Switching Blindings
with a View Towards IDEA

Olaf Neiße and Jürgen Pulkus

Giesecke & Devrient, Department of Cryptology, Prinzregentenstr. 159,
81677 Munich, Germany

Abstract. Cryptographic algorithms implemented on smart-cards must be protected against side-channel attacks. Some encryption schemes and hash functions like IDEA, RC6, MD5, SHA-1 alternate various arithmetic and boolean operations, each of them requiring a different kind of blinding. Hence the maskings have to be changed frequently. How to switch reasonably between standard arithmetic masking and boolean masking was shown in [2], [3], [5] and [9].

In this paper we propose more space-efficient table-based conversion methods. Furthermore, we deal with some non-standard arithmetic operations, namely arithmetic modulo $2^k + 1$ for some $k \in \mathbb{N}$ and a special multiplication used by IDEA.

Keywords: DPA, IDEA, MD5, Masking Techniques, RC6, SHA-1.

1 Introduction

Running cryptographic algorithms on systems vulnerable to side-channel attacks (e.g., on smart-cards), implementation issues become crucial and non-trivial. By side-channel information we mean all information a physical system leaks into the environment, like computation time, power consumption, electromagnetic emissions. In the literature a variety of attacks is known that exploit side-channels to gain sensitive information like secret keys. Attacks based on statistical examinations, such as Differential Power Analysis (DPA) [6], Differential Electromagnetic Analysis (DEMA) [1],[11] and higher-order DPA, force special protection and therefore special and careful implementations. One way to counteract this problem is to randomly split sensitive data into at least two shares such that computations can be carried out by manipulating only the shares, never reconstructing the original data. In case of two shares we call the two parts *masked data* and *mask*.

Some encryption schemes like IDEA [7], TWOFISH [4] or RC6 [12] and some hash functions like MD5, SHA-1 (see [8]) or SHA-2 [10] use the concept of "incompatible mixed group operations" to achieve the desired confusion and diffusion. The most frequent group structures employed by the designers are the boolean operation $\mathbf{Xor}_L$ (addition on $(\mathbb{Z}/2\mathbb{Z})^L$) and the standard arithmetic operation $\mathbf{Add}_{2^L}$ (addition on $\mathbb{Z}/2^L\mathbb{Z}$) for some integer L, mainly $L = 8, 16, 32$ or 64. But non-standard arithmetic operations like multiplication $\mathbf{Mult}_{2^L+1}$ on

$\mathbb{Z}/(2^L+1)\mathbb{Z}$ are used as well. The masking has to be adapted to these group structures. If such group operations are mixed in a scheme, one has to convert the masking. Methods for switching masks between boolean and standard arithmetic are presented in [2], [3], [5] and [9]. Unlike the conversion from $\mathbf{Xor}_L$ to $\mathbf{Add}_{2^L}$ (using the very efficient method proposed by Goubin [5]) the contrary direction still waits for a time- and space-efficient method.

In Section 3 we present Algorithm 3.2 which transforms $\mathbf{Add}_{2^L}$-maskings into $\mathbf{Xor}_L$-maskings using a compressed table. In comparison to the method proposed by Coron and Tchulkine in [3] our algorithm requires only half the memory for tables. In Section 4 we explain how to use our table to enable switching from $\mathbf{Add}_{2^L}$-maskings to $\mathbf{Add}_{2^L+1}$-maskings and conversely. As an application we show in Sect. 5 how to implement the block cipher IDEA [7] efficiently and securely.

2 Notation

Since several kinds of operations with varying length or modulus have to be mixed in our investigation, we use the following notation: For $l \in \mathbb{N}$ let

$V(l)$	set of bit sequences of length l also viewed as set of integral numbers $\{0,\dots,2^l-1\}$.	
$x \oplus_l y$	infix-notation for bitwise exclusive or	$(x, y \in V(l))$
$x +_l y$	infix-notation for addition `modulo` l	$(x, y \in \{0,\dots,l-1\})$
$\mathbf{Sub}_l(x,y)$	prefix-notation for subtraction `modulo` l	$(x, y \in \{0,\dots,l-1\})$
$x -_l y$	infix-notation for subtraction `modulo` l	$(x, y \in \{0,\dots,l-1\})$
$x \cdot_l y$	infix-notation for multiplication `modulo` l	$(x, y \in \{0,\dots,l-1\})$
-0	bit sequence $(0,0,\dots,0)$	
-1	bit sequence $(1,1,\dots,1)$	
$\overline{x}$	1-complement of x	$(x \in V(l))$
$(x \mid y)$	concatenation of the bit sequences	
$x \gg 1$	logical shift right by 1: $(x_{l-1}, x_{l-2}, \dots, x_0) \mapsto (0, x_{l-1}, \dots, x_1)$	
$\mathbb{1}_<(x,y)$	comparison function $\mathbb{1}_<(x,y) = \begin{cases} 1 & \text{if } x<y \\ 0 & \text{else} \end{cases}$	$(x, y \in \mathbb{Z})$

3 From $\mathbf{Add}_{2^L}$ to $\mathbf{Xor}_L$

Throughout the paper we start with some sensitive data $d \in V(L)$ (with $L \in \mathbb{N}$, mainly $L = 8, 16, 32$) given by a mask x and the masked data y using one type of masking. Our goal is always to switch blinding so that d is then represented by a new mask u and masked data v with respect to another masking. For this we use randomized tables of size $m = 2^l$, where l is a small divisor of L, e.g. $(L,l) = (32,8)$, $(16,8)$ or $(16,4)$. An additional random bit z will be needed to make the calculation DPA-resistant. For variables we signal a certain dependency on z by a tilde. Setting $M = 2^L$ and $k = L/l$, the elements of $V(L)$ are viewed as the k-digit m-ary numbers from 0 to $M - 1$.

Now suppose that d is represented using a standard arithmetic masking. We describe algorithms that perform the conversion into a boolean masking of d:

`Input:`	$(x, y) \in V(L)^2$ such that $y -_{2^L} x = d$
	$u \in V(L)$
`Output:`	$v \in V(L)$ such that $u \oplus_L v = d$

The easiest DPA-resistant algorithm precomputes, depending on x and u, one big table $S : V(L) \to V(L)$ which contains for every value $y \in V(L)$ the corresponding value $v \in V(L)$. The conversion is done by a simple table-lookup. This is fast during the computation; however, for L large this algorithm is not practical because generating the table S needs too much time and space. We overcome this problem by working with such a table S only virtually – i.e. the entries of S are calculated using smaller tables T and C.

The basic idea is similar to the approach presented by Coron and Tchulkine in [3, 3.2]: one precomputes a small randomized table $T : V(l) \to V(l)$ which turns an $+_m$-masked digit for some input-falsifier r into an $\oplus_l$-masked digit for an output-falsifier s. With the help of this table, the result is calculated iteratively digit by digit from the lowest to the highest. More precisely, the digits d_j of $d = (d_k|\dots|d_1)$ are calculated in the additively masked form $d_j +_m r$, and then plugged into the table T to switch securely into the $\oplus_l$-blinding $d_j \oplus_l s$ with falsifier s. So one needs to find the masked digits $d_j +_m r$ using x, y and r. Let $a := (r|\dots|r)$ and $g := x -_M a$. Then $y -_M g = (d_k|\dots|d_1) +_M (r|\dots|r) = (d_k + r + c_{k-1} - c_k m|\dots|d_2 + r + c_1 - c_2 m|d_1 + r - c_1 m)$ for some carry bits $c_j \in \{0, 1\}$. In order to correct the j-th digit of $d + a$ to $d_j +_m r$, one needs to determine the carry bit c_{j-1}, which equals 1 in d_{j-1} out of m cases and therefore is susceptible to DPA. Hence a carry bit table providing the information, if the addition $d_j + r$ yields a carry $\bmod m$ or not, has to be randomized.

Coron and Tchulkine propose in [3, Algorithm 4] to work with a random mask $\gamma \in V(l)$ and the table $\mathbf{C} : V(l) \to V(l)$, $\mathbf{C}[i] = \mathbb{1}_{<}(i, r) +_m \gamma$.

Our solution works with the much smaller bit-table $C : V(l) \to \{0, 1\}$, $C[i] = \mathbb{1}_{<}(i, r)$. We do not randomize the carry bit table directly, instead we blind the carry bit information during the conversion itself by a single random bit z (described below).

Furthermore, the operations $\mathbf{Add}_{2^l}$ and $\mathbf{Xor}_l$ coincide on the least significant bit (LSB). Hence we can spare the LSB of each entry $T[i]$ and replace it by the carry bit $C[i]$. This enables us to store $T[i]$ and $C[i]$ in one table, whereas Coron and Tchulkine needed twice the space.

This explains our precomputation:

ALGORITHM 3.1: PRECOMPUTATION FOR $\mathbf{Add}_{2^L} \rightarrow \mathbf{Xor}_L$

```
(a) Generate uniformly distributed random values $r \in V(l)$, $s \in V(l-1)$
(b) For $i = 0$ to $m-1$ do
(c)     $T[i] = \mathbf{Xor}_{l-1}(\mathbf{Sub}_m(i, r) \gg 1,\ s)$
(d)     $C[i] = \mathbb{1}_<(i, r)$
```

Our switching algorithm 3.2 uses the following fact: the addition $\overline{d}_j + r$ contributes in $m - d_j - 1$ out of m cases a carry bit $= 1$, balancing out the d_j cases when the carry bit is 1 for $d_j + r$. Therefore we work with d_j or its one-complements, depending on a randomly chosen bit z.

For a variable w, let $\tilde{w}$ denote w if $z = 0$, and $\overline{w}$ if $z = 1$, so $\tilde{w} := \mathbf{Xor}(w, -z)$. The computations are based on the following observation: For $p, q \in V(L)$ let $w = p +_{M} q$. Then

$$\tilde{w} = \tilde{p} +_{M} \tilde{q} +_{M} z. \qquad (*)$$

Especially, $\tilde{y} = \tilde{d} +_{M} \tilde{x} +_{M} z$. Solving for $\tilde{d}$ and adding a on both sides gives $\tilde{d} +_{M} a = \tilde{y} -_{M} (\tilde{x} -_{M} a) -_{M} z$, which is in case $z = 0$ just what we wanted to calculate above. In case $z = 1$ we do the same calculation, but the role of d is taken by its complement $\overline{d}$ with two consequences: each additively masked digit plugged into the table yields the complement of the result of the case $z = 0$. This can be corrected easily. And, instead of reading the carry c_j of the addition $d_j + r$ from the carry table, we read the carry of the addition $\overline{d}_j + r$.

ALGORITHM 3.2: $\mathbf{Add}_{2^L} \rightarrow \mathbf{Xor}_L$

```
 (1) Generate a random bit $z \in \{0, 1\}$
 (2) Create $\widetilde{s} = (s|r_0) \oplus_l -z$ with $r_0 =$ the LSB of $r$
 (3) Calculate $g = \mathbf{Sub}_M(\mathbf{Xor}_L(x, -z), (r|\dots|r))$; let $g = (g_k|\dots|g_1)$ with $g_j \in V(l)$
 (4) Calculate $h = \mathbf{Xor}_L(\ (\widetilde{s}|\dots|\widetilde{s})\ ,\ u)$; let $h = (h_k|\dots|h_1)$ with $h_j \in V(l)$
 (5) Set $f = \widetilde{y} = y \oplus_L -z$
 (6) Initialize $c = z$
 (7) For $j = 1$ to $k$ do
 (8)     Subtract $f = \mathbf{Sub}_{2^{(k-j)l}}(f, c)$
 (9)     Subtract $f = \mathbf{Sub}_{2^{(k-j)l}}(f, g_j)$; let $f_j = f \bmod m$
(10)     Look up $o = T[f_j]$, $c = C[f_j]$
(11)     Calculate $v_j = h_j \oplus_l (o|c')$ with $c'$ the LSB of $f$
(12)     Redefine $f = f \gg l$
(13) Let $v = (v_k|\dots|v_1)$
```

Although the steps look rather complicated, they mostly describe elementary operations on a processor.[1] Steps (3) and (4) are precalculations supporting the mask changes in Steps (8), (9) and (11) from x and $\widetilde{s}$ to $\widetilde{r}$ and u, respectively. They could also be part of the loop. All intermediate variables and all carry bits

[1] Steps (8) and (9) could even be done at the same time by a subtraction with carry, but then the change of the carry bit would not be independent of the data d, which might cause troubles on certain platforms.

in (8)-(11), as well as all changes of these values are independent of d, whence this algorithm turns out to be DPA-resistant. Indeed, all values depending on d for $z = 0$ are calculated in the case $z = 1$ using the complement of d.

The algorithm can be varied in many ways: instead of starting with a mask x and a masked value $y = d +_M x$ one can also work with two shares x and $y = d -_M x$. For this, one just has to replace some subtractions by additions. Or, one can calculate in the loop instead of $d + a$ the value $d - a$. For this, one has to modify the tables by using additions instead of subtractions.

For practical implementations on a chip card one might use either one table of size 256 bytes or sixteen tables of size 16 nibbles each. In the latter case one can generate all tables with pairwise different input and output randomizations (in random order) and then use different tables in each round of the loop. As for all possible input randomizations there is a table available in RAM, one can choose in Step (3) instead of $(r|\dots|r)$ an arbitrary value. For example one can make g to zero, such that the subtraction in Step (9) in the loop can be skipped.

4 From $\mathbf{Add}_{\mathbf{2}^L}$ to $\mathbf{Add}_{\mathbf{2}^L+\mathbf{1}}$ and back

In this section we deal with the question how to switch between standard arithmetic masking modulo 2^L and additive maskings modulo $2^L + 1$ without leaking any information about the data. Our solution is based on the following fact:

Proposition 4.1. *Let $M \in \mathbb{N}$.*
i) For $a, d \in \{0, 1, \dots, M - 1\}$ let $b = \mathbf{Add}_M(d, a)$ and $c = \mathbb{1}_<(b, a)$.
Then $\mathbf{Add}_{M+1}(d, a) = \mathbf{Sub}_{M+1}(b, c)$.
ii) For $a, d \in \{0, 1, \dots, M\}$ let $b = \mathbf{Add}_{M+1}(d, a)$ and $c = \mathbb{1}_<(b, a)$.
Then $\mathbf{Add}_M(d, a) = \mathbf{Add}_M(b, c)$.

Proof. We first examine the case $c = 0$, that is $b \geq a$ and therefore $b = d + a$. Both conclusions are obvious except if $a + d = b = M$, where in fact $b +_M c = 0 +_M 0 = 0 = d +_M a$. The case $c = 1$ follows immediately from $d + a = M + b = (M + 1) + (b - 1)$ for i) and $d + a = (M + 1) + b = M + (b + 1)$ for ii). □

So we just have to add respectively subtract the carry bit c. As before, this bit is correlated to the sensitive data d and therefore has to be randomized.

Starting from an additive mask modulo 2^L, the carry bit can be determined employing the same ideas we have seen in Algorithm 3.2: for masks a of the special form $(r|\dots|r)$, the carry bit $c = \mathbb{1}_<(d, a)$ equals the bit c_k in $d +_M a = (d_k + r + c_{k-1} - c_k m|\dots|d_2 + r + c_1 - c_2 m|d_1 + r - c_1 m)$. In Algorithm 3.2, Step (10) for $j = k$ yields c, masked with the chosen bit z. Hence we adapt Algorithm 3.2 to our new problem such that $\mathbf{Sub}_{M+1}(b, c)$ is computed without leaking information.

Starting from an additive mask modulo $2^L + 1$, additional to the ideas of Sect. 3 more ideas are needed, since the special value 2^L has to be treated differently. However, since 2^m is not a divisor of $2^L + 1$, not all intermediate values are totally independent of d. Few values show weak dependency which should not leak side channel information in practice.

Both conversion algorithms employ the unmasked carry bit table C:

```
ALGORITHM 4.2: PRECOMPUTATION $\mathbf{Add}_M \leftrightarrow \mathbf{Add}_{M+1}$
(a) Generate a random value $r \in V(l)$
(b) For $i = 1$ to $m$ do
(c)     $C[i] = \mathbb{1}_<(i, r)$
```

Before going into details of the algorithms, we give an analogue of the formula $(*)$ for calculations modulo $2^L + 1$. We use the notation

$$\widetilde{x} = \mathbf{Sub}_{M+1}(\mathbf{Xor}_L((x_k|\ldots|x_1), -z), x_{k+1} \oplus z) = \begin{cases} x & \text{if } z = 0 \\ -3 -_{M+1} x & \text{if } z = 1 \end{cases}$$

where, as usual, $x = (x_{k+1}|x_k|\ldots|x_1)$ with $x_{k+1} \in \{0,1\}$ and $x_j \in V(l)$ for $x \in \{0, \ldots, M\}$. For $x \in V(L)$ and $z = 1$ this is the usual one-complement.

Let $w = x +_{M+1} y$. Then an easy calculation shows:

$$\widetilde{w} = \widetilde{x} +_{M+1} \widetilde{y} +_{M+1} 3 \cdot z. \qquad (**)$$

This formula is helpful in verifying the correctness of our algorithms. Another fact one should keep in mind is: instead of adding two elements of $V(L)$ modulo $M + 1$ one can add them modulo M, if the occuring carry is later subtracted modulo $M + 1$. For subtraction the carry has accordingly to be added modulo $M + 1$.

Subsequently we propose our conversion Algorithm 4.3 for the transformation $\mathbf{Add}_M \to \mathbf{Add}_{M+1}$:

```
Input:   $(x, y) \in V(L) \times V(L)$ such that $y -_M x = d$
         $u \in \{0, 1, \ldots, M\} \subset V(L+1)$
Output:  $v \in V(L+1)$ such that $v -_{M+1} u = d$
```

As mentioned above, we modify Algorithm 3.2 for our purpose. Clearly Steps (2), (4), (11), (13) and the first half of (10) in 3.2 can be skipped. The new steps will be explained below.

```
ALGORITHM 4.3: $\mathbf{Add}_M \to \mathbf{Add}_{M+1}$
 (1) Generate a random bit $z \in \{0,1\}$
 (2) Calculate $g = \mathbf{Sub}_M(\mathbf{Xor}_L(x, -z),\ (r|\ldots|r))$; let $g = (g_k|\ldots|g_1)$ with $g_j \in V(l)$
 (3) Set $\widetilde{u} = \mathbf{Sub}_{M+1}(\mathbf{Xor}_L((u_k|\ldots|u_1),\ -z),\ u_{k+1} \oplus_1 z)$
 (4) Initialize $v = \mathbf{Sub}_{M+1}(\widetilde{u}, (r|\ldots|r))$
 (5) Set $f = \widetilde{y} = \mathbf{Xor}_L(y, -z)$
 (6) Initialize $c = z$
 (7) For $j = 1$ to $k$ do
 (8)     Subtract $f = \mathbf{Sub}_{2^{(k-j)l}}(f, c)$
 (9)     Subtract $f = \mathbf{Sub}_{2^{(k-j)l}}(f, g_j)$; let $f_j = f \bmod m$
(10)     $v = \mathbf{Add}_{M+1}(v, c << (l \cdot j))$
(11)     $v = \mathbf{Add}_{M+1}(v, f_j << (l \cdot j))$
```

```
(12)     Look up $c = C[f_j]$
(13)     Redefine $f = f >> l$
(14) $v = \mathbf{Sub}_{M+1}(v, c)$
(15) $v = \mathbf{Add}_{M+1}(v, z)$
(16) $v = \mathbf{Sub}_{M+1}(\mathbf{Xor}_L((v_k|\dots|v_1),\ -z),\ v_{k+1} \oplus_1 z)$ with $v = (v_{k+1}|v_k|\dots|v_1)$
```

During the loop of 4.3, we have to determine the digits $\widetilde{d_j} +_m r$ (namely f_j in Step (9)) to obtain the intermediate carry bits c_j and the required carry bit $c = c_k$ (Step (12)). Therefore we have to exchange $y = d +_M x$ for $b = d +_M a$ (Steps (8) and (9)), where $a = (r|\dots|r)$ denotes the concatenation of k copies of r. Moreover, the masked data $v = d +_{M+1} u = (d +_M a) -_{M+1} c -_{M+1} a$ is calculated in Steps (4), (10), (11), (14) and (15). These calculation have to be balanced via the two cases $z = 0$ and $z = 1$:

(z=0) $\quad v = d + u = d + a + u - a \overset{4.1}{=} b - c + u - a \bmod M + 1$

(z=1) $\quad v = \overline{d} + u = (M - 1) - d - a + u + a \overset{4.1}{=} c - b + u + a - 2 \bmod M + 1$

From these equations one sees that what has to be added to u to get v in the case $z = 0$ has to be subtracted in the case $z = 1$ and vice versa. The simplest solution is to invert u at the beginning and at the end of the calculation if $z = 1$. That is done in Steps (3) and (16) using $(**)$.

Clearly, the digits $\widetilde{d_j} + r$ give no information about d, and the carry bits are balanced for the same reason as they are in 3.2 . Additionally, all those values are combined independently with u. So all variables are independent of the sensitive data d. Hence our algorithm does not leak any side-channel information.

Although Algorithm 4.3 looks rather complicated, our method works fast, since most calculations decribe basic operations on a processor and the carry bits occur automatically within those calculations.

In the same manner we also deal with the converse $\mathbf{Add}_{M+1} \to \mathbf{Add}_M$:

`Input:` $x, y \in \{0, 1, \dots, M\}$ such that $y -_{M+1} x = d$, $d \in V(L)$
$u \in V(L)$
`Output:` $v \in V(L)$ such that $v -_M u = d$.

The Algorithm 4.4 looks very similar to Algorithm 4.3 since we use the same technique to obtain the carry bit c in a blinded way. A difficult part of the Algorithm 4.4 is how to calculate the digits $d_j +_m r$ using $b = d +_{M+1} a$ ($a = (r|\dots|r)$) from below in the case $b = M$ without having to correct the result at the end, because this correction term would give vulnerable information about d.

This difficulty we solve by comparing $(f_k|\dots|f_1)$ and $(g_k|\dots|g_1)$, where $f = (f_{k+1}|f_k|\dots|f_1) := y +_{M+1} 1$ resp. its complement, and $g = (g_{k+1}|g_k|\dots|g_1) := \widetilde{x} -_{M+1} a$. This information is not totally, but well enough balanced – at least in the interesting case $L = 16$ and $l = 4$ or $l = 8$ – such that it should not provide sufficient information for an effective DPA-attack.

Algorithm 4.4: $\mathbf{Add}_{M+1} \rightarrow \mathbf{Add}_{M}$

```
 (1) Generate a random bit $z \in \{0,1\}$
 (2) Calculate $\widetilde{x} = \mathbf{Sub}_{M+1}(\mathbf{Xor}_L((x_k|\ldots|x_1),-z),\ x_{k+1} \oplus_1 z)$
 (3) Calculate $g = \mathbf{Sub}_{M+1}(\widetilde{x},\ (r|\ldots|r))$
     let $g = (g_{k+1}|\ldots|g_1)$ with $g_{k+1} \in \{0,1\}$ and $g_j \in V(l)$
 (4) Adjust $y = \mathbf{Sub}_{M+1}(y,1)$
 (5) Calculate $f = \widetilde{y} = \mathbf{Sub}_{M+1}(\mathbf{Xor}_L((y_k|\ldots|y_1),\ -z), y_{k+1} \oplus_1 z)$
     let $f = (f_{k+1}|\ldots|f_1)$ with $f_{k+1} \in \{0,1\}$ and $f_j \in V(l)$
 (6) Determine $c' = \mathbb{1}_{<}((f_k|\ldots|f_1),(g_k|\ldots|g_1))$
 (7) Calculate $\widetilde{u} = \mathbf{Xor}_L(u,\ -z)$
 (8) Initialize $v = \widetilde{u} -_M (r|\ldots|r)$
 (9) $v = v -_M 1 +_M z +_M (f_k|\ldots|f_1) -_M f_{k+1} -_M (g_k|\ldots|g_1) +_M g_{k+1} +_M c'$
(10) Initialize $c = f_{k+1}$
(11) For $j = 1$ to $k$ do
(12)     Subtract $f = \mathbf{Sub}_{2(k-j)l}(f, g_j)$; let $f_j = f \bmod m$
(13)     Set $c_{\mathrm{Aux}} = \mathbb{1}_{<}(f_j, c)$, subtract $f_j = \mathbf{Sub}_m(f_j, c)$ and set $c = c_{\mathrm{Aux}}$
(14)     Look up and redefine $c = c \oplus C[f_j]$
(15)     Redefine $\widetilde{f} = \widetilde{f} >> l$
(16) Calculate $v = \mathbf{Add}_M(v, c)$
(17) $v = \mathbf{Xor}_L(v, -z)$
```

The basic equations read as follows:

$$\begin{array}{ll} \text{(z=0)} & v = u + d = u + d + a - a \overset{4.2}{=} u + b + c - a \mod M \\ \text{(z=1)} & v = u + \widetilde{d} = u + (-3 -_{M+1} d) - a + a \text{ and} \\ & u - d - a + a \overset{4.2}{=} u - b - c + a \mod M \end{array}$$

So, for $z = 1$, we invert u at the beginning and at the end of the calculations. Moreover, we have to transform $u + (-3 -_{M+1} d) - a + u + a$ to $u - d - a + a$, which explains those tricky calculations.

Instead of manipulating the masked data $v = d + u$ inside the loop (see Steps (10) and (11) of Algorithm 4.3), we suggest to exchange masks outside the loop in Steps (8) and (9) of Algorithm 4.3. The more complicate calculation is due to the arithmetic $\bmod M + 1$.

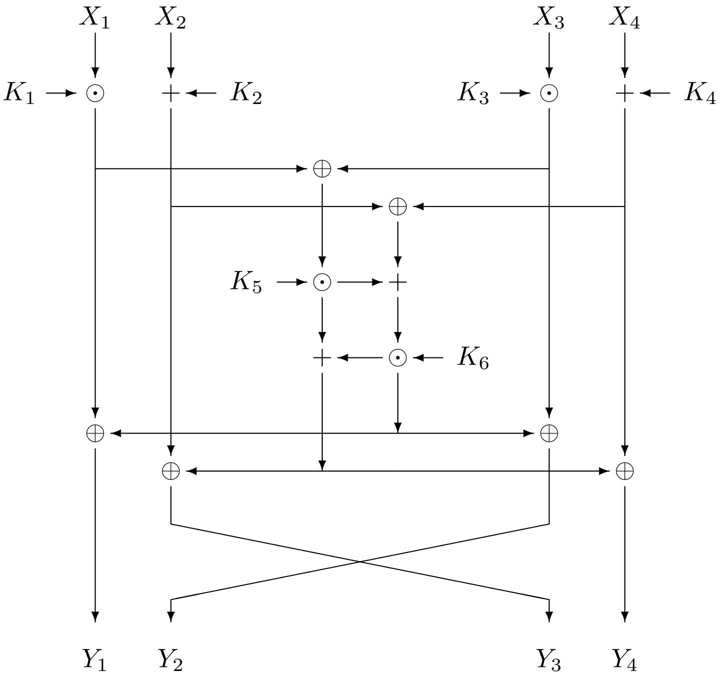
As for Algorithm 4.3, all steps of Algorithm 4.4 can easily be realised on a processor. Inside the loop, we need just two subtractions and one table lookup.

5 Application to IDEA

One of the algorithms using the principle of "mixed group operations" is the block cipher IDEA (International Data Encryption Algorithm, see [7]). Besides the boolean and standard arithmetic on $L = 16$ bits, the designers employ an operation $\odot$ based on the multiplication modulo $2^L + 1$. Since $2^{16} + 1$ is a prime number, multiplication on $\mathbb{Z}/(2^{16} + 1)\mathbb{Z}$ yields a cryptographically good operation on the set $\{1, 2, 3, \ldots, M = 2^{16}\}$. Identifying the value $0 \in V(L)$ with 2^{16} this defines the third operation $\odot$ on $V(16)$:

$$d \odot d' = \mathbf{I}^{-1}(\mathbf{I}(d) \cdot_{2^{16}+1} \mathbf{I}(d')) \text{ with } \mathbf{I}(0) = 2^{16} \text{ and } \mathbf{I}(d) = d \text{ else.}$$

IDEA has a block size of 64 bit. It iterates eight rounds followed by an output transformation. Each round takes as input 64 bits, given by four values $X_1, \ldots, X_4 \in V(16)$, and six key values (obtained by some key scheduling) $K_1, \ldots, K_6 \in V(16)$. Combining the three operations $\oplus = \oplus_{16}$, $+ = +_{2^{16}}$ and $\odot$ on $V(16)$ the output $Y_1, Y_2, Y_3, Y_4 \in V(16)$ of each round is calculated as follows:



The data of all three operation have to be blinded. For $\oplus$ and $+$ we use the boolean and standard arithmetic masking, respectively. As a blinding for the $\odot$-operation we recommend arithmetic blinding modulo $M+1 = 2^{16}+1$; a sensitive data $d \in V(L)$ is given by a mask $x \in \{0, 1, \ldots, 2^{16}\}$ and the masked data $y = \mathbf{I}(d) +_{M+1} x$. Then it is clear how to calculate $\mathbf{I}(d \cdot d') = \mathbf{I}(d) \cdot_{M+1} \mathbf{I}(d')$ using addition and multiplication modulo $M+1$. So we just have to develop a method to switch maskings between this kind of blinding and the arithmetic masking modulo M. As it turns out, this is an application of the methods presented in Sect. 4. This is based on the observation

$$\mathbf{I}(d) = (d -_M 1) +_{M+1} 1 \qquad \text{for } d \in V(L) \ .$$

Suppose we have $x, y \in V(L)$ with $y = d +_M x$, and $u \in \{0, 1, \ldots, M\}$. Then $(y -_M 1, x)$ represents $d -_M 1$. We use the algorithms of Sect. 4 to obtain the

"$+_{M+1}$"-masking (u, v) of $d -_M 1$, that is $v = (d -_M 1) +_{M+1} u$. Finally $v +_{M+1} 1 = \mathbf{I}(d) +_{M+1} u$. So $v = \mathbf{I}(d) +_{M+1} u$ can be computed easily without leaking any information about d.

Conversely, let (u, v) represent $\mathbf{I}(d)$ for some sensitive data $d \in V(L)$. Then we apply the algorithm $\mathbf{Add}_{M+1} \to \mathbf{Add}_M$ on $(u, v -_{M+1} 1)$ and some mask $x \in V(L)$ to receive $y \in V(L)$ such that $y = d -_L 1 +_L x$, i.e $y +_M 1 = d +_L x$ is the masked value of d with mask x. Indeed, $v -_{M+1} 1 = (d -_M 1) +_{M+1} u$. This shows how to convert some "$\odot$"-masking into a blinding for standard arithmetic.

6 Conclusion

Motivated by the task of a DPA-resistant implementation of IDEA and other encryption schemes using mixed group operations, we presented algorithms to switch the mask between blindings of boolean, standard arithmetic and some non-standard arihmetic operations. Our methods are based on small randomized tables with an additional random bit deciding the computation mode.

References

1. R. Anderson, M. Kuhn: *Soft Tempest: Hidden Data Transmission Using Electromagnetic Emanations.* Proc. of Information Hiding (1998).
2. J.-S. Coron, L. Goubin: *On Boolean and Arithmetic Masking against Differential Power Analysis.* CHES 2000, LNCS **1965** pp. 231–237.
3. J.-S. Coron, A. Tchulkine: *A New Algorithm for Switching from Arithmetic to Boolean Masking.* CHES 2003, LNCS **2779** pp. 89–97.
4. N. Ferguson, C. Hall, J. Kelsey, B. Schneier, D. Wagner, D. Whiting: *Twofish: A* 128-*Bit Block Cipher.*
5. L. Goubin: *A Sound Method for Switching between Boolean and Arithmetic Masking.* CHES 2001, LNCS **2162** pp. 3–15.
6. P. Kocher, J. Jaffe, B. Jun: *Differential Power Analysis.* Advances in Cryptology – CRYPTO 1999, Proceedings, LNCS **1666**, pp. 388-397.
7. X. Lai, J.L. Massey: *A Proposal for a New Block Encryption Standard.* Advances in Cryptology – EUROCRYPT 1990, Proceedings, LNCS **473**, pp. 389-404.
8. A.J. Menezes, P.C. van Oorschot, S.A. Vanstone: *Handbook of Applied Cryptography.* CRC, (1996).
9. T.S. Messerges: *Securing the AES Finalists Against Power Analysis Attacks.* FSE 2000, LNCS **1978** (Springer).
10. NIST: *FIPS 180-2: Secure Hash Standard.* August 2001.
11. J.-J. Quisquater, D. Samyde: *A new tool for non-intrusive analysis of Smart Cards based on electro-magnetic emmisions: the SEMA and DEMA methods.* Eurocrypt rump session, Bruges, Belgium, May 2000.
12. R.L. Rivest, M.J.B. Robshaw, R. Sidney, Y.L. Yin: *The RC6 Block Cipher.*