use of a trick already used in [20]. It exploits the distributive property of the AND over the XOR and evaluates an AND as a series of four AND operations calculated pairwise between masked operands and masks (operations are carried

out with masked operands and masks independent from each other). Specifically we implement the calculation of Ω_0 as the left-to-right evaluation of:

$$\begin{aligned}\Omega_0 &= \gamma \oplus (x \,\&\, y) = \gamma \oplus \left[(\boldsymbol{x} \oplus r_x) \,\&\, (\boldsymbol{y} \oplus r_y)\right] \\ &= \gamma \oplus (\boldsymbol{x} \,\&\, \boldsymbol{y}) \oplus (\boldsymbol{x} \,\&\, r_y) \oplus (\boldsymbol{y} \,\&\, r_x) \oplus (r_x \,\&\, r_y) \ .\end{aligned}$$

Putting all together we obtain a DPA-protected addition algorithm. Our secure addition algorithm is depicted in Alg. 2. It makes use of 3 additional temporary k-bit variables (C, T, and Ω), generates one random mask, and requires $\underline{5k+8}$ operations:

- $(2k+6)$ XORs ;
- $(2k+2)$ ANDs ;
- k logical shifts .

Algorithm 2 Secure addition with blinded operands

Input: $(\boldsymbol{x}, \boldsymbol{y}, r_x, r_y, \gamma) \in \mathbb{Z}_{2^k}^5$ such that $\boldsymbol{x} = x \oplus r_x$, $\boldsymbol{y} = y \oplus r_y$ and γ a pre-computed random integer
Output: $(\boldsymbol{s}, r_s)$ where $\boldsymbol{s} = (x+y) \oplus r_s \pmod{2^k}$ and $r_s = r_x \oplus r_y$

/* $\Omega_0 = (x \,\&\, y) \oplus \gamma$ */
1: $\mathsf{C} \leftarrow \gamma$
2: $\mathsf{T} \leftarrow \boldsymbol{x} \,\&\, \boldsymbol{y}$; $\Omega \leftarrow \mathsf{C} \oplus \mathsf{T}$
3: $\mathsf{T} \leftarrow \boldsymbol{x} \,\&\, r_y$; $\Omega \leftarrow \Omega \oplus \mathsf{T}$
4: $\mathsf{T} \leftarrow \boldsymbol{y} \,\&\, r_x$; $\Omega \leftarrow \Omega \oplus \mathsf{T}$
5: $\mathsf{T} \leftarrow r_x \,\&\, r_y$; $\Omega \leftarrow \Omega \oplus \mathsf{T}$ ▷ $\Omega \leftarrow \Omega_0$
/* $\boldsymbol{c}^{(1)} = 2\Omega_0$ and $\Omega = 2\gamma \,\&\, (x \oplus y) \oplus \Omega_0$ */
6: $\mathsf{B} \leftarrow \Omega \ll 1$; $\mathsf{C} \leftarrow \mathsf{C} \ll 1$ ▷ $\mathsf{B} \leftarrow \boldsymbol{c}^{(1)}$; $\mathsf{C} \leftarrow 2\gamma$
7: $\mathsf{A}_0 \leftarrow \boldsymbol{x} \oplus \boldsymbol{y}$; $\mathsf{A}_1 \leftarrow r_x \oplus r_y$
8: $\mathsf{T} \leftarrow \mathsf{C} \,\&\, \mathsf{A}_0$; $\Omega \leftarrow \Omega \oplus \mathsf{T}$
9: $\mathsf{T} \leftarrow \mathsf{C} \,\&\, \mathsf{A}_1$; $\Omega \leftarrow \Omega \oplus \mathsf{T}$ ▷ $\Omega \leftarrow \Omega$
/* Main loop */
10: **for** $i = 2$ to $k-1$ **do**
11: $\quad \mathsf{T} \leftarrow \mathsf{B} \,\&\, \mathsf{A}_0$; $\mathsf{B} \leftarrow \mathsf{B} \,\&\, \mathsf{A}_1$
12: $\quad \mathsf{B} \leftarrow \mathsf{B} \oplus \Omega$
13: $\quad \mathsf{B} \leftarrow \mathsf{B} \oplus \mathsf{T}$
14: $\quad \mathsf{B} \leftarrow \mathsf{B} \ll 1$
15: **end for** ▷ $\mathsf{B} \leftarrow c \oplus 2\gamma$
/* Aggregation */
16: $\mathsf{A}_0 \leftarrow \mathsf{A}_0 \oplus \mathsf{B}$
17: $\mathsf{A}_0 \leftarrow \mathsf{A}_0 \oplus \mathsf{C}$
18: **return** $(\mathsf{A}_0, \mathsf{A}_1)$

4 Security analysis

In this section, we study more formally the DPA resistance of the proposed algorithm. An algorithm is *first-order secure* if the intermediate variables do not reveal any information about the sensitive data. To prove first-order DPA resistance, we first list all intermediate variables of the algorithm. We show then that no variable exhibits dependency on the sensitive data. In the sequel, we let V_i denote the values resulting from the intermediate operation performed at Line i in Alg. 2.

Table 1: Intermediate variables

V_i	Values		
V_1		$\mathsf{C} = \gamma$	
V_2		$\mathsf{T} = (x \oplus r_x)\ \&\ (y \oplus r_y)$	$\Omega = (x \oplus r_x)\ \&\ (y \oplus r_y) \oplus \gamma$
V_3		$\mathsf{T} = (x \oplus r_x)\ \&\ r_y$	$\Omega = (x \oplus r_x)\ \&\ y \oplus \gamma$
V_4		$\mathsf{T} = (y \oplus r_y)\ \&\ r_x$	$\Omega = x\ \&\ y \oplus r_x\ \&\ r_y \oplus \gamma$
V_5		$\mathsf{T} = r_x\ \&\ r_y$	$\Omega = x\ \&\ y \oplus \gamma$
V_6	$\mathsf{B} = 2 \cdot (x\ \&\ y \oplus \gamma) = c^{(1)} \oplus 2\gamma$	$\mathsf{C} = 2\gamma$	
V_7		$\mathsf{A}_0 = (x \oplus r_x) \oplus (y \oplus r_y)$	$\mathsf{A}_1 = r_x \oplus r_y$
V_8		$\mathsf{T} = 2\gamma\ \&\ (x \oplus r_x \oplus y \oplus r_y)$	$\Omega = \gamma \oplus 2\gamma\ \&\ (x \oplus y \oplus r_x \oplus r_y) \oplus x\ \&\ y$
V_9		$\mathsf{T} = 2\gamma\ \&\ (r_x \oplus r_y)$	$\Omega = \gamma \oplus 2\gamma\ \&\ (x \oplus y) \oplus x\ \&\ y$
V_{11}	$\mathsf{B}^{(\mathsf{i})} = (c^{(i-1)} \oplus 2\gamma)\ \&\ (r_x \oplus r_y)$	$\mathsf{T} = (c^{(i-1)} \oplus 2\gamma)\ \&\ (x \oplus y \oplus r_x \oplus r_y)$	
V_{12}	$\mathsf{B}^{(\mathsf{i})} = (c^{(i-1)} \oplus 2\gamma)\ \&\ (r_x \oplus r_y) \oplus \big(\gamma \oplus 2\gamma\ \&\ (x \oplus y) \oplus x\ \&\ y\big)$		
V_{13}	$\mathsf{B}^{(\mathsf{i})} = \gamma \oplus c^{(i-1)}\ \&\ (x \oplus y) \oplus x\ \&\ y$		
V_{14}	$\mathsf{B}^{(\mathsf{i})} = 2 \cdot \big(\gamma \oplus c^{(i-1)}\ \&\ (x \oplus y) \oplus x\ \&\ y\big) = c^{(i)} \oplus 2\gamma$		
V_{16}		$\mathsf{A}_0 = x \oplus y \oplus r_x \oplus r_y \oplus c^{(k-1)} \oplus 2\gamma$	
V_{17}		$\mathsf{A}_0 = (x \oplus y \oplus c^{(k-1)}) \oplus r_x \oplus r_y = (x + y) \oplus r_x \oplus r_y$	

We consider that inputs x and y are respectively masked with r_x and r_y, two random variables uniformly distributed over $\mathbb{Z}_{2^k}$. The sensitive variables and their associated masks are then assumed mutually independent. Furthermore, the two random variables r_x and r_y are chosen independently, which gives $r_s = r_x \oplus r_y$ to be also uniform over $\mathbb{Z}_{2^k}$.

A bitwise AND operation between two independent Boolean masked variables does not give a uniformly distributed result. It has however the same distribution as an AND applied to two random variables. Values of T in steps V_2, V_3, V_4, V_5 and values of $\mathsf{T} = (c^{(i-1)} \oplus 2\gamma)\ \&\ (r_s \oplus x \oplus y)$ and $\mathsf{B}^{(\mathsf{i})} = (c^{(i-1)} \oplus 2\gamma)\ \&\ r_s$ in V_{11} are therefore not related to unmasked data (namely x or y). None of these values can act as a mask, and XORing such values would leak through DPA. However, if these are XORed with a random k-bit mask γ, the resulting value retains the uniform distribution of the mask. Consequently, the value of Ω in steps V_2, V_3, V_4 and V_5 and $\mathsf{B}^{(\mathsf{i})} = \gamma \oplus c^{(i-1)}\ \&\ (x \oplus y) \oplus x\ \&\ y$ in V_{13} are all uniformly distributed over $\mathbb{Z}_{2^k}$. The same holds true for Ω in steps V_8 and V_9 because, as already stated in Sect. 3.2, Ω is uniform on $\mathbb{Z}_{2^k}$ when γ is. As Ω is uniform, $\mathsf{B}^{(\mathsf{i})} = (c^{(i-1)} \oplus 2\gamma)\ \&\ r_s \oplus \Omega$ in V_{12} is also uniform on $\mathbb{Z}_{2^k}$.

Calculating an XOR between independent masked values or between random masks has the same distribution as the XOR of two random numbers and so is not related to the unmasked data. The variables $\mathsf{A_0} = r_s \oplus (x \oplus y)$ and $\mathsf{A_1} = r_s$ in steps V_7 are therefore not leaking. Likewise, performing an AND between masked values and an independent mask does not reveal anything about unmasked data. Hence, the values of T in steps V_8 and V_9 are not related to unmasked data.

Also, since $2a \bmod 2^k = 2(a \bmod 2^{k-1})$ for any $a \in \mathbb{Z}_{2^k}$, the values of B in V_6 and V_{14} and value of C in V_6 are uniformly distributed over $2\mathbb{Z}_{2^{k-1}}$. Finally, it clearly appears that $\mathsf{A_0} = 2\gamma \oplus r_s \oplus (x + y)$ and $\mathsf{A_0} = r_s \oplus (x + y)$, in steps V_{16} and V_{17} respectively do not leak information about unmasked data.

5 Application to XTEA

5.1 XTEA overview

XTEA is a 64-bit block cipher that has 32 rounds and operates with a key size of 128 bits. It uses similar routines for encryption and decryption module.

Round function Let $\mathsf{rk}[2i]$ and $\mathsf{rk}[2i+1]$ be the round keys (for $0 \leqslant i \leqslant 31$). Let also v_0 and v_1 denote the two 32-bit inputs. The outputs are then updated as

$$\begin{cases} v_0 \leftarrow v_0 + (F(v_1) + v_1) \oplus \mathsf{rk}[2i] \\ v_1 \leftarrow v_1 + (F(v_0) + v_0) \oplus \mathsf{rk}[2i+1] \end{cases} \quad \text{where } F(v) = [(v \ll 4) \oplus (v \gg 5)] \ .$$

The pair (v_0, v_1) is initialized with 64-bit plaintext m, $m = v_0 \| v_1$. The above procedure is run for $i = 0, \ldots, 31$, the ciphertext is the output of round 31.

Key schedule The master secret key is a 128-bit value $K = (K[0], K[1], K[2], K[3])$ where $K[j]$'s are 32-bit values $(0 \leqslant j \leqslant 3)$. The round keys are defined as

$$\begin{cases} \mathsf{rk}[2i] = K[a_i] + \delta_i \ , \ a_i = \delta_i \ \& \ 3 \\ \mathsf{rk}[2i+1] = K[b_i] + \delta_{i+1} \ , \ b_i = (\delta_{i+1} \gg 11) \ \& \ 3 \end{cases} \quad \text{where } \delta_i = i \cdot \delta \pmod{2^{32}}$$

with $\delta =$ `0x9E3779B9`.

5.2 Preventing first-order DPA

A round of XTEA involves Boolean operations (XORs and Shifts) and six additions. Boolean operations are easily masked through an XOR whereas the addition operations are securely evaluated using Alg. 2. We assume that fresh 32-bit masks w_0, w_1 and Γ are uniformly picked at random for each encryption process. The pair (w_0, w_1) is applied to the input plaintext. The mask Γ is used with the secure addition algorithm. The same masks are then maintained across all rounds, as in the transformed masking method ([1]). At the end of the algorithm

the masks (w_0, w_1) are applied to the output data to recover the matching, unmasked ciphertext. For an implementation that is secure against first-order DPA, the round keys do not need to be masked. In such a case, the key schedule is normally implemented. Only 4 additions per round will be then evaluated using Alg. 2:

- two additions with the evaluation of F,
- two additions when updating the pair (v_0, v_1).

A round of XTEA can be written as

$$v_b \leftarrow v_b + (F(v_{\bar{b}}) + v_{\bar{b}}) \oplus \mathsf{rk}[2i+b] \quad \text{where } F(v_{\bar{b}}) = [(v_{\bar{b}} \lll 4) \oplus (v_{\bar{b}} \ggg 5)]$$

for $b \in \{0, 1\}$. Its masked version is implemented as

$$\begin{aligned}
\Gamma &\leftarrow \texttt{random}(2^k) \\
(\mathsf{A}_0, \mathsf{A}_1) &\leftarrow \left(F(\boldsymbol{v_{\bar{b}}}), F(w_{\bar{b}})\right) \\
(\mathsf{A}_0, \mathsf{A}_1) &\leftarrow \texttt{SecADD}(\mathsf{A}_0, v_{\bar{b}}, \mathsf{A}_1, w_{\bar{b}}, \Gamma) \\
(\mathsf{A}_0, \mathsf{A}_1) &\leftarrow (\mathsf{A}_0 \oplus \mathsf{rk}[2i+b], \mathsf{A}_1 \oplus \mathsf{rk}[2i+b]) \\
(\mathsf{A}_0, \mathsf{A}_1) &\leftarrow \texttt{SecADD}(\mathsf{A}_0, v_b, \mathsf{A}_1, w_b, \Gamma)
\end{aligned}$$

where the operations on the masked variables and the masks are processed separately without leaking information about the original variables.

As the two input variables are split into two shares, the number of Boolean operations related to those variables is doubled. This gives 22 basic operations per round plus 4 secure additions. Since the XTEA block-cipher operates on words of size 32 bits, we used the addition algorithm with $k = 32$. Therefore each round requires $4 \times 168 + 22 = 684$ word operations.

5.3 Performance analysis

We implemented an unmasked version, and a DPA resistant version of XTEA using different addition algorithms. With previous mask-switching methods we implemented the three conversions steps in a row. The addition algorithm includes the two secure B→A conversions (both use the same pre-computed 32-bit random), followed by the secure A→B conversion. The code was written in C and a 32-bit Intel based processor was used for evaluating the implementation. The compilation options were chosen to favor small code size. Optimal code would be possible if written in assembly, but the goal was to determine the relative costs between different secure implementation of XTEA. Details such as the code size, RAM overhead and the cycle count are given in Table 2.

The ROM size represents the complete XTEA code size including the addition algorithm. We also give the size of the addition algorithm inside parenthesis. As the goal is to compare algorithms with smallest memory requirement, for the table-based algorithms we tested the 4-bit nibble size version. The RAM size represents the 16-byte key, the size of the look-up table if any, as well as the size

Table 2: Details of various XTEA implementations

Algorithms	ROM [bytes]	RAM [bytes]	Cycles/byte
XTEA	114	16	60
masked XTEA (Alg. 2)	**379**(**80**)	**28**	**2410**
masked XTEA ([8] + Alg. 3)	395(96)	**28**	**2515**
masked XTEA ([18])	620(262)	45	3180
masked XTEA ([5])	664(304)	51	3403

of the random numbers used for masking the inputs, generating the LUTs and computing the secure additions. We assume that a random generator is available to provide the random numbers needed by all algorithms. Although some algorithms need more random numbers, those numbers are only computed once per execution. This is therefore not a determining factor in the comparison. For this reason, the time for their generation was not taken into account.

Table 2 shows that the protected version of XTEA, using the table-based algorithms, requires a ROM space that is almost six times larger than the unprotected version. These versions also have a RAM overhead of at least 45 bytes (*i.e.*, three times bigger than the unprotected version). Our implementation of XTEA requires a ROM space that is only 3.3 times larger than the unprotected version. Remarkably, our method is faster than the methods of [5] and [18] which require a memory space (ROM + RAM) that is at least 1.6 times larger. Goubin's version has a comparable performance especially when considering the improved A→B conversion given in Appendix A, but our algorithm remains better. Indeed, the overall computation cost for one secure addition using Goubin's optimized variant is $5 \cdot k + 1 + 2 \cdot 7 + 2 = \underline{5 \cdot k + 17}$. Our algorithm enables us to save 9 operations per addition, which yields to 1152 operations saved per XTEA execution. Algorithm 2 provides then the best choice regarding the memory versus speed complexity and makes it suitable for resource-constrained devices. This advantage obviously depends on the internal structure of the cryptographic algorithm. If the structure combines additions and Boolean operations such that several additions are performed in a row, the mask-switching methods are then faster. Section 6.1 presents other examples where our addition algorithm presents an advantage over the methods of [8], [18] and [5].

6 Further Results

6.1 Other applications

For larger operands, our algorithm is also applicable to the SKEIN hash function or Threefish block-cipher [6]. These algorithms work with 64-bit variables and make extensive use of the MIX function that combines an XOR, an addition modulo 2^{64} and a rotation by a constant. For smaller k, the gain with our algorithm is even more significant. For instance, one could use our algorithm for protecting SAFER [15]. SAFER encryption alternates between use of a byte-word XORs and additions modulo 256 (*i.e.*, $k = 8$).

6.2 Addition over the integers

Although described for adding over $\mathbb{Z}_{2^k}$, our secure addition algorithm (Alg. 2) readily extends to output the result $\boldsymbol{s} = s \oplus r_s$ over the integers. For this purpose, it suffices to run the algorithm by seeing the involved operands as elements in $\mathbb{Z}_{2^{k+1}}$. Indeed, as the input values x and y are smaller than 2^k their sum, $z = x + y$, over the integers is smaller than 2^{k+1} and so is an element of $\mathbb{Z}_{2^{k+1}}$. More generally, our algorithm can accommodate integers x, y of arbitrary length and compute their blinded sum $\boldsymbol{s}$ over $\mathbb{Z}$ by running it over $\mathbb{Z}_{2^{k+1}}$ for any $k \geqslant \max(|x|_2, |y|_2)$ —where $|\cdot|_2$ denotes the binary length.

6.3 Subtraction

Algorithm 2 can also be used for subtraction (which is useful for the XTEA decryption process for example). We use the notation $\overline{x}$ to denote the bitwise complementation of x, namely $\overline{x} = x \oplus (-1)$. From the definition, we immediately get the following identity $\overline{x \oplus y} = x \oplus y \oplus (-1) = x \oplus \overline{y} = \overline{x} \oplus y$.

Our secure subtraction algorithm builds on Alg. 2 and runs in three steps. The input is $(\boldsymbol{x}, \boldsymbol{y}, r_x, r_y) \in \left(\mathbb{Z}_{2^k}\right)^4$ such that $\boldsymbol{x} = x \oplus r_x$ and $\boldsymbol{y} = y \oplus r_y$ and the output is $(\boldsymbol{w}, r_w)$ where $\boldsymbol{w} = (x - y) \oplus r_w \pmod{2^k}$ and $r_w = r_x \oplus r_y$.

1. Compute $\overline{\boldsymbol{x}}$;
2. Call Algorithm 2 on input $(\overline{\boldsymbol{x}}, \boldsymbol{y}, r_x, r_y)$ and obtain $(\boldsymbol{s}, r_s)$ where $\boldsymbol{s} = (\overline{x} + y) \oplus r_s$ and $r_s = r_x \oplus r_y$;
3. Set $\boldsymbol{w} = \overline{\boldsymbol{s}}$ and $r_w = r_s$, and return $(\boldsymbol{w}, r_w)$.

The correctness follows by observing that $\overline{\boldsymbol{x}} = \overline{x \oplus r_x} = \overline{x} \oplus r_x$. Hence, since $\boldsymbol{y} = y \oplus r_y$, at Step 2, we indeed have $\boldsymbol{s} = (\overline{x} + y) \oplus r_s$ with $r_s = r_x \oplus r_y$. The final step exploits the identity $-x = \overline{x} + 1$; the complementation of $\boldsymbol{s}$ yielding

$$\overline{\boldsymbol{s}} = \overline{(\overline{x} + y)} \oplus r_s = \overline{(-x - 1 + y)} \oplus r_s = \big(-(-x - 1 + y) - 1\big) \oplus r_s = (x - y) \oplus r_s \ .$$

7 Conclusion

This paper presented a secure addition algorithm for preventing DPA-like attacks in symmetric-key cryptosystems. Remarkably, the developed method involves only Boolean operations (converting to arithmetic masking is not needed) and does not need pre-computed tables. As an illustration, we provided a countermeasure to protect XTEA, which proved well adapted to 32-bit microprocessors.

References

1. Akkar, M.L., Giraud, C.: An implementation of DES and AES, secure against some attacks. In: Cryptographic Hardware and Embedded Systems − CHES 2001. LNCS, vol. 2162, pp. 309–318. Springer (2001)

2. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Advances in Cryptology − CRYPTO '99. LNCS, vol. 1666, pp. 398–412. Springer (1999)
3. Coron, J.S., Goubin, L.: On Boolean and arithmetic masking against differential power analysis. In: Cryptographic Hardware and Embedded Systems − CHES 2000. LNCS, vol. 1965, pp. 231–237. Springer (2000)
4. Coron, J.S., Tchulkine, A.: A new algorithm for switching from arithmetic to Boolean masking. In: Cryptographic Hardware and Embedded Systems − CHES 2003. LNCS, vol. 2779, pp. 89–97. Springer (2003)
5. Debraize, B.: Efficient and provably secure methods for switching from arithmetic to Boolean masking. In: Cryptographic Hardware and Embedded Systems − CHES 2012. LNCS, vol. 7428, pp. 107–121. Springer (2012)
6. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family. Submission to NIST (Round 3) (Oct 2010), http://www.skein-hash.info/sites/default/files/skein1.3.pdf
7. Golić, J.D.: Techniques for random masking in hardware. IEEE Transactions on Circuits and Systems 54(2), 291–300 (2007)
8. Goubin, L.: A sound method for switching between Boolean and arithmetic masking. In: Cryptographic Hardware and Embedded Systems − CHES 2001. LNCS, vol. 2162, pp. 3–15. Springer (2001)
9. Goubin, L., Patarin, J.: DES and differential power analysis (The "duplication" method). In: Cryptographic Hardware and Embedded Systems (CHES '99). LNCS, vol. 1717, pp. 158–172. Springer (1999)
10. Kelsey, J., Schneier, B., Wagner, D.: Related-key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. In: Information and Communication Security (ICICS '97). LNCS, vol. 1334, pp. 233–246. Springer (1997)
11. Knuth, D.E.: The Art of Computer Programming, vol. 2. Addison-Wesley, 2nd edn. (1981)
12. Knuth, D.E.: The Art of Computer Programming, vol. 4A. Addison-Wesley (2011)
13. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Advances in Cryptology − CRYPTO '99. LNCS, vol. 1666, pp. 388–397. Springer (1999)
14. Lipmaa, H., Moriai, S.: Efficient algorithms for computing differential properties of addition. In: Fast Software Encryption (FSE 2001). LNCS, vol. 2355, pp. 336–350. Springer (2002)
15. Massey, J.L.: SAFER K-64: A byte-oriented block-ciphering algorithm. In: Fast Software Encryption (FSE '93). LNCS, vol. 809, pp. 1–17. Springer (1994)
16. Messerges, T.S.: Securing the AES finalists against power analysis attacks. In: Fast Software Encryption (FSE 2000). LNCS, vol. 1978, pp. 150–164. Springer (2000)
17. Needham, R.M., Wheeler, D.J.: TEA extensions. Tech. rep., Computer Laboratory, University of Cambridge (Oct 1997), available at URL http://www.cl.cam.ac.uk/ftp/users/djw3/xtea.ps
18. Neiße, O., Pulkus, J.: Switching blindings with a view towards IDEA. In: Cryptographic Hardware and Embedded Systems − CHES 2004. LNCS, vol. 3156, pp. 230–239. Springer (2004)
19. Örs, S.B., Gürkaynak, F.K., Oswald, E., Preneel, B.: Power-analysis attack on an ASIC AES implementation. In: International Conference on Information Technology: Coding and Computing (ITCC '04), Volume 2. pp. 546–552. IEEE Computer Society (2004)
20. Trichina, E.: Combinational logic design for AES SubByte transformation on masked data. Cryptology ePrint Archive, Report 2003/236 (2003), http://eprint.iacr.org/2003/236

21. Wheeler, D.J., Needham, R.M.: TEA, a tiny encryption algorithm. In: Fast Software Encryption (FSE '94). LNCS, vol. 1008, pp. 363–366. Springer (1995)
22. Wheeler, D.J., Needham, R.M.: Corrections to XTEA. Tech. rep., Computer Laboratory, University of Cambridge (Oct 1998), available at URL `http://www.movable-type.co.uk/scripts/xxtea.pdf`

A Optimized Variant of Goubin's Method

We show in this appendix how to rearrange the operations in the secure A→B algorithm used for converting $A = x - r$ to $x' = x \oplus r$. As a result, the algorithm cost is slightly reduced.

The carry expansion formula expressed using t_i, $0 \leq i \leq k-1$ (see [8, Corollary 2.1]) can be simplified. The idea is to start the recursion with $t_0 = 0$ instead of $t_0 = 2\gamma$. The value of t_1 then simplifies to $t_1 = 2\big[t_0 \& (A \oplus r) \oplus \omega\big] = 2\omega$. The recursion formula can so be re-written as

$$t_i = \begin{cases} 2\omega & \text{if } i = 1\,, \\ 2\left[t_i \,\&\, (A \oplus r) \oplus \omega\right] & \text{for } 2 \leqslant i \leqslant k-1\,. \end{cases}$$

The main loop within the secure A→B conversion algorithm becomes then:

$\mathsf{T} \leftarrow 2\Omega$
for $i = 2$ to $k-1$ **do**
  $\Gamma \leftarrow \mathsf{T} \,\&\, r$; $\Gamma \leftarrow \Gamma \oplus \Omega$; $\mathsf{T} \leftarrow \mathsf{T} \,\&\, A$
  $\Gamma \leftarrow \Gamma \oplus \mathsf{T}$; $\mathsf{T} \leftarrow 2\Gamma$
end for

We extract the first loop iteration and trade five operations against one logical shift operation. This reduces the algorithm cost to $\underline{5k+1}$ operations. This small change has no impact on the security of the algorithm.

Algorithm 3 Improved Goubin's A→B conversion

Input: (A, r), such that $A = x - r \bmod 2^k$
Output: (x', r), such that $x' = x \oplus r$

1: $\Gamma \leftarrow \texttt{random}(2^k)$
2: $\mathsf{T} \leftarrow 2\Gamma$; $x' \leftarrow \Gamma \oplus r$; $\Omega \leftarrow \Gamma \,\&\, x'$; $x' \leftarrow \mathsf{T} \oplus A$; $\Gamma \leftarrow \Gamma \oplus x'$; $\Gamma \leftarrow \Gamma \,\&\, r$; $\Omega \leftarrow \Omega \oplus \Gamma$
3: $\Gamma \leftarrow \mathsf{T} \,\&\, A$; $\Omega \leftarrow \Omega \oplus \Gamma$; $\mathsf{T} \leftarrow 2\Omega$
4: **for** $i = 2$ to $k-1$ **do**
5: $\Gamma \leftarrow \mathsf{T} \,\&\, r$; $\Gamma \leftarrow \Gamma \oplus \Omega$; $\mathsf{T} \leftarrow \mathsf{T} \,\&\, A$
6: $\Gamma \leftarrow \Gamma \oplus \mathsf{T}$; $\mathsf{T} \leftarrow 2\Gamma$
7: **end for**
8: $x' \leftarrow x' \oplus \mathsf{T}$
9: **return** (x', r)