Secure Multiplication for Bitslice Higher-Order Masking: Optimisation and Comparison

Dahmun Goudarzi[1,2], Anthony Journault[3], Matthieu Rivain[1], and François-Xavier Standaert[3]

[1] CryptoExperts, Paris, France
[2] ENS, CNRS, INRIA and PSL Research University, Paris, France
[3] ICTEAM/ELEN/Crypto Group, Université catholique de Louvain, Belgium

dahmun.goudarzi@cryptoexperts.com
anthony.journault@uclouvain.be
matthieu.rivain@cryptoexperts.com
fstandae@uclouvain.be

Abstract. In this paper, we optimize the performances and compare several recent masking schemes in bitslice on 32-bit arm devices, with a focus on multiplication. Our main conclusion is that efficiency (or randomness) gains always come at a cost, either in terms of composability or in terms of resistance against horizontal attacks. Our evaluations should therefore allow a designer to select a masking scheme based on implementation constraints and security requirements. They also highlight the increasing feasibility of (very) high-order masking that are offered by increasingly powerful embedded devices, with new opportunities of high-security devices in various contexts.

1 Introduction

Nowadays, higher-order masking is one of the soundest approaches to protect the implementation of a block cipher against side-channel attacks. Recent studies have shown that the bitslice implementation strategy can provide the best performances in software [DPV01,BGRV15,GR17,JS17]. This strategy allows to perform parallel evaluations of a Boolean circuit where the logical gates are replaced by instructions working on registers of several bits. Then, higher-order masking is applied at the Boolean level, where the linear gates become linear instructions working on registers and non-linear gates become calls to secure bitwise non-linear operations.

Since secure non-linear operations are quadratic in the masking order d (whereas for linear operation the cost is in $O(d)$), their evaluation is the main bottleneck for implementers. In the past couple of years, several multiplication schemes have been proposed in the literature offering different tradeoffs between security (*e.g.*, in terms of composability or resistance against so-called horizontal attacks) and performances (timings, randomness requirements).

One of the most popular algorithms is the so-called ISW multiplication scheme introduced in the seminal work of Ishai, Sahai and Wagner at Crypto

2003 [ISW03]. It provides composable security captured by the notion of *Strong Non Interference* (SNI). Based on this construction, Belaïd *et al.* proposed at Eurocrypt 2016 [BBP+16] a variant with randomness savings at the cost of only satisfying the (weaker) *Non Interference* (NI) notion. At CHES 2016, Battistello *et al.* [BCPZ16] went the other way by proposing not only SNI security but also improved resistance against horizontal attacks, at the cost of increased randomness requirements. Eventually, at Eurocrypt 2017, Barthe *et al.* introduced an alternative approach to the ISW-based multiplications. This approach is optimized for parallel implementations such as bitslicing and handles registers that hold all the shares of a given bit. It also comes with different security risks in terms of assumptions. Namely, storing the shares of a single bit potentially allows better resistance against shares re-combinations due to transitions [CGP+12,BGG+14], while leading to higher risks of re-combinations due to couplings [CBG+17]. Journault and Standaert [JS17] compared this new approach with the ISW approach for the AES S-box implemented by Goudarzi and Rivain [GR17], showing that for the optimal case, *i.e.* the masking order is equal to the size of the register, Barthe *et al.* 's approach slightly outperforms ISW multiplication. However, no comparison has been made with other masking orders.

In this paper, we aim to optimize and compare these different masking schemes, in order to better understand the performance gains and overheads that correspond to their different security guarantees. For this purpose, we first try to increase the efficiency of these four schemes, not only at the algorithmic level, but also by taking into account the implementation perspective and possible implementation tricks. We also propose an efficient way to evaluate the Barthe *et al.* multiplication when the masking order is lower than the architectures size. Subsequently, we propose a comparison regarding different aspects such as timing cost, memory overhead, randomness usage and the given security level for each of these multiplications. Ultimately, the goal of this paper is therefore to provide insight to designers and developers who wish to protect efficiently a block cipher with higher-order masking (*i.e.*, which multiplication scheme to use depending on their needs, depending on their hardware limitations or security requirements).

This paper is organized as follows. Section 2 gives some preliminaries on bitslice higher-order masking and security notions. We then introduce in Section 3 the four multiplications studied and discuss the proposed optimization either at the algorithmic level or for the implementation prospective. Section 4 presents the two refresh mask algorithms (ISW and Barthe *et al.* based) that are needed when implementing a block cipher. Finally, Section 5 describes our implementations and the obtained performances to compare the multiplications as well as the refreshing procedures.

The code source of all our implementations is available on Github [GJRS18] under the GPL licence (v3).

2 Preliminaries

2.1 Bitsliced Higher-Order Masking

One of the most studied countermeasure against said-channel attacks is *masking*, a.k.a. *secret sharing*. It consists in splitting a secret value x into d shares $x_1, x_2, \ldots, x_d$ satisfying

$$x = x_1 \oplus x_2 \oplus \cdots \oplus x_d$$

where $x_2, \cdots, x_d$ are randomly distributed and x_1 is computed accordingly. The parameter d is then called the *masking order*.

Recently, bitslicing has been shown to give excellent performances for block cipher implementations protected with masking in software [GR17,JS17]. The bitslice implementation strategy is to perform parallel evaluations of a Boolean circuit where the logic gates are replaced by instructions working on registers of several bits. In the context of masked implementations of block ciphers, this strategy is applied to speed up the evaluations of S-boxes, which are then computed in parallel. Each XOR gate in the underlying Boolean circuit gives rise to d bitwise XOR instructions and each AND gate is replaced by a secure bitwise AND operation based on a secure multiplication scheme such as the ones studied in this paper.

2.2 Security Notions

In the following we informally recall the different security models usually considered in the side-channel community, starting from the abstract probing model (NI/SNI security), then the intermediate bounded moment model and finally the practical noisy leakage model.

At Crypto 2003, Ishai, Sahai and Wagner introduced in their seminal paper [ISW03] the so-called *probing model*. In this model, the adversary is allowed to probe a limited number of wires in a target (protected) circuit. If no adversary is able to recover secret information using up to t probes, the circuit is said to be *t-probing secure*. In their paper, the authors show how to achieve t-probing security using masking of order $d = 2t + 1$. It was later shown that a t-probing secure multiplication can be obtained with $d = t + 1$ shares [RP10] but this approach might result in some security flaws while composing several masked operations without proper mask refreshing [CPRR14]. The stronger notion of SNI has then been introduced in [BBD+16]: it allows to prove t-probing security with only $d = t + 1$ shares for the composition of several gadgets. In particular, any masked circuit composed of t-SNI secure gadgets is also t-SNI secure (and therefore t-probing secure).

Next, the bounded moment model was introduced in [BDF+17] as a relaxation of the probing model in order to capture parallel implementations of masking schemes where all the shares might be contained in a single register and processed in a single cycle (which is hardly captured with the probing model). The idea of the bounded moment model is to look at the higher-order moments to

get a security parameter (since the security of a masked implementation usually comes from the need to evaluate higher-order statistical moments).

Eventually, the noisy leakage model was introduced by Prouff and Rivain in [PR13] and reflects concrete adversaries who obtain an intermediate value perturbed by a noisy leakage function. This is a more realistic side-channel model as this is typically what an attacker can recover from a side-channel analysis. Masking can be formally shown to be a sound countermeasure in such a model since the information revealed on a variable x by noisy leakages on the shares $x_1, \ldots, x_d$ decreases exponentially with the masking order d [CJRR99,PR13]. The latter model is in general more tricky to manipulate, but is strictly needed to evaluate security against so-called horizontal attacks, where the repeated manipulation of the shares enable to get rid of a part of the noise as the masking order d grows. Under the condition of sufficient noise and independence, security in the noisy leakage model is implied by probing security [DDF14].

2.3 ARMv7 Architectures

We made our implementation in the generic ARM v7 assembly language. Most of the ARM processors are composed of 16 registers of 32 bits, ranging from $R0$ to $R15$: registers $R0$ to $R12$ are known as variable registers and are available for computation. The three last registers are usually reserved for special purposes: $R13$ is used as the stack pointer (SP), $R14$ is the link register (LR) storing the return address during a function call, and $R15$ is the program counter (PC).

The ARM instruction set is essentially composed of three classes (summarized in Table 1): the data instructions which performs arithmetic operations on the register, the memory instructions which allows to load and store data and the branching instructions which are used for loops, conditional statements and function calls. One important feature of the ARM assembly is the *barrel shifter* allowing any data instruction to shift one of its operands at no extra cost in terms of clock cycles. However to fully benefit from its efficiency, the rotation offset for the barrel shifter needs to be defined with immediate values instead of registers.

Table 1. ARM instructions.

Class	Examples	Clock cycles
Data instructions	`EOR`, `ADD`, `SUB`, `AND`, `MOV`	1
Memory instructions	`LDR`, `STR` / `LDM`, `STM`	3 or $n + 2$
Branching instructions	`B`, `BX`, `BL`	3 or 4

Eventually, we assume that our target architecture include a True Random Number Generator (TRNG), that frequently fills a register with a fresh 32-bit random string. We consider two different settings for this TRNG: the setting of [GR17] where one needs to wait 10 clock cycles to get a new random string;

and the one of [JS17] where one needs to wait 80 clock cycles to get a new random string. The TRNG register can then be read at the cost of a single load instruction.

3 Secure Multiplications

In this section we describe optimized low-level implementations of the four following secure multiplications:

ISW (Ishai-Sahai-Wagner, Crypto'03): probing secure multiplication,
BDF$^+$ (Barthe *et al.* , Eurocrypt'17): bounded-moment secure multiplication,
BBP$^+$ (Belaïd *et al.* , Eurocrypt'16): ISW gadget with randomness saving,
BCPZ (Battistello *et al.* , CHES'16): ISW gadget with additional refreshing.

Each multiplication is described at the algorithmic level and at the implementation level (with possible implementation tricks). We further give implementation results (clock cycles and code size) in the first TRNG setting. The four schemes are then compared in terms of performances, randomness consumption, and security guarantees in Section 5 in both TRNG settings.

3.1 ISW: the Standard Probing-Secure Multiplication

At Crypto 2003, Ishai, Sahai and Wagner [ISW03] proposed an algorithm to securely compute an AND gate for any number of shares d, the so-called ISW multiplication is described in Algorithm 1. They also introduced the probing model and proved that their multiplication has a security order $t = \lfloor (d-1)/2 \rfloor$ in this model. The security proof was extended to the order $t = d-1$ in [RP10] and to the stronger t-SNI property in [BBD$^+$16], both extensions assuming independent input sharings.

Algorithm 1 ISW (Ishai-Sahai-Wagner, Crypto'03)

Input: sharings $(a_1, \ldots, a_d) \in \{0,1\}^{32\times d}$ and $(b_1, \ldots, b_d) \in \{0,1\}^{32\times d}$
Output: sharing $(c_1, \ldots, c_d) \in \{0,1\}^{32\times d}$ such that $\bigoplus_i c_i = (\bigoplus_i a_i) \wedge (\bigoplus_j b_j)$

1. **for** $i = 1$ to d **do**
2. $\quad c_i \leftarrow a_i \wedge b_i$
3. **end for**
4. **for** $i = 1$ to d **do**
5. $\quad$ **for** $j = i+1$ to d **do**
6. $\quad\quad s \leftarrow \{0,1\}^{32}$
7. $\quad\quad s' \leftarrow (s \oplus (a_i \wedge b_j)) \oplus (a_j \wedge b_i)$
8. $\quad\quad c_i \leftarrow c_i \oplus s$
9. $\quad\quad c_j \leftarrow c_j \oplus s'$
10. $\quad$ **end for**
11. **end for**
12. **return** $(c_1, \ldots, c_d)$

From two sharings $(a_1, \ldots, a_d)$ and $(b_1, \ldots, b_d)$, the ISW multiplication simply computes all the d^2 crossed products $a_i \cdot b_j$ which are then summed in d new shares c_i with new random elements $r_{i,j}$. Each new random element is involved twice in the new shares implying $\bigoplus_i c_i = \bigoplus_{i,j} a_i \cdot b_j = (\bigoplus_i a_i) \cdot (\bigoplus_j b_j)$. The ISW scheme is pictured in Algorithm 1 for the bitwise setting, where $\wedge$ and $\oplus$ denote the (32-bit) bitwise AND and XOR.

From the implementation viewpoint, we follow the work of [GR17] and implement the scheme without any particular implementation trick for any masking order d. In order to push forward the optimization, we also propose a version of the code where the nested loops are unrolled for specific values of d, namely when d is a power of 2. The performances of our low-levels implementations are summarized in Table 2. We observe that unrolling the loops allows us to save 15% to 23% clock cycles with an overhead factor from 3 to 200 for the code size. The only case where the unrolling fully benifits in both time and memory is for $d = 2$.

Table 2. Implementation results for the ISW multiplication

	clock cycles					code size (bytes)					register usage	random usage
d	2	4	8	16	32	2	4	8	16	32		
Straight ISW	75	291	1155	4611	18435	164	164	164	164	164	10	$d\,(d-1)/2$
Unrolled ISW	58	231	949	3876	15682	132	464	1848	7500	30324	8	$d\,(d-1)/2$

3.2 BDF$^+$: a Bounded-Moment Secure Multiplication

At Eurocrypt 2017, Barthe *et al.* introduced a new way to compute a secure multiplication specifically tailored for the bitwise context (*i.e.* for bitslice implementations) [BDF$^+$17]. Their scheme handles registers holding all the shares of a given bit whereas in traditional ISW-based scheme, the shares of a variable are stored in different registers for security reasons. Nevertheless, Barthe *et al.* show that their multiplication is secure in the relaxed bounded moment model, which is argued to be sound in practice.

Intuitively the BDF$^+$ multiplication can be decomposed in different steps: the loading of the input shares a and b; the computation of the partial products between a and b; the loading of fresh randomness r; and the compression phase where these partial products are XORed all together and separated by the fresh randomness.

Its implementation is especially efficient when the number of shares d is equal to the size of the registers in the target architecture. This has been shown in [JS17] for the case $d = 32$. However, a question left open in the latter work is the scenario where the number of shares mismatches the register size. This issue is addressed hereafter.

For this purpose, we generalize the BDF$^+$ algorithm to a scenario where d can be lower than the register size. We propose a parallel version of this algorithm in which several sharings are stored in a register (*e.g.* 4 sharings of order $d = 8$ in one 32-bit register) and we describe an efficient way to perform sharing-wise rotations to keep good performances in such a non-optimal scenario. The main restriction is that our generalization only works for masking order that are power of 2 (so that the sharing size divides the register size), including the case $d = 2$ which was not taken into account in the original publication. The optimized BDF$^+$ multiplication is described in Algorithm 2.

Algorithm 2 BDF$^+$ (Barthe *et al.* , Eurocrypt'17)

Input: shares $a = (a_1, \cdots, a_d) \in \{0,1\}^{32}$, shares $b = (b_1, \cdots, b_d) \in \{0,1\}^{32}$
Output: shares $c = (c_1, \cdots, c_d) \in \{0,1\}^{32}$

1. $x_1 \leftarrow a \wedge b$
2. $r \leftarrow \{0,1\}^{32}$
3. $y_1 \leftarrow x_1 \oplus r$
4. **if** $d = 2$ **then**
5. $\quad x_2 \leftarrow a \wedge \mathsf{ROT}(b, 1)$
6. $\quad y_2 \leftarrow y_1 \oplus x_2$
7. $\quad c \leftarrow y_2 \oplus \mathsf{ROT}(r, 1)$
8. **else**
9. $\quad$ **for** $i = 1$ to $d/2 - 1$ **do**
10. $\quad\quad$ **if** $i \mod 2 = 0$ **then**
11. $\quad\quad\quad r \leftarrow \{0,1\}^{32}$
12. $\quad\quad$ **end if**
13. $\quad\quad x_{2i} \leftarrow a \wedge \mathsf{ROT}(b, i)$
14. $\quad\quad x_{2i+1} \leftarrow \mathsf{ROT}(a, i) \wedge b$
15. $\quad\quad y_{3i-1} \leftarrow y_{3i-2} \oplus x_{2i}$
16. $\quad\quad y_{3i} \leftarrow y_{3i-1} \oplus x_{2i+1}$
17. $\quad\quad y_{3i+1} \leftarrow y_{3i} \oplus \mathsf{ROT}(r, i \mod 2)$
18. $\quad$ **end for**
19. $\quad x_d \leftarrow a \wedge \mathsf{ROT}(b, d/2)$
20. $\quad c \leftarrow y_{3\lfloor (d-1)/2 \rfloor + 1} \oplus x_d$
21. **end if**
22. **return** c

Encoding. In order to make full use of the register when d is less than 32 (*i.e.* d is not equal to the architecture size), but d is a power of 2, we fill the input registers with $k = 32/d$ words of d shares. We thus process k secure multiplications in parallel. More specifically, let us denote $w_0, \ldots, w_{31}$ the bits of a 32-bit register $\boldsymbol{w}$ (from MSB to LSB). For $d = 16$, $\boldsymbol{w}$ encodes 2 secret bits z_0 and z_1 such that $\bigoplus_{i=0}^{15} w_i = z_0$ and $\bigoplus_{i=16}^{31} w_i = z_1$. For $d = 8$, $\boldsymbol{w}$ encodes 4 secret bits z_0, z_1, z_2 and z_3 such that $\bigoplus_{i=0}^{7} w_i = z_0$ and $\bigoplus_{i=8}^{15} w_i = z_1$ and $\bigoplus_{i=16}^{23} w_i = z_2$ and $\bigoplus_{i=24}^{31} w_i = z_3$, and so on.

Efficient sharing-wise rotation. Algorithm 2 can directly be applied on multi-sharing input registers. The only operation which needs to be modified accordingly is the rotation $\mathsf{ROT}(w, i)$. We propose an efficient low-level implementation for such a sharing-wise rotation. Our method relies on the observation that applying an i-bit rotation to every d-bit chunck in a word w can be obtained by the following equation:

$$\mathsf{ROT}(w, i) = \big((w \ll i) \wedge \mathtt{mask}_{d,i}\big) \oplus \big((w \gg d - i) \wedge \overline{\mathtt{mask}}_{d,i}\big) \tag{1}$$

where $\mathtt{mask}_{d,i}$ is a selection mask defined as

$$\mathtt{mask}_{d,i} = \underbrace{\mathtt{11\ldots1}}_{d-i}\underbrace{\mathtt{00\ldots0}}_{i} \,\|\, \cdots \,\|\, \underbrace{\mathtt{11\ldots1}}_{d-i}\underbrace{\mathtt{00\ldots0}}_{i} \ ,$$

and $\overline{\mathtt{mask}}_{d,i}$ denotes its complement. From this equation we can directly compute the sharing-wise rotation. The main trick in the implementation is to efficiently deal with the generation of $\mathtt{mask}_{d,i}$ and the sharing-wise rotation.

The mask generation is decomposed into two steps. The first step allows to setup the mask correctly: $\mathtt{mask}_{d,0}$ is initialized with the value `0xFFFFFFFF`. We then need a `correction` value which will be used to update the mask correctly. `correction` is initialized with values given in Table 3. Note that these operations are performed only once at the beginning of the multiplication. The second step will update the mask for the rotation according to the offset of the rotation given by the following formula:

$$\mathtt{mask}_{d,i} = \mathtt{mask}_{d,0} \oplus (\mathtt{correction} \ll i)$$

In practice, we only store $\mathtt{mask}_{d,0}$ and `correction` in two registers and we update them accordingly in each iteration of the loop. The cost of the update is 2 cycles.

```
;;mask update
EOR     $mask, $mask, $correction
LSL     $correction, $correction, #1
```

Note that we make use of another register in order to store $\mathtt{mask}_{d,1}$ (*i.e.* the rotation by 1) which is always needed to compute the rotations of the random values (instead of computing it again each time).

Table 3. Possible values for correction

d	2	4	8	16
`correction`	0x5555555	0x11111111	0x01010101	0x00010001

The rotation $\mathsf{ROT}(w, i)$ is then quite straightforward to implement as describes hereafter:

```
;; rotation of $w by $i
    AND     $tmp, $mask, $w, LSL $i
    LSR     $w, $w, $(d-i)
    BIC     $w, $w, $mask
    EOR     $w, $tmp, $w
```

Since the offsets of the shift lie in a register, we cannot benefit from the *barrel shifter*. Hence the overall cost of one rotation is 5 cycles.

In Table 4, we report results of our implementation of the BDF$^+$ multiplication for d ranging from 2 to 32 for the generic version and an unrolled version (where the main advantage is to be able to hardcode the masks and values for the shifts). We observe that the unrolled version for $d = 32$ is faster and has less code size than for $d = 16$. This is easily explained by the fact that we can make full use of the *barrel shifter* in the case $d = 32$. Moreover, we observe that the unrolled version is 40% to 80% faster than the regular version. This is due to the fact that we can hardcode the masks, which makes the *barrel shifter* work again. The code size of the unrolled version ranges from 0.3 to 3 times the generic one. Note also that the code size of the generic version is decreasing as d grows because we compute the `correction` value iteratively (*i.e.* it needs $\log(32/d)$ iterations).

Table 4. Performance results for BDF$^+$ (generic and unrolled)

	clock cycles					code size (bytes)					registers	random usage
d	2	4	8	16	32	2	4	8	16	32		
BDF$^+$ generic	n/a	77	146	285	n/a	n/a	248	244	240	n/a	13	$\lceil (d-1)/4 \rceil$
BDF$^+$ unrolled	34	47	81	149	120	280	356	504	808	748	13	$\lceil (d-1)/4 \rceil$

3.3 BBP$^+$: Towards Optimal Randomness Consumption

Belaïd *et al.* at Eurocrypt 2016 [BBP$^+$16] tackled the problem of minimizing the amount of randomness required in a secure multiplication. They described a generic algorithm which makes use of less randomness than ISW, reducing the former randomness requirement from $\frac{d\,(d-1)}{2}$ to $\frac{d^2}{4} + d$. As opposed to the ISW multiplication (which achieves $(d-1)$-SNI security), this algorithm is only proven $(d-1)$-NI secure. The original description of this secure multiplication (see [BBP$^+$16]) is generic for any masking order $d \geq 4$ (specific algorithms for the case where $d = 2$ and 3 are given in their paper). However, it makes use of several conditional branches to process additional operations depending on the parity of the order d and/or of the loop index i.

We rewrote the algorithm such that all the conditional branches are removed, without affecting the correctness (see Algorithm 3). These changes lead to sev-

eral improvements in practice: first replacing if/else statement with loops allows avoiding several conditional branches treatment that are quit expensive in ARM assembly. Moreover, by rewriting the algorithm in such a way, we can compute all the randomness on-the-fly and avoid multiple load and store instructions for the correction step. Such improvements come at the cost of a less genereic algorithm (it only works for even orders d). For the sake of comparison, we have implemented both algorithms to show the performance gained in clock cycles and code size (see Table 5). We can see that our improvements allow a gain in timing ranging from 18% to 20% with an overhead of only 80 bytes of memory. Furthermore, we also unrolled the nested loops in order to get better results in timings. The timing gain ranges from 17% to 60% with an overhead factor between 3.5 and 50 for the code size for $d \geq 8$ only. For smaller d's, the unrolled version is better for both timing and code size.

Algorithm 3 BBP^+ (Belaïd *et al.* , Eurocrypt'16) w/o conditional branches

Input: sharings $(a_1, \ldots, a_d) \in \{0,1\}^{32\times d}$ and $(b_1, \ldots, b_d) \in \{0,1\}^{32\times d}$
Output: sharing $(c_1, \ldots, c_d) \in \{0,1\}^{32\times d}$ such that $\bigoplus_i c_i = (\bigoplus_i a_i) \wedge (\bigoplus_j b_j)$

1. $c_1 \leftarrow a_1 \wedge b_1$
2. $c_2 \leftarrow a_2 \wedge b_2$
3. **for** $i = 3$ to $d-1$ by 2 **do**
4. $\quad c_i \leftarrow a_i \wedge b_i$
5. $\quad c_{i+1} \leftarrow a_{i+1} \wedge b_{i+1}$
6. $\quad s_i \leftarrow \{0,1\}^{32}$
7. **end for**
8. **for** $i = 1$ to $d-1$ by 2 **do**
9. $\quad r_{i,i+1} \leftarrow \{0,1\}^{32}$
10. $\quad \text{LoopRow}(i, i+3)$
11. $\quad c_i \leftarrow c_i \oplus (r_{i,i+1} \oplus a_i \wedge b_{i+1} \oplus a_{i+1} \wedge b_i)$
12. $\quad \text{LoopRow}(i+1, i+3)$
13. $\quad c_{i+1} \leftarrow c_{i+1} \oplus r$
14. **end for**

Algorithm 4 LoopRow Procedure

Input: indexes i, t randoms $(s_j)_{j\in\{3,\ldots,d-1\}}$

1. **for** $j = d$ down to t by 2 **do**
2. $\quad r_{i,j} \leftarrow \{0,1\}^{32}$
3. $\quad c_i \leftarrow c_i \oplus \big(r \oplus (a_i \wedge b_j \oplus a_j \wedge b_i) \oplus s_{j-1} \oplus (a_i \wedge b_{j-1} \oplus a_{j-1} \wedge b_i)\big)$
4. $\quad c_j \leftarrow c_j \oplus r_{i,j}$
5. **end for**

Table 5. Implementation results for the BBP$^+$ multiplication

	clock cycles					code size (bytes)					register usage	random usage
d	2	4	8	16	32	2	4	8	16	32		
Original BBP$^+$	n/a	334	1204	4552	17680	n/a	344	344	344	344	12	$d + d^2/4$
Optimized BBP$^+$	88	274	970	3658	14218	428	428	428	428	428	12	$d + d^2/4$
Unrolled BBP$^+$	36	161	775	3018	11920	100	344	1544	5996	23732	11	$d + d^2/4$

3.4 BPCZ: Towards Security against Horizontal Attacks

At CHES 2016, Battistello *et al.* described a horizontal side-channel attack on the standard ISW multiplication [BCPZ16]. This attack essentially consists in reducing the noise in the targeted values by averaging them. More precisely, during the computation of Algorithm 1, each share a_i (resp. b_i) is manipulated d times. Hence one can *average* the noise and reduce it by a factor $\sqrt{d}$ (in a standard deviation metric). Such an attack is inherent to the ISW scheme and implies that despite the probing-security, increasing the masking order d implies increasingly high noise requirements for the masking countermeasure to bring security improvements (i.e., for the noise to be large enough after averaging, it has to increase before averaging).

Battistello *et al.* also proposed a mitigation of such a horizontal attack. Their multiplication, given in Algorithm 5, is similar to the standard ISW multiplication but the matrix of the crossed products $a_i \cdot b_j$ is computed differently (see Algorithm 6): refreshings are regularly inserted to avoid the multiple apparition of each share a_i (resp. b_i). The RefreshMasks operation is a simple ISW-based refreshing as described later in Section 4. The authors also proved that their multiplication is $(d-1)$-SNI secure.

Algorithm 5 BCPZ (Battistello *et al.* , CHES'16)

Input: shares a_i such that $\sum_i a_i = a$, shares b_i such that $\sum_i b_i = b$
Output: shares c_i such that $\sum_i c_i = a \cdot b$

1. $M_{i,j} \leftarrow \text{MatMult}((x_1, \ldots, x_d), (y_1, \ldots, y_d))$
2. **for** $i = 1$ to d **do**
3. $\quad c_i \leftarrow M_{i,i}$
4. **end for**
5. **for** $i = 1$ to d **do**
6. $\quad$ **for** $j = i + 1$ to d **do**
7. $\quad\quad s \leftarrow \mathbb{F}$
8. $\quad\quad s' \leftarrow (s + M_{i,j}) + M_{j,i}$
9. $\quad\quad c_i \leftarrow c_i + s$
10. $\quad\quad c_j \leftarrow c_j + s'$
11. $\quad$ **end for**
12. **end for**
13. **return** $c_1, ..., c_d$

Algorithm 6 MatMult

Input: the n-sharings $(x_i)_{i\in[1..n]}$ and $(y_i)_{i\in[1..n]}$ of x^* and y^* respectively
Output: the n^2-sharing $(M_{i,j})_{i\in[1..n],j\in[1..n]}$ of $x^* \cdot y^*$

1. **if** $n = 1$ **then**
2. $\quad \boldsymbol{M} \leftarrow [x_1 \cdot y_1]$
3. **else**
4. $\quad \boldsymbol{X}^{(1)} \leftarrow (x_1, \ldots, x_{n/2})$, $\boldsymbol{X}^{(1)} \leftarrow (x_{n/2+1}, \ldots, x_n)$
5. $\quad \boldsymbol{Y}^{(1)} \leftarrow (y_1, \ldots, y_{n/2})$, $\boldsymbol{Y}^{(1)} \leftarrow (y_{n/2+1}, \ldots, y_n)$
6. $\quad \boldsymbol{M}^{(1,1)} \leftarrow \text{MatMult}(\boldsymbol{X}^{(1)}, \boldsymbol{Y}^{(1)})$
7. $\quad \boldsymbol{X}^{(1)} \leftarrow \text{RefreshMasks}(\boldsymbol{X}^{(1)})$, $\boldsymbol{Y}^{(1)} \leftarrow \text{RefreshMasks}(\boldsymbol{Y}^{(1)})$
8. $\quad \boldsymbol{M}^{(1,2)} \leftarrow \text{MatMult}(\boldsymbol{X}^{(1)}, \boldsymbol{Y}^{(2)})$
9. $\quad \boldsymbol{M}^{(2,1)} \leftarrow \text{MatMult}(\boldsymbol{X}^{(2)}, \boldsymbol{Y}^{(1)})$
10. $\quad \boldsymbol{X}^{(2)} \leftarrow \text{RefreshMasks}(\boldsymbol{X}^{(2)})$, $\boldsymbol{Y}^{(2)} \leftarrow \text{RefreshMasks}(\boldsymbol{Y}^{(2)})$
11. $\quad \boldsymbol{M}^{(2,2)} \leftarrow \text{MatMult}(\boldsymbol{X}^{(2)}, \boldsymbol{Y}^{(2)})$
12. $\quad \boldsymbol{M} \leftarrow \begin{pmatrix} \boldsymbol{M}^{(1,1)} & \boldsymbol{M}^{(1,2)} \\ \boldsymbol{M}^{(2,1)} & \boldsymbol{M}^{(2,2)} \end{pmatrix}$
13. **end if**
14. **return** $\boldsymbol{M}$

The implementation of Algorithm 5 is straightforward (same as ISW). The main challenge is to efficiently implement Algorithm 6 in a recursive way. In fact, due to the restrictive amount of registers available, using functions to perform the recursion in ARM assembly becomes very costly. Each recursive call needs to have access to several informations: the correct set of input sharings, namely the start of $\boldsymbol{X}_1$, $\boldsymbol{X}_2$, $\boldsymbol{Y}_1$ and $\boldsymbol{Y}_2$ as well as the correct adresses for the output sharings. This means that several registers containing those information needs to be pushed to the stack prior to each call to a recursive function and poped before the computation. As push and pop are basically load and store in ARM assembly the total cost of managing the inputs and outputs of a recursive function is approximately equal to a dozen of clock cycles for each recursive calls. This costs, on top of the associated jumps for each recursive fucntion, is equivalent to the computation of a complete ISW multiplication. Therefore and since we restrict ourselves in this study to $d \leq 32$, we developed the MatMult procedure with macros. Specifically, for each masking order d that is a power of 2, we simply implements Algorithm 6 using macros for each possible input sharing size $n \in \{2, 4, \ldots, 32\}$, which allows us to save several clock cycles. However the main drawback of implementing the MatMult procedure in such way is that the code size exponentially grows. To lower down the explosion of the code size, we have also implemented a version of the code where the terminal case macro (for $n = 2$) is implemented as a function. This allows us to divide by up to 5 the code size while having a performance decrease of around 20%. Both timing and code size for the BPCZ multiplication with the two versions of the MatMult procedure are given in Table 6.

Table 6. Implementation results for the BCPZ multiplication

	clock cycles					code size (bytes)					register	random usage
d	2	4	8	16	32	2	4	8	16	32		
BCPZ (macros)	108	498	2106	8698	35386	240	648	2324	9368	38168	13	$(\log(d)-1)d^2/2-(d/2-1)d$
BCPZ (functions)	134	593	2529	10473	42649	400	476	780	1996	6860	13	$(\log(d)-1)d^2/2-(d/2-1)d$

4 Refresh Masks

Most of the multiplication gadgets rely on the condition that their two inputs have to be independently shared in order to guaranty full security (and avoid doubling the number of shares instead). But in complex circuit involving many multiplications and other linear operations, this condition might not always be fulfilled (*e.g.* the two inputs of a multiplication might be linearly related) leading to security flaws as pointed out by Coron *et al.* [CPRR14]. Refreshing gadgets (in particular SNI ones) allow avoiding such kind of behavior if used systematically on one of the input of the multiplication. In this section, we describe and compare the refresh gadgets associated to their multiplication, *i.e.* the ISW refresh and the BDF$^+$ refresh.

4.1 ISW

A sound refresh can be performed by using the ISW multiplication: it simply consist in multiplying the shares a_i by the vector $(1, 0, \cdots, 0)$ and has been proven $(d-1)$-SNI secure. The ISW refresh needs $d(d-1)/2$ random bits and performs $d(d-1)$ additions. The overall algorithm is described in Algorithm 7.

Algorithm 7 ISW Refresh

Input: shares $a_1, a_2, \ldots, a_d$
Output: shares $c_1, c_2, \ldots, c_d$ such that $\sum_{i=1}^{d} c_i = \sum_{i=1}^{d} a_i$

1. **for** $i = 1$ to d **do**
2. $\quad c_i \leftarrow a_i$
3. **end for**
4. **for** $i = 1$ to d **do**
5. $\quad$ **for** $j = i + 1$ to d **do**
6. $\quad\quad r \leftarrow \{0,1\}^{32}$
7. $\quad\quad c_i \leftarrow c_i \oplus r$
8. $\quad\quad c_j \leftarrow c_j \oplus r$
9. $\quad$ **end for**
10. **end for**
11. **return** $c_1, c_2, \ldots, c_d$

As shown by Goudarzi and Rivain [GR17], this refreshing procedure can be optimized by partially unrolling the nested loops by taking advantages of

Table 7. Implementation results for the ISW refresh

	clock cycles					code size (bytes)					register usage	random usage
d	2	4	8	16	32	2	4	8	16	32		
ISW Refresh	51	72	239	933	3761	224	224	224	224	224	10	$d\,(d-1)/2$

available registers. This allows to load multiple shares at once and perform the sound operations on all of them, instead of doing it one by one. Namely, for masking orders equal to power of 2, this allows to load the a_i's four by four, namely loading $a_i, a_{i+1}, a_{i+2}, a_{i+3}$ and have the number of operations in the nested loop divided by 4. As in ARM assembly, the multiple load instruction is more efficient that several single loads, this improvement yields a very efficient ISW-based refresh implementation. The performance results of the ISW refresh can be found in Table 7.

4.2 BDF$^+$ Refresh

Barthe *et al.* in [BDF$^+$17], along with their multiplication gadget, also provide a refreshing gadget described in Algorithm 8. It simply consists in XORing the share to refresh by a random value and a rotation of it. The iteration of the BDF$^+$ refresh $\lceil (d-1)/3 \rceil$ times makes it SNI secure. The overall BDF$^+$ refresh needs $d\lceil (d-1)/3 \rceil$ random bits and performs $2\lceil (d-1)/3 \rceil$ additions and $\lceil (d-1)/3 \rceil$ ROT. There is no particular implementations tricks except we use the same ROT algorithm introduced in Section 3.2 in order to keep the correctness with the specific encoding. Implementation results can be found in Table 8.

Algorithm 8 BDF$^+$ Refresh

Input: shares a
Output: shares c
1. $r \leftarrow \{0,1\}^{32}$
2. $c \leftarrow a \oplus r \oplus \mathsf{ROT}(r, 1)$
3. **return** c

Table 8. Implementation results for the BDF$^+$ refresh

	clock cycles					code size (bytes)					register usage	random usage
d	2	4	8	16	32	2	4	8	16	32		
BDF$^+$ Refresh	25	25	25	25	16	116	116	116	116	110	10	d

5 Comparisons and discussion

We conclude the paper by comparing the different implementations and discussing their pros and cons regarding both the security properties they guarantee and the performances they allow.

5.1 High level comparison

In Table 9 we gather the four multiplications we studied in this paper and we compare them at an algorithmic level. Namely, we give the operation counts (in terms of 32-bit XOR, 32-bit AND, and sharing-wise ROT) to perform a secure 32-bit AND between two sharings. The NI/SNI row specifies if the considered multiplication is SNI- or NI-secure. The row "max use of shares" represents (informally) the level of protection against horizontal side-channels attacks: $O(d)$ means that each shares is processed a linear number in d times (*i.e.* no protection) and $O(1)$ means that each shares is processed a constant number of times (*i.e.* protection).

We differentiate two cases for the BDF$^+$ multiplication. A first case where we consider the multiplication alone, which is SNI until $d = 3$ and only NI secure afterwards. A second case where we consider the composition of the multplication with one iteration of the BDF$^+$ refresh (described in Section 4), which is SNI secure up to $d = 8$ and only NI secure afterwards (see [BDF$^+$17]). The cost difference between these two versions is simply the cost of an elementary refresh (i.e., the addition of a share of zero). Finding the number of such refreshes that are required to be SNI at any order is an open problem. Note that for BDF$^+$, the results are given for d calls to the multiplication (since each call allows to compute $32/d$ elements).

We note that we did not perform the same addition for the BBP$^+$ multiplication since it would imply the need of a more expensive SNI refresh on the output, which would contradict the goal of [BBP$^+$16] to minimize randomness by mixing NI and SNI multiplications instead of solely SNI multiplications (and in particular, if an SNI multiplication is then required, one could use the ISW one, or the BDF$^+$ up to order 8).

Algorithm:	ISW	BDF$^+$ (BM model)	BDF$^+$ w. refresh (BM model)	BBP$^+$	BCPZ
NI/SNI:	SNI	SNI (up to $d = 3$)	SNI (up to $d = 8$)	NI	SNI
Max use of shares:	$O(d)$	$O(d)$	$O(d)$	$O(d)$	$O(1)$
XOR-32 count:	$2d(d-1)$	$d(3d/2-1)$	$d(3d/2+1)$	$(7d^2-6d)/4$	$d^2\log(d)+2d$
AND-32 count:	d^2	d^2	d^2	d^2	d^2
ROT count:	0	$d(5d/4-1)$	$5d^2/4$	0	0
Random bits:	$16d(d-1)$	$32d\lceil(d-1)/4\rceil$	$32d\lceil(d-1)/4\rceil+32$	$8d^2+16d-1$	$16d^2\log(d)+d$

Table 9. Comparison of the multiplications at the algorithmic level.

We also recall that this table does not mention the different risks of unsatisfied (independence) assumption mentioned in introduction. Namely the fact that the BDF$^+$ multiplication can suffer from a reduced security order due to couplings while for the other algorithms, the main risk of security order reduction comes from transition-based leakages.

5.2 Implementation-based comparison

Based on the results in the previous sections, we can compare the performances of our implementations of the multiplications for bitsliced inputs with higher-order masking in ARM v7. We make the comparison for five masking orders, namely 2, 4, 8, 16 and 32. Moreover, we also give the performance results for two sets of TRNG. For the first one (called the TRNG-1 settings in the following), we make the same assumption as in [GR17] that we need to wait 10 clock cycles to get a fresh 32-bit random word. For the second one (called the TRNG-2 settings in the following), we make the same assumption as in [JS17] that we need to wait 80 clock cycles to get a fresh 32-bit random word. Finally, in order to have a fair comparison between the four algorithms the implementation results are given for the computation of a multiplication between two shared 32-bit operands. This means that for the 3 ISW-based multiplication (ISW, BCPZ, BBP$^+$) the results are given for a single call to their respective functions, whereas for the BDF$^+$ multiplication the results are given for d calls to the function (since each calls allows to compute $32/d$ elements). The overall results are given in Tables 10 and 11 for respectively the TRNG-1 and the TRNG-2 settings. As illustration, we also plot the performances in clock cycles (log scale) for both TRNG-1 and TRNG-2 settings in Figure 1 and Figure 2 respectively.

Table 10. Multiplication performances for TRNG-1.

	TRNG-1									
	clock cycles					code size (bytes)				
d	2	4	8	16	32	2	4	8	16	32
ISW	75	291	1155	4611	18435	164	164	164	164	164
ISW unrolled	58	231	949	3876	15682	132	464	1848	7500	30324
BDF$^+$	n/a	308	1168	4560	n/a	n/a	248	244	240	n/a
BDF$^+$ unrolled	68	188	648	2384	3840	280	356	504	808	748
BDF$^+$ (+ refresh)	n/a	408	1568	5360	n/a	n/a	360	356	352	n/a
BDF$^+$ unrolled (+ refresh)	118	288	1048	3184	5440	392	468	616	920	960
BBP$^+$	88	274	970	3658	14218	428	428	428	428	428
BBP$^+$ unrolled	36	161	775	3018	11910	100	344	1544	5996	23732
BCPZ (macros)	108	498	2106	8698	35386	240	648	2334	9368	38168
BCPZ (macros + functions)	134	593	2529	10473	42649	400	476	780	1996	6860
ISW Refresh	51	72	239	933	3761	236	236	236	236	236
BDF$^+$ Refresh	50	50	50	50	50	128	128	128	128	128

As expected the BCPZ offers the worst performances because of the many refreshings which intend to provide resistance to horizontal side-channel attacks, for both of the TRNG settings.

Table 11. Multiplication performances for TRNG-2.

	TRNG-2									
	clock cycles					code size (bytes)				
d	2	4	8	16	32	2	4	8	16	32
ISW	166	837	3703	15531	63571	500	500	500	500	500
ISW unrolled	149	777	3497	14796	60818	480	872	2508	9264	36600
BDF$^+$	n/a	672	2624	10384	n/a	n/a	596	592	588	n/a
BDF$^+$ unrolled	250	552	2104	8208	27136	448	500	876	1204	1192
BDF$^+$ (+ refresh)	n/a	1136	3552	12240	n/a	n/a	1016	1012	1008	n/a
BDF$^+$ unrolled (+ refresh)	482	1016	3032	10064	30848	868	920	1296	1624	1612
BBP$^+$	270	820	2790	10210	38970	800	800	800	800	800
BBP$^+$ unrolled	127	525	2504	9479	36581	436	716	2096	7172	27776
BCPZ (macros)	199	1408	7202	32358	136942	576	1032	2988	11372	45932
BCPZ (macros + functions)	225	1503	7625	34133	144205	760	836	1128	2344	7208
ISW Refresh	142	345	2241	10761	46713	412	412	412	412	412
BDF$^+$ Refresh	116	116	116	116	116	420	420	420	420	420

The BBP$^+$ multiplication outperforms the ISW multiplication (up to 25% faster) even in the case where the randomness is cheap. The difference becomes more significant in the TRNG-2 context (up to 40% faster), since BBP$^+$ have reduced randomness requirements.

For the TRNG-2 settings, we can also observe that unrolling the loops does not offer an interesting tradeoff as the gain in timing is not very significant compared to the code size overhead.

As shown in Table 2 of [BDF$^+$17], the iteration of the BDF$^+$ refresh requires a bit less randomness than ISW one but is more computationally involved. This is well reflected in Tables 10: the ISW refresh has better performances than the BDF$^+$ refresh for the TRNG-1 setting while it is the opposite for the TRNG-2 setting.

Fig. 1. Multiplication performances for TRNG-1 in clock cycles

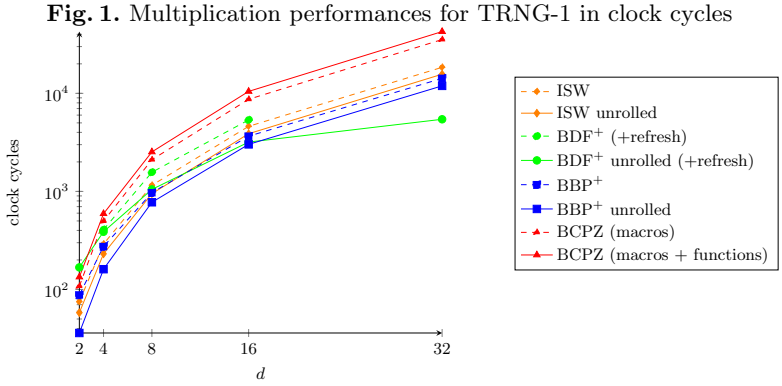
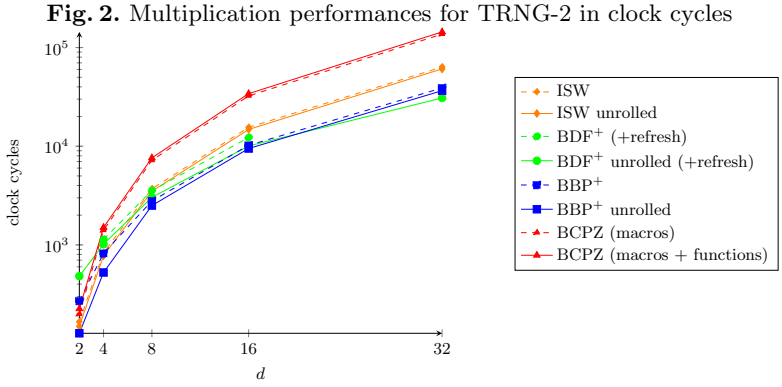


Fig. 2. Multiplication performances for TRNG-2 in clock cycles



Overall, BDF$^+$ and BBP$^+$ multiplications provide the best performances in both TRNG settings thanks to their lower randomness requirements (compared to the classical ISW). Of course these two multiplications also have weaker security guaranties (in terms of composability and resistance against horizontal attacks). On the other hand, ISW and BCPZ offer better security guaranties and hence are more involved in terms of randomness requirements, making these differences more visible in the TRNG-2 setting.

Conclusion and Future Work. One interesting consequence of this observation is that it raises interesting optimization problems on how to best exploit different multiplications in order to obtain the best security vs. performance tradeoff for full implementations (e.g., of block ciphers), which is a nice scope for further research.

Our implementations heavily relies on the use of the barrel shifter of the ARM 32-bit architecture. Comparing these schemes on different architectures and with different register sizes could lead to different performance results (even though the general trend should not differ due to the randomness requirements of the different schemes).

Of course these schemes should also be evaluated considering their practical side-channel security and not only software performances. By providing the code on an open source platform, we hope that this will provide good material for future research in that direction.

Acknowledgments. This work has been funded in part by the European Commission and the Walloon Region through the FEDER project USERMedia (convention number 501907-379156) and by the INNOVIRIS project SCAUT . François-Xavier Standaert is a research associate of the Belgian Fund for Scientific Research.

References

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16: 23rd Conference on Computer and Communications Security*, pages 116–129, Vienna, Austria, October 24–28, 2016. ACM Press.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, volume 9813 of *Lecture Notes in Computer Science*, pages 23–39, Santa Barbara, CA, USA, August 17–19, 2016. Springer, Heidelberg, Germany.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, Paris, France, May 8–12, 2017. Springer, Heidelberg, Germany.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *CARDIS*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.

[BGRV15] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. DPA, bitslicing and masking at 1 GHz. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 599–619, Saint-Malo, France, September 13–16, 2015. Springer, Heidelberg, Germany.

[CBG+17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does coupling affect the security of masked implementations? In *COSADE*, volume 10348 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2017.

[CGP+12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of security proofs from one leakage model to another: A new issue. In *COSADE*, volume 7275 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2012.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany.

[CPRR14] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *Fast Software Encryption – FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 410–424, Singapore, March 11–13, 2014. Springer, Heidelberg, Germany.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440. Springer, 2014.

[DPV01] Joan Daemen, Michael Peeters, and Gilles Van Assche. Bitslice ciphers and power analysis attacks. In Bruce Schneier, editor, *Fast Software Encryption – FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 134–149, New York, NY, USA, April 10–12, 2001. Springer, Heidelberg, Germany.

[GJRS18] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and Francois-Xavier Standaert. Source code. `https://github.com/CryptoExperts/bitslice-masking-multiplication`, 2018.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 567–597, Paris, France, May 8–12, 2017. Springer, Heidelberg, Germany.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany.

[JS17] Anthony Journault and François-Xavier Standaert. Very high order masking: Efficient implementation and security evaluation. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 623–643, Taipei, Taiwan, September 25–28, 2017. Springer, Heidelberg, Germany.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427, Santa Barbara, CA, USA, August 17–20, 2010. Springer, Heidelberg, Germany.