How Fast Can Higher-Order Masking Be in Software?

Dahmun Goudarzi[1,2] and Matthieu Rivain[1]

[1] CryptoExperts, Paris, France
[2] ENS, CNRS, INRIA and PSL Research University, Paris, France

dahmun.goudarzi@cryptoexperts.com
matthieu.rivain@cryptoexperts.com

Abstract. Higher-order masking is widely accepted as a sound countermeasure to protect implementations of blockciphers against side-channel attacks. The main issue while designing such a countermeasure is to deal with the nonlinear parts of the cipher *i.e.* the so-called s-boxes. The prevailing approach to tackle this issue consists in applying the Ishai-Sahai-Wagner (ISW) scheme from CRYPTO 2003 to some polynomial representation of the s-box. Several efficient constructions have been proposed that follow this approach, but higher-order masking is still considered as a costly (impractical) countermeasure. In this paper, we investigate efficient higher-order masking techniques by conducting a case study on ARM architectures (the most widespread architecture in embedded systems). We follow a bottom-up approach by first investigating the implementation of the base field multiplication at the assembly level. Then we describe optimized low-level implementations of the ISW scheme and its variant (CPRR) due to Coron *et al.* (FSE 2013). Finally we present improved state-of-the-art polynomial decomposition methods for s-boxes with custom parameters and various implementation-level optimizations. We also investigate an alternative to these methods which is based on bitslicing at the s-box level. We describe new masked bitslice implementations of the AES and PRESENT ciphers. These implementations happen to be significantly faster than (optimized) state-of-the-art polynomial methods. In particular, our bitslice AES masked at order 10 runs in 0.48 megacycles, which makes 8 milliseconds in presence of a 60 MHz clock frequency.

1 Introduction

Since their introduction in the late 1990's, side-channel attacks have been considered as a serious threat against cryptographic implementations. Among the existing protection strategies, one of the most widely used relies on applying *secret sharing* at the implementation level, which is known as (*higher-order*) *masking*. This strategy achieves provable security in the so-called *probing security model* [24] and *noisy leakage model* [32, 17], which makes it a prevailing way to get secure implementations against side-channel attacks.

Higher-Order Masking. Higher-order masking consists in sharing each internal variable x of a cryptographic computation into d random variables x_1, x_2, $\ldots$, x_d, called *the shares* and satisfying $x_1 + x_2 + \cdots + x_d = x$ for some group operation $+$, such that any set of $d-1$ shares is randomly distributed and independent of x. In this paper, we will consider the prevailing *Boolean masking* which is based on the bitwise addition of the shares. It has been formally demonstrated that in the noisy leakage model, where the attacker gets noisy information on each share, the complexity of recovering information on x grows exponentially with the number of shares [12, 32]. This number d, called *the masking order*, is hence a sound security parameter for the resistance of a masked implementation.

When dth-order masking is involved to protect a blockcipher, a so-called dth-order masking scheme must be designed to enable computation on masked data. To be sound, a dth order masking scheme must satisfy the two following properties: *(i) completeness*, at the end of the encryption/decryption, the sum of the d shares must give the expected result; *(ii) probing security*, every tuple of $d-1$ or less intermediate variables must be independent of any sensitive variable.

Most blockcipher structures are composed of one or several linear transformation(s), and a non-linear function, called the *s-box* (where the linearity is considered w.r.t. the bitwise addition). Computing a linear transformation $x \mapsto \ell(x)$ in the masking world can be done in $O(d)$ complexity by applying ℓ to each share independently. This clearly maintains the probing security and the completeness holds by linearity since we have $\ell(x_1) + \ell(x_2) + \cdots + \ell(x_d) = \ell(x)$. On the other hand, the non-linear operations (such as s-boxes) are more tricky to compute on the shares while ensuring completeness and probing security.

Masked S-boxes. In [24], Ishai, Sahai, and Wagner tackled this issue by introducing the first generic higher-order masking scheme for the multiplication over $\mathbb{F}_2$ in complexity $O(d^2)$. The here-called ISW scheme was later used by Rivain and Prouff to design an efficient masked implementation of AES [34]. Several works then followed to improve this approach and to extend it to other SPN blockciphers [10, 14, 15, 26]. The principle of these methods consists in representing an n-bit s-box as a polynomial $\sum_i a_i\, x^i$ in $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$, whose evaluation is then expressed as a sequence of linear functions (*e.g.* squaring, additions, multiplications by constant coefficients) and *nonlinear multiplications* over $\mathbb{F}_{2^n}$. The former are simply masked in complexity $O(d)$ (thanks to their linearity), whereas the latter are secured using ISW in complexity $O(d^2)$. The total complexity is hence mainly impacted by the number of nonlinear multiplications involved in the underlying polynomial evaluation. This observation led to a series of publications aiming at conceiving polynomial evaluation methods with the least possible nonlinear multiplications [10, 35, 15]. The so-called CRV method, due to Coron, Roy and Vivek [15], is currently the best known generic method with respect to this criteria.

Recently, an alternative to previous ISW-based polynomial methods was proposed by Carlet, Prouff, Rivain and Roche in [11]. They introduce a so-called *algebraic decomposition method* that can express an s-box in terms of polynomials of low algebraic degree. They also show that a variant of ISW due to Coron, Prouff, Rivain and Roche [14] can efficiently be used to secure the computation of any quadratic function. By combining the here-called CPRR scheme together with their algebraic decomposition method, Carlet *et al.* obtain an efficient alternative to existing ISW-based masking schemes. In particular, their technique is argued to beat the CRV method based on the assumption that an efficiency gap exists between an ISW multiplication and a CPRR evaluation. However, no optimized implementation is provided to back up this assumption.

Despite these advances, higher-order masking still implies strong performance overheads on protected implementations, and it is often believed to be impractical beyond small orders. On the other hand, most published works on the subject focus on theoretical aspects without investigating optimized low-level implementations. This raises the following question: *how fast can higher-order masking be in software?*

Our Contribution. In this paper, we investigate this question and present a case study on ARM (v7) architectures, which are today the most widespread in embedded systems (privileged targets of side-channel attacks). We provide an extensive and fair comparison between the different methods of the state of the art and a benchmarking on optimized implementations of higher-order masked blockciphers. For such purpose, we follow a bottom-up approach and start by investigating the efficient implementation of the base-field multiplication, which is the core elementary operation of the ISW-based masking schemes. We propose several implementations strategies leading to different time-memory trade-offs. We then investigate the two main building blocks of existing masking schemes, namely the ISW and CPRR schemes. We optimize the implementation of these schemes and we describe parallelized versions that achieve significant gains in performances. From these results, we propose fine-tuned variants of the CRV and algebraic decomposition methods, which allows us to compare them in a practical and optimized implementation context. We also investigate efficient polynomial methods for the specific s-boxes of two important blockciphers, namely AES and PRESENT.

As an additional contribution, we put forward an alternative strategy to polynomial methods which consists in applying bitslicing at the s-box level. More precisely, the s-box computations within a blockcipher round are bitsliced so that the core nonlinear operation is not a field multiplication anymore (nor a quadratic polynomial) but a bitwise logical AND between two m-bit registers (where m is the number of s-box computations). This allows us to translate compact hardware implementations of the AES and PRESENT s-boxes into efficient masked implementations in software. This approach has been previously used to design blockciphers well suited for masking [21] but, to the best of our knowledge, has never been used to derive efficient higher-order masked implementations of existing standard blockciphers such as AES or PRESENT. We further provide implementation results for full blockciphers and discuss the security aspects of our implementations.

Our results clearly demonstrate the superiority of the bitslicing approach (at least on 32-bit ARM architectures). Our masked bitslice implementations of AES and PRESENT are significantly faster than state-of-the-art polynomial methods with fine-tuned low-level implementations. In particular, an encryption masked at the order 10 only takes a few milliseconds with a 60 MHz clock frequency (specifically 8ms for AES and 5ms for PRESENT).

Other Related Works. Our work focuses on the optimized implementation of *polynomial methods* for efficient higher-order masking of s-boxes and blockciphers, as well as on the bitslice alternative. All these schemes are based on Boolean masking with the ISW construction (or the CPRR variant) for the core non-linear operation (which is either the field multiplication or the bitwise logical AND). Further masking techniques exist with additional features that should be adverted here.

Genelle, Prouff and Quisquater suggest mixing Boolean masking and *multiplicative masking* [19]. This approach is especially effective for blockciphers with inversion-based s-boxes such as AES. Prouff and Roche turn classic constructions from multi-party computation into a higher-order masking scheme resilient to glitches [33]. A software implementation study comparing these two schemes and classical polynomial methods for AES has been published in [23]. Compared to this previous work, our approach is to go deeper in the optimization (at the assembly level) and we further investigate generic methods (*i.e.* methods that apply to any s-box and not only to AES). Another worth-mentioning line of works is the field of *threshold implementations* [29, 30] in which the principle of threshold cryptography is applied to get secure hardware masking in the presence of glitches (see for instance [31, 28, 6]). Most of threshold implementations target first-order security but recent works discuss the extension to higher orders [5]. It should be noted that in the context of hardware implementations, the occurrence of glitches prevents the straight use of classic ISW-based Boolean masking schemes (as considered in the present work). Threshold implementations and the Prouff-Roche scheme are therefore the main solutions for (higher-order) masking in hardware. On the other hand, these schemes are not competitive for the software context (due to limited masking orders and/or to an increased complexity) and they are consequently out of the scope of our study.

Finally, we would like to mention that subsequently to the first version of this work, and motivated by the high performances of our bitslice implementations of AES and PRESENT, we have extended the bitslice higher-order masking approach to any s-box by proposing a generic decomposition method in [20]. New blockcipher designs with efficient masked bitslice implementation have also been recently proposed in [25].

Paper Organization. The next section provides some preliminaries about ARM architectures (Section 2). We then investigate the base field multiplication (Section 3) and the ISW and CPRR schemes (Section 4). Afterward, we study polynomial methods for s-boxes (Section 5) and we introduce our masked bitslice implementations of the AES and PRESENT s-boxes (Section 6). Eventually, we describe our implementations of the full ciphers (Section 7). The security aspects of our implementations are further discussed in the full version of the paper.

Source Code and Performances. For the sake of illustration, the performances of our implementations are mostly displaid on graphics in the present version. Exact performance figures (in terms of clock cycles, code size and RNG consumption) are provided in the full version of the paper (available on IACR ePrint). The source code of our implementations is also available on GitHub.

2 Preliminaries on ARM Architectures

Most ARM cores are RISC processors composed of sixteen 32-bit registers, labeled `R0`, `R1`, ..., `R15`. Registers `R0` to `R12` are known as *variable registers* and are available for computation.[3] 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 link register `R14` can also be used as additional variable register by saving the return address on the stack (at the cost of push/pop instructions). The gain of having a bigger register pool must be balanced with the saving overhead, but this trick enables some improvements in many cases.

In ARM v7, most of the instructions can be split into the following three classes: *data instructions*, *memory instructions*, and *branching instructions*. The data instructions are the arithmetic and bitwise operations, each taking one clock cycle (except for the multiplication which takes two clock cycles). The memory instructions are the load and store (from and to the RAM) which require 3 clock cycles, or their variants for multiple loads or stores ($n + 2$ clock cycles). The last class of instructions is the class of branching instructions used for loops, conditional statements and function calls. These instructions take 3 or 4 clock cycles.

One important specificity 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. Four kinds of shifting are supported: the logical shift left (`LSL`), the logical shift right (`LSR`), the arithmetic shift right (`ASR`), and the rotate-right (`ROR`). All these shifting operations are parameterized by a shift length in $[\![1, 32]\!]$ (except for the logical shift left `LSL` which lies in $[\![0, 31]\!]$). The latter can also be relative by using a register but in that case the instruction takes an additional clock cycle.

[3] Note that some conventions exist for the first four registers `R0`–`R3`, also called *argument registers*, and serving to store the arguments and the result of a function at call and return respectively.

Eventually, we assume that our target architecture includes a fast True Random Number Generator (TRNG), that frequently fills a register with a fresh 32-bit random strings (*e.g.* every 10 clock cycles). The TRNG register can then be read at the cost of a single load instruction.[4]

3 Base Field Multiplication

In this section, we focus on the efficient implementation of the multiplication over $\mathbb{F}_{2^n}$ where n is small (typically $n \in [\![4, 10]\!]$). The fastest method consists in using a precomputed table mapping the 2^{2n} possible pairs of operands (a, b) to the output product $a \cdot b$.

In the context of embedded systems, one is usually constrained on the code size and spending several kilobytes for (one table in) a cryptographic library might be prohibitive. That is why we investigate hereafter several alternative solutions with different time-memory trade-offs. Specifically, we look at the classical binary algorithm and exp-log multiplication methods. We also describe a tabulated version of Karatsuba multiplication, and another table-based method: the *half-table multiplication.* The obtained implementations are compared in terms of clock cycles, register usage, and code size (where the latter is mainly impacted by precomputed tables).

In the rest of this section, the two multiplication operands in $\mathbb{F}_{2^n}$ will be denoted a and b. These elements can be seen as polynomials $a(x) = \sum_{i=0}^{n-1} a_i x^i$ and $b(x) = \sum_{i=0}^{n-1} b_i x^i$ over $\mathbb{F}_2[x]/p(x)$ where the a_i's and the b_i's are binary coefficients and where p is a degree-n irreducible polynomial over $\mathbb{F}_2[x]$. In our implementations, these polynomials are simply represented as n-bit strings $a = (a_{n-1}, \ldots, a_0)_2$ or equivalently $a = \sum_{i=0}^{n-1} a_i \, 2^i$ (and similarly for b).

3.1 Binary Multiplication

The binary multiplication algorithm is the most basic way to perform a multiplication on a binary field. It consists in evaluating the following formula:

$$a(x) \cdot b(x) = \big(\cdots \big(\big(b_{n-1} a(x) x + b_{n-2} a(x)\big) x + b_{n-3} a(x)\big) \cdots \big) x + b_0 a(x) \ , \quad (1)$$

by iterating over the bits of b. A formal description is given in Algorithm 1.

[4] This is provided that the TRNG address is already in a register. Otherwise one must first load the TRNG address, before reading the random value. Our code ensures a gap of at least 10 clock cycles between two readings of the TRNG.

Algorithm 1 Binary multiplication algorithm

```
Input: $a(x), b(x) \in \mathbb{F}_2[x]/p(x)$
Output: $a(x) \cdot b(x) \in \mathbb{F}_2[x]/p(x)$
1. $r(x) \leftarrow 0$
2. for $i = n - 1$ down to $0$ do
3.     $r(x) \leftarrow x \cdot r(x) \bmod p(x)$
4.     if $b_i = 1$ then $r(x) \leftarrow r(x) + a(x)$
5. end for
6. return $r(x) \bmod p(x)$
```

The reduction modulo $p(x)$ can be done either inside the loop (at Step 3 in each iteration) or at the end of the loop (at Step 6). If the reduction is done inside the loop, the degree of $x \cdot r(x)$ is at most n in each iteration. So we have

$$x \cdot r(x) \bmod p(x) = \begin{cases} x \cdot r(x) - p(x) & \text{if } r_{n-1} = 1 \\ x \cdot r(x) & \text{otherwise} \end{cases} \tag{2}$$

The reduction then consists in subtracting $p(x)$ to $x \cdot r(x)$ if and only if $r_{n-1} = 1$ and doing nothing otherwise. In practice, the multiplication by x simply consists in left-shifting the bits of r and the subtraction of p is a simple XOR. The tricky part is to conditionally perform the latter XOR with respect to the bit r_{n-1} as we aim to a branch-free code. This is achieved using the *arithmetic right shift*[5] instruction (sometimes called signed shift) to compute $(r \ll 1) \oplus (r_{n-1} \times p)$ by putting r_{n-1} at the sign bit position, which can be done in 3 ARM instructions (3 clock cycles) as follows:

```
LSL $tmp, $res, #(32-n)        ;; tmp = r_{n-1}
AND $tmp, $mod, $tmp, ASR #32  ;; tmp = p & (tmp ASR 32)
EOR $res, $tmp, $res, LSL #1   ;; r = (r_{n-1} * p)^(r << 1)
```

Step 4 consists in conditionally adding a to r whenever b_i equals 1. Namely, we have to compute $r \oplus (b_i \times a)$. In order to multiply a by b_i, we use the rotation instruction to put b_i in the sign bit and the arithmetic shift instruction to fill a register with b_i. The latter register is then used to mask a with a bitwise AND instruction. The overall Step 4 is performed in 3 ARM instructions (3 clock cycles) as follows:

```
ROR $opB, #31                  ;; b_i = sign(opB)
AND $tmp, $opA, #opB, ASR #32  ;; tmp = a & (tmp ASR 32)
EOR $res, $tmp                 ;; r = r^(a * b_i)
```

[5] This instruction performs a logical right-shift but instead of filling the vacant bits with 0, it fills these bits with the leftmost bit operand (*i.e.* the sign bit).

Variant. If the reduction is done at the end of the loop, Step 3 then becomes a simple left shift, which can be done together with Step 4 in 3 instructions (3 clock cycles) as follows:

```
ROR $opB , #31                   ;; b_i = sign(opB)
AND $tmp , $opA , $opB , ASR #32  ;; tmp = a & (tmp ASR 32)
EOR $res , $tmp , $res , LSL #1   ;; r = (a * b_i)^(r << 1)
```

The reduction must then be done at the end of the loop (Step 6), where we have $r(x) = a(x) \cdot b(x)$ which can be of degree up to $2n-2$. Let r_h and r_ℓ be the polynomials of degree at most $n-2$ and $n-1$ such that $r(x) = r_h(x) \cdot x^n + r_\ell(x)$. Since we have $r(x) \bmod p(x) = (r_h(x) \cdot x^n \bmod p(x)) + r_\ell(x)$, we only need to reduce the high-degree part $r_h(x) \cdot x^n$. This can be done by tabulating the function mapping the $n-1$ coefficients of $r_h(x)$ to the $n-2$ coefficients of $r_h(x) \cdot x^n \bmod p(x)$. The overall final reduction then simply consists in computing $T[r \gg n] \oplus (r \wedge (2^n - 1))$, where T is the corresponding precomputed table.

3.2 Exp-Log Multiplication

Let $g \in \mathbb{F}_{2^n}$ be a generator of the multiplicative group $\mathbb{F}_{2^n}^*$. We shall denote by $\exp_g$ the exponential function defined over $[\![0, 2^n - 1]\!]$ as $\exp_g(\ell) = g^\ell$, and by $\log_g$ the discrete logarithm function defined over $\mathbb{F}_{2^n}^*$ as $\log_g = \exp_g^{-1}$. Assume that these functions can be tabulated (which is usually the case for small values of n). The multiplication between field elements a and b can then be efficiently computed as

$$a \cdot b = \begin{cases} \exp_g(\log_g(a) + \log_g(b) \bmod 2^n - 1) & \text{if } a \neq 0 \text{ and } b \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Le us denote $t = \log_g(a) + \log_g(b)$. We have $t \in [\![0, 2^{n+1} - 2]\!]$ giving

$$t \bmod 2^n - 1 = \begin{cases} t - 2^n + 1 & \text{if } t_n = 1 \\ t & \text{otherwise} \end{cases} \tag{4}$$

where t_n is the most significant bit in the binary expansion $t = \sum_{i=0}^{n} t_i\, 2^i$, which can be rewritten as $t \bmod 2^n - 1 = (t + t_n) \wedge (2^n - 1)$. This equation can be evaluated with 2 ARM instructions[6] (2 clock cycles) as follows:

```
ADD $tmp , $tmp , LSR #n        ;;tmp = tmp + tmp>>n
AND $tmp , #(2^n-1)             ;;tmp = tmp & (2^n-1)
```

[6] Note that for $n > 8$, the constant $2^n - 1$ does not lie in the range of constants enabled by ARM (*i.e.* rotated 8-bit values). In that case, one can use the BIC instruction to perform a logical AND where the second argument is complemented. The constant to be used is then 2^n which well belongs to ARM constants whatever the value of n.

Variant. Here again, a time-memory trade-off is possible: the $\exp_g$ table can be doubled in order to handle a $(n+1)$-bit input and to perform the reduction. This simply amounts to consider that $\exp_g$ is defined over $[\![0, 2^{n+1}-2]\!]$ rather than over $[\![0, 2^n - 1]\!]$.

Zero-testing. The most tricky part of the exp-log multiplication is to manage the case where a or b equals 0 while avoiding any conditional branch. Once again we can use the arithmetic right-shift instruction to propagate the sign bit and use it as a mask. The test of zero can then be done with 4 ARM instructions (4 clock cycles) as follows:

```
RSB $tmp, $opA, #0               ;; tmp = 0 - a
AND $tmp, $opB, $tmp, ASR #32    ;; tmp = b & (tmp ASR 32)
RSB $tmp, #0                     ;; tmp = 0 - tmp
AND $res, $tmp, ASR #32          ;; r = r & (tmp ASR 32)
```

3.3 Karatsuba Multiplication

The Karatsuba method is based on the following equation:

$$a \cdot b = (a_h + a_\ell)(b_h + b_\ell)\, x^{\frac{n}{2}} + a_h\, b_h\, (x^n + x^{\frac{n}{2}}) + a_\ell\, b_\ell\, (x^{\frac{n}{2}} + 1) \bmod p(x) \qquad (5)$$

where a_h, a_ℓ, b_h, b_ℓ are the $\frac{n}{2}$-degree polynomials such that $a(x) = a_h\, x^{\frac{n}{2}} + a_\ell$ and $b(x) = b_h\, x^{\frac{n}{2}} + b_\ell$. The above equation can be efficiently evaluated by tabulating the following functions:

$$\begin{aligned} (a_h + a_\ell, b_h + b_\ell) &\mapsto (a_h + a_\ell)(b_h + b_\ell)\, x^{\frac{n}{2}} \bmod p(x) \ , \\ (a_h, b_h) &\mapsto a_h\, b_h\, (x^n + x^{\frac{n}{2}}) \bmod p(x) \ , \\ (a_\ell, b_\ell) &\mapsto a_\ell\, b_\ell\, (x^{\frac{n}{2}} + 1) \bmod p(x) \ . \end{aligned}$$

We hence obtain a way to compute the multiplication with 3 look-ups and a few XORs based on 3 tables of 2^n elements.

In practice, the most tricky part is to get the three pairs $(a_h || b_h)$, $(a_\ell || b_\ell)$ and $(a_h + a_\ell || b_h + b_\ell)$ to index the table with the least instructions possible. The last pair is a simple addition of the two first ones. The computation of the two first pairs from the operands $a \equiv (a_h || a_\ell)$ and $b \equiv (b_h || b_\ell)$ can then be seen as the transposition of a 2×2 matrix. This can be done with 4 ARM instructions (4 clock cycles) as follows:

```
EOR $tmp0, $opA, $opB, LSR #(n/2)   ;; tmp0 = [a_h|a_l^b_h]
EOR $tmp1, $opB, $tmp0, LSL #(n/2)  ;; tmp1 = [a_h|a_l|b_l]
BIC $tmp1, #(2^n*(2^(n/2)-1))       ;; tmp1 = [a_l|b_l]
EOR $tmp0, $tmp1, LSR #(n/2)        ;; tmp0 = [a_h|b_h]
```

3.4 Half-Table Multiplication

The half-table multiplication can be seen as a trade-off between the Karatsuba method and the full-table method. While Karatsuba involves 3 look-ups in three 2^n-sized tables and the full-table method involves 1 look-up in a 2^{2n}-sized table, the half-table method involves 2 look-ups in two $2^{\frac{3n}{2}}$-sized tables. It is based on the following equation:

$$a \cdot b = b_h \, x^{\frac{n}{2}} \, (a_h \, x^{\frac{n}{2}} + a_\ell) + b_\ell \, (a_h \, x^{\frac{n}{2}} + a_\ell) \bmod p(x) \; , \tag{6}$$

which can be efficiently evaluated by tabulating the functions:

$$\begin{aligned} (a_h, a_\ell, b_h) &\mapsto b_h \, x^{\frac{n}{2}} \, (a_h \, x^{\frac{n}{2}} + a_\ell) \bmod p(x) \; , \\ (a_h, a_\ell, b_\ell) &\mapsto b_\ell \, (a_h \, x^{\frac{n}{2}} + a_\ell) \bmod p(x) \; . \end{aligned}$$

Once again, the barrel shifter is useful to get the input triplets efficiently. Each look-up can be done with two ARM instructions (for a total of 8 clock cycles) as follows:

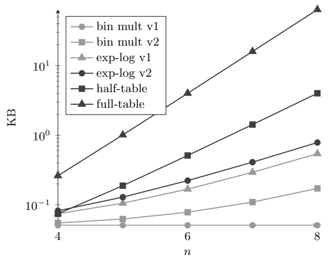
```
EOR  $tmp,$opB,$opA,LSL#n              ;;tmp=[a_h|a_l|b_h|b_l]
LDRB $res,[$tab1,$tmp,LSR#(n/2)        ;;res=T1[a_h|a_l|b_h]
EOR  $tmp,$opA,$opB,LSL#(32-n/2)       ;;tmp=[b_l|0..|a_h|a_l]
LDRB $tmp,[$tab2,$tmp,ROR#(32-n/2)]    ;;tmp=T2[a_h|a_l|b_l]
```

3.5 Performances

The obtained performances are summarized in Table 1 in terms of clock cycles, register usage, and code size. For clock cycles, the number in brackets indicates instructions that need to be done only once when multiple calls to the multiplication are performed (as in the secure multiplication procedure described in the next section). These are initialization instructions such as loading a table address in a register. For $n > 8$, elements take two bytes to be stored (assuming $n \leq 16$) which implies an overhead in clock cycles and a doubling of the table size. For most methods, the clock cycles and register usage are constant w.r.t. $n \geq 8$, whereas the code size depends on n. For the sake of illustration, we therefore additionally display the code size (and corresponding LUT sizes) in Figure 1 for several values of n.

Table 1. Multiplication performances.

	bin mult v1	bin mult v2	exp-log v1	exp-log v2	kara.	half-tab	full-tab
clock cycles ($n \leq 8$)	$10n + 3$ (+3)	$7n + 3$ (+3)	18 (+2)	16 (+2)	19 (+2)	10 (+3)	4 (+3)
clock cycles ($n > 8$)	$10n + 4$ (+3)	$7n + 15$ (+3)	35 (+2)	31 (+2)	38 (+2)	n/a	n/a
registers	5	5	5 (+1)	5 (+1)	6 (+1)	5 (+1)	5
code size ($n \leq 8$)	52	$2^{n-1} + 48$	$2^{n+1} + 48$	$3 \cdot 2^n + 40$	$3 \cdot 2^n + 42$	$2^{\frac{3n}{2}+1} + 24$	$2^{2n} + 12$



n	4	6	8	10
Binary v1	0	0	0	0
Binary v2	8 B	32 B	128 B	1 KB
Exp-log v1	32 B	128 B	0.5 KB	4 KB
Exp-log v2	48 B	192 B	0.75 KB	6 KB
Karatsuba	48 B	192 B	0.75 KB	6 KB
Half-table	0.13 KB	1 KB	8 KB	128 KB
Full-table	0.25 KB	4 KB	64 KB	2048 KB

Fig. 1. Full code size (left graph) and LUT size (right table) w.r.t. n.

We observe that all the methods provide different time-memory trade-offs except for Karatsuba which is beaten by the exp-log method (v1) both in terms of clock cycles and code size. The latter method shall then always be preferred to the former (at least on our architecture). As expected, the full-table method is by far the fastest way to compute a field multiplication, followed by the half-table method. However, depending on the value of n, these methods might be too consuming in terms of code size due to their large precomputed tables. On the other hand, the binary multiplication (even the improved version) has very poor performances in terms of clock cycles and it should only be used for extreme cases where the code size is very constrained. We consider that the exp-log method v2 (*i.e.* with doubled exp-table) is a good compromise between code size an speed whenever the full-table and half-table methods are not affordable (which might be the case for *e.g.* $n \geq 8$). In the following, we shall therefore focus our study on secure implementations using the exp-log (v2), half-table or full-table method for the base field multiplication.

4 Secure Multiplications and Quadratic Evaluations

We have seen several approaches to efficiently implement the base-field multiplication. We now investigate the secure multiplication in the masking world where the two operands $a, b \in \mathbb{F}_{2^n}$ are represented as random d-sharings $(a_1, a_2, \ldots, a_d)$ and $(b_1, b_2, \ldots, b_d)$. We also address the secure evaluation of a function f of algebraic degree 2 over $\mathbb{F}_{2^n}$ (called *quadratic function* in the following). Specifically, we focus on the scheme proposed by Ishai, Sahai, and Wagner (ISW scheme) for the secure multiplication [24], and its extension by Coron, Prouff, Rivain and Roche (CPRR scheme) to secure any quadratic function [14, 11].

4.1 Algorithms

ISW multiplication. From two d-sharings $(a_1, a_2, \ldots, a_d)$ and $(b_1, b_2, \ldots, b_d)$, the ISW scheme computes an output d-sharing $(c_1, c_2, \ldots, c_d)$ as follows:

1. for every $1 \leq i < j \leq d$, sample a random value $r_{i,j}$ over $\mathbb{F}_{2^n}$;
2. for every $1 \leq i < j \leq d$, compute $r_{j,i} = (r_{i,j} + a_i \cdot b_j) + a_j \cdot b_i$;

3. for every $1 \leq i \leq d$, compute $c_i = a_i \cdot b_i + \sum_{j \neq i} r_{i,j}$.

One can check that the output $(c_1, c_2, \ldots, c_d)$ is well a d-sharing of the product $c = a \cdot b$. We indeed have $\sum_i c_i = \sum_{i,j} a_i \cdot b_j = (\sum_i a_i)(\sum_j b_j)$ since every random value $r_{i,j}$ appears exactly twice in the sum and hence vanishes.

Mask refreshing. The ISW multiplication was originally proved probing secure at the order $t = \lfloor (d-1)/2 \rfloor$ (and not $d-1$ as one would expect with masking order d). The security proof was later made tight under the condition that the input d-sharings are based on independent randomness [34]. In some situations, this independence property is not satisfied. For instance, one might have to multiply two values a and b where $a = \ell(b)$ for some linear operation ℓ. In that case, the shares of a are usually derived as $a_i = \ell(b_i)$, which clearly breaches the required independence of input shares. To deal with this issue, one must refresh the sharing of a. However, one must be careful doing so since a bad refreshing procedure might introduce a flaw [14]. A sound method for mask-refreshing consists in applying an ISW multiplication between the sharing of a and the tuple $(1, 0, 0, \ldots, 0)$ [17, 2]. This gives the following procedure:

1. for every $1 \leq i < j \leq d$, randomly sample $r_{i,j}$ over $\mathbb{F}_{2^n}$ and set $r_{j,i} = r_{i,j}$;
2. for every $1 \leq i \leq d$, compute $a'_i = a_i + \sum_{j \neq i} r_{i,j}$.

It is not hard to see that the output sharing $(a'_1, a'_2, \ldots, a'_d)$ well encodes a. One might think that such a refreshing implies a strong overhead in performances (almost as performing two multiplications) but this is still better than doubling the number of shares (which roughly quadruples the multiplication time). Moreover, we show hereafter that the implementation of such a refreshing procedure can be very efficient in practice compared to the ISW multiplication.

CPRR evaluation. The CPRR scheme was initially proposed in [14] as a variant of ISW to securely compute multiplications of the form $x \mapsto x \cdot \ell(x)$ where ℓ is linear, without requiring refreshing. It was then shown in [11] that this scheme (in a slightly modified version) could actually be used to securely evaluate any quadratic function f over $\mathbb{F}_{2^n}$. The method is based on the following equation

$$f(x_1 + x_2 + \cdots + x_d) = \sum_{1 \leq i < j \leq d} f(x_i + x_j + s_{i,j}) + f(x_j + s_{i,j}) + f(x_i + s_{i,j}) + f(s_{i,j}) \\ + \sum_{i=1}^{d} f(x_i) + (d + 1 \bmod 2) \cdot f(0) \quad (7)$$

which holds for every $(x_i)_i \in (\mathbb{F}_{2^n})^d$, every $(s_{i,j})_{1 \leq i < j \leq d} \in (\mathbb{F}_{2^n})^{d(d-1)/2}$, and every quadratic function f over $\mathbb{F}_{2^n}$.

From a d-sharing $(x_1, x_2, \ldots, x_d)$, the CPRR scheme computes an output d-sharing $(y_1, y_2, \ldots, y_d)$ as follows:

1. for every $1 \leq i < j \leq d$, sample two random values $r_{i,j}$ and $s_{i,j}$ over $\mathbb{F}_{2^n}$,
2. for every $1 \leq i < j \leq d$, compute $r_{j,i} = r_{i,j} + f(x_i + s_{i,j}) + f(x_j + s_{i,j}) + f((x_i + s_{i,j}) + x_j) + f(s_{i,j})$,
3. for every $1 \leq i \leq d$, compute $y_i = f(x_i) + \sum_{j \neq i} r_{i,j}$,
4. if d is even, set $y_1 = y_1 + f(0)$.

According to (7), we then have $\sum_{i=1}^{d} y_i = f\big(\sum_{i=1}^{d} x_i\big)$, which shows that the output sharing $(y_1, y_2, \ldots, y_d)$ well encodes $y = f(x)$.

In [14, 11] it is argued that in the gap where the field multiplication cannot be fully tabulated (2^{2n} elements is too much) while a function $f : \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ can be tabulated (2^n elements fit), the CPRR scheme is (likely to be) more efficient than the ISW scheme. This is because it essentially replaces (costly) field multiplications by simple look-ups. We present in the next section the results of our study for our optimized ARM implementations.

4.2 Implementations and Performances

For both schemes we use the approach suggested in [13] that directly accumulates each intermediate result $r_{i,j}$ in the output share c_i so that the memory cost is $O(d)$ instead of $O(d^2)$ when the $r_{i,j}$'s are stored. Detailed algorithms can be found in the appendix. The ARM implementation of these algorithms is rather straightforward and it does not make use of any particular trick.

As argued in Section 3.5, we consider three variants for the base field multiplication in the ISW scheme, namely the full-table method, the half-table method and the exp-log method (with doubled exp table). The obtained ISW variants are labeled ISW-FT, ISW-HT and ISW-EL in the following. The obtained performances are illustrated in Figure 2 with respect to d. Note that we did not consider ISW-FT for $n > 8$ since the precomputed tables are too huge.

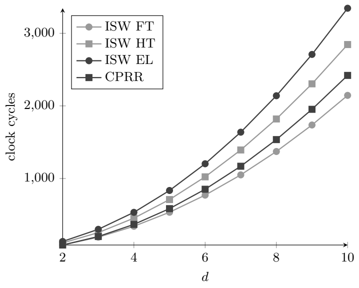


Fig. 2. Timings of ISW and CPRR schemes.

These results show that CPRR indeed outperforms ISW whenever the field multiplication cannot be fully tabulated. Even the half-table method (which is more consuming in code-size) is slower than CPRR. For $n \leq 8$, a CPRR evaluation asymptotically costs 1.16 ISW-FT, 0.88 ISW-HT, and 0.75 ISW-EL.

4.3 Parallelization

Both ISW and CPRR schemes work on n-bit variables, each of them occupying a full 32-bit register. Since in most practical scenarios, we have $n \in [\![4, 8]\!]$, this situation is clearly suboptimal in terms of register usage, and presumably suboptimal in terms of timings. A natural idea to improve this situation is to use parallelization. A register can simultaneously store $m := \lfloor 32/n \rfloor$ values, we can hence try to perform m ISW/CPRR computations in parallel (which would in turn enable to perform m s-box computations in parallel). Specifically, each input shares is replaced by m input shares packed into a 32-bit value. The ISW (resp. CPRR) algorithm load packed values, and perform the computation on each unpacked n-bit chunk one-by-one. Using such a strategy allows us to save multiple load and store instructions, which are among the most expensive instructions of ARM assembly (3 clock cycles). Specifically, we can replace m load instructions by a single one for the shares a_i, b_j in ISW (resp. x_i, x_j in CPRR) and the random values $r_{i,j}$, $s_{i,j}$ (read from the TRNG), we can replace m store instructions by a single one for the output shares, and we can replace m XOR instructions by a single one for some of the addition involved in ISW (resp. CPRR). On the other hand, we get an overhead for the extraction of the n-bit chunks from the packed 32-bit values. But each of these extractions takes a single clock cycle (thanks to the barrel shifter), which is rather small compared to the gain in load and store instructions.

We implemented parallel versions of ISW and CPRR for $n = 4$ and $n = 8$. For the former case, we can perform $m = 8$ evaluations in parallel, whereas for the later case we can perform $m = 4$ evaluations in parallel. For $n = 4$, we only implemented the full-table multiplication for ISW, since we consider that a 256-byte table in code is always affordable. For $n = 8$ on the other hand, we did not implement the full-table, since we consider that a 64-KB table in code would be to much in most practical scenarios. Figures 3 and 4 give the obtained performances in terms of clock cycles.

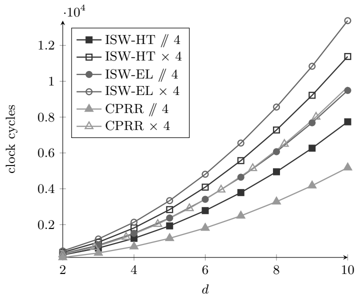


Fig. 3. Timings of (parallel) ISW and CPRR schemes for $n = 8$.

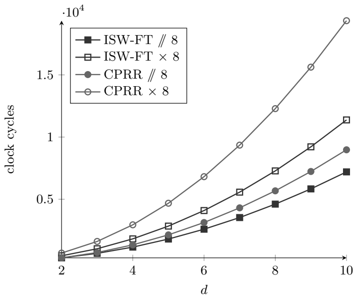


Fig. 4. Timings of (parallel) ISW and CPRR schemes for $n = 4$.

These results show the important gain obtained by using parallelism. For ISW, we get an asymptotic gain around 30% for 4 parallel evaluations ($n = 8$) compared to 4 serial evaluations, and we get a 58% asymptotic gain for 8 parallel evaluations ($n = 4$) compared to 8 serial evaluations. For CPRR, the gain is around 50% (timings are divided by 2) in both cases ($n = 8$ and $n = 4$). We also observe that the efficiency order keeps unchanged with parallelism, that is: ISW-FT > CPRR > ISW-HT > ISW-EL.

Remark 1. Note that using parallelization in our implementations does not compromise the probing security. Indeed, we pack several bytes/nibbles within one word of the cipher state but we never pack (part of) different shares of the same variable together. The probing security proofs hence apply similarly to the parallel implementations.[7]

4.4 Mask-Refreshing Implementation

The ISW-based mask refreshing is pretty similar to an ISW multiplication, but it is actually much faster since it involves no field multiplications and fewer additions (most terms being multiplied by 0). It simply consists in processing:

$$\text{for } i = 1 \mathinner{..} d : \text{for } j = i+1 \mathinner{..} d : r \leftarrow \$;\ a_i \leftarrow a_i + r;\ a_j \leftarrow a_j + r;$$

A straightforward implementation of this process is almost 3 times faster than the fastest ISW multiplication, namely the full-table one (see Figure 5).

We can actually do much better. Compared to a standard ISW implementation, the registers of the field multiplication are all available and can hence be used in order to save several loads and stores. Indeed, the straightforward implementation performs $d - i + 1$ loads and stores for every $i \in [\![1, d]\!]$, specifically 1 load-store for a_i and $d - i$ for the a_j's. Since we have some registers left, we can actually pool the a_j's loads and stores for several a_i's. To do so, we load several shares a_i, a_{i+1}, ..., a_{i+k} with the `LDM` instruction (which has a cost of $k + 2$ instead of $3k$) and process the refreshing between them. Then, for every $j \in [\![i + k + 1, d]\!]$, we load a_j, performs the refreshing between a_j and each of the a_i, a_{i+1}, ..., a_{i+k}, and store a_j back. Afterwards, the shares a_i, a_{i+1}, ..., a_{i+k} are stored back with the `STM` instruction (which has a cost of $k + 2$ instead of $3k$). This allows us to load (and store) the a_j only once for the k shares instead of k times, and to take advantage of the `LDM` and `STM` instructions. In practice, we could deal with up to $k = 8$ shares at the same time, meaning that for $d \leq 8$ all the shares could be loaded and stored an single time using `LDM` and `STM` instructions.

[7] Putting several shares of the same variable in a single register would induce a security flaw in the probing model where full registers can be probed. For this reason, we avoid doing so and we stress that parallelization does not result in such an undesired result. However, it should be noted that in some other relevant security models, such as the single-bit probing model or the *bounded moment leakage model* [3], this would not be an issue anyway.

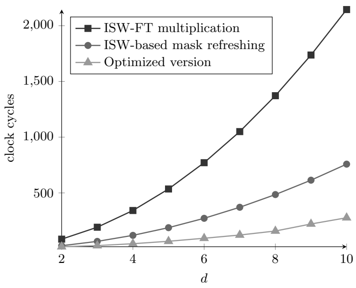


Fig. 5. Timings of mask refreshing.

The performances of our implementations of the ISW-based mask refreshing are plotted in Figure 5. Our optimized refreshing is up to 3 times faster than the straightforward implementation and roughly 10 times faster that the full-table-based ISW multiplication.

5 Polynomial Methods for S-boxes

This section addresses the efficient implementation of polynomial methods for s-boxes based on ISW and CPRR schemes. We first investigate the two best known generic methods, namely the *CRV method* [15], and the *algebraic decomposition method* [11], for which we propose some improvements. We then look at specific methods for the AES and PRESENT s-boxes, and finally provide extensive comparison of our implementation results.

5.1 CRV Method

The CRV method was proposed by Coron, Roy and Vivek in [15]. Before recalling its principle, let us introduce the notion of *cyclotomic class*. For a given integer n, the cyclotomic class of $\alpha \in [\![0, 2^n-2]\!]$ is defined as $C_\alpha = \{\alpha \cdot 2^i \bmod 2^n - 1 \; ; \; i \in \mathbb{N}\}$. We have the following properties: (i) cyclotomic classes are equivalence classes partitioning $[\![0, 2^n - 2]\!]$, and (ii) a cyclotomic class has at most n elements. In the following, we denote by x^L the set of monomials $\{x^\alpha \; ; \; \alpha \in L\}$ for some set $L \subseteq [\![0, 2^n - 1]\!]$.

The CRV method consists in representing an s-box $S(x)$ over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$ as

$$S(x) = \sum_{i=1}^{t-1} p_i(x) \cdot q_i(x) + p_t(x) \; , \tag{8}$$

where $p_i(x)$ and $q_i(x)$ are polynomials with monomials in x^L for some set $L = C_{\alpha_1=0} \cup C_{\alpha_2=1} \cup C_{\alpha_3} \cup \ldots \cup C_{\alpha_\ell}$ such that for every $i \geq 3$, $\alpha_i = \alpha_j + \alpha_k \bmod 2^n - 1$

for some $j, k < i$ (or more generally $\alpha_i = 2^w \cdot \alpha_j + \alpha_k \bmod 2^n - 1$ with $k \in [\![0, n-1]\!]$). Such polynomials can be written as:

$$p_i(x) = \sum_{j=2}^{\ell} l_{i,j}(x^{\alpha_j}) + c_{i,0} \quad \text{and} \quad q_i(x) = \sum_{j=2}^{\ell} l'_{i,j}(x^{\alpha_j}) + c'_{i,0} \ , \tag{9}$$

where the $l_{i,j}, l'_{i,j}$ are linearized polynomials over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$ and where the $c_{i,0}, c'_{i,0}$ are constants in $\mathbb{F}_{2^n}$.

In [15], the authors explain how to find such a representation. In a nutshell, one randomly picks the q_i's and search for p_i's satisfying (8). This amounts to solve a linear system with 2^n equations and $t \cdot |L|$ unknowns (the coefficients of the p_i's). Note that when the choice of the classes and the q_i's leads to a solvable system, then it can be used with any s-box (since the s-box is the target vector of the linear system). We then have two necessary (non sufficient) conditions for such a system to be solvable: (1) the set L of cyclotomic classes is such that $t \cdot |L| \geq 2^n$, (2) all the monomials can be reached by multiplying two monomials from x^L, that is $\{x^i \cdot x^j \bmod (x^{2^n} - x) \ ; \ i, j \in L\} = x^{[\![0, 2^n - 1]\!]}$. For the sake of efficiency, the authors of [15] impose an additional constraint for the choice of the classes: (3) every class (but $C_0 = \{0\}$) have the maximal cardinality of n. Under this additional constraint, condition (1) amounts to the following inequality: $t \cdot \big(1 + n \cdot (\ell - 1)\big) \geq 2^n$. Minimizing the number of nonlinear multiplications while satisfying this constraint leads to parameters $t \approx \sqrt{2^n/n}$ and $\ell \approx \sqrt{2^n/n}$.

Based on the above representation, the s-box can be evaluated using $(\ell - 2) + (t - 1)$ nonlinear multiplications (plus some linear operations). In a first phase, one generates the monomials corresponding to the cyclotomic classes in L. Each x^{α_i} can be obtained by multiplying two previous x^{α_j} and x^{α_k} (where x^{α_j} might be squared w times if necessary). In the masking world, each of these multiplications is performed with a call to ISW. The polynomials $p_i(x)$ and $q_i(x)$ can then be computed according to (9). In practice the linearized polynomials are tabulated so that at masked computation, applying a $l_{i,j}$ simply consists in performing a look-up on each share of the corresponding x^{α_j}. In the second phase, one simply evaluates (8), which takes $t - 1$ nonlinear multiplications plus some additions. We recall that in the masking world, linear operation such as additions or linearized polynomial evaluations can be applied on each share independently yielding a $O(d)$ complexity, whereas nonlinear multiplications are computed by calling ISW with a $O(d^2)$ complexity. The performances of the CRV method is hence dominated by the $\ell + t - 3$ calls to ISW.

Mask refreshing. As explained in Section 4.1, one must be careful while composing ISW multiplications with linear operations. In the case of the CRV method, ISW multiplications are involved on sharings of values $q_i(x)$ and $p_i(x)$ which are linearly computed from the sharings of the x^{α_j} (see (9)). This contradicts the independence requirement for the input sharings of an ISW multiplication, and this might presumably induce a flaw as the one described in [14]. In order to avoid such a flaw in our masked implementation of CRV, we systematically refreshed one of the input sharings, namely the sharing of $q_i(x)$. As shown in Section 4.4, the overhead implied by such a refreshing is manageable.

Improving CRV with CPRR. As suggested in [11], CRV can be improved by using CPRR evaluations instead of ISW multiplications in the first phase of CRV, whenever CPRR is faster than ISW (*i.e.* when full-table multiplication cannot be afforded). Instead of multiplying two previously computed powers x^{α_j} and x^{α_k}, the new power x^{α_i} is derived by applying the quadratic function $x \mapsto x^{2^w+1}$ for some $w \in [\![1, n-1]\!]$. In the masking world, securely evaluating such a function can be done with a call to CPRR. The new chain of cyclotomic classes $C_{\alpha_1=0} \cup C_{\alpha_2=1} \cup C_{\alpha_3} \cup \ldots \cup C_{\alpha_\ell}$ must then satisfy $\alpha_i = (2^w+1)\alpha_j$ for some $j < i$ and $w \in [\![1, n-1]\!]$.

We have implemented the search of such chains of cyclotomic classes satisfying conditions (1), (2) and (3). We could validate that for every $n \in [\![4, 10]\!]$ and for the parameters (ℓ, t) given in [15], we always find such a chain leading to a solvable system. For the sake of code compactness, we also tried to minimize the number of CPRR exponents $2^w + 1$ used in these chains (since in practice each function $x \mapsto x^{2^w+1}$ is tabulated). For $n \in \{4, 6, 7\}$ a single CPRR exponent (either 3 or 5) is sufficient to get a *satisfying chain* (*i.e.* a chain of cyclotomic class fulfilling the above conditions and leading to a solvable system). For the other values of n, we could prove that a single CPRR exponent does not suffice to get a satisfying chain. We could then find satisfying chains for $n = 5$ and $n = 8$ using 2 CPRR exponents (specifically 3 and 5). For $n > 8$, we tried all the pairs and triplets of possible CPRR exponents without success, we could only find a satisfying chain using the 4 CPRR exponents 3, 5, 9 and 17.

Optimizing CRV parameters. We can still improve CRV by optimizing the parameters (ℓ, t) depending on the ratio $\theta = \frac{C_{\text{CPRR}}}{C_{\text{ISW}}}$, where C_{CPRR} and C_{ISW} denote the costs of ISW and CPRR respectively. The cost of the CRV method satisfies

$$\begin{aligned} C_{\text{CRV}} &= (\ell-2)\, C_{\text{CPRR}} + (t-1)\, C_{\text{ISW}} = \big((\ell-2)\cdot\theta + t - 1\big)\, C_{\text{ISW}} \\ &\geq \left((\ell-2)\cdot\theta + \left\lceil \frac{2^n}{(\ell-1)\cdot n + 1} \right\rceil - 1\right) C_{\text{ISW}} \end{aligned}$$

where the inequality holds from conditions (1) and (3) above. This lower bound ensures that the system contains enough unknowns to be solvable. In practice, it was observed in [15] that this is a sufficient condition most of the time to get a solvable system (and our experiments corroborate this fact). Our optimized version of CRV hence consists in using the parameter ℓ minimizing the above lower bound and the corresponding $t = \left\lceil \frac{2^n}{(\ell-1)\cdot n+1} \right\rceil$ as parameters for given bit-length n and cost ratio θ.

It can be checked (see full version) that a ratio slightly lower than 1 implies a change of optimal parameters for all values of n except 4 and 9. In other words, as soon as CPRR is slightly faster than ISW, using a higher ℓ (*i.e.* more cyclotomic classes) and therefore a lower t is a sound trade. For our implementations of ISW and CPRR (see Section 4), we obtained a ratio θ greater than 1 only when ISW is based on the full-table multiplication. In that case, no gain can be obtain from using CPRR in the first phase of CRV, and one should use the original CRV parameters. On the other hand, we obtained θ-ratios of 0.88 and 0.75 for half-table-based ISW and exp-log-based ISW respectively. For the parallel versions, these ratios become 0.69 (half-table ISW) and 0.58 (exp-log ISW). For such ratios, the optimal parameter ℓ is greater than in the original CRV method (see full version for details).

For $n \in \{6, 8, 10\}$, we checked whether we could find satisfying CPRR-based chains of cyclotomic classes, for the obtained optimal parameters. For $n = 6$, the optimal parameters are $(\ell, t) = (5, 3)$ (giving 3 CPRR plus 2 ISW) which are actually the original CRV parameters. We could find a satisfying chain for these parameters. For $n = 8$, the optimal parameters are $(\ell, t) = (9, 4)$ (giving 7 CPRR plus 3 ISW). For these parameters we could not find any satisfying chain. We therefore used the second best set of parameters that is $(\ell, t) = (8, 5)$ (giving 6 CPRR plus 4 ISW) for which we could find a satisfying chain. For $n = 10$, the optimal parameters are $(\ell, t) = (14, 8)$ (giving 12 CPRR plus 7 ISW). For these parameters we could neither find any satisfying chain. So once again, we used the second best set of parameters, that is $(\ell, t) = (13, 9)$ (giving 11 CPRR plus 8 ISW) and for which we could find a satisfying chain. All the obtained satisfying CPRR-based chains of cyclotomic classes are provided in the full version of the paper.

Table 2 compares the performances of the original CRV method and the improved versions for our implementation of ISW (half-table and exp-log variants) and CPRR.[8] For the improved methods, we give the ratio of asymptotic performances with respect to the original version. This ratio ranks between 79% and 94% for the improved version and between 75% and 93% for the improved version with optimized parameters.

Table 2. Performances of CRV original version and improved version (with and without optimized parameters).

	Original CRV ([15])			CRV with CPRR ([11])				Optimized CRV with CPPR			
	# ISW	# CPRR	clock cycles	# ISW	# CPRR	clock cycles	ratio	# ISW	# CPRR	clock cycles	ratio
$n = 6$ (HT)	5	0	$142.5\,d^2 + O(d)$	2	3	$132\,d^2 + O(d)$	93%	2	3	$132\,d^2 + O(d)$	93%
$n = 6$ (EL)	5	0	$167.5\,d^2 + O(d)$	2	3	$142\,d^2 + O(d)$	85%	2	3	$142\,d^2 + O(d)$	85%
$n = 8$ (HT)	10	0	$285\,d^2 + O(d)$	5	5	$267.5\,d^2 + O(d)$	94%	4	6	$264\,d^2 + O(d)$	93%
$n = 8$ (EL)	10	0	$335\,d^2 + O(d)$	5	5	$292.5\,d^2 + O(d)$	87%	4	6	$284\,d^2 + O(d)$	85%
$n = 10$ (EL)	19	0	$997.5\,d^2 + O(d)$	10	9	$858\,d^2 + O(d)$	86%	8	11	$827\,d^2 + O(d)$	83%
$n = 8$ (HT) $/\!\!/4$	10	0	$775\,d^2 + O(d)$	5	5	$657.5\,d^2 + O(d)$	85%	4	6	$634\,d^2 + O(d)$	82%
$n = 8$ (EL) $/\!\!/4$	10	0	$935\,d^2 + O(d)$	5	5	$737.5\,d^2 + O(d)$	79%	4	6	$698\,d^2 + O(d)$	75%

[8] We only count the calls to ISW and CPRR since other operations are similar in the three variants and have linear complexity in d.

5.2 Algebraic Decomposition Method

The algebraic decomposition method was recently proposed by Carlet, Prouff, Rivain and Roche in [11]. It consists in using a basis of polynomials $(g_1, g_2, \dots, g_r)$ that are constructed by composing polynomials f_i as follows

$$\begin{cases} g_1(x) = f_1(x) \\ g_i(x) = f_i\big(g_{i-1}(x)\big) \end{cases} \tag{10}$$

The f_i's are of given algebraic degree s. In our context, we consider the algebraic decomposition method for $s = 2$, where the f_i's are (algebraically) quadratic polynomials. The method then consists in representing an s-box $S(x)$ over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$ as

$$S(x) = \sum_{i=1}^{t} p_i\big(q_i(x)\big) + \sum_{i=1}^{r} \ell_i\big(g_i(x)\big) + \ell_0(x) \ , \tag{11}$$

with

$$q_i(x) = \sum_{j=1}^{r} \ell_{i,j}\big(g_j(x)\big) + \ell_{i,0}(x) \ , \tag{12}$$

where the p_i's are quadratic polynomials over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$, and where the ℓ_i's and the $\ell_{i,j}$'s are linearized polynomials over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$.

As explain in [11], such a representation can be obtained by randomly picking some f_i's and some $\ell_{i,j}$'s (which fixes the q_i's) and then search for p_i's and ℓ_i's satisfying (11). As for the CRV method, this amounts to solve a linear system with 2^n equations where the unknowns are the coefficients of the p_i's and the ℓ_i's. Without loss of generality, we can assume that only ℓ_0 has a constant terms. In that case, each p_i is composed of $\frac{1}{2}n(n+1)$ monomials, and each ℓ_i is composed of n monomials (plus a constant term for ℓ_0). This makes a total of $\frac{1}{2}\, n\,(n+1) \cdot t + n \cdot r + 1$ unknown coefficients. In order to get a solvable system we hence have the following condition: (1) $\frac{1}{2}\, n\,(n+1) \cdot t + n \cdot r + 1 \geq 2^n$. A second condition is (2) $2^{r+1} \geq n$, otherwise there exists some s-box with algebraic degree greater than 2^{r+1} that cannot be achieved with the above decomposition *i.e.* the obtained system is not solvable for every target S.

Based on the above representation, the s-box can be evaluated using $r + t$ evaluations of quadratic polynomials (the f_i's and the q_i's). In the masking world, this is done thanks to CPRR evaluations. The rest of the computation are additions and (tabulated) linearized polynomials which are applied to each share independently with a complexity linear in d. The cost of the algebraic decomposition method is then dominated by the $r + t$ calls to CPRR.

We implemented the search of sound algebraic decompositions for $n \in [\![4, 10]\!]$. Once again, we looked for *full rank* systems *i.e.* systems that would work with any target s-box. For each value of n, we set r to the smallest integer satisfying condition (2) *i.e.* $r \geq \log_2(n) - 1$, and then we looked for a t starting from the lower bound $t \geq \frac{2(2^n - rn - 1)}{n(n+1)}$ (obtained from condition (1)) and incrementing until a solvable system can be found. We then increment r and reiterate the process with t starting from the lower bound, and so on. For $n \leq 8$, we found the same parameters as those reported in [11]. For $n = 9$ and $n = 10$ (these cases were not considered in [11]), the best parameters we obtained were $(r, t) = (3, 14)$ and $(r, t) = (4, 22)$ respectively.

Saving linear terms. In our experiments, we realized that the linear terms $\ell_i\big(g_i(x)\big)$ could always be avoided in (11). Namely, for the best known parameters (r, t) for every $n \in [\![4, 10]\!]$, we could always find a decomposition $S(x) = \sum_{i=1}^{t} p_i\big(q_i(x)\big)$ hence saving $r + 1$ linearized polynomials. This is not surprising if we compare the number of degrees of freedom brought by the p_i's in the linear system ($i.e.$ $\frac{1}{2}\, n\, (n+1) \cdot t$) to those brought by the ℓ_i's ($i.e.$ $n \cdot r$). More details are given in the full version of the paper.

5.3 Specific Methods for AES and PRESENT

Rivain-Prouff (RP) method for AES. Many works have proposed masking schemes for the AES s-box and most of them are based on its peculiar algebraic structure. It is the composition of the *inverse function* $x \mapsto x^{254}$ over $\mathbb{F}_{2^8}$ and an affine function: $S(x) = \text{Aff}(x^{254})$. The affine function being straightforward to mask with linear complexity, the main issue is to design an efficient masking scheme for the inverse function.

In [34], Rivain and Prouff introduced the approach of using an efficient addition chain for the inverse function that can be implemented with a minimal number of ISW multiplications. They show that the exponentiation to the 254 can be performed with 4 nonlinear multiplications plus some (linear) squarings, resulting in a scheme with 4 ISW multiplications. In [14], Coron *et al.* propose a variant where two of these multiplications are replaced CPRR evaluations (of the functions $x \mapsto x^3$ and $x \mapsto x^5$).[9] This was further improved by Grosso *et al.* in [22] who proposed the following addition chain leading to 3 CPRR evaluations and one ISW multiplications: $x^{254} = (x^2 \cdot ((x^5)^5)^5)^2$. This addition chain has the advantage of requiring a single function $x \mapsto x^5$ for the CPRR evaluation (hence a single LUT for masked implementation). Moreover it can be easily checked by exhaustive search that no addition chain exists that trades the last ISW multiplication for a CPRR evaluation. We therefore chose to use the Grosso *et al.* addition chain for our implementation of the RP method.

Kim-Hong-Lim (KHL) method for AES. This method was proposed in [26] as an improvement of the RP scheme. The main idea is to use the tower

[9] The original version of the RP scheme [34] actually involved a weak mask refreshing procedure which was exploited in [14] to exhibit a flaw in the s-box processing. The CPRR variant of ISW was originally meant to patch this flaw but the authors observed that using their scheme can also improve the performances. The security of the obtained variant of the RP scheme was recently verified up to masking order 4 using program verification techniques [2].

field representation of the AES s-box [36] in order to descend from $\mathbb{F}_{2^8}$ to $\mathbb{F}_{2^4}$ where the multiplications can be fully tabulated. Let δ denote the isomorphism mapping $\mathbb{F}_{2^8}$ to $(\mathbb{F}_{2^4})^2$ with $\mathbb{F}_{2^8} \equiv \mathbb{F}_{2^4}[x]/p(x)$, and let $\gamma \in \mathbb{F}_{2^8}$ and $\lambda \in \mathbb{F}_{2^4}$ such that $p(x) = x^2 + x + \lambda$ and $p(\gamma) = 0$. The tower field method for the AES s-box works as follows:

1. $a_h\gamma + a_l = \delta(x),\ a_h, a_l \in \mathbb{F}_{2^4}$
2. $d = \lambda\, a_h^2 + a_l \cdot (a_h + a_l) \in \mathbb{F}_{2^4}$
3. $d' = d^{14} \in \mathbb{F}_{2^4}$
4. $a'_h = d'\, a_j \in \mathbb{F}_{2^4}$
5. $a'_l = d'(a_h + a_l) \in \mathbb{F}_{2^4}$
6. $S(x) = \mathrm{Aff}(\delta^{-1}(a'_h\gamma + a'_l)) \in \mathbb{F}_{2^8}$

At the third step, the exponentiation to the 14 can be performed as $d^{14} = (d^3)^4 \cdot d^2$ leading to one CPRR evaluation (for $d \mapsto d^3$) and one ISW multiplication (plus some linear squarings).[10] This gives a total of 4 ISW multiplications and one CPRR evaluation for the masked AES implementation.

***F*∘*G* method for PRESENT.** As a 4-bit s-box, the PRESENT s-box can be efficiently secured with the CRV method using only 2 (full table) ISW multiplications. The algebraic decomposition method would give a less efficient implementation with 3 CPRR evaluations. Another possible approach is to use the fact that the PRESENT s-box can be expressed as the composition of two quadratic functions $S(x) = F \circ G(x)$. This representation was put forward by Poschmann *et al.* in [31] to design an efficient *threshold implementation* of PRESENT. In our context, this representation can be used to get a masked s-box evaluation based on 2 CPRR evaluations. Note that this method is asymptotically slower than CRV with 2 full-table ISW multiplications. However, due to additional linear operations in CRV, $F \circ G$ might actually be better for small values of d.

5.4 Implementations and Performances

We have implemented the CRV method and the algebraic decomposition method for the two most representative values of $n = 4$ and $n = 8$. For $n = 4$, we used the full-table multiplication for ISW (256-byte table), and for $n = 8$ we used the half-table multiplication (8-KB table) and the exp-log multiplication (0.75-KB table). Based on our analysis of Section 5.1, we used the original CRV method for $n = 4$ (*i.e.* $(\ell, t) = (3, 2)$ with 2 ISW multiplications), and we used the improved CRV method with optimized parameters for $n = 8$ (*i.e.* $(\ell, t) = (8, 5)$ with 6 CPRR evaluations and 4 ISW multiplications). We further implemented parallel versions of these methods, which mainly consisted in replacing calls to ISW and CPRR by calls to their parallel versions (see Section 4.3), and replacing linear operations by their parallel counterparts.

We also implemented the specific methods described in Section 5.3 for the AES and PRESENT s-boxes, as well as their parallel counterparts. Specifically,

[10] The authors of [26] suggest to perform $d^3 = d^2 \cdot d$ with a full tabulated multiplication but this would actually imply a flaw as described in [14]. That is why we use a CPRR evaluation for this multiplication.

we implemented the $F \circ G$ method for PRESENT and the RP and KHL methods for AES. The RP method was implemented with both the half-table and the exp-log methods for the ISW multiplication. For the KHL method, the ISW multiplications and the CPRR evaluation are performed on 4-bit values. It was then possible to perform 8 evaluations in parallel. Specifically, we first apply the isomorphism δ on 8 s-box inputs to obtain 8 pairs (a_h, a_l). The a_h values are grouped in one register and the a_l values are then grouped in a second register. The KHL method can then be processed in a 8-parallel version relying on the parallel ISW and CPRR procedures for $n = 4$.

Our implementation results (in terms of clock cycles) are depicted in Figures 6 and 7 for $n = 4$ (with the $F \circ G$ method as a particular case), in Figures 8 and 9 for $n = 8$, and in Figures 10 and 11 for the AES s-box.

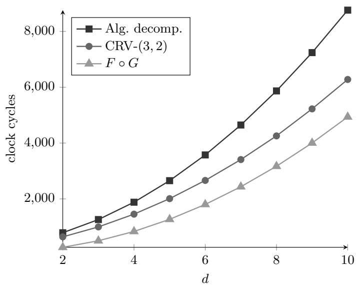


Fig. 6. Timings for one s-box ($n = 4$).

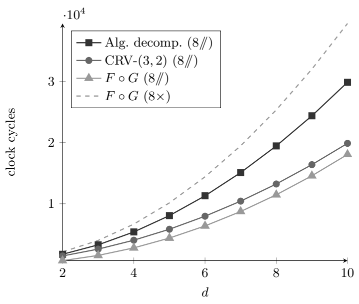


Fig. 7. Timings for 8 s-boxes ($n = 4$).

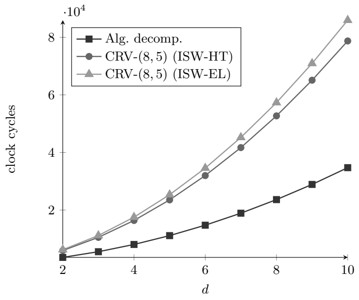


Fig. 8. Timings for one s-box ($n = 8$).

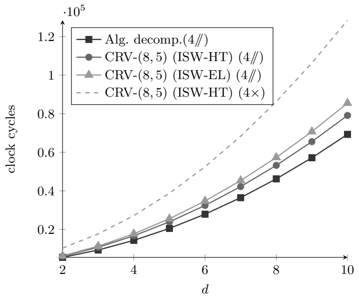


Fig. 9. Timings for 4 s-boxes ($n = 8$).

We observe that the CRV method is clearly better than the algebraic decomposition method for $n = 4$ in both the serial and parallel case. This is not surprising since the former involves 2 calls to ISW-FT against 3 calls to CPRR for the latter. For $n = 8$, CRV is only slightly better than the algebraic decomposition, which is due to the use of CPRR and optimized parameters, as explained in Section 5.1. On the other hand, the parallel implementation of the algebraic decomposition method becomes better than CRV which is due to the efficiency of the CPRR parallelization.

Regarding the specific case of PRESENT, we see that the $F \circ G$ method is actually better than CRV for $d \in [\![2, 10]\!]$ thought it is asymptotically slower. It can be checked (see full version) that CRV becomes faster only after $d \geq 38$. In parallel, $F \circ G$ is also faster than CRV until $d \geq 11$. This shows that the $F \circ G$ method offers a valuable alternative to the CRV method for PRESENT in practice. Note that many 4-bit s-boxes have a similar decomposition (see [6] for an extensive analysis), so this method could be applied to further blockciphers.

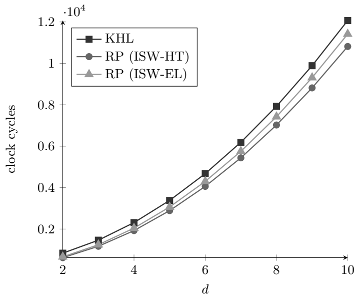


Fig. 10. Timings for one AES s-box.

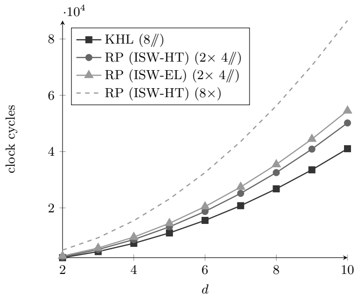


Fig. 11. Timings for 8 AES s-boxes.

For the AES, we observe that the RP method is better than KHL, which means that the gain obtained by using full-table multiplications does not compensate the overhead implied by the additional multiplication required in KHL compared to RP. We also see that the two versions of RP are very closed, which is not surprising since the difference regards a single multiplication (the other ones relying on CPRR). Using ISW-HT might not be interesting in this context given the memory overhead. For the parallel versions, KHL becomes better since it can perform 8 evaluations simultaneously, whereas RP is bounded to a parallelization degree of 4. This shows that though the field descent from $\mathbb{F}_{2^8}$ to $\mathbb{F}_{2^4}$ might be nice for full tabulation, it is mostly interesting for increasing the parallelization degree.

Eventually as a final and global observation, we clearly see that using parallelism enables significant improvements. The timings of parallel versions rank between 40% and 60% of the corresponding serial versions. In the next section, we push the parallelization one step further, namely we investigate bitslicing for higher-order masking implementations.

6 Bitslice Methods for S-boxes

In this section, we focus on the secure implementation of AES and PRESENT s-boxes using bitslice. Bitslice is an implementation strategy initially proposed by Biham in [4]. It consists in performing several parallel evaluations of a Boolean circuit in software where the logic gates can be replaced by instructions working on registers of several bits. As nicely explained in [27], "*in the bitslice implementation one software logical instruction corresponds to simultaneous execution of m hardware logical gates, where m is a register size* [...] *Hence bitslice can be efficient when the entire hardware complexity of a target cipher is small and an underlying processor has many long registers.*"

In the context of higher-order masking, bitslice can be used at the s-box level to perform several secure s-box computations in parallel. One then need a compact Boolean representation of the s-box, and more importantly a representation with the least possible nonlinear gates. These nonlinear gates can then be securely evaluated in parallel using the ISW scheme as detailed hereafter. Such an approach was applied in [21] to design blockciphers with efficient masked computations. To the best of our knowledge, it has never been applied to get fast implementations of classical blockciphers such as AES or PRESENT. Also note that a bitsliced implementation of AES masked at first and second orders was described in [1] and used as a case study for practical side-channel attacks on a ARM Cortex-A8 processor running at 1 GHz.

6.1 ISW Logical AND

The ISW scheme can be easily adapted to secure a bitwise logical AND between two m-bit registers. From two d-sharings $(a_1, a_2, \ldots, a_d)$ and $(b_1, b_2, \ldots, b_d)$ of two m-bit strings $a, b \in \{0,1\}^m$, the ISW scheme computes an output d-sharing $(c_1, c_2, \ldots, c_d)$ of $c = a \wedge b$ as follows:

1. for every $1 \leq i < j \leq d$, sample an m-bit random value $r_{i,j}$,
2. for every $1 \leq i < j \leq d$, compute $r_{j,i} = (r_{i,j} \oplus a_i \wedge b_j) \oplus a_j \wedge b_i$,
3. for every $1 \leq i \leq d$, compute $c_i = a_i \wedge b_i \oplus \bigoplus_{j \neq i} r_{i,j}$.

On the ARM architecture considered in this paper, registers are of size $m = 32$ bits. We can hence perform 32 secure logical AND in parallel. Moreover a logical AND is a single instruction of 1 clock cycle in ARM so we expect the above ISW logical AND to be faster than the ISW field multiplications. The detailed performances of our ISW-AND implementation are provided in the full version. We observe that the ISW-AND is indeed faster than the fastest ISW field multiplication (*i.e.* ISW-FT). Moreover it does not require any precomputed table and is hence lighter in code than the ISW field multiplications (except for the binary multiplication which is very slow).

6.2 Secure Bitslice AES S-box

For the AES s-box, we based our work on the compact representation proposed by Boyar *et al.* in [8]. Their circuit is obtained by applying logic minimization techniques to the tower-field representation of Canright [9]. It involves 115 logic gates including 32 logical AND. The circuit is composed of three parts: the *top linear transformation* involving 23 XOR gates and mapping the 8 s-box input bits x_0, x_1, …, x_7 to 23 new bits x_7, y_1, y_2, …, y_{21}; the *middle non-linear transformation* involving 30 XOR gates and 32 AND gates and mapping the previous 23 bits to 18 new bits z_0, z_1, …, z_{17}; and the *bottom linear transformation* involving 26 XOR gates and 4 XNOR gates and mapping the 18 previous bits to the 8 s-box output bits $s_0, s_1, \ldots, s_7$. In particular, this circuit improves the usual count of 34 AND gates involved in previous tower-field representations of the AES s-box.

Using this circuit, we can perform the 16 s-box computations of an AES round in parallel. That is, instead of having 8 input bits mapped to 8 output bits, we have 8 (shared) input 16-bit words X_0, $X_1, \ldots$, X_7 mapped to 8 (shared) output 16-bit words S_1, S_2, …, S_8. Each word X_i (resp. S_i) contains the ith bits input bit (resp. output bit) of the 16 s-boxes. Each XOR gate and AND gate of the original circuit is then replaced by the corresponding (shared) bitwise instruction between two 16-bit words.

Parallelizing AND gates. For our masked bitslice implementation, a sound complexity unit is one call to the ISW-AND since this is the only nonlinear operation, *i.e.* the only operation with quadratic complexity in d (compared to other operations that are linear in d). In a straightforward bitslice implementation of the considered circuit, we would then have a complexity of 32 ISW-AND. This is suboptimal since each of these ISW-AND is applied to 16-bit words whereas it can operates on 32-bit words. Our main optimization is hence to group together pairs of ISW-AND in order to replace them by a single ISW-AND with fully filled input registers. This optimization hence requires to be able to group AND gates by pair that can be computed in parallel. To do so, we reordered the gates in the middle non-linear transformation of the Boyar *et al.* circuit, while keeping the computation consistent. We were able to fully parallelize the AND gates, hence dropping our bitslice complexity from 32 down to 16 ISW-AND. We thus get a parallel computation of the 16 AES s-boxes of one round with a complexity of 16 ISW-AND, that is one single ISW-AND per s-box. Since an ISW-AND is (significantly) faster than any ISW multiplication, our masked bitslice implementation breaks through the barrier of one ISW field multiplication per s-box. Our reordered version of the Boyar *et al.* circuit is provided in the full version of the paper.

Mask refreshing. As for the CRV method, our bitslice AES s-box makes calls to ISW with input sharings that might be linearly related. In order to avoid any flaw, we systematically refreshed one of the input sharings in our masked implementation. Here again, the implied overhead is mitigated (between 5% and 10%).

6.3 Secure Bitslice PRESENT S-box

For our masked bitslice implementation of the PRESENT s-box, we used the compact representation given by Courtois *et al.* in [16], which was obtained from Boyar *et al.* 's logic minimization techniques improved by involving OR gates. This circuit is composed of 4 nonlinear gates (2 AND and 2 OR) and 9 linear gates (8 XOR and 1 XNOR).

PRESENT has 16 parallel s-box computations per round, as AES. We hence get a bitslice implementation with 16-bit words that we want to group for the calls to ISW-AND. However for the chosen circuit, we could not fully parallelize the nonlinear gates because of the dependency between three of them. We could however group the two OR gates after a slight reordering of the operations. We hence obtain a masked bitslice implementation computing the 16 PRESENT s-boxes in parallel with 3 calls to ISW-AND. Our reordered version of the circuit is depicted in the full version of the paper. For the sake of security, we also refresh one of the two input sharings in the 3 calls to ISW-AND. As for the bitslice AES s-box, the implied overhead is manageable.

6.4 Implementation and Performances

Figures 12 and 13 plot the performances obtained for our masked bitslice implementations of the AES and PRESENT s-boxes. For comparison, we also recall the performances of the fastest polynomial methods for AES and PRESENT (*i.e.* parallel versions of KHL and $F \circ G$) as well as the fastest generic methods for $n = 8$ and $n = 4$ (*i.e.* parallel versions of the algebraic decomposition method for $n = 8$ and CRV for $n = 4$).

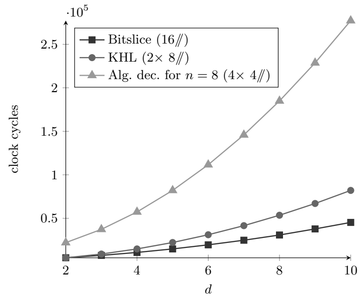


Fig. 12. Timings for 16 AES s-boxes.

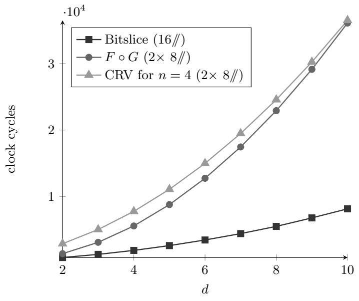


Fig. 13. Timings for 16 PRESENT s-boxes.

These results clearly demonstrate the superiority of the bitslicing approach. Our masked bitslice implementations of the AES and PRESENT s-boxes are significantly faster than state-of-the art polynomial methods finely tuned at the assembly level.

7 Cipher Implementations

This section finally describes masked implementations of the full PRESENT and AES blockciphers. These blockciphers are so-called *substitution-permutation networks*, where each round is composed of a key addition layer, a nonlinear layer and a linear diffusion layer. For both blockciphers, the nonlinear layer consists in the parallel application of 16 s-boxes. The AES works on a 128-bit state (which divides into sixteen 8-bit s-box inputs) whereas PRESENT works on a 64-bit state (which divides into sixteen 4-bit s-box inputs). For detailed specifications of these blockciphers, the reader is referred to [18] and [7]. For both blockciphers, we follow two implementation strategies: the standard one (with parallel polynomial methods for s-boxes) and the bitslice one (with bitslice s-box masking).

For the sake of efficiency, we assume that the key is already expanded, and for the sake of security we assume that each round key is stored in (non-volatile) memory under a shared form. In other words, we do not perform a masked key schedule. Our implementations start by masking the input plaintext with $d-1$ random m-bit strings (where m is the blockcipher bit-size) and store the d resulting shares in memory. These d shares then compose the sharing of the blockcipher state that is updated by the masked computation of each round. When all the rounds have been processed, the output ciphertext is recovered by adding all the output shares of the state. For the bitslice implementations, the translation from standard to bitslice representation is performed before the initial masking so that it is done only once. Similarly, the translation back from the bitslice to the standard representation is performed a single time after unmasking.

The secure s-box implementations are done as described in previous sections. It hence remains to deal with the key addition and the linear layers. These steps are applied to each share of the state independently. The key-addition step simply consists in adding each share of the round key to one share of the state. The linear layer implementations are described in the full version of the paper.

7.1 Performances

In our standard implementation of AES, we used the parallel versions of KHL and RP (with ISW-EL) for the s-box. For the standard implementation of PRESENT, we used the parallel versions of the $F \circ G$ method and of the CRV method. The obtained performances are summarized in Table 3. The timings are further plotted in Figures 14 and 15 for illustratetion.

These results clearly confirm the superiority of the bitslice implementations in our context. The bitslice AES implementation asymptotically takes 38% of the timings of the standard AES implementation using the best parallel polynomial method for the s-box (namely KHL). This ratio reaches 18% for PRESENT (compared to the $F \circ G$ method). It is also interesting to observe that PRESENT is slower than AES for standard masked implementations whereas it is faster for masked bitslice implementations. In the latter case, a PRESENT computation asymptotically amounts to 0.58 AES computation. This ratio directly results from the number of calls to ISW-AND which is $10 \times 16 = 160$ for AES (16 per round) and $31 \times 3 = 93$ for PRESENT (3 per round).

Table 3. Performances of masked blockciphers implementation.

	Clock cycles	Code (KB)	Random (bytes)
Bitslice AES	$3280\,d^2 + 14075\,d + 12192$	7.5	$640d(d+1)$
Standard AES (KHL $/\!\!/$)	$7640\,d^2 + 6229\,d + 6311$	4.8	$560d(d+1)$
Standard (AES RP-HT $/\!\!/$)	$9580\,d^2 + 5129\,d + 7621$	12.4	$400d(d+1)$
Standard (AES RP-EL $/\!\!/$)	$10301\,d^2 + 6561\,d + 7633$	4.1	$400d(d+1)$
Bitslice PRESENT	$1906.5\,d^2 + 10972.5\,d + 7712$	2.2	$372d(d+1)$
Standard PRESENT ($F \circ G$ $/\!\!/$)	$11656\,d^2 + 341\,d + 9081$	1.9	$496d(d+1)$
Standard PRESENT (CRV $/\!\!/$)	$9145\,d^2 + 45911\,d + 11098$	2.6	$248d(d+1)$

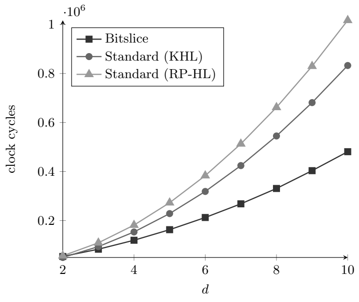


Fig. 14. Timings of masked AES.

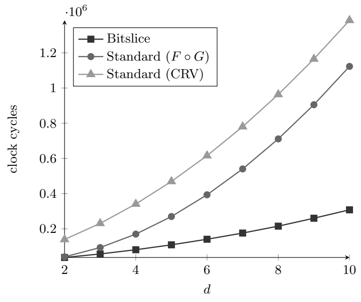


Fig. 15. Timings of masked PRESENT.

In order to illustrate the obtained performances in practice, Table 4 gives the corresponding timings in milliseconds for a clock frequency of 60 MHz. For a masking order of 10, our bitslice implementations only take a few milliseconds.

Table 4. Timings for masked bistlice AES and PRESENT with a 60 Mhz clock.

	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 10$
Bitslice AES	0.89 ms	1.39 ms	1.99 ms	2.7 ms	8.01 ms
Bitslice PRESENT	0.62 ms	0.96 ms	1.35 ms	1.82 ms	5.13 ms

References

1. J. Balasch, B. Gierlichs, O. Reparaz, and I. Verbauwhede. DPA, bitslicing and masking at 1 GHz. In T. Güneysu and H. Handschuh, editors, *Cryptographic*

Hardware and Embedded Systems – CHES 2015, volume 9293 of *Lecture Notes in Computer Science*, pages 599–619, Saint-Malo, France, Sept. 13–16, 2015. Springer, Heidelberg, Germany.

2. G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub. Verified proofs of higher-order masking. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485, Sofia, Bulgaria, Apr. 26–30, 2015. Springer, Heidelberg, Germany.
3. G. Barthe, F. Dupressoir, S. Faust, B. Grégoire, F.-X. Standaert, and P.-Y. Strub. Parallel implementations of masking schemes and the bounded moment leakage model. Cryptology ePrint Archive, Report 2016/912, 2016. `http://eprint.iacr.org/2016/912`.
4. E. Biham. A fast new DES implementation in software. In E. Biham, editor, *Fast Software Encryption – FSE'97*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272, Haifa, Israel, Jan. 20–22, 1997. Springer, Heidelberg, Germany.
5. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Higher-order threshold implementations. In P. Sarkar and T. Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 326–343, Kaoshiung, Taiwan, R.O.C., Dec. 7–11, 2014. Springer, Heidelberg, Germany.
6. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz. Threshold implementations of all 3 × 3 and 4 × 4 S-boxes. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 76–91, Leuven, Belgium, Sept. 9–12, 2012. Springer, Heidelberg, Germany.
7. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466, Vienna, Austria, Sept. 10–13, 2007. Springer, Heidelberg, Germany.
8. J. Boyar, P. Matthews, and R. Peralta. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 26(2):280–312, Apr. 2013.
9. D. Canright. A very compact S-box for AES. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455, Edinburgh, UK, Aug. 29 – Sept. 1, 2005. Springer, Heidelberg, Germany.
10. C. Carlet, L. Goubin, E. Prouff, M. Quisquater, and M. Rivain. Higher-order masking schemes for S-boxes. In A. Canteaut, editor, *Fast Software Encryption – FSE 2012*, volume 7549 of *Lecture Notes in Computer Science*, pages 366–384, Washington, DC, USA, Mar. 19–21, 2012. Springer, Heidelberg, Germany.
11. C. Carlet, E. Prouff, M. Rivain, and T. Roche. Algebraic decomposition for probing security. In R. Gennaro and M. J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 742–763, Santa Barbara, CA, USA, Aug. 16–20, 2015. Springer, Heidelberg, Germany.
12. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412, Santa Barbara, CA, USA, Aug. 15–19, 1999. Springer, Heidelberg, Germany.

13. J.-S. Coron. Higher order masking of look-up tables. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 441–458, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.
14. J.-S. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-order side channel security and mask refreshing. In S. Moriai, editor, *Fast Software Encryption – FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 410–424, Singapore, Mar. 11–13, 2014. Springer, Heidelberg, Germany.
15. J.-S. Coron, A. Roy, and S. Vivek. Fast evaluation of polynomials over binary finite fields and application to side-channel countermeasures. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 170–187, Busan, South Korea, Sept. 23–26, 2014. Springer, Heidelberg, Germany.
16. N. T. Courtois, D. Hulme, and T. Mourouzis. Solving circuit optimisation problems in cryptography and cryptanalysis. Cryptology ePrint Archive, Report 2011/475, 2011. `http://eprint.iacr.org/2011/475`.
17. A. Duc, S. Dziembowski, and S. Faust. Unifying leakage models: From probing attacks to noisy leakage. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.
18. FIPS PUB 197. *Advanced Encryption Standard*, Nov. 2001.
19. L. Genelle, E. Prouff, and M. Quisquater. Thwarting higher-order side channel analysis with additive and multiplicative maskings. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 240–255, Nara, Japan, Sept. 28 – Oct. 1, 2011. Springer, Heidelberg, Germany.
20. D. Goudarzi and M. Rivain. On the multiplicative complexity of boolean functions and bitsliced higher-order masking. In B. Gierlichs and A. Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, volume 9813 of *Lecture Notes in Computer Science*, pages 457–478, Santa Barbara, CA, USA, Aug. 17–19, 2016. Springer, Heidelberg, Germany.
21. V. Grosso, G. Leurent, F.-X. Standaert, and K. Varici. LS-designs: Bitslice encryption for efficient masked software implementations. In C. Cid and C. Rechberger, editors, *Fast Software Encryption – FSE 2014*, volume 8540 of *Lecture Notes in Computer Science*, pages 18–37, London, UK, Mar. 3–5, 2015. Springer, Heidelberg, Germany.
22. V. Grosso, E. Prouff, and F.-X. Standaert. Efficient masked S-boxes processing - A step forward -. In D. Pointcheval and D. Vergnaud, editors, *AFRICACRYPT 14: 7th International Conference on Cryptology in Africa*, volume 8469 of *Lecture Notes in Computer Science*, pages 251–266, Marrakesh, Morocco, May 28–30, 2014. Springer, Heidelberg, Germany.
23. V. Grosso, F.-X. Standaert, and S. Faust. Masking vs. multiparty computation: How large is the gap for AES? In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 400–416, Santa Barbara, CA, USA, Aug. 20–23, 2013. Springer, Heidelberg, Germany.
24. Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In D. Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481, Santa Barbara, CA, USA, Aug. 17–21, 2003. Springer, Heidelberg, Germany.

25. A. Journault, F. Standaert, and K. Varici. Improving the security and efficiency of block ciphers based on LS-designs. *Des. Codes Cryptography*, 82(1-2):495–509, 2017.
26. H. Kim, S. Hong, and J. Lim. A fast and provably secure higher-order masking of AES S-box. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 95–107, Nara, Japan, Sept. 28 – Oct. 1, 2011. Springer, Heidelberg, Germany.
27. M. Matsui and J. Nakajima. On the power of bitslice implementation on intel core2 processor. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134, Vienna, Austria, Sept. 10–13, 2007. Springer, Heidelberg, Germany.
28. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the limits: A very compact and a threshold implementation of AES. In K. G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.
29. S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of non-linear functions in the presence of glitches. In P. J. Lee and J. H. Cheon, editors, *ICISC 08: 11th International Conference on Information Security and Cryptology*, volume 5461 of *Lecture Notes in Computer Science*, pages 218–234, Seoul, Korea, Dec. 3–5, 2009. Springer, Heidelberg, Germany.
30. S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of non-linear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, Apr. 2011.
31. A. Poschmann, A. Moradi, K. Khoo, C.-W. Lim, H. Wang, and S. Ling. Side-channel resistant crypto for less than 2,300 GE. *Journal of Cryptology*, 24(2):322–345, Apr. 2011.
32. E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In T. Johansson and P. 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.
33. E. Prouff and T. Roche. Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 63–78, Nara, Japan, Sept. 28 – Oct. 1, 2011. Springer, Heidelberg, Germany.
34. M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427, Santa Barbara, CA, USA, Aug. 17–20, 2010. Springer, Heidelberg, Germany.
35. A. Roy and S. Vivek. Analysis and improvement of the generic higher-order masking scheme of FSE 2012. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 417–434, Santa Barbara, CA, USA, Aug. 20–23, 2013. Springer, Heidelberg, Germany.
36. A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A compact Rijndael hardware architecture with S-box optimization. In C. Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254, Gold Coast, Australia, Dec. 9–13, 2001. Springer, Heidelberg, Germany.