Algebraic Decomposition for Probing Security[⋆]

Claude Carlet[1], Emmanuel Prouff[2], Matthieu Rivain[3], and Thomas Roche[2]

[1] LAGA, UMR 7539, CNRS, Department of Mathematics,
University of Paris XIII and University of Paris VIII
`claude.carlet@univ-paris8.fr`

[2] ANSSI
`firstname.name@ssi.gouv.fr`

[3] CryptoExperts
`matthieu.rivain@cryptoexperts.com`

Abstract. The probing security model is very popular to prove the side-channel security of cryptographic implementations protected by masking. A common approach to secure nonlinear functions in this model is to represent them as polynomials over a binary field and to secure their nonlinear multiplications thanks to a method introduced by Ishai, Sahai and Wagner at Crypto 2003. Several schemes based on this approach have been published, leading to the recent proposal of Coron, Roy and Vivek which is currently the best known method when no particular assumption is made on the algebraic structure of the function. In the present paper, we revisit this idea by trading nonlinear multiplications for low-degree functions. Specifically, we introduce an algebraic decomposition approach in which a nonlinear function is represented as a sequence of functions with low algebraic degrees. We therefore focus on the probing-secure evaluation of such low-degree functions and we introduce three novel methods to tackle this particular issue. The paper concludes with a comparative analysis of the proposals, which shows that our algebraic decomposition method outperforms the method of Coron, Roy and Vivek in several realistic contexts.

1 Introduction

Since their introduction in [16, 17], side-channel attacks are known to be a serious threat against implementations of cryptographic algorithms. Among the existing strategies to secure an implementation, one of the most widely used relies on *secret sharing* (aka *masking*). Using secret sharing at the implementation level enables to achieve provable security in the so-called *probing security model* [13]. In this model, it is assumed that an adversary can recover information on a limited number of intermediate variables of the computation. This model has been argued to be practically relevant to address so-called *higher-order* side-channel attacks and it has been the basis of several efficient schemes to protect block ciphers [1, 6, 11, 12, 25, 26]. More recently, it has been shown in [10] that the probing security of an implementation actually implies its security in the more realistic *noisy leakage model* introduced in [24]. This makes probing security a very appealing feature for the design of secure implementations against side-channel attacks.

The first generic probing secure scheme, here called the ISW scheme, was designed by Ishai, Sahai and Wagner in [13]. It was later used by Rivain and Prouff to design an efficient probing-secure implementation of AES.[1] Several works then followed to improve this approach and to extend it to other SPN block ciphers [6, 8, 9, 14, 27]. In a nutshell, these methods consist in representing the nonlinear functions of such ciphers (the so-called *s-boxes*) as polynomials over a binary field. Evaluating such polynomials then involves some *linear* operations (*e.g.* squaring, addition, multiplication by constants) which are secured with straightforward and efficient methods, and some so-called *nonlinear multiplications* which are secured using ISW. The proposed

[⋆] This article is the full version of the extended abstract appearing in the proceedings of CRYPTO 2015.

[1] The original proposal of [26] actually involved a weak mask refreshing algorithm and the weakness was exploited in [8] to exhibit a flaw in the s-box processing. The authors of [8] also proposed a correction, which was recently verified for $d \leqslant 4$ using program verification techniques [2].

polynomial evaluation methods then aim at minimizing the number of nonlinear multiplications. The method recently introduced by Coron, Roy and Vivek in [9] (called CRV in the following) is currently the most efficient scheme for the probing-secure evaluation of nonlinear functions without particular algebraic structure.

In this paper we introduce a new *algebraic decomposition* approach for the probing-secure evaluation of nonlinear functions. More precisely, we propose a method — inspired from [9] — to efficiently compute a function from the evaluation of several other functions having a low algebraic degree. We subsequently study the problem of designing efficient probing-secure evaluation methods for low-degree functions. Specifically, we introduce three methods to tackle this issue with different and complementary assets. The first one relies on a recursive higher-order derivation of the target function. Remarkably, it enables to get a t-probing secure evaluation of a function of algebraic degree s (for arbitrary order t) from several s-probing secure evaluations of the function. Namely, for degree-s functions, it reduces t-probing security to s-probing security. This method has a complexity exponential in s and is hence only interesting for low-degree functions. For the case $s = 2$, we show how to specialize it into an efficient algorithm which happens to be a generalization of a previous method proposed in [8] to secure a peculiar type of multiplications (specifically $x \mapsto x \cdot \ell(x)$ where ℓ is linear). Our generalization of this algorithm can be applied to secure *any* quadratic function more efficiently than a *single* multiplication with the ISW scheme. Our second approach also applies a recursive derivation technique, but in a more direct way which allows us to design a stand-alone probing-secure evaluation method. Interestingly, this method does not rely on the polynomial representation of the target function h, and its processing only involves additions and evaluations of h. When the latter evaluation can be tabulated (which is often the case in the context of s-boxes), the proposed algorithm can be a valuable alternative to the state-of-the-art solutions based on field multiplications. However, this method also has a complexity exponential in the algebraic degree of h, which makes it only interesting for low degree functions. Eventually, the third method consists in tweaking the CRV method for the case of low-degree functions. The tweak is shown to substantially reduce the number of nonlinear multiplications for functions of a given (low) algebraic degree.

The theoretical analysis of our proposals is completed by an extensive comparative analysis considering several ratios for the cost of a field multiplication over the cost of other elementary operations. The results show that for functions of algebraic degree 2 our generalization of [8] is always the most efficient. On the other hand, for functions of algebraic degree 3, the tweaked CRV method achieves the best performances except for small probing orders for which our second method takes the advantage. For high-degree functions, our algebraic decomposition method reaches its best performances when it is combined with our generalization of [8]. For some cost ratios, our method is more efficient than CRV, which makes it the currently best known method for the probing-secure evaluation of nonlinear functions in these scenarios. As an example, for functions of dimension $n = 8$, our method outperforms CRV whenever a field multiplication takes more than 5 elementary operations (which we believe to be a realistic scenario).

From a general point of view, this paper shows that the issue of designing efficient probing-secure schemes for nonlinear functions may be reduced to the secure evaluation of functions of small algebraic degrees. This approach, and the first encouraging results reported in this paper, opens a promising avenue for further research on this topic. Our work also has strong connections with the field of *threshold implementations* in which secret sharing techniques are used to design hardware masking schemes resistant to glitches [20, 21]. An efficient threshold implementation is indeed usually obtained by decomposing the target nonlinear function into lower-degree functions (ideally quadratic functions). In this context, some decompositions have been proposed for specific functions, such as the PRESENT s-box [22], the AES s-box [19], and the DES s-boxes [3]. Some works have also followed an exhaustive approach to provide decompositions for small-size s-boxes, specifically all bijective 3-bit and 4-bit s-boxes [3, 18]. Certain 5-bit and 6-bit s-boxes have also been considered in [4].

2 Preliminaries

2.1 Functions in Finite Fields and Derivation

Along this paper, we shall denote by $[\![i,j]\!]$ the integer interval $[i,j]\cap\mathbb{Z}$ for any pair of integers (i,j) with $i\leqslant j$, and we shall denote by $\mathbb{F}_{2^n}$ the finite field with 2^n elements for any integer $n\geqslant 1$. Choosing a basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2^n$ enables to represent elements of $\mathbb{F}_{2^n}$ as elements of the vector space $\mathbb{F}_2^n$ and *vice versa*. In the following, we assume that the same basis is always used to represent elements of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$. For any positive integers $m\leqslant n$, a function from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$ is called an (n,m)*-function* (the dimensions could be omitted if they are clear from the context). Nonlinear functions used in block-ciphers are usually called *s-boxes*. The set of (n,m)-functions is denoted by $\mathcal{B}_{n,m}$. When m divides n, any function $h\in\mathcal{B}_{n,m}$ can be represented by a polynomial function $x\in\mathbb{F}_{2^n}\mapsto\sum_{i=0}^{2^n-1}a_ix^i$ where the a_i's are constant coefficients in $\mathbb{F}_{2^n}$ that can be obtained by applying Lagrange's Interpolation. When m does not divide n, the m-bit outputs of h can be embedded into $\mathbb{F}_2^n$ by padding them to n-bit outputs (*e.g.* by setting the most significant bits to 0). This ensures that any function $h\in\mathcal{B}_{n,m}$ can be evaluated as a polynomial over $\mathbb{F}_{2^n}$. If padding has been used, then it can be removed after the polynomial evaluation by mapping the output from $\mathbb{F}_{2^n}$ to $\mathbb{F}_2^n$, and then from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$ (by removing the $n-m$ most significant 0's). The *algebraic degree* of a function $h\in\mathcal{B}_{n,m}$ is the integer value $\max_{a_i\neq 0}(\mathtt{HW}(i))$ where the a_i's are the coefficients of the polynomial representation of h and where $\mathtt{HW}(i)$ denotes the Hamming weight of i (see *e.g.* [5] for more details about this notion). The algebraic degree must not be confused with the classical notion of polynomial degree which is the integer value $\max_{a_i\neq 0}(i)$. A function of algebraic degree s will sometimes be called a *degree-s function.*

This paper intensively uses the notion of *higher-order derivative* of a (n,m)-function. It is is recalled hereafter.

Definition 1 (Higher-Order Derivative). *Let n and m be two positive integers such that $m\leqslant n$ and let $h\in\mathcal{B}_{n,m}$. For any positive integer t and any t-tuple $(a_1,a_2,\ldots,a_t)\in(\mathbb{F}_2^n)^t$, the (n,m)-function $D_{a_1,a_2,\ldots,a_t}h$ which maps any $x\in\mathbb{F}_{2^n}$ to $\sum_{I\subseteq[\![1,t]\!]}h\Big(x+\sum_{i\in I}a_i\Big)$ is called the* t^{th}-order derivative *of h with respect to $(a_1,a_2,\ldots,a_t)$.*

Any t^{th}-order derivative of a degree-s function has an algebraic degree bounded above by $s-t$. In the following we shall denote by $\varphi_h^{(t)}$ the (n,m)-function defined by

$$\varphi_h^{(t)}:(a_1,a_2,\ldots,a_t)\mapsto D_{a_1,a_2,\ldots,a_t}\,h(0)\ . \tag{1}$$

This function has some well-known properties recalled hereafter.

Proposition 1. *Let $h\in\mathcal{B}_{n,m}$ be a degree-s function. Then, the function $\varphi_h^{(s)}$ is s-linear symmetric and equals zero for any family of a_i linearly dependent over $\mathbb{F}_2$. Conversely, if $\varphi_h^{(s)}$ is s-linear, then the algebraic degree of h is at most s.*

Proof. $\varphi_h^{(s)}$ is clearly symmetric and we have

$$\varphi_h^{(s)}(a_1,a_2,\ldots,a_s)=D_{a_1,a_2,\ldots,a_{s-1}}\,h(0)+D_{a_1,a_2,\ldots,a_{s-1}}\,h(a_s)\ .$$

Then the function $a_s\mapsto\varphi_h^{(s)}(a_1,a_2,\ldots,a_s)$ has algebraic degree at most 1 for every $a_1,a_2,\ldots,a_{s-1}$. Since it vanishes at 0, it is then linear. By symmetry, we deduce that $\varphi_h^{(s)}$ is s-linear. If two among the a_i's are equal, then $\varphi_h^{(s)}(a_1,a_2,\ldots,a_s)$ equals the sum (in characteristic 2) of twice the same values and therefore vanishes. If more generally the a_i's are not linearly independent over $\mathbb{F}_2$, then one of them is a linear combination of the others and by s-linearity and thanks to the latter property, it also vanishes. Conversely, if $\varphi_h^{(s)}$ is s-linear, then for every $a_1,a_2,\ldots,a_{s-1}$, $D_{a_1,a_2,\ldots,a_{s-1}}\,h$ has algebraic degree at most 1 and it is then a simple matter to check by decreasing induction on t that $D_{a_1,a_2,\ldots,a_t}\,h$ has algebraic degree at most $s-t$ fot every $t\leqslant s$. For $t=0$ this implies that h has algebrac degree at most s. $\square$

2.2 Sharing and Probing Security

For two positive integers k and d, a (k,d)*-sharing* of a variable x defined over some finite field $\mathbb{K}$ is a random vector $(x_1, x_2, \ldots, x_k)$ over $\mathbb{K}^k$ such that $x = \sum_{i=1}^{k} x_i$ holds (*completeness equality*) and any tuple of $d-1$ shares x_i is a uniform random vector over $\mathbb{K}^{d-1}$. The variable x is also called the *plain value* of the sharing $(x_i)_i$. If $k = d$, the terminology simplifies to d*-sharing*.

An algorithm with domain $\mathbb{K}^d$ is said t*-probing secure* if on input a d-sharing $(x_1, x_2, \ldots, x_d)$ of some variable x, it admits no tuple of t or less intermediate variables that depends on x. An algorithm achieving t-probing security is resistant to the class of t^{th}-order side-channel attacks (see for instance [7, 24, 26]). In this paper we will further use the following non-standard notion.

Definition 2 (perfect t-probing security). *An algorithm is said to achieve* perfect t-probing security *if every ℓ-tuple of its intermediate variables can be perfectly simulated from an ℓ-tuple of its input shares for every $\ell \leqslant t$.*

It can be shown that for the tight security case $t = d-1$, the notion above is equivalent to the usual t-probing security. This is formalized in the following lemma.

Lemma 1. *An algorithm taking a d-sharing as input achieves $(d-1)$-probing security if and only if it achieves perfect $(d-1)$-probing security.*

Proof. Perfect $(d-1)$-probing security clearly implies $(d-1)$-probing security. Now assume that some algorithm taking a d-sharing $(x_1, x_2, \ldots, x_d)$ as input does not achieve perfect $(d-1)$-probing security. This means that there exists a ℓ-tuple $(v_1, v_2, \ldots, v_\ell)$ of its intermediate variables that requires at least $\ell + 1$ input shares $(x_i)_{i \in I}$ to be perfectly simulated, where $I \subseteq [\![1, d]\!]$ and $|I| \geqslant \ell + 1$. Then consider the tuple of intermediate variables composed of $(v_1, v_2, \ldots, v_\ell)$ and the inputs $(x_i)_{i \notin I}$. This tuple has at most $d-1$ elements but it requires the full input sharing $(x_1, x_2, \ldots, x_d)$ to be perfectly simulated. Therefore the algorithm cannot be $(d-1)$-probing secure. By contrapositive, the $(d-1)$-probing security implies the perfect $(d-1)$-probing security for algorithms taking a d-sharing as input. $\square$

We shall simply say that an algorithm taking a d-sharing as input is (perfect) probing secure when it achieves (perfect) $(d-1)$-probing security. We will also say that an algorithm admits a t-order flaw when t of its intermediate variables jointly depend on the plain input (hence contradicting t-probing security).

It is worth noticing that achieving (perfect) probing security for the evaluation of a linear function g is pretty simple. Computing a d-sharing of $g(x)$ from a d-sharing $(x_1, x_2, \ldots, x_d)$ of x can indeed be done by applying g to every share. The process is clearly (perfectly) t-probing secure with $t = d-1$ and the completeness holds by linearity of g since we have $g(x) = g(x_1) + g(x_2) + \cdots + g(x_d)$. The same holds for an affine function but the constant term $g(0)$ must be added to the right side if and only if d is odd (without impacting the probing security).

Probing security is more tricky to achieve for nonlinear functions. In [13], Ishai, Sahai, and Wagner tackled this issue by introducing the first generic t-probing secure scheme for the multiplication over $\mathbb{F}_2$ which can be easily extended to the multiplication over any finite field. Let $(x_i)_i$ and $(y_i)_i$ be the d-sharings of two variables x and y over some binary field $\mathbb{K}$, the ISW scheme proceeds as follows:

1. for every $1 \leqslant i < j \leqslant d$, sample a random value $r_{i,j}$ over $\mathbb{K}$,
2. for every $1 \leqslant i < j \leqslant d$, compute $r_{j,i} = (r_{i,j} + a_i \cdot b_j) + a_j \cdot b_i$,
3. for every $1 \leqslant i \leqslant d$, compute $c_i = a_i \cdot b_i + \sum_{j \neq i} r_{i,j}$.

The completeness holds from $\sum_i c_i = \sum_{i,j} a_i \cdot b_j = (\sum_i a_i)(\sum_j b_j)$ since every random value $r_{i,j}$ appears exactly twice in the sum and hence vanishes.[2] The above multiplication procedure was proved t-probing secure with $t \leqslant (d-1)/2$ in [13]. This was later improved in [26] to a tight t-probing security with $t \leqslant d-1$.

[2] This is true since $\mathbb{K}$ is a binary field, but the method can easily be extended to any field by defining $r_{j,i} = (a_i \cdot b_j - r_{i,j}) + a_j \cdot b_i$ in Step 2.

2.3 Secure Evaluation of Nonlinear Functions

In the original scheme of Ishai *et al.*, a computation is represented as a Boolean circuit composed of logical operations NOT and AND. In [26], Rivain and Prouff generalized their approach to larger fields which allowed them to design an efficient probing-secure computation of the AES cipher. The main issue in securing SPN ciphers like AES lies in the s-box computations (the rest of the cipher being purely linear operations). The Rivain-Prouff scheme computes the AES s-box using 4 multiplications over $\mathbb{F}_{2^8}$ (secured with ISW) plus some linear operations (specifically squaring over $\mathbb{F}_{2^8}$, and the initial affine transformation). This approach was then extended in [6] to secure any s-box in $\mathcal{B}_{n,m}$. The principle is to represent the s-box as a polynomial $\sum_i a_i \cdot x^i$ in $\mathbb{F}_{2^n}[x]$ whose evaluation is then expressed as a sequence of linear functions (*e.g.* squaring over $\mathbb{F}_{2^n}$, additions, multiplications by coefficients) and *nonlinear multiplications* over $\mathbb{F}_{2^n}$. The former operations can be simply turned into probing-secure operations of complexity $O(d)$ as described in the previous section, whereas the latter operations are secured using ISW with complexity $O(d^2)$. The total complexity is hence mainly impacted by the number of nonlinear multiplications involved in the underlying polynomial evaluation method. This observation led to a series of publications aiming at developing polynomial evaluation methods with least possible costs in terms of nonlinear multiplications. Some heuristics in this context were proposed by Carlet *et al.* in [6], and then improved by Roy and Vivek in [27] and by Coron, Roy and Vivek in [9]. In the latter reference, a method is introduced that we shall call the CRV method in the following. It is currently the most efficient way to get probing-secure evaluations of nonlinear functions.

Tabulated multiplications. Further works have shown that secure evaluation methods could be improved by relying on specific kind of nonlinear multiplications. Assume for instance that the dimension n is such that a lookup table with 2^{2n} entries is too big for some given architecture whereas a lookup table with 2^n entries fits (*e.g.* for $n = 8$ one needs 64 kilobytes for the former and 256 bytes for the latter). This means that a multiplication over $\mathbb{F}_{2^{n/2}}$ can be computed using a single lookup, whereas a multiplication over $\mathbb{F}_{2^n}$ must rely on more costly arithmetic (*e.g.* schoolbook method, log-exp tables, Karatsuba, etc.). This observation was used by Kim, Hong, and Lim in [14] to design an efficient alternative to the Rivain-Prouff scheme based on the so-called *tower-field representation* of the AES s-box [28]. Using this representation, the AES s-box can be computed based on 5 multiplications over $\mathbb{F}_{2^4}$ instead of 4 multiplications over $\mathbb{F}_{2^8}$, which results in a significant gain of performance when the former are tabulated while the latter are computed using log-exp tables (although one more multiplication is involved). Another approach, proposed in [8] by Coron, Prouff, Rivain, and Roche, is to consider the multiplications of the form $x \cdot \ell(x)$ where ℓ is a linear function. Such multiplications can be secured using a variant of ISW relying on a table for the function $x \mapsto x \cdot \ell(x)$. This variant that we shall call the CPRR scheme, allowed the authors to design a faster probing-secure scheme for the AES s-box. It was further used in [12] to reduce the complexity of the probing-secure evaluation of power functions in $\mathbb{F}_{2^n}$ with $n \leqslant 8$.

3 The Algebraic Decomposition Method

In this section, we introduce a new algebraic decomposition method that split the evaluation of a nonlinear function with arbitrary algebraic degree into several evaluations of functions with given (low) algebraic degree s. Our proposed method is inspired from the CRV method but relies on low-degree polynomial evaluations instead of nonlinear multiplications.

We consider a function $h \in \mathcal{B}_{n,m}$ which is seen as a polynomial $h(x) = \sum_{j=0}^{2^n-1} a_j x^j$ over $\mathbb{F}_{2^n}[x]$. We start by deriving a family of generators $(g_i)_i$ as follows:

$$\begin{cases} g_1(x) = f_1(x) \\ g_i(x) = f_i\big(g_{i-1}(x)\big) \end{cases} , \tag{2}$$

where the f_i's are random polynomials of given algebraic degree s. Then we randomly generate t polynomials $(q_i)_i$ over the subspace of polynomials $\sum_{j=1}^{r} \ell_j \circ g_j$ where the ℓ_j's are linearized polynomials (*i.e.* polynomials of algebraic degree 1). This is done by sampling random linearized polynomials $(\ell_{i,j})_{i,j}$ and by computing

$$q_i(x) = \sum_{j=1}^{r} \ell_{i,j}\big(g_j(x)\big) + \ell_{i,0}(x) \ . \tag{3}$$

Eventually, we search for t polynomials p_i of algebraic degree s and for $r+1$ linearized polynomials ℓ_i such that

$$h(x) = \sum_{i=1}^{t} p_i\big(q_i(x)\big) + \sum_{i=1}^{r} \ell_i\big(g_i(x)\big) + \ell_0(x) \ . \tag{4}$$

From such polynomials, we get a method to compute $h(x)$ by subsequently evaluating (2), (3) and (4). This method involves $r+t$ evaluations of degree-s polynomials (the f_i and the p_i), plus some linear operations. The following table gives the exact operation counts (where "#eval deg-s" denotes the number of evaluations of degree-s functions, "#eval LP" denotes the number of evaluations of linearized polynomials, and "#add" denotes the number of additions).

#eval deg-s	#eval LP	#add
$r+t$	$(t+1)(r+1)$	$r \cdot t + t + r$

The complexity of the resulting d-probing secure evaluation can be obtained by multiplying by d the number of additions and the number of evaluations of linearized polynomials (which might be tabulated) and by adding $r+t$ secure evaluations of degree-s functions.

Remark 1. The generation step of our method can be generalized as follows:

$$\begin{cases} g_1(x) = f_1(x) \\ g_i(x) = f_i\Big(\sum_{j=1}^{i-1} \ell'_{i,j}\big(g_j(x)\big)\Big) \end{cases} \tag{5}$$

where the f_i's are random polynomials of given algebraic degree s and the $\ell'_{i,j}$'s are random linearized polynomials. This generalization has no impact when $r \leqslant 2$. In particular, it has no impact on our experimental results for $s \in \{2,3\}$ and $n \in [\![4,8]\!]$ (indeed, the best counts are always obtained with $r \leqslant 2$ since we stop at $g_2(x) = f'_2 \circ f_1(x)$ where $f'_2 = f_2 \circ \ell_{2,1}$ is of degree s_1 – see hereafter –). However, (5) might give better results than (2) for higher values of n and/or s.

As in the CRV method, the search of polynomials $(p_i)_i$ and $(\ell_i)_i$ satisfying (4) for given polynomials $(g_i)_i$ and $(q_i)_i$ amounts to solve a system of linear equations over $\mathbb{F}_{2^n}$:

$$A \cdot \boldsymbol{b} = \boldsymbol{c} \ . \tag{6}$$

The target vector $\boldsymbol{c}$ has 2^n coordinates which are the values taken by $h(x)$ for all x over $\mathbb{F}_{2^n}$, that is

$$\boldsymbol{c} = (h(e_1), h(e_2), h(e_3), \ldots, h(e_{2^n})) \ ,$$

where $\{e_1, e_2, \ldots, e_{2^n}\} = \mathbb{F}_{2^n}$. The coordinates of the vector $\boldsymbol{b}$ are the variables of the system that represents the solutions for the coefficients of the polynomials $(p_i)_i$ and $(\ell_i)_i$. The matrix A is then defined as the concatenation of several submatrices:

$$A = (\mathbf{1} \mid A_{p_1} \mid A_{p_2} \mid \cdots \mid A_{p_t} \mid A_{\ell_0} \mid A_{\ell_1} \mid \cdots \mid A_{\ell_r}) \ ,$$

where $\mathbf{1}$ is the 2^n-dimensional column vector with all coordinates equal to $1 \in \mathbb{F}_{2^n}$, where the A_{p_i} are $2^n \times N_s$ submatrices, with $N_s = \sum_{d=2}^{s} \binom{n}{d}$, and where the A_{ℓ_i} are $2^n \times n$ submatrices. For every $i \in [\![1,t]\!]$, the $2^n \times N_s$ matrix A_{p_i} is defined as:

$$A_{p_i} = \begin{pmatrix} q_i(e_1)^{\beta_1} & q_i(e_1)^{\beta_2} & \cdots & q_i(e_1)^{\beta_{N_s}} \\ q_i(e_2)^{\beta_1} & q_i(e_2)^{\beta_2} & \cdots & q_i(e_2)^{\beta_{N_s}} \\ \vdots & \vdots & \ddots & \vdots \\ q_i(e_{2^n})^{\beta_1} & q_i(e_{2^n})^{\beta_2} & \cdots & q_i(e_{2^n})^{\beta_{N_s}} \end{pmatrix} ,$$

where $\{\beta_i\} = \{\beta \mid 2 \leqslant \mathtt{HW}(\beta) \leqslant s\} \subseteq [\![0, 2^n - 1]\!]$ *i.e.* the β_i are the powers with non-zero coefficients in a degree-s polynomial with null linear and constant parts. We can indeed search for p_i's with null linear parts (*i.e.* with 0-coefficients for the powers x^{2^j} and 0-constants) since these linear parts would not add any degrees of freedom compared to the linear combinations of the g_i's. On the other hand, A_{ℓ_i} is defined as the $2^n \times n$ matrix:

$$A_{\ell_0} = \begin{pmatrix} e_1^{\alpha_1} & e_1^{\alpha_2} & \cdots & e_1^{\alpha_n} \\ e_2^{\alpha_1} & e_2^{\alpha_2} & \cdots & e_2^{\alpha_n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{2^n}^{\alpha_1} & e_{2^n}^{\alpha_2} & \cdots & e_{2^n}^{\alpha_n} \end{pmatrix} \quad \text{and} \quad A_{\ell_i} = \begin{pmatrix} g_i(e_1)^{\alpha_1} & g_i(e_1)^{\alpha_2} & \cdots & g_i(e_1)^{\alpha_n} \\ g_i(e_2)^{\alpha_1} & g_i(e_2)^{\alpha_2} & \cdots & g_i(e_2)^{\alpha_n} \\ \vdots & \vdots & \ddots & \vdots \\ g_i(e_{2^n})^{\alpha_1} & g_i(e_{2^n})^{\alpha_2} & \cdots & g_i(e_{2^n})^{\alpha_n} \end{pmatrix}$$

for $i \geqslant 1$, where $\{\alpha_i\} = \{2^i \mid 0 \leqslant i \leqslant n-1\}$ (*i.e.* the α_i are the powers with non-zero coefficients in a linearized polynomial).

System (6) has 2^n equations and $t \cdot N_s + (r+1)\,n$ unknowns. Such a system admits a solution for every choice of $h(x)$ if and only if its rank is 2^n, which implies the following inequality as necessary condition:

$$t \cdot N_s + (r+1)\,n \geqslant 2^n \quad \Leftrightarrow \quad t \geqslant \frac{2^n - (r+1)\,n}{N_s} \ . \tag{7}$$

In other words our method requires at least $\big(2^n - (r+1)\,n\big)/N_s$ secure evaluations of degree-s polynomials. Another necessary condition is that the algebraic degree of the $p_i \circ q_i$ reaches n, that is r verifies $s^{r+1} \geqslant n$. This constraint becomes $s^{r+1} \geqslant n-1$ if we only focus on bijective functions (since their algebraic degree is at most $n-1$).

Remark 2. We stress that once a full-rank system has been found for some given parameters r and t, it can be applied to get a decomposition for *every* n-bit nonlinear function (the considered function being the target vector of the system). Note however that for some specific function it might be possible to find a decomposition involving less than $r+t$ degree-s polynomials. For instance, the 4-bit s-box of PRESENT can be decomposed into 2 quadratic functions [22], whereas we need 3 quadratic functions to decompose any 4-bit s-box.

Experimental results. We provide hereafter some experimental results for our algebraic decomposition method. We experimented our method for $n \in [\![4, 8]\!]$ and $s \in \{2, 3\}$. The following table summarizes the best results that we obtained and compares them to the lower bound resulting from the above constraints.

	$n=4$	$n=5$	$n=6$	$n=7$	$n=8$
#eval-2 (achieved)	**3**	**4**	**5**	**8**	**11**
#eval-2 (lower bound)	2	4	5	7	10
#eval-3 (achieved)	**2**	**3**	**3**	**4**	**4**
#eval-3 (lower bound)	2	3	3	4	4

Note that our method can be generalized to involve the evaluation of functions with different (low) algebraic degrees. In particular, in our experiments, we considered the hybrid case where the f_i are of degree s_1 and the p_i are of degree s_2. In that case, the constraint on r becomes $s_1^r \cdot s_2 \geqslant n$ (resp. $s_1^r \cdot s_2 \geqslant n-1$ for bijective functions), and the constraint on t remains as in (7) with s_2 instead of s. The following table summarizes the best results for the hybrid case $(s_1, s_2) = (2, 3)$. This case was always more favorable than the case $(s_1, s_2) = (3, 2)$.

	$n=4$	$n=5$	$n=6$	$n=7$	$n=8$
#eval-2 + #eval-3 (achieved)	**1+1**	**2+1**	**1+2**	**2+2**	**2+3**
#eval-2 + #eval-3 (lower bound)	$1+1$	$2+1$	$1+2$	$2+2$	$2+3$

The following table gives the exact parameters (s_1, s_2), r, and t, that we achieved for every $n \in [\![4, 8]\!]$. Note that the entries 4^*, 5^*, and 7^* stand for bijective functions of size $n \in \{4, 5, 7\}$, which enjoy more efficient decompositions. For the other considered values of n, the bijective property did not enable any improvement.

n	#eval-2	#eval-3	(s_1, s_2)	r	t
4	3	-	$(2,2)$	1	2
	3	-	$(2,2)$	2	1
	1	1	$(2,3)$	1	1
	-	2	$(3,3)$	1	1
4^*	-	1	$(1,3)$	0	1
5	4	-	$(2,2)$	2	2
	2	1	$(2,3)$	2	1
	-	3	$(3,3)$	1	2
5^*	3	-	$(2,2)$	1	2
6	5	-	$(2,2)$	2	3
	1	2	$(2,3)$	1	2
	-	3	$(3,3)$	1	2
7	8	-	$(2,2)$	2	6
	2	2	$(2,3)$	2	2
	-	4	$(3,3)$	1	3
	-	4	$(3,3)$	2	2
7^*	1	2	$(2,3)$	1	2
8	11	–	$(2,2)$	2	9
	11	–	$(2,2)$	3	8
	2	3	$(2,3)$	2	3
	-	4	$(3,3)$	1	3

Remark 3. For the case $n = 4$, it was shown in [3] that every cubic bijective function in $\mathcal{B}_{4,4}$ can be either decomposed as $h(\cdot) = f_1 \circ f_2(\cdot)$ or as $h(\cdot) = f_1 \circ f_2 \circ f_3(\cdot)$, where the f_i are quadratic functions, if and only if it belongs to the alternating group of permutations in $\mathcal{B}_{4,4}$. It was then shown in [18] that the so-called *optimal s-boxes* in $\mathcal{B}_{4,4}$ can be decomposed as $h(\cdot) = f_1(\cdot) + f_2 \circ f_3(\cdot)$. The authors also suggested to use a decomposition of the form $h(\cdot) = f_1(\cdot) + f_2 \circ f_3(\cdot) + f_4 \circ f_5(\cdot) + \ldots$ for other functions in $\mathcal{B}_{4,4}$. Our results demonstrate that *every* function in $\mathcal{B}_{4,4}$ can be decomposed using 3 quadratic functions plus some linear functions. Moreover, we know from [3] that there exist functions in $\mathcal{B}_{4,4}$ that cannot be decomposed using only two quadratic functions. This show the optimality of our method for the case $n = 4$ and $s = 2$. This also suggests that the lower bound (7) might not be tight.

4 Reducing the Probing Order down to the Algebraic Degree

In this section we start by showing that arbitrary t-probing security can always be reduced to s-probing security where s is the algebraic degree of the function to protect. Specifically, we give a method to construct a t-probing secure processing of a degree-s function from its s-probing secure processing. Then, we apply our method for the case $s = 2$, where a simple 2-probing secure processing can be turned into an efficient t-probing secure evaluation of any (algebraically) quadratic function.

4.1 General Case

Let n and m be two positive integers such that $m \leqslant n$ and let $h \in \mathcal{B}_{n,m}$ be the vectorial function whose processing must be secured at the order $t = d - 1$ for some d. By definition of $\varphi_h^{(s)}$, for every tuple

$(a_1, \cdots, a_s) \in (\mathbb{F}_2^n)^s$ we have:

$$h\Big(\sum_{i=1}^{s} a_i\Big) = \varphi_h^{(s)}(a_1, a_2, \ldots, a_s) + \sum_{I \subsetneq [\![1,s]\!]} h\Big(\sum_{i \in I} a_i\Big) \ . \tag{8}$$

Iterating (8) we obtain the following theorem where, by convention, $h(\sum_{i\in\emptyset} a_i)$ equals $h(0)$. The proof is given in Appendix A.

Theorem 1. *Let $h \in \mathcal{B}_{n,m}$ be a vectorial function of algebraic degree at most s. Then, for every $d \geqslant s$ we have:*

$$h\Big(\sum_{i=1}^{d} a_i\Big) = \sum_{1\leqslant i_1<\cdots<i_s\leqslant d} \varphi_h^{(s)}(a_{i_1}, \ldots, a_{i_s}) + \sum_{j=0}^{s-1} \eta_{d,s}(j) \sum_{\substack{I\subseteq[\![1,d]\!] \\ |I|=j}} h\Big(\sum_{i\in I} a_i\Big) \ ,$$

where $\eta_{d,s}(j) = \binom{d-j-1}{s-j-1} \bmod 2$ for every $j \leqslant s-1$.

From Theorem 1 we get the following corollary (see proof in Appendix B).

Corollary 1. *Let $h \in \mathcal{B}_{n,m}$ be a vectorial function of algebraic degree at most s. Then, for every $d > s$ we have:*

$$h\Big(\sum_{i=1}^{d} a_i\Big) = \sum_{j=0}^{s} \mu_{d,s}(j) \sum_{\substack{I\subseteq[\![1,d]\!] \\ |I|=j}} h\Big(\sum_{i\in I} a_i\Big) \ , \tag{9}$$

where $\mu_{d,s}(j) = \binom{d-j-1}{s-j} \bmod 2$ for every $j \leqslant s$.

Corollary 1 states that, for any d, the evaluation of a degree-s function $h \in \mathcal{B}_{n,m}$ on the sum of d shares can be expressed as several evaluations of h on sums $\sum_{i\in I} a_i$ with $|I| \leqslant s$. We can then reduce a $(d-1)$-probing secure evaluation of h to several $(j-1)$-probing secure evaluations of h where $j = |I| \leqslant s$. Doing so, each evaluation takes $j = |I|$ shares $(a_i)_{i\in I}$ and computes a j-sharing of $h(\sum_{i\in I} a_i)$. Afterwards, one needs a secure scheme to combine the obtained shares of all the $h(\sum_{i\in I} a_i)$, with $I \subseteq [\![1,d]\!]$ such that $|I| \leqslant s$ and $\mu_{d,s}(|I|) = 1$, into a d-sharing of $h(a)$.

The overall process is summarized in the following algorithm, where SecureEval is a primitive that performs a $(j-1)$-probing secure evaluation of h on a j-sharing input for any $j \leqslant s$, and where SharingCompress is a primitive that on input $(x_i)_{i\leqslant[\![1,k]\!]}$ produces a d-sharing of $\sum_{i=1}^{k} x_i$ for any $k \leqslant d$. The description of such SharingCompress algorithm which achieves perfect t-probing security is given in Appendix C.

```
Algorithm 1 : Secure evaluation of a degree-$s$ function
  Input  : a $d$-sharing $(x_1, x_2, \cdots, x_d)$ of $x \in \mathbb{F}_{2^n}$
  Output: a $d$-sharing $(y_1, y_2, \cdots, y_d)$ of $y = h(x)$
1 for $I \subseteq [\![1,d]\!]$ with $|I| \leqslant s$ and $\mu_{d,s}(|I|) = 1$ do
2     $(r_{I,k})_{k\leqslant|I|} \leftarrow$ SecureEval$(h, (x_i)_{i\in I})$
3 $(y_1, y_2, \ldots, y_d) \leftarrow$ SharingCompress$\big((r_{I,k})_{k\leqslant|I|, I\subseteq[\![1,d]\!], \mu_{d,s}(|I|)=1}\big)$
4 return $(y_0, y_1, \ldots, y_d)$
```

Security. The following theorem states the (perfect) probing security of our method:

Theorem 2. *If* SecureEval *and* SharingCompress *are perfect probing-secure, then Algorithm 1 is perfect probing-secure.*

Proof. By assumption, the SecureEval primitive called on j input shares is $(j-1)$-probing secure. From Lemma 1, it is then *perfect* $(j-1)$-probing secure. Moreover, since the input is a j-sharing, any set of intermediate variables of arbitrary size ℓ can be perfectly simulated from the knowledge of ℓ or fewer inputs (here ℓ may be greater than or equal to j).

Let us consider $\ell < d$ intermediate variables from Algorithm 1. Part of these intermediate variables are located in the SharingCompress algorithm (including its output), say t of them. By the perfect probing-security of SharingCompress, the t intermediate variables can be perfectly simulated from t of SharingCompress algorithm inputs. These t inputs are actually outputs of the SecureEval primitives. Adding these t outputs to the rest of the intermediate variables (there are $\ell - t$ left), the properties of SecureEval primitives allow to select ℓ inputs of Algorithm 1 that together allow to perfectly simulate the whole set of intermediate variables. Therefore, Algorithm 1 is perfect $(d-1)$-probing secure. $\square$

Complexity. For every s and every $d > s$, the term $\mu_{d,s}(j)$ always equals 1 when $j = s$. On the other hand, whenever $d \equiv s \bmod 2^\ell$ with $\ell = \lfloor \log_2 s \rfloor + 1$, the term $\mu_{d,s}(j)$ equals 0 for every $j < s$. For the sake of efficiency, we suggest using such a value of d for our method.[3] The complexity of Algorithm 1 is then of $\binom{d}{s}$ calls to SecureEval with s shares and one call to SharingCompress from k shares to d shares where $k = s\binom{d}{s}$. From the complexity of the sharing compression method, we obtain the following operation count (where "#add" and "#rand" respectively denote the number of additions and the number of sampled random values in the sharing compression).

	#SecureEval	#add	#rand
Exact count	$\binom{d}{s}$	$\big(s\binom{d}{s} - d\big)(d+1)$	$\frac{1}{2}\big(s\binom{d}{s} - d\big)(d-1)$
Approximation	$\big(\frac{1}{s!}\big)d^s$	$\big(\frac{1}{(s-1)!}\big)d^{s+1}$	$\big(\frac{1}{2(s-1)!}\big)d^{s+1}$

4.2 Quadratic Case

For degree-2 (aka quadratic) functions h, it may be observed that we have

$$\varphi_h^{(2)}(x_i, x_j) = h(x_i + x_j + r) + h(x_i + r) + h(x_j + r) + h(r) \tag{10}$$

for every $(x_i, x_j, r) \in \big(\mathbb{F}_{2^n}\big)^3$ (this holds since $\varphi_h^{(3)}(x_i, x_j, r) = 0$ for quadratic functions). This observation and Theorem 1 imply the following equality for any integer $d \geqslant s$ and any $r_{i,j}$ values in $\mathbb{F}_{2^n}$:

$$h\Big(\sum_{i\in[\![1,d]\!]} x_i\Big) = \sum_{1\leqslant i<j\leqslant d} \big(h(x_i + x_j + r_{i,j}) + h(x_j + r_{i,j}) + h(x_i + r_{i,j}) + h(r_{i,j})\big) + \sum_{i\in[\![1,d]\!]} h(x_i) + \big((d+1) \bmod 2\big)\cdot h(0) \ . \tag{11}$$

Equality (11) shows that the evaluation of a quadratic function in a sum of d shares can be split into a sum of terms depending on at most two shares x_i and x_j. In this particular case it is then possible to use an improved sharing compression inspired from ISW, which gives Algorithm 2. Note that in Step 4 the additions must be computed from left to right in order to ensure the probing security.

This algorithm is actually already known from [8]. However the authors only suggest to use it for the secure evaluation of a multiplication of the form $x \cdot \ell(x)$ where ℓ is a degree-1 function (linear or affine). We show here that this algorithm can actually be used to securely compute *any* degree-2 function. The only difference is that one must add $h(0)$ to one output share whenever d is even (this term does not appear in [8] since by definition of $h : x \mapsto x \cdot \ell(x)$ one always get $h(0) = 0$). We also describe in Appendix D a variant of this algorithm that trades some random number generation against a few more additions. The probing security of Algorithm 2 holds from the security proof provided in [8].

[3] This is a weak constraint for low algebraic degrees. For instance $s \leqslant 3$ gives $d \equiv s \bmod 4$, $s \leqslant 7$ gives $d \equiv s \bmod 8$, etc. The complexity for any $d > s$ is given in Appendix G.1.

Algorithm 2 : Secure evaluation of a quadratic function

```
Input  : the $d$-sharing $(x_1, x_2, \ldots, x_d)$ of $x$ in $\mathbb{F}_{2^n}$
Output: a $d$-sharing $(y_1, y_2, \ldots, y_d)$ of $y = h(x)$
1  for $i = 1$ to $d$ do
2      for $j = i + 1$ to $d$ do
3          $r_{i,j} \leftarrow^{\$} \mathbb{F}_{2^n}$; $r'_{i,j} \leftarrow^{\$} \mathbb{F}_{2^n}$
4          $r_{j,i} \leftarrow r_{i,j} + h(x_i + r'_{i,j}) + h(x_j + r'_{i,j}) + h((x_i + r'_{i,j}) + x_j) + h(r'_{i,j})$
5  for $i = 1$ to $d$ do
6      $y_i \leftarrow h(x_i)$
7      for $j = 1$ to $d$, $j \neq i$ do
8          $y_i \leftarrow y_i + r_{i,j}$
9  if $d$ is even then $y_1 = y_1 + h(0)$
10 return $(y_1, y_2, \ldots, y_d)$
```

Complexity. The following table summarizes the complexity of Algorithm 2 in terms of additions, evaluations of h, and sampled random values (we consider the worst case of d being even). For comparison, we also give the complexity of the ISW scheme for a single multiplication.

	# add	# eval$_h$	# mult	# rand
Algorithm 2	$\frac{9}{2}\,d(d-1)+1$	$d(2d-1)$	-	$d(d-1)$
ISW multiplication	$2d(d-1)$	-	d^2	$\frac{1}{2}d(d-1)$

As explained in Section 2.3, for some values of n, a lookup table of 2^n entries might be affordable while a lookup table of 2^{2n} entries is not (typically for $n = 8$ giving 256 bytes *vs.* 64 kilobytes). In such a situation, the cost of one evaluation of h is expected to be significantly lower than the cost of a multiplication. We hence expect Algorithm 2 to be more efficient than the ISW scheme. Moreover, as shown above, Algorithm 2 can be used to securely evaluate a degree-2 function with a polynomial representation possibly involving *many* multiplications, whereas the ISW scheme securely evaluates a *single* multiplication.

5 A Tree-based Method to Secure Low-Degree Functions

In this section we introduce another new scheme to secure the evaluation of any function $h \in \mathcal{B}_{n,m}$ of given algebraic degree s. The method only involves additions and evaluations of h (that may be tabulated depending on the context). As the previous method, its complexity is exponential in s which makes it suitable for low-degree functions only. We start by introducing the core ideas of our method with the simple case of a 2-probing secure evaluation (*i.e.* taking a 3-shared input) of a degree-2 function. Then, we show how to extend the approach to achieve arbitrary probing security. The study is completed in Appendix E by a generalization of our result to functions of any algebraic degree.

5.1 Two-Probing Security for Quadratic Functions

Let $h \in \mathcal{B}_{n,m}$ be a degree-2 function. We present hereafter a method to securely construct a 3-sharing (y_1, y_2, y_3) of $y = h(x)$ from a 3-sharing (x_1, x_2, x_3) of x. Since h is quadratic, its third-order derivatives are null (see Definition 1) which implies the following equality for every $x \in \mathbb{F}_{2^n}$ and every triplet $(r_1, r_2, r_3) \in \mathbb{F}_{2^n}^3$:

$$h(x) = \sum_{1\leqslant i\leqslant 3} h(x + r_i) + \sum_{1\leqslant i<j\leqslant 3} h(x + r_i + r_j) + h(x + r_1 + r_2 + r_3) \ , \qquad (12)$$

or equivalently

$$h(x) = \sum_{i=1}^{7} h(x + e^{(i)}) \ , \tag{13}$$

where $e^{(i)}$ denotes the scalar product $\omega_i \cdot (r_1, r_2, r_3)$ with ω_i being the binary representation of the index i (*e.g.* $\omega_3 = (0,1,1)$). We shall say in the following that the family $(e^{(i)})_{i\in[\![1,7]\!]}$ is 3-*spanned* from (r_1, r_2, r_3) (a formal definition is given hereafter). Replacing x by the sum of its shares then leads to:

$$h(x_1 + x_2 + x_3) = \sum_{i=1}^{7} h(x_1 + x_2 + x_3 + e^{(i)}) \ . \tag{14}$$

It may be checked that the tuple $(h_i)_{i\in[\![1,7]\!]} = \big(h(x + e^{(i)})\big)_{i\in[\![1,7]\!]}$ is a $(7,3)$-sharing of $h(x)$ (this holds since the rank of $(e^{(i)})_{i\in[\![1,7]\!]}$ is at most 3). However, a direct evaluation of the h_i would yield an obvious second-order flaw. Indeed, for every $i \in \{1,2,3\}$, the evaluation of at least one of the terms in (14) implies the computation of $x + r_i$ which can be combined with r_i to recover x. A natural solution to avoid such a second-order flaw is to split the processing of each $x + e^{(i)}$ into several parts, which actually amounts to randomly split each $e^{(i)}$ into 3 shares $e_1^{(i)}$, $e_2^{(i)}$, $e_3^{(i)}$. This leads to the following expression:

$$h(x) = \sum_{i=1}^{7} h\Big((x_1 + e_1^{(i)}) + (x_2 + e_2^{(i)}) + (x_3 + e_3^{(i)})\Big) \ . \tag{15}$$

It then just remains to securely turn the $(7,3)$-sharing $(h_i)_{i\in[\![1,7]\!]}$ into a 3-sharing $(y_i)_{i\in[\![1,3]\!]}$. This is done by using the following sharing compression procedure specific to the case 7 to 3:

1. $(m_1, m_2, m_3) \leftarrow^{\$} \mathbb{F}_{2^n}^q$
2. $y_1 \leftarrow (h_1 + m_1) + (h_4 + m_2) + h_7$
3. $y_2 \leftarrow (h_2 + m_1) + (h_5 + m_3)$
4. $y_3 \leftarrow (h_3 + m_2) + (h_6 + m_3)$

The algorithmic description of the overall method for 3-sharing is then depicted in Algorithm 3 (where the RandSpan procedure is given hereafter).

Algorithm 3 : Secure evaluation of h on a 3-sharing.

```
Input: a 3-sharing $(x_1, x_2, x_3)$ of $x \in \mathbb{F}_{2^n}$
Output: a 3-sharing $(y_1, y_2, y_3)$ of $h(x)$
1 for $j = 1$ to 3 do
2     $(e_j^{(1)}, e_j^{(2)}, \ldots, e_j^{(7)}) \leftarrow \mathsf{RandSpan}(3, n)$
3 for $i = 1$ to 7 do
4     $c_i \leftarrow (x_1 + e_1^{(i)}) + (x_2 + e_2^{(i)}) + (x_3 + e_3^{(i)})$
5     $h_i \leftarrow h(c_i)$
6 $(y_1, y_2, y_3) \leftarrow \mathsf{SharingCompress}[7:3](h_1, h_2, \ldots, h_7)$
7 return $(y_1, y_2, y_3)$
```

5.2 Arbitrary Probing Security for Quadratic Functions

Before addressing the general case of an arbitrary d-sharing, let us introduce the notion of *random s-spanned family* which will be useful in the following.

Definition 3 (Random s-spanned family over $\mathbb{F}_{2^n}$). *Let n and s be two positive integers such that $s \leqslant n$. A* random s-spanned family over $\mathbb{F}_{2^n}$ *is a family of $2^s - 1$ random variables $\{e^{(i)} = \omega_i \cdot (r_1, r_2, \ldots, r_s) \mid 1 \leqslant i \leqslant 2^s - 1\}$ where $(r_1, r_2, \ldots, r_s)$ is a uniform random vector over $\mathbb{F}_{2^n}^s$.*

A random s-spanned family E is a linear subspace of $\mathbb{F}_{2^n}$ with $\dim(E) \leqslant s$. The following algorithm shall be used for sampling a random s-spanned family:

```
Algorithm 4 : RandSpan
   Input: two integers $s$ and $n$ such that $s \leqslant n$
   Output: a random $s$-spanned family $(e^{(1)}, e^{(2)}, \dots, e^{(2^s-1)})$ over $\mathbb{F}_{2^n}$
1  for $i = 1$ to $s$ do
2      $r_i \xleftarrow{\$} \mathbb{F}_{2^n}$
3  for $i = 1$ to $2^s - 1$ do
4      $e^{(i)} \leftarrow \omega_i \cdot (r_1, r_2, \dots, r_s)$
5  return $(e^{(1)}, e^{(2)}, \dots, e^{(2^s-1)})$
```

Remark 4. Note that d random s-spanned families form a d-sharing of a random s-spanned family, whereas the converse is false: a d-sharing of a random s-spanned family does not necessarily give d random s-spanned families. However, for our purpose, it will be sufficient to generate d random s-spanned families whenever we need a d-sharing of a random s-spanned family.

The idea in previous section can be extended to any security order d by starting from the following equation which generalizes (15):

$$h(x) = \sum_{i=1}^{7} h\Big(\sum_{j=1}^{d} (x_j + e_j^{(i)})\Big) \ , \tag{16}$$

where $(x_j)_{j \in [\![1,d]\!]}$ (resp. $(e_j^{(i)})_{j \in [\![1,d]\!]}$) is a d-sharing of x (resp. of the ith element of a random 3-spanned family).

It may be checked that the family $(h_i)_i := (h(\sum_{j=1}^{d} x_j + e^{(i)}))_i$ is a $(7,3)$-sharing of $h(x)$. However, the direct evaluation of (16) is not yet satisfactory since it implies the occurrence of several triplets of elements whose sum equals x. Indeed, for any triplet of distinct indices $(i_1, i_2, i_3) \in [\![1,7]\!]$ such that $i_1 = i_2 \oplus i_3$, we have:

$$\Big(\sum_{j=1}^{d} x_j + e_j^{(i_1)}\Big) + \Big(\sum_{j=1}^{d} x_j + e_j^{(i_2)}\Big) + \Big(\sum_{j=1}^{d} x_j + e_j^{(i_3)}\Big) = \sum_{j=1}^{d} x_j = x \ .$$

To deal with the above issue, each evaluation of h in (16) is split over a different (and independent) space. This leads to the following evaluation formula:

$$h(x) = \sum_{i_1=1}^{7} \sum_{i_2=1}^{7} h\Big(\sum_{j=1}^{d} x_j + e_j^{(i_1)} + e_j^{(i_1,i_2)}\Big) \ , \tag{17}$$

where, for every $j \in [\![1,d]\!]$, the family $(e_j^{(i_1)})_{i_1 \in [\![1,7]\!]}$ is randomly 3-spanned and, for every $i_1 \in [\![1,7]\!]$, the family $(e_j^{(i_1,i_2)})_{i_2 \in [\![1,7]\!]}$ is randomly 3-spanned.

Now, the family composed of all the intermediate results appearing during the processing of the sum in (17) is free from 3rd-order flaws. However, by the same reasoning as above, one can reconstruct $\sum_j x_j + e_j^{(i_1)}$ for some i_1 by summing three elements $\sum_j x_j + e_j^{(i_1)} + e_j^{(i_1,i_2)}$ corresponding to three different values of i_2 which are linearly dependent. Repeating this for three different values of i_1 which are linearly dependent eventually leads to the recovery of $x = \sum_j x_j$. This flaw, which involves 3×3 intermediate variables of the processing, is of order 9. To deal with this issue, we propose to apply the derivation approach (Equation (16)) recursively t times, leading to:

$$h(x) = \sum_{i_1=1}^{7} \sum_{i_2=1}^{7} \cdots \sum_{i_t=1}^{7} h\Big(\sum_{j=1}^{d} x_j + e_j^{(i_1)} + e_j^{(i_1,i_2)} + \cdots + e_j^{(i_1,i_2,\cdots,i_t)}\Big) \ , \tag{18}$$

where for every $(j, q) \in [\![1, d]\!] \times [\![1, t]\!]$ and every tuple $(i_1, \cdots, i_{q-1}) \in [\![1, 7]\!]^{q-1}$ the family $(e_j^{(i_1,\cdots,i_{q-1},i_q)})_{i_q \in [\![1,7]\!]}$ is randomly 3-spanned. A similar flaw as the one previously exhibited can still be applied but its order is now 3^t. Taking $t = \lceil \log_3 d \rceil$ is hence sufficient to avoid any nontrivial flaw. For the sake of simplicity, we assume hereafter that d is a power of 3 (so that $t = \log_3 d$) and we detail all the steps of our approach to securely evaluate a quadratic function at any order $d - 1$.

Our method first consists in computing the elements $x_j + e_j^{(i_1)} + e_j^{(i_1,i_2)} + \cdots + e_j^{(i_1,i_2,\cdots,i_t)}$ for every j and every $(i_1, i_2, \cdots, i_t) \in [\![1, 7]\!]^t$. For security reasons, this computation must be done carefully and can be represented by the descent of a tree whose nodes have all 7 children which are indexed from 1 to 7 (left to right). A node is itself labeled by a tuple listing the indices of the edges connecting it to the root. By convention, the label of the root is fixed to (0). The notations are illustrated in Figure 1 for $t = 3$.

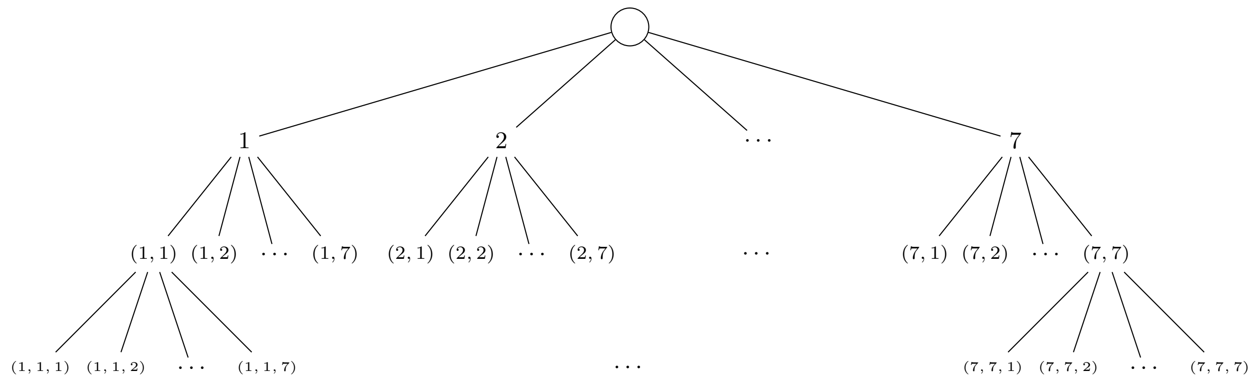

Fig. 1. Tree of indices $(i_1, i_2, \ldots, i_t)$ for $t = 3$.

Each node takes a d-tuple as input (starting with the d-sharing $(x_1, x_2, \cdots, x_d)$ of x for the root) and it outputs 7 new d-tuples (one per child node) generated from d random 3-spanned families. This sequence of operations is detailed hereafter for a node labeled by a tuple $u \in [\![1, 7]\!]^*$ which receives $(z_1, z_2, \ldots, z_d)$ as input:

- for every $j \in [\![1, d]\!]$ generate a random 3-spanned family $\{e_j^{(u||i)} \mid i \in [\![1, 7]\!]\}$
- for every $i \in [\![1, 7]\!]$ send $(z_1 + e_1^{(u||i)},\ z_2 + e_2^{(u||i)}, \ldots,\ z_d + e_d^{(u||i)})$ to the child with label $u||i$.

By construction, the leaf labeled $(i_1, i_2, \ldots, i_t) \in [\![1, 7]\!]^t$ receives a d-tuple $(z_1^{(i_1,i_2,\ldots,i_t)}, \cdots, z_d^{(i_1,i_2,\ldots,i_t)})$ whose coordinates satisfy

$$z_j^{(i_1,i_2,\ldots,i_t)} = x_j + e_j^{(i_1)} + e_j^{(i_1,i_2)} + \cdots + e_j^{(i_1,i_2,\cdots,i_t)} \quad , \tag{19}$$

for every $j \in [\![1, d]\!]$.

The second step of the tree processing simply consists for each leaf to sum the d elements of the received tuple and to apply the function h to the result. A leaf with label $(i_1, i_2, \ldots, i_t)$ hence outputs a value $h^{(i_1,i_2,\cdots,i_t)}$ satisfying:

$$h^{(i_1,i_2,\cdots,i_t)} := h(\sum_{j=1}^{d} z_j^{(i_1,i_2,\cdots,i_t)}) = h\Big(\sum_{j=1}^{d} x_j + e_j^{(i_1)} + e_j^{(i_1,i_2)} + \cdots + e_j^{(i_1,i_2,\cdots,i_t)}\Big) \ .$$

Due to (18) the family $\{h^{(i_1,i_2,\cdots,i_t)} \mid (i_1, i_2, \cdots, i_t) \in [\![1, 7]\!]^t\}$ forms a $(7^t, 3^t)$-sharing of $h(x)$. To turn this $(7^t, 3^t)$-sharing into a 3^t-sharing of $h(x)$ (which leaves us with a d-sharing of $h(x)$ since $t = \log_3(d)$) the tree is browsed from the leaves to the root, converting a 7-tuple associated to each node into a triplet thanks to SharingCompress[7 : 3] algorithm. Specifically, on level $t-1$, a node labeled $(i_1, i_2, \cdots, i_{t-1}) \in [\![1,7]\!]^{t-1}$ processes the following conversion:

$$\big(h_{(1)}^{(i_1,i_2,\cdots,i_{t-1})},\ h_{(2)}^{(i_1,i_2,\cdots,i_{t-1})},\ h_{(3)}^{(i_1,i_2,\cdots,i_{t-1})}\big) \leftarrow \mathsf{SharingCompress}[7:3]\big(h^{(i_1,i_2,\cdots,i_{t-1},1)}, \cdots, h^{(i_1,i_2,\cdots,i_{t-1},7)}\big)$$

Then, on level $t-2$, the node labeled $(i_1, i_2, \cdots, i_{t-2})$ gets the following (7×3)-tuple of shares

$$\big(h_{(j)}^{(i_1,i_2,\cdots,i_{t-2},1)},\ h_{(j)}^{(i_1,i_2,\cdots,i_{t-2},2)},\ \ldots,\ h_{(j)}^{(i_1,i_2,\cdots,i_{t-2},7)}\big)_{j\in\{1,2,3\}} .$$

For every $j \in \{1,2,3\}$, the underlying 7-tuple is converted into a triplet of shares by calling once again the sharing compression algorithm. We hence get the following (3×3)-tuple of shares

$$\big(h_{(j_1,j_2)}^{(i_1,i_2,\cdots,i_{t-2})}\big)_{(j_1,j_2)\in\{1,2,3\}^2} .$$

Similarly, on level $t-k$, each node transforms a $(7 \times 3^{k-1})$-tuple of shares into a 3^k-tuple

$$\big(h_{(j_1,j_2,\cdots,j_k)}^{(i_1,i_2,\cdots,i_{t-k})}\big)_{(j_1,j_2,\cdots,j_k)\in\{1,2,3\}^k} .$$

Eventually, the root receives a 3^t-sharing $(h_{(j_1,\cdots,j_t)}^{(\emptyset)})_{(j_1,j_2,\cdots,j_t)\in\{1,2,3\}^t}$ of $h(x)$.

The overall process can be performed by calling the following TreeExplore algorithm on the input d-sharing $(x_1, x_2, \ldots, x_d)$. This algorithm is recursive with a depth parameter k ranging from $t = \lceil \log_3(d) \rceil$ to 0. It first generates seven new tuples, then it calls itself on the seven generated tuples with depth parameter $k-1$ and get seven (3^{k-1})-tuples in return, and finally it converts the obtained $(7 \times 3^{k-1})$-tuple of shares in a 3^k-tuple of shares. If the depth parameter is 0, meaning that the recursion has reached a leaf, the algorithm just sums the elements of input tuple and applies h.

Algorithm 5 : TreeExplore

```
Input: a $d$-sharing $(z_1, z_2, \ldots, z_d)$ of $z \in \mathbb{F}_{2^n}$, a depth parameter $k$
Output: a $3^k$-sharing of $h(z)$
1 if $k = 0$ then
2     return $h(z_1 + z_2 + \cdots + z_d)$
3 for $j = 1$ to $d$ do
4     $(e_j^{(1)}, e_j^{(2)}, \ldots, e_j^{(7)}) \leftarrow$ RandSpan(3)
5 for $i = 1$ to 7 do
6     $(h_{(w)}^{(i)})_{w\in\{1,2,3\}^{k-1}} \leftarrow$ TreeExplore$(z_1 + e_1^{(i)}, z_2 + e_2^{(i)}, \ldots, z_d + e_d^{(i)}, k-1)$
7 for $w$ in $\{1,2,3\}^{k-1}$ do
8     $(h_{(w||1)}, h_{(w||2)}, h_{(w||3)}) \leftarrow$ SharingCompress[7 : 3]$(h_{(w)}^{(1)}, h_{(w)}^{(2)}, \ldots, h_{(w)}^{(7)})$
9 return $(h_{(w)})_{w\in\{1,2,3\}^k}$
```

Complexity. The tree contains $\sum_{k=0}^{t-1} 7^k = \frac{7^t-1}{6}$ nodes and 7^t leafs (where $7^t = d^{\ln 7/\ln 3} \approx d^{1.77}$). Each node makes d calls to RandSpan(3), we hence have a total of $\frac{d}{6}(7^t-1)$ calls, each involving 5 additions and the generation of 3 random elements. A node at level $t-k$ also makes 3^{k-1} calls to SharingCompress[7 : 3], which makes a total of $\sum_{i=0}^{t-1} 7^i 3^{t-1-i} = \frac{7^t-3^t}{4}$ calls, each involving 10 additions and the generation of 3 random elements. Additionally, each node performs $7d$ additions to generate the new sharings and each leaf performs d additions and one evaluation of h.

The following table summarizes the complexity of our method in terms of additions, evaluation of h, and random generation over $\mathbb{F}_{2^n}$.

	# add	# eval$_h$	# rand
Exact count	$3d \cdot 7^t - 2d + \frac{5}{2}(7^t - 3^t)$	7^t	$\frac{d}{2}7^t - \frac{d}{2} + \frac{3(7^t-3^t)}{4}$
Approximation	$3d^{2.77}$	$d^{1.77}$	$\frac{1}{2}d^{2.77}$

Security. The probing security of our tree-based method is proven for the general case in Appendix F.

6 Adapting the CRV Method to Low Algebraic Degrees

This section proposes an adaptation of Coron-Roy-Vivek's method (CRV) with improved complexity for functions with given (low) algebraic degree. We start by recalling the original method [9].

6.1 The CRV Method

In the following, we shall view functions over $\mathcal{B}_{n,m}$ as polynomials over $\mathbb{F}_{2^n}[x]$. Let $h(x) = \sum_{j=0}^{2^n-1} a_j x^j$ be the function that must be securely evaluated. To find a representation of $h(x)$ that minimizes the number of nonlinear multiplications, CRV starts by building the union set

$$\mathcal{L} = \bigcup_{i=1}^{\ell} \mathcal{C}_{\alpha_i} \quad \text{s.t. } \alpha_1 = 0,\ \alpha_2 = 1, \text{ and } \alpha_{i+1} \in \bigcup_{j=1}^{i} \mathcal{C}_{\alpha_j} + \bigcup_{j=1}^{i} \mathcal{C}_{\alpha_j} \text{ for every } i \leqslant \ell \ , \tag{20}$$

where $\mathcal{C}_{\alpha_i}$ is the cyclotomic class of α_i defined as $\mathcal{C}_{\alpha_i} = \{2^j \cdot \alpha_i \bmod (2^n - 1)\,;\, j \in [\![0, n-1]\!]\}$. The elements in $\{x^\alpha;\ \alpha \in \mathcal{L}\}$ can then be processed with only $\ell - 2$ nonlinear multiplications (since for $\alpha_1 = 0$ and $\alpha_2 = 1$ the building requires no nonlinear multiplication). The set $\mathcal{L}$ must satisfy the constraint $\mathcal{L} + \mathcal{L} = [\![0, 2^n - 1]\!]$ and, if possible, the ℓ classes $\mathcal{C}_{\alpha_i}$ in $\mathcal{L}$ are chosen such that $|\mathcal{C}_{\alpha_i}| = n$.

Let $\mathcal{P} \subseteq \mathbb{F}_{2^n}[x]$ be the subspace spanned by the monomials x^α with $\alpha \in \mathcal{L}$. The second step of CRV consists in randomly generating $t - 1$ polynomials $q_i(x) \in \mathcal{P}$ and in searching for t polynomials $p_i(x) \in \mathcal{P}$ such that

$$h(x) = \sum_{i=1}^{t-1} p_i(x) \times q_i(x) + p_t(x) \ . \tag{21}$$

This gives a linear system with 2^n equations (one for each $x \in \mathbb{F}_{2^n}$) and $t \times |\mathcal{L}|$ unknowns (the coefficients of the p_i). Such a system admits a solution for every choice of h if its rank is 2^n, which leads to the necessary condition $t \times |\mathcal{L}| \geqslant 2^n$. Finding such a solution provides a method to evaluate h involving $\ell + t - 3$ multiplications: $\ell - 2$ multiplications to generate the monomial $(x^j)_{j \in \mathcal{L}}$, from which $p_i(x)$ and $q_i(x)$ are computed as linear combinations for every $i \leqslant t$, and $t - 1$ multiplications to evaluate (21). In order to optimize the number of linear operations, the polynomials p_i can be represented as $p_i(x) = \sum_{\alpha_j \in \mathcal{L}} \ell_{i,j}(x^{\alpha_j})$ where the $\ell_{i,j}$ are linearized polynomials (*i.e.* polynomials of algebraic degree 1) that might be tabulated.

Complexity. Assuming that all the cyclotomic classes in $\mathcal{L}$ except $\mathcal{C}_0$ have maximum size n (*i.e.* $|\mathcal{L}| = 1 + n \times (\ell - 1)$) and that the lower bound $t \times |\mathcal{L}| \geqslant 2^n$ is reached, it is argued in [9] that the complexity of CRV is minimized for $t = \left\lceil \sqrt{2^n/n}\, \right\rceil$ and $\ell = \left\lceil \sqrt{2^n/n} - 1/n + 1 \right\rceil$. Moreover, it is empirically shown in [9] that these lower bounds are often achieved for $n \leqslant 12$. Using ISW to secure nonlinear multiplications we get the following complexity (where #eval LP denotes the number of evaluations of a linearized polynomial):

#add	#eval LP	#rand	#mult
$2d^2(t + \ell - 3) + d(t\ell - 2t - 3\ell + 5)$	$d\ell t$	$\frac{d(d-1)(t+\ell-3)}{2}$	$d^2(t + \ell - 3)$

Remark 5. Some of the $\ell - 2$ nonlinear multiplications used to generate the set of powers $\{x^\alpha; \alpha \in \mathcal{L}\}$ might take the form $x^{\alpha_i} \cdot (x^{\alpha_i})^{2^j}$ with $\alpha_i \in \mathcal{L}$. In that case, the CPRR scheme (Algorithm 2) may be preferred to

ISW. Indeed, as discussed in Section 4.2, this algorithm may be more efficient than ISW when $x \mapsto x \cdot x^{2^j}$ can be tabulated whereas $(x, y) \mapsto x \cdot y$ cannot. The same observation applies for the tweaked CRV method proposed in the next section for low-degree functions. In the complexity comparisons discussed in Section 7, this optimization is used (we have actually observed that it was always possible to entirely build $\mathcal{L}$ based on $\ell - 2$ multiplications of the form $x \mapsto x \cdot x^{2^j}$).

6.2 The CRV Method for Degree-s Functions

We propose hereafter an adaptation of the CRV method for functions with (low) algebraic degree s. For the generation of $\mathcal{L}$, the constraint becomes $\mathcal{L} + \mathcal{L} = \{\alpha \in [\![0, 2^n - 1]\!]\ ;\ \mathtt{HW}(\alpha) \leqslant s\}$. When s is even, we impose the additional condition that every cyclotomic class $\mathcal{C}_\alpha \subseteq \mathcal{L}$ verifies $\mathtt{HW}(\alpha) \leqslant \frac{s}{2}$ (the odd case is addressed later).

Then, as in the original method, we randomly generate $t - 1$ polynomials $q_i(x)$ in the subspace $\mathcal{P}$ (*i.e.* the subspace of polynomials spanned by the monomials x^j with $j \in \mathcal{L}$), and we try to solve a linear system obtained from (21). The difference is that the obtained system is of rank at most $\sum_{r=0}^{s} \binom{n}{r}$, *i.e.* the maximum number of non-null coefficients in a degree-s function. Here again if this maximal rank is achieved, then the system has a solution for every target vector *i.e.* for every degree-s function $h(x)$. The necessary condition to get a full-rank system then becomes $t \times |\mathcal{L}| \geqslant \sum_{r=0}^{s} \binom{n}{r}$. Assuming that the cyclotomic classes in $\mathcal{L}$ except $\mathcal{C}_0$ have maximum size n, this gives

$$t \geqslant \frac{\sum_{r=0}^{s} \binom{n}{r}}{n(\ell - 1) + 1} \ .$$

If this bound is reached, the number $t + \ell - 3$ of nonlinear multiplications is minimized by taking

$$t = (\ell - 1) \approx \Big(\frac{1}{n} \sum_{r=0}^{s} \binom{n}{r}\Big)^{1/2} \tag{22}$$

and we get $t + \ell - 3 \approx 2\big(\frac{1}{n} \sum_{r=0}^{s} \binom{n}{r}\big)^{1/2} - 2$. When s is odd, the method is similar but $\mathcal{L}$ must contain some cyclotomic classes $\mathcal{C}_\alpha$ such that $\mathtt{HW}(\alpha) \leqslant \frac{s+1}{2}$ and the q_i are constructed from powers x^α such that $\mathtt{HW}(\alpha) \leqslant \frac{s-1}{2}$ (this ensures that the algebraic degree of $p_i(x) \cdot q_i(x)$ is at most s).

The complexity for this method is the same as for CRV (see table in the previous section), but for low-degree functions, the obtained parameters (t, ℓ) are significantly smaller. The obtained number of nonlinear multiplications are compared in the following table.

	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$
Original CRV	2	4	5	7	10	14	19
CRV for $s = 2$	2	2	2	3	3	3	3
CRV for $s = 3$	2	3	4	5	5	6	7

7 Comparison

In this section, we first study the practical complexity of the methods introduced in Sections 4, 5, and 6 to secure functions of low algebraic degrees (specifically of degrees $s = 2$ and $s = 3$). Then, we compare our algebraic decomposition method exposed in Section 3 with the CRV method (which is the best known alternative). The comparisons are done on specific examples where the dimension n fits with classical symmetric ciphers' s-boxes (*i.e.* $n \in \{4, 6, 8\}$) and the number d of shares ranges over $[\![2, 9]\!]$. The study is completed in Appendix G.2 by an asymptotic analysis (w.r.t parameters n, d, and s).

7.1 Low-Degree Functions

For the case $s = 2$, we compare Algorithm 2 (generalization of the CPRR scheme – see Section 4), Algorithm 5 (aka `TreeExplore` – see Section 5), and the tweaked CRV method for low-degree functions (aka `CRV-LD` – see Section 6). For the case $s = 3$, our first method (Algorithm 1) must be combined with a third-order secure evaluation method (primitive `SecureEval`) of degree-3 functions. For such a purpose, we either use Algorithm 5 (`TreeExplore`) or the tweaked CRV method (`CRV-LD`). The exact operations counts for these methods, which are recalled in Appendix G.1, are illustrated in Figures 2–4. Note that when $d \leqslant s$, Algorithm 1 is not specified and therefore cannot be applied. In our comparisons, we assumed that an addition, a table lookup and a random generation of n bits have the same cost 1.[4] For the multiplication, we considered various possible costs C. The case $C = 1$ corresponds to a device where the multiplication of two elements in $\mathbb{F}_{2^n}$ can be precomputed and stored in a table, hence taking $n2^{2n}$ bits of memory. When this is not possible (in a constraint environment and/or for too large values of n), the multiplication must be implemented and its cost essentially depends on the device architecture.[5] We believe to encompass most realistic scenarios by considering $C \in \{5, 10, 20\}$. Note that for the case $s = 3$, we used the improved CRV method based on CPRR for the generation of powers as suggested in Remark 5. We did not use it for $s = 2$ since a CPRR multiplication is similar to one call to Algorithm 2.

Case $(\mathbf{s = 2})$. Figures 2–4 clearly illustrate the superiority of Algorithm 2 for the secure evaluation of degree-2 functions at any order. The ranking between our second method (Algorithm 5 aka `TreeExplore`) and the tweaked CRV method (`CRV-LD`) depends on the sharing order d and the multiplication cost ratio C. For high values of C (*i.e.* when the multiplication cannot be tabulated), `TreeExplore` outperforms the tweaked CRV method. It is particularly interesting for a sharing order d lower than 4. However, due to its tree structure of depth $\log_3(d)$, it becomes slower as soon as d reaches 4. Moreover for higher values of d, it would not be competitive since its asymptotic complexity is more than quadratic in the sharing order.

Case $(\mathbf{s = 3})$. Figures 2–4 show the superiority of the tweaked CRV method for sharing orders greater than 3 and even for costly multiplications (*i.e.* $C = 20$). For small sharing orders $d \in \{2, 3\}$, Algorithm 5 (`TreeExplore`) is competitive (for the same reasons as for the degree-2 case). It appears that the use of our first method (Algorithm 1) for lowering the sharing order down to the algebraic degree is never interesting in this setting. We however think that this is not a dead-end point for this method and that the ideas used to obtain Algorithm 2 from the general description of the method could be used to get an improved version for the cubic case as well. We let this question open for further research on this subject.

7.2 High-Degree Functions

We now consider the algebraic decomposition method described in Section 3 and compare it to the CRV method. The following table summarizes the number $(\ell - 2) + (t - 1)$ of secure nonlinear multiplications involved in CRV, as well as the numbers r and t of secure evaluations of degree-s_1 functions and degree-s_2 functions in the algebraic decomposition method.

[4] In the context of side-channel countermeasures, generating n random bits usually amounts to read a n-bit value in a TRNG register.

[5] In [12], the authors explain that the processing of a field multiplication with the `CPU` instructions set requires between 20 and 40 cycles and they give some examples of implementations.

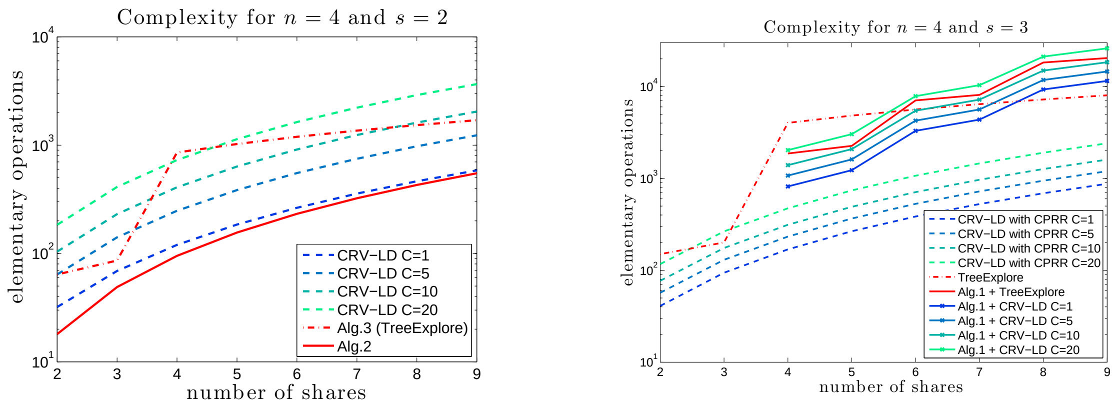

Fig. 2. Secure evaluation of quadratic and cubic functions for $n = 4$

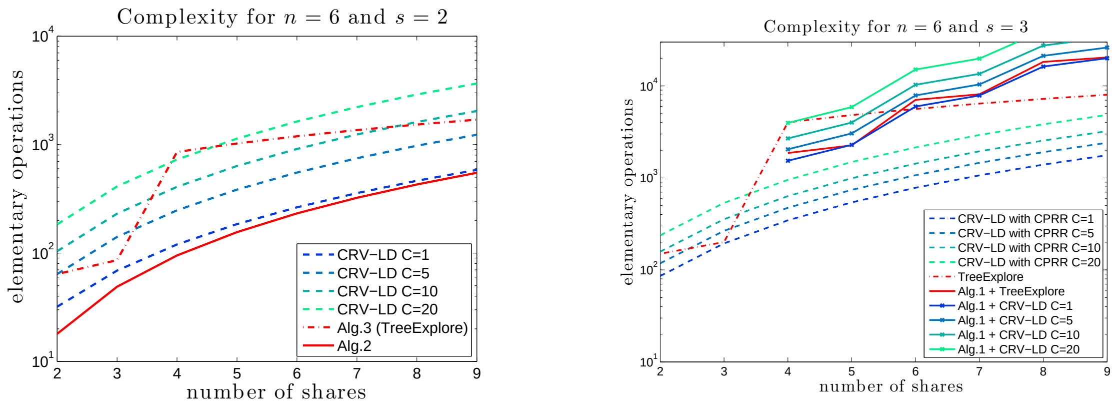

Fig. 3. Secure evaluation of quadratic and cubic functions for $n = 6$

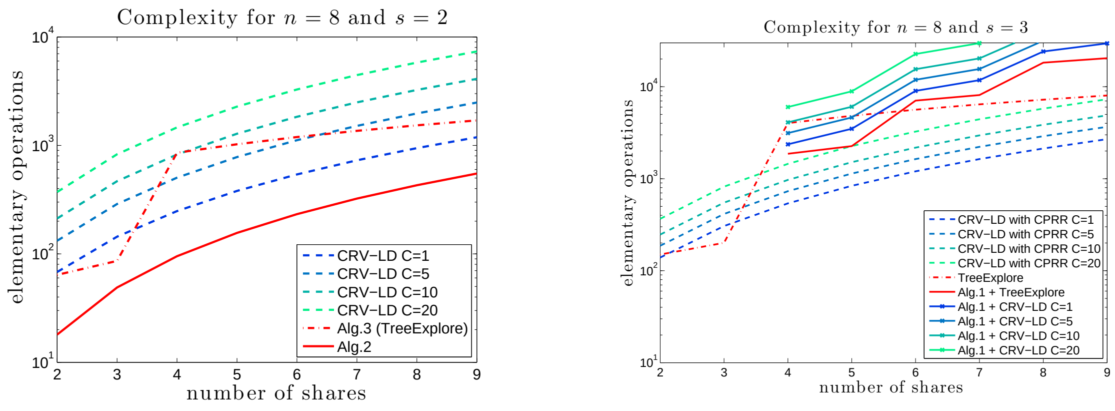

Fig. 4. Secure evaluation of quadratic and cubic functions for $n = 8$

	This Paper					CRV		
n	#eval-2	#eval-3	(s_1, s_2)	r	t	#mult	$\ell - 2$	$t - 1$
	3	-	$(2,2)$	1	2			
4	3	-	$(2,2)$	2	1	2	1	1
	1	1	$(2,3)$	1	1			
	-	2	$(3,3)$	1	1			
4*	-	1	$(1,3)$	0	1			
	4	-	$(2,2)$	2	2			
5	2	1	$(2,3)$	2	1	4	2	2
	-	3	$(3,3)$	1	2			
5*	3	-	$(2,2)$	1	2			
	5	-	$(2,2)$	2	3			
6	1	2	$(2,3)$	1	2	5	3	2
	-	3	$(3,3)$	1	2			
	8	-	$(2,2)$	2	6			
7	2	2	$(2,3)$	2	2	6	4	3
	-	4	$(3,3)$	1	3			
	-	4	$(3,3)$	2	2			
7*	1	2	$(2,3)$	1	2			
	11	-	$(2,2)$	2	9			
8	11	-	$(2,2)$	3	8	10	5	5
	2	3	$(2,3)$	2	3			
	-	4	$(3,3)$	1	3			

Figures 5–7 give the overall cost of the CRV method and of our algebraic decomposition method for various values of n, d and C. We used the improved version of CRV suggested in Remark 5 (*i.e.* using CPRR multiplications for the first phase and ISW multiplications for the second phase). The right-side graphics illustrate the gain obtained by using this improved version of CRV, while the left-side graphics compare our method to the improved CRV method. For our method, we considered quadratic decomposition (*i.e.* $s_1 = s_2 = 2$) combined with Algorithm 2 for secure quadratic evaluations, as well as cubic decomposition ($s_1 = s_2 = 3$) with `TreeExplore` for secure cubic evaluation. We further considered hybrid decomposition combined with both Algorithm 2 and `TreeExplore`.

We observe that the quadratic decomposition is always more efficient than the cubic and hybrid decompositions. This is due to the gap of efficiency between the secure quadratic evaluation (Algorithm 2) and the secure cubic evaluation (`TreeExplore`). Compared to CRV, the quadratic decomposition offers some efficiency gain depending on the multiplication cost. For $n = 4$, we observe that it is more efficient than CRV whenever a multiplication takes at least 10 elementary operations. For $n = 8$, the quadratic decomposition is better than CRV whenever the multiplication cost exceeds 5 elementary operations. This shows that our algebraic decomposition approach is the best known method for the probing secure evaluation of nonlinear functions (such as s-boxes) in a context where the field multiplication is a costly operation.

Acknowledgments

We would like to thank Jürgen Pulkus for a relevant remark that allowed us to make (7) tighter.

References

1. J. Balasch, S. Faust, and B. Gierlichs. Inner Product Masking Revisited. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 486–510. Springer, 2015.

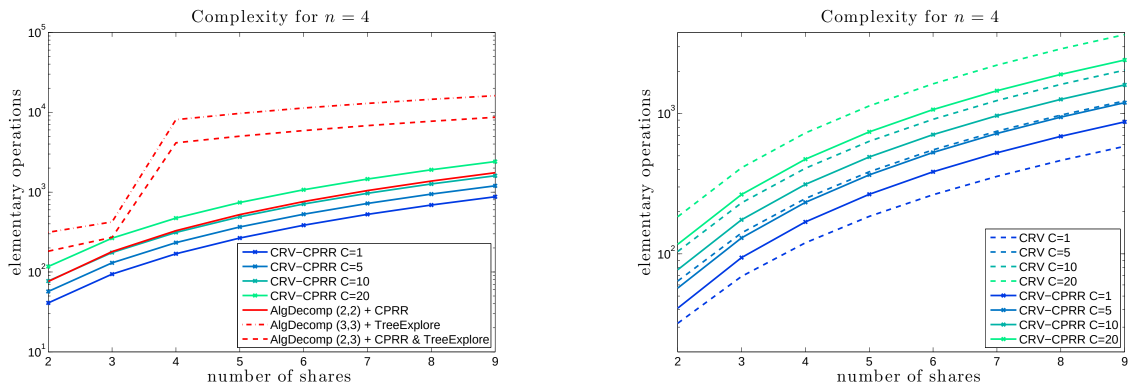

Fig. 5. Secure evaluation of nonlinear functions (arbitrary degree) for $n = 4$

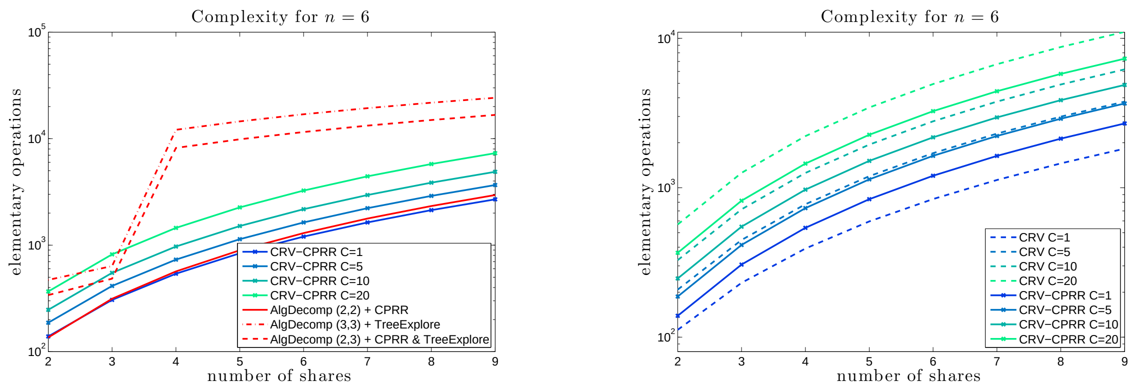

Fig. 6. Secure evaluation of nonlinear functions (arbitrary degree) for $n = 6$

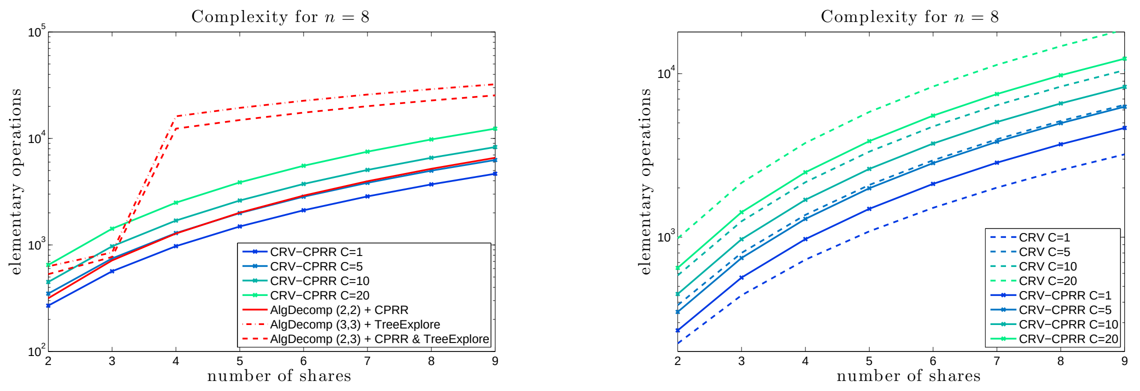

Fig. 7. Secure evaluation of nonlinear functions (arbitrary degree) for $n = 8$

2. G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, and P. Strub. Verified proofs of higher-order masking. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.
3. 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, *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2012.
4. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, N. Tokareva, and V. Vitkup. Threshold implementations of small S-boxes. *Cryptography and Communications*, 7(1):3–33, 2015.
5. C. Carlet. *Vectorial Boolean Functions for Cryptography*, chapter 9, pages 398–469. Boolean Models and Methods in Mathematics, Computer Science, and Engineering. Cambridge University Press, June 2010.
6. C. Carlet, L. Goubin, E. Prouff, M. Quisquater, and M. Rivain. Higher-order masking schemes for s-boxes. In A. Canteaut, editor, *FSE*, volume 7549 of *Lecture Notes in Computer Science*, pages 366–384. Springer, 2012.
7. J.-S. Coron. Fast Evaluation of Polynomials over Finite Fields and Application to Side-channel Countermeasures. Personnal communication., February 2014.
8. J.-S. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-order side channel security and mask refreshing. In S. Moriai, editor, *FSE*, Lecture Notes in Computer Science. Springer, 2013. To appear.
9. J.-S. Coron, A. Roy, and S. Vivek. Fast Evaluation of Polynomials over Finite Fields and Application to Side-channel Countermeasures. In *CHES*, 2014. To appear.
10. A. Duc, S. Dziembowski, and S. Faust. Unifying leakage models: from probing attacks to noisy leakage. In P. Q. Nguyen and E. Oswald, editors, *Eurocrypt*, Lecture Notes in Computer Science. Springer, 2014. To appear.
11. L. Genelle, E. Prouff, and M. Quisquater. Thwarting higher-order side channel analysis with additive and multiplicative maskings. In Preneel and Takagi [23], pages 240–255.
12. Grosso, Vincent and Prouff, Emmanuel and Standaert, François-Xavier . Efficient Masked S-Boxes Processing, A Step Forward. In D. Poincheval and D. Vergnaud, editors, *Africacrypt*, Lecture Notes in Computer Science. Springer, 2014. To appear.
13. Y. Ishai, A. Sahai, and D. Wagner. 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. Springer, 2003.
14. H. Kim, S. Hong, and J. Lim. A fast and provably secure higher-order masking of aes s-box. In Preneel and Takagi [23], pages 95–107.
15. D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, third edition, 1988.
16. P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
17. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
18. S. Kutzner, P. H. Nguyen, and A. Poschmann. Enabling 3-Share Threshold Implementations for all 4-Bit S-Boxes. In H. Lee and D. Han, editors, *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, volume 8565 of *Lecture Notes in Computer Science*, pages 91–108. Springer, 2013.
19. 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, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011.
20. 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*, volume 5461 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2009.
21. S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptology*, 24(2):292–321, 2011.
22. A. Poschmann, A. Moradi, K. Khoo, C. Lim, H. Wang, and S. Ling. Side-Channel Resistant Crypto for Less than 2, 300 GE. *J. Cryptology*, 24(2):322–345, 2011.
23. B. Preneel and T. Takagi, editors. *Cryptographic Hardware and Embedded Systems, 13th International Workshop – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*. Springer, 2011.
24. E. Prouff and M. Rivain. Higher-Order Side Channel Security and Mask Refreshing. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013 - 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2013.

25. E. Prouff and T. Roche. Higher-order glitches free implementation of the aes using secure multi-party computation protocols. In Preneel and Takagi [23], pages 63–78.
26. M. Rivain and E. Prouff. Provably secure higher-order masking of aes. In S. Mangard and F.-X. Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.
27. 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, *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 417–434. Springer, 2013.
28. A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In E. Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001.

A Proof of Theorem 1

Proof. Let us prove the theorem by induction on $d \geqslant s$. For $d = s$, the relation results in (8). Assuming it is verified for $s, \cdots, d$ we have:

$$\begin{aligned}
h\left(\sum_{i=1}^{d+1} a_i\right) &= h\left(\sum_{i=1}^{d-1} a_i + (a_d + a_{d+1})\right) \\
&= \sum_{1\leqslant i_1<\cdots<i_s\leqslant d-1} \varphi_h^{(s)}(a_{i_1},\ldots,a_{i_s}) + \sum_{1\leqslant i_1<\cdots<i_{s-1}\leqslant d-1} \varphi_h^{(s)}(a_{i_1},\ldots,a_{i_{s-1}},a_d+a_{d+1}) \\
&+ \sum_{j=0}^{s-1}\eta_{d,s}(j) \sum_{\substack{I\subseteq[\![1,d-1]\!]\\|I|=j}} h\left(\sum_{i\in I} a_i\right) + \sum_{j=1}^{s-1}\eta_{d,s}(j) \sum_{\substack{I\subseteq[\![1,d-1]\!]\\|I|=j-1}} h\left(\sum_{i\in I} a_i + a_d + a_{d+1}\right) \\
&= \sum_{1\leqslant i_1<\cdots<i_s\leqslant d-1} \varphi_h^{(s)}(a_{i_1},\ldots,a_{i_s}) + \sum_{1\leqslant i_1<\cdots<i_{s-1}\leqslant d-1} \varphi_h^{(s)}(a_{i_1},\ldots,a_{i_{s-1}},a_d) \\
&+ \sum_{1\leqslant i_1<\cdots<i_{s-1}\leqslant d-1} \varphi_h^{(s)}(a_{i_1},\ldots,a_{i_{s-1}},a_{d+1}) + \sum_{j=0}^{s-1}\eta_{d,s}(j) \sum_{\substack{I\subseteq[\![1,d-1]\!]\\|I|=j}} h\left(\sum_{i\in I} a_i\right) \\
&\qquad + \sum_{j=1}^{s-2}\eta_{d,s}(j) \sum_{\substack{I\subseteq[\![1,d+1]\!]\\|I|=j+1, d\in I, d+1\in I}} h\left(\sum_{i\in I} a_i\right) + \eta_{d,s}(s-1) \sum_{\substack{I\subseteq[\![1,d+1]\!]\\|I|=s, d\in I, d+1\in I}} h\left(\sum_{i\in I} a_i\right) .
\end{aligned}$$

The last equation above holds as $\varphi_h^{(s)}$ is s-linear. Eventually, applying (8) to develop the term of the last sum we get:

$$\begin{aligned}
h\left(\sum_{i=1}^{d+1} a_i\right) &= \sum_{\substack{1\leqslant i_1<\cdots<i_s\leqslant d+1\\ \{d,d+1\}\not\subseteq\{i_1,\ldots,i_s\}}} \varphi_h^{(s)}(a_{i_1},\ldots,a_{i_s}) + \sum_{j=0}^{s-1}\eta_{d,s}(j) \sum_{\substack{I\subseteq[\![1,d-1]\!]\\|I|=j}} h\left(\sum_{i\in I} a_i\right) \\
&\qquad + \sum_{j=1}^{s-2}\eta_{d,s}(j) \sum_{\substack{I\subseteq[\![1,d+1]\!]\\|I|=j+1, \{d,d+1\}\subseteq I}} h\left(\sum_{i\in I} a_i\right) \\
&\qquad + \eta_{d,s}(s-1) \sum_{\substack{1\leqslant i_1<\cdots<i_s\leqslant d+1\\ \{d,d+1\}\subseteq\{i_1,\ldots,i_s\}}} \left(\varphi_h^{(s)}(a_{i_1},\ldots,a_{i_s}) + \sum_{J\subsetneq\{1,\ldots,s\}} h\left(\sum_{j\in J} a_{i_j}\right)\right) ,
\end{aligned}$$

where "$\{d, d+1\} \not\subseteq \{i_1, \ldots, i_s\}$" means that at least one among $d, d+1$ is not in $\{i_1, \ldots, i_s\}$.

We recall that $\eta_{d,s}(j) = \binom{d-j-1}{s-j-1}$, therefore $\eta_{d,s}(s-1) = 1$ and we deduce:

$$\begin{aligned}
h\left(\sum_{i=1}^{d+1} a_i\right) &= \sum_{1\leqslant i_1<\cdots<i_s\leqslant d+1} \varphi_h^{(s)}(a_{i_1},\ldots,a_{i_s}) + \sum_{j=0}^{s-1}\eta_{d,s}(j) \sum_{\substack{I\subseteq[\![1,d-1]\!]\\|I|=j}} h\left(\sum_{i\in I} a_i\right) \\
&\qquad + \sum_{j=2}^{s-1}\eta_{d,s}(s-1) \sum_{\substack{I\subseteq[\![1,d+1]\!]\\|I|=j, \{d,d+1\}\subseteq I}} h\left(\sum_{i\in I} a_i\right) + \sum_{\substack{I\subseteq\{1,\ldots,d+1\}\\|I|=s, \{d,d+1\}\subseteq I}} \left(\sum_{I'\subsetneq I} h\left(\sum_{i\in I'} a_i\right)\right).
\end{aligned}$$

We have:

$$\sum_{\substack{I\subseteq [\![1,d+1]\!]\\ |I|=s,\{d,d+1\}\subseteq I}} \left(\sum_{I'\subsetneq I} h\left(\sum_{j\in I'} a_i\right)\right) = \sum_{j=0}^{s-2}\binom{d-j-1}{s-j-2} \sum_{\substack{I'\subseteq [\![1,d-1]\!]\\ |I'|=j}} h\left(\sum_{i\in I'} a_i\right)$$
$$+\sum_{j=2}^{s-1}\binom{d-j+1}{s-j} \sum_{\substack{I'\subseteq [\![1,d+1]\!]\\ |I'|=j, d\in I', d+1\in I'}} h\left(\sum_{i\in I'} a_i\right)$$
$$+\sum_{j=1}^{s-1}\binom{d-j}{s-j-1}\sum_{\substack{I'\subseteq [\![1,d]\!]\\ |I'|=j, d\in I'}} h\left(\sum_{i\in I'} a_i\right) + \sum_{j=1}^{s-1}\binom{d-j}{s-j-1} \sum_{\substack{I'\subseteq [\![1,d+1]\!]\\ |I'|=j, d\notin I', d+1\in I'}} h\left(\sum_{i\in I} a_i\right) ,$$

where the coefficients have to be taken modulo 2. This completes the proof of the induction, since $\eta_{d,s}(j) + \binom{d-j-1}{s-j-2} \equiv \eta_{d,s}(j-1) + \binom{d-j+1}{s-j} \equiv \binom{d-j}{s-j} + \binom{d-j+1}{s-j} \equiv \binom{d-j}{s-j-1}$ [mod 2]. Hence the property is satisfied for $d \geqslant s$. □

B Proof of Corollary 1

Proof. According to Theorem 1, we have:

$$h\left(\sum_{i=1}^{d} a_i\right) = \sum_{1\leqslant i_1<\cdots<i_s\leqslant d} \varphi_h^{(s)}(a_{i_1},\ldots,a_{i_s}) + \sum_{j=0}^{s-1}\eta_{d,s}(j) \sum_{\substack{I\subseteq [\![1,d]\!]\\ |I|=j}} h\left(\sum_{i\in I} a_i\right)$$
$$= \sum_{1\leqslant i_1<\cdots<i_s\leqslant d} \left(\sum_{J\subseteq\{1,\ldots,s\}} h\left(\sum_{j\in J} a_{i_j}\right)\right) + \sum_{j=0}^{s-1}\eta_{d,s}(j) \sum_{\substack{I\subseteq [\![1,d]\!]\\ |I|=j}} h\left(\sum_{i\in I} a_i\right)$$
$$= \sum_{j=0}^{s} \sum_{\substack{I\subseteq [\![1,d]\!]\\ |I|=j}} \binom{d-j}{s-j} h\left(\sum_{i\in I} a_i\right) + \sum_{j=0}^{s-1}\eta_{d,s}(j) \sum_{\substack{I\subseteq [\![1,d]\!]\\ |I|=j}} h\left(\sum_{i\in I} a_i\right)$$

with the convention $\binom{d-s}{0} = 1$ if $d \geqslant s$

$$= \sum_{\substack{I\subseteq [\![1,d]\!]\\ |I|=s}} h\left(\sum_{i\in I} a_i\right) + \sum_{j=0}^{s-1}\mu_{d,s}(j) \sum_{\substack{I\subseteq [\![1,d]\!]\\ |I|=j}} h\left(\sum_{i\in I} a_i\right),$$

since $\binom{d-j}{s-j} + \eta_{d,s}(j) \equiv \binom{d-j-1}{s-j}$ [mod 2]. □

C Sharing Compression

For every $k > d$, we propose hereafter a (k, d) compression primitive that takes a k-tuple $(x_1, x_2, \ldots, x_k)$ at input and computes a d-sharing of the value $\sum_{i=1}^{k} x_i$. We then show that this primitive achieves perfect $(d-1)$-probing security and give its complexity in terms of quantity of randomness and number of operations. In the specific case $d = 3$, we further propose an improved sharing compression method.

C.1 General Case

The generic (k, d) compression primitive is processed by calling $\lceil \frac{k-d}{d} \rceil$ times a $(2d, d)$ compression primitive — achieving perfect $(d-1)$-probing security — that we will use as building block. Our $(2d, d)$ sharing compression (*i.e.* SharingCompress$[2d : d]$) proceeds as follows:

1. for every $1 \leqslant i < j \leqslant d$, randomly sample $r_{i,j} \in \mathbb{F}_{2^n}$ and set $r_{j,i} = r_{i,j}$,
2. for every $1 \leqslant i, j \leqslant d$, compute $y_i = (x_i + (\sum_{j \neq i} r_{i,j})) + x_{i+d}$,

where the operations order is given by the brackets. The correctness holds from the fact that $\sum_{1 \leqslant i \leqslant d} y_i$ equals $\sum_{1 \leqslant i \leqslant d}(x_i + x_{d+i}) + \sum_{1 \leqslant i \leqslant d} \sum_{j \neq i} r_{i,j}$ that is $\sum_{1 \leqslant i \leqslant 2d} x_i$.

Now that we have a $(2d, d)$ sharing compression method, we simply iterate it to create a (k, d) sharing compression method. Let $K = d \cdot \lceil k/d \rceil$. Our (k, d) sharing compression method starts by setting $(y_1, y_2, \ldots, y_d) \leftarrow (x_1, x_2, \ldots, x_d)$ and then iterates

$$(y_1, \ldots, y_d) \leftarrow \mathsf{SharingCompress}[2d : d](y_1, \ldots, y_d, x_{jd+1}, \ldots, x_{jd+d})$$

for $j = 2, 3, \ldots, \lceil k/d \rceil$, where $x_i = 0$ for every $k < i \leqslant K$.

The algorithmic description of the method is given hereafter:

Algorithm 6 : $\mathsf{SharingCompress}[2d : d]$

```
Input   : a $2d$-sharing $(x_1, x_2, \ldots, x_{2d})$
Output: a $d$-sharing $(y_1, y_2, \cdots, y_d)$ s.t. $\sum_{i=1}^{d} y_i = \sum_{i=1}^{2d} x_i$
1  Randomly generate $\frac{d(d-1)}{2}$ elements in $\mathbb{F}_{2^n}$, $\{r_{i,j}\}_{1 \leqslant i < j \leqslant d}$
2  for $i = 1$ to $d$ do
3      for $j = 1$ to $i - 1$ do
4          $v_i \leftarrow v_i + r_{j,i}$
5      for $j = i + 1$ to $d$ do
6          $v_i \leftarrow v_i + r_{i,j}$
7  for $i = 1$ to $d$ do
8      $y_i \leftarrow x_i + v_i$
9      $y_i \leftarrow y_i + x_{i+d}$
10 return $(y_1, \ldots, y_d)$
```

Algorithm 7 : $\mathsf{SharingCompress}[k : d]$

```
Input   : a $k$-sharing $(x_1, x_2, \ldots, x_k)$
Output: a $d$-sharing $(y_1, y_2, \cdots, y_d)$ s.t. $\sum_{i=1}^{d} y_i = \sum_{i=1}^{k} x_i$
1  $K \leftarrow d \cdot \lceil k/d \rceil$
2  for $j = k$ to $K$ do
3      $x_j \leftarrow 0$
4  for $j = 1$ to $d$ do
5      $y_j \leftarrow x_j$
6  for $j = 1$ to $\frac{K-d}{d}$ do
7      $(y_1, \cdots, y_d)$
8          $\leftarrow \mathsf{SharingCompress}[2d : d](y_1, \cdots, y_d, x_{j*d+1}, x_{j*d+2}, \cdots, x_{j*d+d})$
9  return $(y_1, \ldots, y_d)$
```

Complexity. The method makes $\lceil k/d \rceil - 1$ calls to the $(2d, d)$ sharing compression procedure. Each of these calls samples $\frac{1}{2}d(d-1)$ fresh random values from $\mathbb{F}_{2^n}$ and processes $d(d+1)$ additions over $\mathbb{F}_{2^n}$. This makes a total of $(K - d)(d + 1) \approx k \cdot d$ additions and $\frac{1}{2}(K - d)(d - 1) \approx \frac{1}{2}k \cdot d$ random values.

#add	#rand
$d\lceil \frac{k}{d} \rceil(d+1)$	$\frac{1}{2}d\lceil \frac{k}{d} \rceil(d-1)$

Security. We state the perfect probing security of our sharing compression techniques in what follows:

Proposition 2. *The (k, d) sharing compression procedure (Algorithm 7) achieves perfect $(d-1)$-probing security.*

We first prove the security of our $(2d, d)$ sharing compression procedure.

Lemma 2. *The $(2d, d)$ sharing compression procedure (Algorithm 6) achieves perfect $(d-1)$-probing security.*

Proof. Let us consider a set of $\ell < d$ intermediate variables $\{v_1, v_2, \ldots, v_\ell\}$. First observe that an intermediate variable v_h can only take a limited number of forms which we classify in three categories:

1. $r_{i,j}$ and x_i
2. $\sum_{\substack{j\neq i \\ j\leqslant n}} r_{i,j}$ for some $n \leqslant d$ and $x_i + \sum_{j\neq i} r_{i,j}$
3. $x_i + \sum_{j\neq i} r_{i,j} + x_{i+d}$

We will show that such set of intermediate variables can be perfectly simulated with no more than ℓ inputs among $x_1, x_2, \ldots, x_{2d}$.

To this purpose we will first show that if more than ℓ inputs enter in the computation of $\{v_1, v_2, \ldots, v_\ell\}$ then there exists an intermediate variable $v_h \in \{v_1, v_2, \ldots, v_\ell\}$ and a random value $r_{i,j}$ such that $r_{i,j}$ enters in the computation of v_h but not in the computation of $v_{h'}$ for $h' \in [\![1, \ell]\!] \backslash \{h\}$.

Let $v_h \in \{v_1, v_2, \ldots, v_\ell\}$ be such that two input shares x_i and x_{i+d} enter in its computation (*i.e.* v_h is of the third form). Then, it can be checked that one of the following conditions is satisfied:

– (**C1**) there exists a fresh random $r_{i,j}$ that enters in the computation of no other intermediate variable than v_h;
– (**C2**) there exists an element $v_{h'}$ in $\{v_1, v_2, \ldots, v_\ell\} \setminus \{v_h\}$ such that all variables that enter in the computation of $v_{h'}$ also enter in the computation of v_h (in particular any input that enters in the computation of $v_{h'}$ also enters in the computation of v_h).

Indeed, if **C1** is not satisfied, then there exists an intermediate variable $v_{h'}$ in $\{v_1, v_2, \ldots, v_\ell\} \setminus \{v_h\}$ that involves in its computation at least two randoms $r_{i,j}$ and $r_{i,k}$ which also appear in v_h. Otherwise this would contradict[6] $\ell < d$. By construction, this is only possible if $v_{h'}$ is an intermediate sum in the process of building v_h.

From this statement we trivially get that if more than ℓ inputs enter in the computation of $\{v_1, v_2, \ldots, v_\ell\}$ then at least one intermediate variable v_h verifies **C1** and then there exists a fresh random $r_{i,j}$ that enters in the computation of no other intermediate variable than v_h.

Finally, the above identified v_h is independent from all the inputs and all the intermediate variables $\{v_1, v_2, \ldots, v_\ell\} \setminus \{v_h\}$ and therefore can be perfectly simulated by a fresh random value and discarded from the list of intermediate variables yet to simulate. The above argument can be applied iteratively until either there are fewer inputs that enter in the computation of the remaining list of intermediate variable than the number of elements in it or the list is empty.

This concludes the proof, since a set of intermediate variable of size smaller than d can always be reduced to a set of n intermediate variables where less than n inputs enters in their computation and therefore can be perfectly simulated from these inputs and by setting to random values the $r_{i,j}$ involved. □

[6] Since v_h is of form (3), then $d-1$ randoms $r_{i,j}$ enter its computation. For all of them to also enter in the computation of others v_k, while satisfying the condition $\ell < d$, there must exist $v_{h'}$, with $h' \neq h$, that involves two common randoms with v_h.

Proof (Proposition 2). Let us consider a set of $\ell < d$ intermediate variables $\{v_1, v_2, \ldots, v_\ell\}$. We will show that this set of intermediate variables can be perfectly simulated with no more than ℓ inputs among $x_1, x_2, \ldots, x_k$. By construction, any intermediate variable can be seen as an intermediate variable of the $\mathsf{SharingCompress}[2d : d]$ algorithm. Therefore, from Lemma 2, the set of, say n, intermediate variables that appear in a particular instance of $\mathsf{SharingCompress}[2d : d]$ can be mapped to a set of n or less inputs of the $\mathsf{SharingCompress}[2d : d]$ instance such that these inputs can perfectly simulate the n intermediate variables. Each of these n or less inputs is then considered as either input of the $\mathsf{SharingCompress}$ algorithm or as output (*i.e.* intermediate variable) of another instance of $\mathsf{SharingCompress}[2d : d]$. Starting from the last call to algorithm $\mathsf{SharingCompress}[2d : d]$ up to the first call, the intermediate variables $\{v_1, v_2, \ldots, v_\ell\}$ can be iteratively mapped to ℓ or less inputs of the algorithm $\mathsf{SharingCompress}$. This concludes the proof. ☐

C.2 Specific Case (k to 3)

The following algorithm provide an improve sharing compression for the specific case where one wants to compute a 3-sharing from a $(k, 3)$-sharing.

Algorithm 8 : $\mathsf{SharingCompress}[k : 3]$

```
Input: shares $(h_1, h_2, \ldots, h_k)$ with $(k > 3)$
Output: shares $(y_1, y_2, y_3)$
1 $q \leftarrow \lfloor \frac{k}{2} \rfloor$
2 $(m_1, m_2, \ldots, m_q) \leftarrow^{\$} \mathbb{F}_{2^n}^q$
3 $y_1 \leftarrow \sum_{i \bmod 3=1}^{k} (h_i + m_{\lceil i/2 \rceil})$
4 $y_2 \leftarrow \sum_{i \bmod 3=2}^{k} (h_i + m_{\lceil i/2 \rceil})$
5 $y_3 \leftarrow \sum_{i \bmod 3=0}^{k} (h_i + m_{\lceil i/2 \rceil})$
6 return $(y_1, y_2, y_3)$
```

In the above algorithm we define $m_{\lceil i/2 \rceil} := 0$ for every $\lceil \frac{i}{2} \rceil > q$. Since the sharing compression with $(k, d) = (7, 3)$ is intensively used in the paper, we further exhibit the algorithm for $\mathsf{SharingCompress}[7 : 3]$ hereafter:

Algorithm 9 : $\mathsf{SharingCompress}[7 : 3]$

```
Input: shares $(h_1, h_2, \ldots, h_7)$
Output: shares $(y_1, y_2, y_3)$
1 $m_1 \leftarrow^{\$} \mathbb{F}_{2^n}$
2 $m_2 \leftarrow^{\$} \mathbb{F}_{2^n}$
3 $m_3 \leftarrow^{\$} \mathbb{F}_{2^n}$
4 $y_1 \leftarrow (h_1 + m_1) + (h_4 + m_2) + h_7$
5 $y_2 \leftarrow (h_2 + m_1) + (h_5 + m_3)$
6 $y_3 \leftarrow (h_3 + m_2) + (h_6 + m_3)$
7 return $(y_1, y_2, y_3)$
```

Complexity. The complexity of Algorithm 8 is summarized in the following table:

#add	#rand
$(2k - 3) - (k \bmod 2)$	$\lfloor \frac{k}{2} \rfloor$

Security. The security of Algorithm 8 is stated hereafter:

Lemma 3. *The $(k,3)$ sharing compression procedure (Algorithm 8) achieves perfect 2-probing security.*

Proof. The whole set of intermediate variables of $\mathsf{SharingCompress}[k:3]$ algorithm (8) is as follows:

- $h_1, h_2, \ldots, h_k$
- $m_1, m_2, \ldots, m_q$
- $\sum_{i \bmod 3=1}^{n}(h_i + m_{\lceil i/2 \rceil})$ for some $n \leqslant k$
- $\sum_{i \bmod 3=2}^{n}(h_i + m_{\lceil i/2 \rceil})$ for some $n \leqslant k$
- $\sum_{i \bmod 3=0}^{n}(h_i + m_{\lceil i/2 \rceil})$ for some $n \leqslant k$

First case, the attacker observes a unique intermediate variable v:

- if v is of the form h_i, then this very input h_i is enough to perfectly simulate v.
- if v involves m_i in its computation then, since no other information about m_i is given to the attacker, he cannot distinguish v from a random value. In this case no knowledge about the input is necessary; v is perfectly simulated by a fresh random.

Second case, the attacker observe two intermediate variables v_1, v_2:

- if v_1 (resp. v_2) is of the form h_i, then assign h_i to v_1 (resp. v_2). If there is still one variable (say v_1) unassigned, then at least a fresh random m_i enters its computation. Since m_i enters the computation of no other intermediate variable, then assign a fresh random value to v_1.
- otherwise, if m_i enters in the computation of v_1 but not v_2, then both variables are mutually independent of the variables $\{h_1, h_2, h_3, h_4, h_5, h_6, h_7\}$; two fresh random values can be assigned to them.
- otherwise, v_1 and v_2 are necessarily either of the form $(h_i + x, h_j + x)$ or $(h_i + x, x)$, where at least one random m_k enters in the computation of x. In both cases the attacker cannot distinguish x from a random value. Therefore one may assign $(h_i + m, h_j + m)$ to (v_1, v_2) (resp. $(h_i + m, m)$ to (v_1, v_2)), where m is a fresh random value.

□

D A Variant of Algorithm 2

We describe hereafter a variant of Algorithm 2 that trades some random number generations against few more additions. Its security proof, which is similar to that given in [8], is not detailed here.

Algorithm 10 : Scheme for Quadratic Functions (Second Version)

```
Input  : the $d$-sharing $(x_1, x_2, \cdots, x_d)$ of $x$ in $\mathbb{F}_{2^n}$
Output: a $d$-sharing $(y_1, y_2, \cdots, y_d)$ of $y = h(x)$

1  Randomly generate $d(d-1)/2$ elements $r_{i,j} \in \mathbb{F}_{2^n}$ indexed such that $1 \leqslant i < j \leqslant d$
2  Randomly generate $d$ elements $\alpha_i \in \mathbb{F}_{2^n}$ indexed such that $1 \leqslant i \leqslant d$
3  Randomly generate $d-1$ elements $\beta_i \in \mathbb{F}_{2^n}$ indexed such that $1 \leqslant i \leqslant d-1$
4  for $i = 1$ to $d$ do
5      for $j = i+1$ to $d$ do
6          $r'_{i,j} \leftarrow \alpha_i + \beta_j$
7          $t \leftarrow r_{j,i} + h(x_i + r'_{i,j})$
8          $t \leftarrow t + h(x_j + r'_{i,j})$
9          $t \leftarrow t + h((x_i + r'_{i,j}) + x_j)$
10         $t \leftarrow t + h(r'_{i,j})$
11 for $i = 1$ to $d$ do
12     $y_i \leftarrow h(x_i)$
13     for $j = 1$ to $d$, $j \neq i$ do
14         $y_i \leftarrow y_i + r_{i,j}$
15 if $d$ is even then $y_1 = y_1 + h(0)$
16 return $(y_1, y_2, \ldots, y_d)$
```

E General Tree-based Method

In this section, the secure evaluation method previously exhibited for quadratic polynomials is extended to apply for any algebraic degree. It directly follows from the hereafter straightforward generalization of (16) and (18) to a degree-s polynomial $h \in \mathcal{B}_{n,m}$:

$$h\Big(\sum_{j=1}^{d} x_j\Big) = \sum_{i=1}^{2^s-1} h\Big(\sum_{j=1}^{d} x_j + e_j^{(i)}\Big) \ , \tag{23}$$

and

$$h\Big(\sum_{j=1}^{d} x_j\Big) = \sum_{i_1=1}^{2^{s+1}-1} \sum_{i_2=1}^{2^{s+1}-1} \cdots \sum_{i_t=1}^{2^{s+1}-1} h\Big(\sum_{j=1}^{d} x_j + e_j^{(i_1)} + e_j^{(i_1,i_2)} + \cdots + e_j^{(i_1,i_2,\cdots,i_t)}\Big) \ , \tag{24}$$

where for every $(j,q) \in [\![1,d]\!] \times [\![1,t]\!]$ and every tuple $(i_1, \cdots, i_{q-1}) \in [\![1, 2^{s+1}-1]\!]^{q-1}$ the family $(e_j^{(i_1,\cdots,i_{q-1},i_q)})_{i_q \in [\![1,2^{s+1}-1]\!]}$ is randomly $(s+1)$-spanned.

Starting from (24), the same reasoning as in Section 5.2 leads to the definition of the algorithm $\mathsf{TreeExplore}_s$ whose specification and complexity are given hereafter.

TreeExplore$_s$ When called for depth parameter k equal to $t = \lceil \log_3(d) \rceil$ and for a degree-s polynomial h, it builds a d-sharing of $h(x)$ from a d-sharing of x while ensuring $(d-1)$-probing security.

Algorithm 11 : $\mathsf{TreeExplore}_s(n, d)$

```
  Input: a function $h$ of algebraic degree $s$, a $d$-sharing $(z_1, z_2, \dots, z_d)$ of $z \in \mathbb{F}_{2^n}$, a depth parameter $k$
  Output: a $3^k$-sharing of $h(z)$
1 if $k = 0$ then
2     return $h(z_1 + z_2 + \cdots + z_d)$
3 for $j = 1$ to $d$ do
4     $(e_j^{(1)}, e_j^{(2)}, \dots, e_j^{(2^{s+1}-1)}) \leftarrow \mathsf{RandSpan}(s+1)$
5 for $i = 1$ to $2^{s+1} - 1$ do
6     $(h_{(w)}^{(i)})_{w \in \{1,2,3\}^{k-1}} \leftarrow \mathsf{TreeExplore}(h, s, z_1 + e_1^{(i)}, z_2 + e_2^{(i)}, \dots, z_d + e_d^{(i)}, k-1)$
7 for $j = 1$ to $3^{k-1}$ do
8     $(h_{(w||1)}, h_{(w||2)}, h_{(w||3)}) \leftarrow \mathsf{SharingCompress}[2^{s+1} - 1 : 3](h_{(w)}^{(1)}, h_{(w)}^{(2)}, \dots, h_{(w)}^{(2^{s+1}-1)})$
9 return $(h_{(w)})_{w \in \{1,2,3\}^k}$
```

Complexity. Let us denote by $\mu(s)$ the value $2^{s+1} - 1$. The tree contains $\sum_{k=0}^{t-1} \mu(s)^k = \frac{1}{\mu(s)-1}(\mu(s)^t - 1)$ nodes and $\mu(s)^t$ leafs (where $\mu(s)^t = d^{\ln(2^{s+1}-1)/\ln 3} \simeq d^{0.63(s+1)}$). Each node makes d calls to $\mathsf{RandSpan}(s+1)$, we hence have a total of $\frac{1}{\mu(s)-1}(\mu(s)^t - 1)d$ calls, each involving $\sum_{i=1}^{s+1} \binom{s+1}{i}(i-1) = (s-1)2^s + 1$ additions and the generation of $s+1$ random elements. A node at level $t-k$ also makes 3^{k-1} calls to $\mathsf{SharingCompress}[2^{s+1} - 1 : 3]$, which makes a total of $\sum_{i=0}^{t-1} \mu(s)^i 3^{t-1-i} = \frac{\mu(s)^t - 3^t}{\mu(s)-3}$ calls, each involving $2\mu(s) - 4$ additions and the generation of $\lfloor \frac{\mu(s)}{2} \rfloor$ random elements. Additionally, each node performs $\mu(s)d$ additions to generate the new sharings and each leaf performs d additions and one evaluation of h.

		Complexity
$\mathcal{C}_{\mathsf{TE}_s}$	#add	$\frac{\mu(s)^t-1}{\mu(s)-1} d \frac{(s+1)}{2}(\mu(s)+1) + \frac{\mu(s)^t-3^t}{\mu(s)-3}(2\mu(s)-4) + \mu(s)^t d$
	#rand	$\frac{\mu(s)^t-1}{\mu(s)-1} d(s+1) + \frac{\mu(s)^t-3^t}{\mu(s)-3} \left\lfloor \frac{\mu(s)}{2} \right\rfloor$
Exact Count	#mult	0
	#eval$_h$	$\mu(s)^t$
		where $t = \lceil \log_3 d \rceil$ and $\mu(s)^t = d^{\ln(2^{s+1}-1)/\ln 3}$.
$\mathcal{C}_{\mathsf{TE}_s}$	#add	$O(sd^{0.63(s+1)+1})$
	#rand	$O(sd \left(\frac{d^{0.63}}{2}\right)^{s+1})$
Approximation	#mult	0
	#eval$_h$	$O(d^{0.63(s+1)})$
		where $t = \log_3 d$ and $\mu(s)^t \simeq d^{0.63(s+1)}$.

F Security of the Tree-based Method

The argumentation is inspired from simulation proof procedures as done for instance in [13]. In the following, we assume that the security parameter d takes the form 3^t to ease the explanation, but the reasoning straightforwardly extends to the general case $d \in \mathbb{N}$. For any family $\mathcal{O}$ of at most $d-1$ intermediate results probed/observed by the adversary during the processing of $\mathsf{TreeExplore}$, our proof is composed of the following three main steps:

1. **assign**, to each of the $d-1$ observed intermediate variables, a sub-set $\mathcal{E} \subset [\![1,d]\!]$ and/or a sub-set $\mathcal{U} \subset [\![1,7^t]\!]$,
2. **simulate** the behaviour of all the intermediate variables (in a way which is indistinguishable from the adversary view) with only the knowledge of the shares in $\{x_i; i \in \mathcal{E}\}$ and $\{h^{(u)}; u \in \mathcal{U}\}$,
3. **prove** that the knowledge of the shares in $\{x_i; i \in \mathcal{E}\}$ and $\{h^{(u)}; u \in \mathcal{U}\}$ (where the $h^{(u)}$ are viewed as the shares of $h(x)$) gives no information about the secret x.

In the proof below, the recursive execution of TreeExplore is represented by a tree as in Figure 1. Each node (including the leaves) processes one instance of TreeExplore in the recursive processing started by the root. According to the notations introduced for Figure 1, a node at depth k is labelled with a $(t-k)$-tuple $u = (i_1, i_2, \cdots, i_{t-k}) \in [\![1,7]\!]^{t-k}$ that contains the indices of the edges connecting the node to the root. The size of a tuple u is denoted by $|u|$. By convention, we shall assume that the label of the root is 0. The tuple u is also used to label the variables manipulated by the uth node: intermediate results z_j, $e_j^{(i)}$, $h_{(w)}^{(i)}$ and $h_{(w)}$ in Algorithm 5 are therefore respectively denoted by $z_j^{(u)}$, $e_j^{(u||i)}$, $h_{(w)}^{(u||i)}$ and $h_{(w)}^{(u)}$, where $||$ denotes the concatenation. If u (resp. w) is the empty tuple of size 0, then we will simply the notation $h_{(w)}^{(u)}$ in $h_{(w)}$ (resp. $h^{(u)}$). We apply the same notation rule to denote the random values involved during the wth processing of SharingCompress[7 : 3] performed by the node labelled with u: namely the intermediate values m_i are denoted by $m_{(w||i)}^{(u)}$. Eventually, the set of all the tuples with coordinates in $[\![1,7]\!]$ (resp. $[\![1,3]\!]$) and size lower than or equal to t is denoted by $[\![1,7]\!]^\star$ (resp. $[\![1,3]\!]^\star$).

For the assignment and simulation steps of our proof we split the processing into three steps.

– The first step is dedicated to the construction of the values $z_j^{(u)}$ with $j \in [\![1,d]\!]$ and $u \in [\![1,7]\!]^k$ for $k \leqslant t$. This corresponds to the construction of the tree such as described in Figure 1 from the root to the leaves (or equivalently to the steps 3 to 6 – excluding the processing of the outputs $h_{(w)}^{(u||i)}$ – in Algorithm 5).
– The second step relates to the computation of $h(\sum_{j=1}^d z_j^{(u)})$ for every tuple $u \in [\![1,7]\!]^t$. This corresponds to the processing done by a leaf in the tree representation given in Figure 1 (or equivalently to the 2nd Step of Algorithm 5).
– The third step is dedicated to the construction of the values $h_{(w)}^{(u)}$ for $(u,w) \in [\![1,7]\!]^{t-k} \times [\![1,3]\!]^k$ and $k \leqslant t$. This corresponds to the secure conversion of the 7^t shares $h^{(u)}$ of $h(x)$, possessed by the leaves after the two first steps, into $d = 3^t$ shares $h_{(w)}$ of $h(x)$. Equivalently, it corresponds to the recursive processing of SharingCompress[7 : 3].

Assignment. The purpose of this step is to construct two sets $\mathcal{E}$ and $\mathcal{U}$ of labels. The first set will be filled with indices j of shares x_j of x, and the second set will be filled with labels u of the shares $h^{(u)}$ of $h(x)$. Both sets are initially empty.

– First Step: during this step the node with label u manipulates $z_j^{(u)}$, $e_j^{(u||i)}$ or $(z_j^{(u)} + e_j^{(u||i)})$ with $j \in [\![1,d]\!]$. If one of them is probed, then add j to $\mathcal{E}$.
– Second Step: during this step, the uth leaf processes $h^{(u)} = h(\sum_{j=1}^d z_j^{(u)})$. The intermediate results during this processing are $h^{(u)}$ and all the sub-sums $S_q^{(u)} = \sum_{j=1}^q z_j^{(u)}$, with $q \leqslant d$. If the adversary observes one or several intermediate results $S_q^{(u)}$ for $q \in [\![1,d]\!]$ (including $h^{(u)} = h(S_d^{(u)})$), then one starts by adding $(u,0)$ to $\mathcal{U}$. Afterwards, we build a set $S^{(u)}$ containing 0 and all the values q such that $S_q^{(u)}$ has been observed. Finally, one applies the following rule for every non-zero element $q \in S^{(u)}$ in ascending order:
  - let τ be the greatest element of $S^{(u)}$ which is smaller than q; if $|\{\tau+1, \cdots, q\} \cap \mathcal{E}\}| = q - \tau - 1$, then replace[7] $\mathcal{E}$ by $\mathcal{E} \cup \{\tau+1, \cdots, q\}$, otherwise add to $\mathcal{E}$ the greatest element in $\{\tau+1, \cdots, q\} \backslash \mathcal{E}$.

[7] Note that this step adds at most one element to $\mathcal{E}$.

– Third Step: during this step a node at depth $t-k$ and labelled by u manipulates the values $h^{(u)}_{(w||j)}$ and $h^{(u||i)}_{(w)}$ for $(i,j,w) \in [\![1,7]\!] \times [\![1,3]\!] \times [\![1,3]\!]^{k-1}$. It also manipulates all the intermediate values during the 3^{k-1} processings of SharingCompress$[7:3]$ (each one converting the seven shares $h^{(u||1)}_{(w)}$, $h^{(u||2)}_{(w)}$, ..., $h^{(u||7)}_{(w)}$ into the three new shares $h^{(u)}_{(w||1)}$, $h^{(u)}_{(w||2)}$ and $h^{(u)}_{(w||3)}$). The set $\mathcal{U}$ is first filled with all the pairs (u,w) such that $h^{(u)}_{(w)}$ has been probed by the adversary. Then, from the root to the internal nodes with greatest depth (or equivalently for the input parameter k of Algorithm 5 ranging from t to 1), we apply the following treatment to $\mathcal{U}$ (where we recall that $\mathcal{O}$ denotes the set of intermediate results probed by the adversary):
  - for each call to SharingCompress$[7:3]$ (labelled by $w \in [\![1,3]\!]^k$) done by a node with label $u \in [\![1,7]\!]^{t-k}$; if the set $\mathcal{O}$ contains one (resp. two) element(s) corresponding to intermediate values manipulated during SharingCompress$[7:3]$, then apply Lemma 3 to identify the element(s) $h^{(u)}_{(w||i)}$ (resp. ($h^{(u)}_{(w||i)}$ and $h^{(u)}_{(w||j)}$) enabling to perfectly simulate them and add the corresponding pair(s) of labels into $\mathcal{U}$.
  - for each call to SharingCompress$[7:3]$ (labelled by $w \in [\![1,3]\!]^k$) done by a node with label $u \in [\![1,7]\!]^{t-k}$; if $\mathcal{O} \cup \{h^{(u')}_{(w')}; (u',w') \in \mathcal{U}\}$ contains at least 3 values corresponding to internal values during the processing (including the outputs $h^{(u)}_{(w||j)}$, with $j \in [\![1,3]\!]$, and the inputs $h^{(u||i)}_{(w)}$, with $i \in [\![1,7]\!]$), then add the seven new pairs $(u||1,w)$, $(u||2,w)$, ..., $(u||7,w)$ to $\mathcal{U}$.
  - remove from $\mathcal{U}$ all the pairs with a label $u \in [\![1,7]\!]^{t-k}$ as first coordinate.

  At the end of the process above for the third step, it can be checked that the set $\mathcal{U}$ is only filled with labels u of leaves.

The three rules applied during the third step above implicitly assume that SharingCompress$[7:3]$ is such that (1) the observation of $i<3$ intermediate results brings information on at most i inputs of SharingCompress$[7:3]$ and (2) the observation of $i \geqslant 3$ intermediate results is sufficient to recover information on all the inputs/outputs (*i.e.* the processing is perfect 2-probing secure). This property is proved in Lemma 3 which sates the perfect 2-probing security of the algorithm SharingCompress$[N:3]$ for any N.

Simulation. Before starting the simulation step, we introduce the following useful notation: two words u and u' respectively of size k and k' satisfy the relation $u \leqslant u'$ if $k \leqslant k'$ and $u_i = u'_i$ for every $i \leqslant k$. We shall moreover assume that any word contains the null word 0 (of size 0). The simulation then starts by assigning a random value to every $e^{(u)}_j$ entering in the computation of any value in $\mathcal{O}$ (actually, this is exactly what is done in Algorithm 1). Afterwards, every intermediate variable $v \in \mathcal{O}$ (*i.e.* every intermediate value observed by the adversary) is simulated as follows.

– If v is of the form $z^{(u)}_j$ or $(z^{(u)}_j + e^{(u||i)}_j)$ with $j \in [\![1,d]\!]$, then j belongs to $\mathcal{E}$ by construction (first step of the assignment phase). Due to (19), both $z^{(u)}_j$ and $(z^{(u)}_j + e^{(u||i)}_j)$ can therefore be perfectly simulated with the share x_j and the random values $e^{(u||i)}_j$ and $e^{(w)}_j$ such that $w \leqslant u$ (which have been initialized at random during the preliminary step of the simulation).
– If v is of the form $\sum_{j=1}^{q} z^{(u)}_j$, with $(q,u) \in [\![1,d]\!] \times [\![1,7]\!]^k$, then its simulation depends on $\mathcal{E}$. If the latter set contains all the indices j lower than or equal to q, then all the values $z^{(u)}_j$ entering into the sum can be perfectly simulated thanks to the x_j with $j \in \mathcal{E}$ and the random values $e^{(w)}_j$ with $w \leqslant u$. Otherwise, this means that there exists $j \in [\![1,q]\!]$ such that $j \notin \mathcal{E}$, *i.e.* such that no value of the form $z^{(w)}_j$ or $(z^{(w)}_j + e^{(w||i)}_j)$ (with $w \in [\![1,7]\!]^*$ and $i \in [\![1,7]\!]$), has been previously observed. This moreover implies that the missing value $z^{(u)}_j$ cannot be deduced from the other observations v of the form $\sum_{j=1}^{q} z^{(u)}_j$ with $q \in [\![1,d]\!]$ (due to the rule applied during the second step of the assignment). As a consequence, if $[\![1,q]\!] \cap \mathcal{E} \neq [\![1,q]\!]$, then $v = \sum_{j=1}^{q} z^{(u)}_j$ can be assigned a random value.

– If v enters into one of the 3^{k-1} processings of $\mathsf{SharingCompress}[7:3]$ performed by a node with label $u \in [\![1,7]\!]^{t-k}$ (including the outputs but excluding the inputs), then one applies the following rule: if $\mathcal{U}$ contains all the labels $u' \in [\![1,7]\!]^t$ such that $u \leqslant u'$, then v can be perfectly simulated thanks to all the values $(h^{(u')})_{u' \in [\![1,7]\!]^t, u \leqslant u'}$, and the values $(m_i^{(u'')})_{i \leqslant 3, u'' \in [\![1,7]\!]^\star, u \leqslant u''}$ which are assigned at random. Otherwise, this means that at least one $u' \in [\![1,7]\!]^t$ is missing in $\mathcal{U}$, which itself implies (by construction of $\mathcal{U}$) that at least five among the 7 inputs of the call to $\mathsf{SharingCompress}[7:3]$ manipulating v are missing. As a consequence, at least one value entering into the construction of v cannot be deduced from the adversary observations and the latter variable can hence be assigned a random value.
– If v is an input of one of the 3^{k-1} processings of $\mathsf{SharingCompress}[7:3]$ performed by a node with label $u \in [\![1,7]\!]^{t-k}$, then it also corresponds to the output of one of the processings of $\mathsf{SharingCompress}[7:3]$ performed by the nodes at depth $t-k+1$. It has therefore been simulated by the previous step.

Security. Before introducing the main proposition of the security proof, we recall that we have:

$$h^{(u)} = h\left(x + \sum_{u' \in [\![1,7]\!]^*, u' \leqslant u} \mathbf{e}^{(u')}\right) , \tag{25}$$

where $\mathbf{e}^{(u')} = \sum_{j=1}^{3^t} e_j^{(u')}$ (see (18)).

Proposition 3. *For any set $\mathcal{O}$ of strictly less than d observable variables, the sets $\mathcal{E}$ and $\mathcal{U}$ designed from $\mathcal{O}$ by the assignment phase brings no information on the sensitive variable x.*

Proof. By construction, the cardinality of $\{x_i : i \in \mathcal{E}\}$ is at most $d-1$. It is therefore independent of x and of the set $\{h^{(u)}; (u,0) \in \mathcal{U}\}$ since, from (25), the $h^{(u)}$ only involve x (and not the x_i) in their computation. It then remains to prove that $\{h^{(u)}; (u,0) \in \mathcal{U}\}$ is independent of x (and $h(x)$).

Let $\mathcal{U}_k$ denote the content of $\mathcal{U}$ during the third step of the assignment when starting to deal with the nodes at depth k.

– If there exists $i_1 \in [\![1,7]\!]$ such that $(u||i_1, w) \notin \mathcal{U}_{k+1}$, then there exist at least four other indices i_2, i_3, i_4 and i_5 in $[\![1,7]\!]$ such that the $(u||i_j, w)$, $j \in \{1,2,3,4,5\}$, are not in $\mathcal{U}_{k+1}$. This property is called P. It is a consequence of the fact that, by construction of $\mathcal{U}$ (during the assignment), for every $(u,w) \in [\![1,7]\!]^k \times [\![1,3]\!]^{t-k}$ we have $\#\{i \in [\![1,7]\!] \mid (u||i,w) \in \mathcal{U}_{k+1}\} \in \{0,1,2,7\}$.
– By construction of $\mathcal{U}$, if P is satisfied by a node at depth $k+1$, then it is satisfied by another node at depth k.
– If $\#\mathcal{O} < 3^t$ (which is true by hypothesis since $d = 3^t$), then there exists $u \in [\![1,7]\!]^t$ such that $(u,0) \notin \mathcal{U}_t$ (*i.e.* $(u,0) \notin \mathcal{U}$). One can indeed easily prove the contrapositive: if all the possible pairs $(u,0)$ are in $\mathcal{U}$, then by construction and the two first points above this implies that every 7-tuple at input of a 3^{t-1} processings of $\mathsf{SharingCompress}[7:3]$ done at depth 0 is included in $\mathcal{U}_1$, which implies that $\#\mathcal{U}_0$ has cardinality greater than $d = 3^t$ and hence $\#\mathcal{O} \geqslant d$ (since $\mathcal{U}_0 \subseteq \mathcal{O}$).
– From the three points above, we deduce that P is satisfied for every depth $k \in [\![1,t]\!]$. In other terms and without loss of generality, there exists $w' \in [\![1,3]\!]^{t-1}$ such that $h_{(w')}^{(1)}$, $h_{(w')}^{(2)}$, $h_{(w')}^{(3)}$, $h_{(w')}^{(4)}$ and $h_{(w')}^{(5)}$ cannot be deduced from the elements in $\mathcal{U}$, which itself implies that at least one element of $\{h_{(w'||1)}, h_{(w'||1)}, h_{(w'||3)}\}$ cannot be deduced from $\mathcal{U}$. Since $(h_{(w)})_{w \in [\![1,3]\!]^t}$ is a d-sharing of $h(x)$ (for $d = 3^t$), we deduce that $\mathcal{U}$ is independent of $h(x)$ (and hence of x since h can be bijective).

□

G Complexities

G.1 Operation Count

We summarize here in two tables the exact operation count for all discussed methods in this paper. Table 1 gathers the methods for the secure evaluation of low algebraic degree polynomials, whereas Table 2 gives the complexities for secure evaluation of a polynomial of high algebraic degree.

Table 1. Complexity: Methods for Low Algebraic Degree Polynomial

	Operation Count	
$\mathcal{C}_{\mathtt{D2S}_2}$	#add	$\frac{9}{2}\, d(d-1)+1$
Section $4, s=2$	#rand	$d(d-1)$
$(\sim CPRR)$	#mult	0
	#lut	$d(2d-1)$
$\mathcal{C}_{\mathtt{D2S}_s}$	#add	$\sum_{j=1}^{s} \mu_{d,s}(j) \cdot \binom{d}{j} \cdot \#\mathrm{add}_s(j) + (d\lceil \frac{k}{d} \rceil - d)(d+1)$
Section $4, s>2$	#rand	$\sum_{j=1}^{s} \mu_{d,s}(j) \cdot \binom{d}{j} \cdot \#\mathrm{rand}_s(j) + \frac{1}{2} \cdot (d\lceil \frac{k}{d} \rceil - d)(d-1)$
	#mult	$\sum_{j=1}^{s} \mu_{d,s}(j) \cdot \binom{d}{j} \cdot \#\mathrm{mult}_s(j)$
	#lut	$\sum_{j=1}^{s} \mu_{d,s}(j) \cdot \binom{d}{j} \cdot \#\mathrm{lut}_s(j)$
		where $k = \sum_{j=1}^{s} j \cdot \mu_{d,s}(j) \cdot \binom{d}{j}$ and $\mu_{d,s}(j) = \binom{d-j-1}{s-j} \bmod 2$
		where $\#\mathrm{xxx}_s(j)$ are either from $\mathcal{C}_{\mathtt{TE}_s}$ or $\mathcal{C}_{\mathtt{CRV}_s}$ at order j
$\mathcal{C}_{\mathtt{TE}_2}$	#add	$3d \cdot 7^t - 2d + \frac{5}{2}(7^t - 3^t)$
Section $5, s=2$	#rand	7^t
	#mult	0
	#lut	$\frac{d}{2}7^t - \frac{d}{2} + \frac{3(7^t-3^t)}{4}$
		where $t = \lceil \log_3 d \rceil$
$\mathcal{C}_{\mathtt{TE}_s}$	#add	$\frac{\mu(s)^t-1}{\mu(s)-1} d^{\frac{(s+1)}{2}} (\mu(s)+1) + \frac{\mu(s)^t-3^t}{\mu(s)-3}\,(2\mu(s)-4) + \mu(s)^t d$
Section $5, s>2$	#rand	$\frac{\mu(s)^t-1}{\mu(s)-1} d(s+1) + \frac{\mu(s)^t-3^t}{\mu(s)-3} \left\lfloor \frac{\mu(s)}{2} \right\rfloor$
	#mult	0
	#lut	$\mu(s)^t$
		where $t = \lceil \log_3 d \rceil$ and $\mu(s)^t = d^{\ln(2^{s+1}-1)/\ln 3} \simeq d^{0.63(s+1)}$.
$\mathcal{C}_{CRV_s}$	#add	$2d(d-1)(\ell+t-3) + d(t-1)(\ell-2)$
Section 6	#rand	$\frac{d(d-1)}{2}(\ell+t-3)$
	#mult	$d^2(\ell+t-3)$
	#lut	$d \times t \times \ell$
		where $t = \sqrt{\sum_{i=0}^{s} \binom{n}{i}/n}$ and $\ell \geqslant \sqrt{\sum_{i=0}^{s} \binom{n}{i}/n} - \frac{1}{n} + 1$

Table 2. Complexity: Methods for High Algebraic Degree Polynomial

		Operation Count
$\mathcal{C}_{CRV}$	#add	$2d(d-1)(\ell+t-3)+d(t-1)(\ell-2)$
	#rand	$\frac{d(d-1)}{2}(\ell+t-3)$
	#mult	$d^2(\ell+t-3)$
	#lut	$d\times t\times \ell$
		where $t=\sqrt{2^n/n}$ and $\ell \geqslant \sqrt{2^n/n}-\frac{1}{n}+1$
$\mathcal{C}_{\mathsf{PolyDecomp}}$	#add	$(t+r)\cdot \#\text{add}_s + d\cdot(r\cdot t+t+r)$
Decomposition in	#rand	$(t+r)\cdot \#\text{rand}_s$
Polynomials	#mult	$(t+r)\cdot \#\text{mult}_s$
of degree s	#lut	$(t+r)\cdot \#\text{lut}_s + d\cdot(t+1)\cdot(r+1)$
(Section 3)		where t and r are chosen to minimize the overall complexity
		where $\mathcal{C}_{\#\text{xxx}_s}$ are from low algebraic degree methods
$\mathcal{C}_{\mathsf{PolyDecomp}(s_1,s_2)}$	#add	$r\cdot\#\text{add}_{s_1}+t\cdot\#\text{add}_{s_2}+d\cdot(r\cdot t+t+r)$
Decomposition in	#rand	$r\cdot\#\text{rand}_{s_1}+t\cdot\#\text{rand}_{s_2}$
Polynomials	#mult	$r\cdot\#\text{mult}_{s_1}+t\cdot\#\text{mult}_{s_2}$
of degree s_1 and s_2	#lut	$r\cdot\#\text{lut}_{s_1}+t\cdot\#\text{lut}_{s_2}+d\cdot(t+1)\cdot(r+1)$
		where t and r are chosen to minimize the overall complexity
		where $\#\text{xxx}_s$ are from low algebraic degree methods

G.2 Asymptotic Complexities

When comparing the asymptotic complexities of the methods, we can no longer assume that the function h is represented by a look-up table. Indeed the bit-size of such a table is $O(n2^n)$ which is quickly much greater than the capacity of the device. If a compact representation of h is known, it can however be evaluated on-the-fly. This is in particular the case, if h is known to have a small algebraic degree. Indeed, in this case h can be represented by the list of all its non-zero coefficients in its polynomial representation. This gives a memory complexity of $O(n\sum_{i=0}^{s}\binom{n}{i})$. Moreover, for sparse polynomials the evaluation can be done faster than with Horner's scheme which has complexity $O(2^n)$ both in terms of additions and multiplications (see *e.g.* [15]). The evaluation for sparse polynomials simply consists in evaluating each individual term separately (*e.g.* by using a square and multiply algorithm). It requires $O(\sum_{i=0}^{s}\binom{n}{i})$ additions and $O(n\sum_{i=0}^{s}\binom{n}{i})$ multiplications. Based on this discussion, and after denoting by $\kappa(n,s)$ the sum $\sum_{i=0}^{s}\binom{n}{i}$, we exhibit hereafter the asymptotic complexities of the presented methods for a function $h\in\mathcal{B}_{n,n}$ of algebraic degree s:

- $CRV_s(n,d)$: starting from the complexity analysis done in Section 6 and assuming $t=\ell\simeq\sqrt{\frac{1}{n}\kappa(n,s)}$ (which corresponds to the choices of t and ℓ that minimize the number of multiplications) we get $O(d^2\sqrt{\kappa(n,s)/n})$ non-linear multiplications and random values, $O(d^2\sqrt{\kappa(n,s)/n}+d\kappa(n,s)/n)$ additions and $O(\frac{d}{n}\kappa(n,s))$ evaluations of linearized polynomials. When processed on-the-fly, the latter evaluations themselves involve $O(n)$ additions and $O(n^2)$ multiplications. We eventually get:

	#add	#rand	#mult
$\mathcal{C}_{CRV_s}$	$O(d^2\sqrt{\frac{\kappa(n,s)}{n}}+d\kappa(n,s))$	$O(d^2\sqrt{\frac{\kappa(n,s)}{n}})$	$O(d^2\sqrt{\frac{\kappa(n,s)}{n}}+dn\kappa(n,s))$

where we can approximate $\kappa(n,s)$ by $O(2^n)$ if $s\geqslant n/2-\sqrt{n}$ and by $O(\left(1-2s/n\right)^{-1}\binom{n}{s})$ if $s<n/2-\sqrt{n}$. If s is small compared to n, we can approximate $\binom{n}{s}$ by n^s/s^s and $O(\left(1-2s/n\right)^{-1}\binom{n}{s})$ by $O(n^s/s^s)$.

	#add	#rand	#mult
$\mathcal{C}_{CRV_s}$	$O\big(d^2\frac{n^{(s-1)/2}}{s^{s/2}}+d\frac{n^s}{s^s}\big)$	$O\big(d^2\frac{n^{(s-1)/2}}{s^{s/2}}\big)$	$O\big(d^2\frac{n^{(s-1)/2}}{s^{s/2}}+d\frac{n^{s+1}}{s^s}\big)$

– Algorithm $\mathsf{TreeExplore}_s(n,d)$: from the complexity analysis conducted in Appendix E, we deduce the following complexity $\mathcal{C}_{\mathtt{TE}_s}$:

	#add	#rand	#mult
$\mathcal{C}_{\mathtt{TE}_s}$	$O(d^{0.63(s+1)}(sd+\kappa(n,s)))$	$O(sd^{0.63(s+1)+1})$	$O(d^{0.63(s+1)}n\kappa(n,s))$

After approximation, we get:

	#add	#rand	#mult
$\mathcal{C}_{\mathtt{TE}_s}$	$O\big(d^{0.63(s+1)}\frac{n^s}{s^s}+sd^{0.63s+1.63}\big)$	$O\big(sd^{0.63s+1.63}\big)$	$O\big(d^{0.63(s+1)}\frac{n^{s+1}}{s^s}\big)$

– Algorithm 1 with $\mathsf{SecureEval}_j = CRV_s(n,j)$ for $j \leqslant s$ and $s \leqslant d/2$:

	#add	#rand	#mult
$\mathcal{C}^{CRV_s}_{\text{Alg. 1}}$	$O\big(d^{s+1}(d\frac{n^{(s-1)/2}}{s^{3s/2}}+\frac{n^s}{s^{2s}})\big)$	$O\big(d^{s+2}\frac{n^{(s-1)/2}}{s^{3s/2}}\big)$	$O\big(d^{s+2}\frac{n^{\frac{s-1}{2}}}{s^{\frac{3s}{2}}}(1+\frac{n^{\frac{s+3}{2}}}{ds^{\frac{s}{2}}})\big)$

– Algorithm 1 with $\mathsf{SecureEval}_j = \mathsf{TreeExplore}_s(n,j)$ for $j \leqslant s$ and $s \leqslant d/2$:

	#add	#rand	#mult
$\mathcal{C}^{\mathtt{TE}_s}_{\text{Alg. 1}}$	$O(d^{1.63s+0.63}\frac{n^s}{s^{2s}}+d^{1.63s+1.63}\frac{1}{s^{s-1}})$	$O(\frac{d^{1.63s+1.63}}{s^{s-1}})$	$O(d^{1.63s+0.63}\frac{n^{s+1}}{s^{2s}})$