Noname manuscript No.
(will be inserted by the editor)

Derivative-Based Evaluation of Multivariate Polynomials over Product Hamming Balls

Vaibhav Dixit, Santanu Sarkar, Fukang Liu, Willi Meier

the date of receipt and acceptance should be inserted later

Abstract Efficient evaluation of multivariate polynomials over finite spaces is an important problem in algebraic cryptanalysis, particularly in exhaustive-search attacks on multivariate public-key cryptosystems. Existing fast exhaustive-search techniques, including those of Bouillaguet et al. (CHES 2010), Dinur (EUROCRYPT 2021), and Furue and Takagi (PQCrypto 2023), mainly target the complete space $\mathbb{F}_q^n$, whereas many cryptanalytic applications require evaluation only over structured subsets. Recently, Liu et al. proposed a memory-efficient algorithm for evaluating degree-d polynomials over product Hamming balls

$$B(\boldsymbol{n}, \boldsymbol{w}) = B(n_s, w_s) \times \cdots \times B(n_1, w_1) \subseteq \mathbb{F}_2^n,$$

where $\sum_{i=1}^{s} n_i = n$ and $B(n_i, w_i) \subseteq \mathbb{F}_2^{n_i}$ denotes the set of vectors of length n_i and having at most w_i nonzero coordinates. In this work, we extend this structured-subset paradigm by developing a unified derivative-based framework, comprising initialization and evaluation phases, first over prime fields $\mathbb{F}_q$ and subsequently over arbitrary finite fields $\mathbb{F}_{q^\rho}$ through basis expansion. We derive two initialization methods with time complexities $\mathcal{O}(nrM)$ and $\mathcal{O}(rM^2)$, where $M \leq \min\{\binom{n+d}{d}, q^n\}$ and $r = \min\{d, q-1\}$. After initialization, evaluation over $B(\boldsymbol{n}, \boldsymbol{w})$ requires $\mathcal{O}(d|B(\boldsymbol{n}, \boldsymbol{w})|)$ arithmetic operations, apart from a linear setup cost. The corresponding overall memory bounds are $\mathcal{O}((M + r^2)\log q + n\log(nq))$ and $\mathcal{O}(((d+1)M + r^2)\log q + n\log(nq))$, respectively, and are independent of the number of evaluated points.

Keywords Multivariate polynomial evaluation · finite fields · product Hamming balls · discrete derivatives · fast exhaustive search · MPKCs

V. Dixit · S. Sarkar
Department of Mathematics,
Indian Institute of Technology Madras,
Chennai, Tamil Nadu 600036, India.
E-mail: vaibhavrdrk@gmail.com; sarkar.santanu.bir1@gmail.com

F. Liu
Institute of Science Tokyo, Japan
E-mail: liufukangs@gmail.com

W. Meier
University of Applied Sciences and Arts Northwestern Switzerland
E-mail: willimeier48@gmail.com

1 Introduction

The problem of multivariate polynomial evaluation has been investigated from several perspectives, depending on the structure of the evaluation domain and the intended applications. Broadly speaking, existing algorithms may be classified into three research directions according to the evaluation domain they consider and the computational paradigm they employ. Throughout this paper, q denotes a prime; arbitrary finite extensions of $\mathbb{F}_q$ are written as $\mathbb{F}_{q^\rho}$.

The first direction considers evaluation over arbitrary sets of points, without assuming any geometric or combinatorial structure on the evaluation domain. The objective is to design algorithms that work uniformly for any collection of evaluation points. A major line of work in multivariate multipoint evaluation (MME) aims at achieving running times that are nearly linear in the combined size of the dense polynomial representation and the output. Umans (STOC 2008) [28] gave a nearly-linear-time algorithm for MME over finite fields of small characteristic, and Kedlaya and Umans (FOCS 2008) [23] extended this approach to arbitrary finite fields $\mathbb{F}$. In both cases, the nearly-linear behavior requires the number of variables n to satisfy $n \leq d_{\mathrm{var}}^{o(1)}$, where d_{var} bounds the degree in each variable. The running time is nearly linear in $d_{var}^{n} + N$, up to polynomial factors in $n, d_{\mathrm{var}}, \log \mathbb{F}$, where N is the size of the evaluation set. Removing this restriction remained an important open problem for over a decade. It was first resolved for finite fields of small characteristic by Bhargava et al.(STOC 2022) [7], and subsequently for all finite fields by Bhargava et al. (FOCS 2022) [6]. Ghosh et al. (FOCS 2023) [19] extended fast MME beyond finite fields by developing exact and approximate algorithms over the rational, real, and complex numbers. Their approach crucially builds on the finite-field MME algorithm of Bhargava *et al.* [6], together with efficient rational reconstruction and fast Chinese-remaindering techniques. More recently, van der Hoeven and Lecerf [20] studied deterministic algebraic MME over arbitrary fields for a fixed number of variables and derived refined complexity bounds that balance precomputation and evaluation as a function of the number of points. These algorithms rely on sophisticated algebraic reductions, modular composition, interpolation, and finite-field constructions, and deliberately avoid exploiting any combinatorial structure of the evaluation points.

A second direction, motivated primarily by computer algebra, studies polynomial evaluation over structured evaluation domains. Van der Hoeven and Schost (AAECC 2013) [29] developed multipoint evaluation and interpolation algorithms for subsets of tensor-product grids indexed by finite downward-closed sets of $\mathbb{N}^n$. Their framework includes tensor-product and n-dimensional grids and relies on multivariate Newton bases, fast basis transformations, and recursive decomposition of the underlying downward-closed index set. If N denotes the cardinality of this index set, their general complexity bound is quasi-linear in N; in particular, using fast univariate polynomial multiplication, it can be bounded by $\mathcal{O}\left(nN \log^2 N \log\log N\right)$. Fischer *et al.* (ESA 2025) [15] revisited a particular trimmed case of the framework of van der Hoeven and Schost and proposed a simpler recursive algorithm while maintaining essentially the same asymptotic complexity.

The third direction originates from algebraic cryptanalysis [4, 8] and concerns the efficient evaluation of low-degree multivariate polynomials over large finite-field search spaces. Solving systems of multivariate quadratic equations is a fundamental computational problem in multivariate cryptography. Representative constructions include the Unbalanced Oil and Vinegar (UOV) signature scheme [24], together with its variants such as MAYO [5] and QR-UOV [16]. More recently, MQOM [2] constructed a post-quantum signature scheme whose underlying computational assumption is directly based on the difficulty of solving a multivariate quadratic systems. These constructions further motivate the study of efficient algorithms for solving and evaluating low-degree polynomial systems over finite fields.

From the cryptanalytic perspective, enumeration procedures in which a degree-d polynomial is evaluated over a large search space in $\mathbb{F}_q^n$ in order to identify its zeros arise naturally in XL-type and Crossbred attacks [17,22]. Direct enumeration is generally prohibitively expensive; however, the

low algebraic degree of the polynomials can be exploited to obtain substantially faster evaluation procedures.

A notable example is the Fast Exhaustive Search (FES) technique introduced by Bouillaguet et al. for Boolean polynomials [10]. By traversing $\mathbb{F}_2^n$ in Gray-code order and exploiting Boolean derivatives, FES evaluates a degree-d polynomial on all 2^n inputs in time $\mathcal{O}(d \cdot 2^n)$, following a preprocessing phase of cost $\mathcal{O}(\binom{n}{\leq d})^2$ using $\mathcal{O}(\binom{n}{\leq d})$ bits of memory. This approach outperforms the standard Möbius transform in time complexity and improves upon the algorithm of [14] in terms of memory usage. Subsequently, several algorithms were proposed [9, 12, 13] to solve systems of Boolean equations with improved complexities; however, these methods are inherently restricted to the Boolean space $\mathbb{F}_2^n$.

This restriction was later overcome for complete finite-field spaces by Furue and Takagi at PQCrypto 2023 [18]. Their algorithm replaces Gray-code traversal with lexicographic enumeration and maintains higher-order discrete derivatives. They showed that a degree-d polynomial can be evaluated over the full space $\mathbb{F}_q^n$, where q is a prime number in time $\mathcal{O}(d \cdot q^n)$, after a preprocessing phase estimating $\binom{n+d}{d}^2$ arithmetic operations. This improvement comes at the expense of increased memory complexity, namely $\log(q \cdot n) \cdot n \cdot \binom{n+d}{d}$ bits. Nevertheless, this approach applies only to the full finite space $\mathbb{F}_q^n$ and does not exploit additional structure in the evaluation domain.

Hamming-constrained search domains also arise in coding-theoretic problems such as syndrome decoding. This provides a natural example of a search space whose domain is a Hamming ball; we use it only as motivation for the domain structure considered here, rather than claiming an improved algorithm or complexity bound for syndrome-decoding algorithm (SDP). Specifically, in SDP a matrix $H \in \mathbb{F}_q^{(n-k)\times n}$, a syndrome $s \in \mathbb{F}_q^{n-k}$, and an integer weight $w \in \{0, \ldots, n\}$ are given, and the goal is to find a vector $\boldsymbol{x} \in \mathbb{F}_q^n$ such that

$$H\boldsymbol{x}^\top = s^\top \quad \text{and} \quad \mathrm{Hw}(\boldsymbol{x}) \leq w,$$

where the Hamming weight of $\boldsymbol{x}$ ($\mathrm{Hw}(\boldsymbol{x})$) is the number of nonzero coordinates in $\boldsymbol{x}$. SDP is a classical problem in coding theory and is known to be NP-complete [3]. Recently, the signature scheme WAVE was proposed in [11], relying on the hardness of SDP over $\mathbb{F}_3^n$ with large weights. In such problems, the search space is not the full space $\mathbb{F}_q^n$, but a Hamming-constrained subset that consists of vectors with Hamming weight not greater than w. We refer to such spaces as a *Hamming ball* and denote it by $B(n, w) \subseteq \mathbb{F}_q^n$. More generally, similar Hamming-constrained product spaces arise in algebraic cryptanalysis; for example, Dinur [12] considers solving systems of Boolean equations over a domain of the form $B(n - n_1, w) \times B(n_1, n_1) \subseteq \mathbb{F}_2^n$. We formally define these product Hamming balls in Section 4.

Motivated by such settings, Liu et al. [26] revisited fast exhaustive search over Hamming-constrained product spaces by decoupling the notion of an evaluation rule from the evaluation order. Their abstraction permits efficient evaluation over product Hamming balls in $\mathbb{F}_2^n$, but the resulting technique remains inherently Boolean.

Although the second and third research directions both exploit structure in the evaluation domain, they have evolved largely independently and employ fundamentally different computational paradigms. The computer-algebra literature addresses multipoint evaluation using Newton-basis transformations, recursive decompositions, and interpolation over downward-closed index sets, whereas the FES literature propagates evaluations incrementally along carefully chosen traversal orders using discrete derivatives. Consequently, the two directions have largely focused on different applications: symbolic computation and power-series arithmetic on the one hand, and exhaustive-search algorithms in algebraic cryptanalysis on the other.

2 Our Contribution

Our work follows the derivative-based exhaustive-evaluation paradigm and combines the evaluation-rule framework of Liu et al. [26] for structured subsets of $\mathbb{F}_2^n$ with the higher-order derivative techniques of Furue and Takagi [18] over prime fields. This leads to a derivative-based framework for evaluating multivariate polynomials over product Hamming balls in $\mathbb{F}_q^n$, initially for prime q.

The proposed framework consists of an initialization phase and an evaluation phase. For a reduced polynomial of total degree at most d, let

$$M = |\{\boldsymbol{\alpha} \in \{0,\dots,q-1\}^n : |\boldsymbol{\alpha}| \le d\}| \qquad \text{and} \qquad r = \min\{d, q-1\}.$$

For Initialization phase, we develop two methods for computing the derivative values required at the origin. Method 1 uses the formula of [27], which expresses higher-order discrete derivatives at the origin in terms of the polynomial coefficients and Stirling numbers of the second kind. The method organizes this computation as a coordinatewise transformation requiring $\mathcal{O}(nrM)$ arithmetic operations. Method 2 instead constructs the required admissible derivative polynomials recursively and requires $\mathcal{O}(rM^2)$ operations.

For the evaluation phase, we separate enumeration from derivative evaluation. Algorithm `ENU` enumerates exactly the admissible inputs of $B(\boldsymbol{n}, \boldsymbol{w})$ in ascending lexicographic order while maintaining the information required by the evaluation procedure. Algorithm `EVA` then uses this information to reuse previously computed derivative values and performs at most $\mathcal{O}(d)$ updates per evaluated input. Consequently, the overall running times of the two variants are

$$\mathcal{O}(nrM + d|B(\boldsymbol{n}, \boldsymbol{w})|) \qquad \text{and} \qquad \mathcal{O}\Big(rM^2 + d|B(\boldsymbol{n}, \boldsymbol{w})|\Big)$$

respectively. Their corresponding overall memory requirements are

$$\mathcal{O}\Big((M + r^2)\log q + n\log(nq)\Big) \qquad \text{and} \qquad \mathcal{O}\Big(((d+1)M + r^2)\log q + n\log(nq)\Big),$$

which, importantly, are independent of the number of evaluated points.

Compared with the full-space FEA framework of Furue and Takagi, our initialization procedures yield improved complexity bounds in relevant parameter regimes. Method 1 improves both the time and memory requirements of the initialization phase, while Method 2 provides a time improvement whenever

$$M < \frac{1}{\sqrt{r}}\binom{n+d}{d}.$$

The advantage of the proposed framework is more pronounced for product Hamming balls, since its evaluation cost depends on the number of admissible points $|B(\boldsymbol{n}, \boldsymbol{w})|$ rather than on the size q^n of the complete space.

Although our algorithmic approach originates from the FES literature, the product Hamming balls considered here also possess the downward-closed structure exploited in multipoint evaluation [29]. The computational paradigms are nevertheless different: rather than using Newton bases, basis transformations, and recursive decomposition, our method propagates polynomial and derivative values incrementally along a suitable evaluation order. Moreover, over the complete space $\mathbb{F}_q^n$, evaluation using Method 1 has the coarse time bound $\mathcal{O}(ndq^n)$, whereas the multipoint-evaluation bound is $\mathcal{O}\left(nq^n(\log q)^2 \log\log q\right)$. Hence, Method 1 has a smaller asymptotic upper bound whenever $d = o\left((\log q)^2 \log\log q\right)$.

Finally, we extend the framework from prime fields to arbitrary finite fields $\mathbb{F}_{q^\rho}$ by fixing a basis over $\mathbb{F}_q$. Under this representation, extension-field coordinates are expanded into blocks of prime-field coordinates, and product Hamming balls are represented through corresponding basis-block constraints.

2.1 Organization of the paper

The paper is organized as follows. Section 3 introduces the notation and relevant background. Section 4 develops the derivative-based evaluation framework over prime fields and establishes its sufficient condition. Section 5 verifies this condition for product Hamming balls and establishes the required lexicographic traversal and memory-reuse properties. Section 6 presents the initialization and evaluation algorithms, complexity analysis, and comparison with existing methods. Section 7 extends the framework to arbitrary finite fields $\mathbb{F}_{q^\rho}$ through basis expansion. Finally, the Section 8 summarizes the main results, and Section 9 discusses possible extensions of the framework.

3 Preliminaries

3.1 Notation

We introduce below the notation and conventions used throughout the paper.

- Throughout the paper, q denotes a prime and $\mathbb{F}_q$ is identified with $\mathbb{Z}/q\mathbb{Z}$. Whenever an element of $\mathbb{F}_q$ occurs in an integer-valued expression, comparison, or coordinate decrement, its canonical representative in $\{0, 1, \ldots, q-1\}$ is used. In particular, sums such as $\sum_i x_i$ are ordinary integer sums unless stated otherwise.
- Bold characters denote vectors. In particular,
$$\boldsymbol{x} = (x_n, \ldots, x_1) \in \mathbb{F}_q^n.$$
The all-zero vector is denoted by 0^n.
- Vector addition, subtraction, and coordinate-wise multiplication are performed in $\mathbb{F}_q$. Thus, for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^n$,
$$\boldsymbol{x} \pm \boldsymbol{y} = (x_n \pm y_n, \ldots, x_1 \pm y_1),$$
and
$$\boldsymbol{x} \odot \boldsymbol{y} = (x_n y_n, \ldots, x_1 y_1),$$
where each coordinate operation is taken modulo q. Similarly, for $a \in \mathbb{Z}$, with each coordinate reduced modulo q
$$a\boldsymbol{x} = (ax_n, \ldots, ax_1).$$
If $f : \mathbb{F}_q^n \to \mathbb{F}_q$, then arithmetic between values such as $f(\boldsymbol{x}) \pm f(\boldsymbol{y})$ is also performed in $\mathbb{F}_q$.
- For $\boldsymbol{x} \in \mathbb{F}_q^n$, the support of $\boldsymbol{x}$ is the ordered tuple consisting precisely of the indices of its nonzero coordinates. That is,
$$\mathrm{Sup}(\boldsymbol{x}) = (k_1, \ldots, k_h), \qquad k_1 < \cdots < k_h,$$
where $x_{k_j} \neq 0$, $1 \leq j \leq h$, and $x_\ell = 0$ for every $\ell \notin \{k_1, \ldots, k_h\}$. For the zero vector, we set $\mathrm{Sup}(0^n) = \emptyset$. Moreover, for this support, we use
$$\boldsymbol{K}_i = (k_1, \ldots, k_i), \qquad 1 \leq i \leq h,$$
to denote the prefix consisting of the first i support indices, with $\boldsymbol{K}_0 = \emptyset$.
- The Hamming weight of a vector $\boldsymbol{x}$ is denoted by $\mathrm{Hw}(\boldsymbol{x})$ and is defined by
$$\mathrm{Hw}(\boldsymbol{x}) = |\mathrm{Sup}(\boldsymbol{x})|.$$
- For a vector $\boldsymbol{x} \in \mathbb{F}_q^n$, we define the integer-valued function
$$\mathrm{INT}(\boldsymbol{x}) = \sum_{i=1}^{n} q^{i-1} x_i.$$

– For distinct vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^n$, let $j = \max\{i : x_i \neq y_i\}$. We write

$$\boldsymbol{x} <_{\text{lex}} \boldsymbol{y} \quad \Longleftrightarrow \quad x_j < y_j,$$

where the coordinates are compared through their canonical representatives. Equivalently,

$$\boldsymbol{x} <_{\text{lex}} \boldsymbol{y} \quad \Longleftrightarrow \quad \text{INT}(\boldsymbol{x}) < \text{INT}(\boldsymbol{y}).$$

– For a multi-index $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$, we write $|\boldsymbol{\alpha}| = \sum_{i=1}^n \alpha_i$ for its total order.
– For an array A, the notation $A[a : b]$ denotes the consecutive entries $A[a], A[a+1], \ldots, A[b]$.
– The asymptotic notations $\mathcal{O}(g)$, $\Theta(g)$, $o(g)$ have their standard meanings. In particular, $f = \mathcal{O}(g)$ denotes an asymptotic upper bound, $f = \Theta(g)$ denotes matching asymptotic upper and lower bounds, and $f = o(g)$ means $\frac{f}{g} \longrightarrow 0$ in the parameter regime under consideration.
– For $1 \leq i \leq n$, the i-th standard basis vector and its complement vector are denoted, respectively, by

$$\boldsymbol{e}_i = (0, \ldots, 0, 1, \underbrace{0, \ldots, 0}_{i-1}), \qquad \overline{\boldsymbol{e}_i} = (1, \ldots, 1, 0, \underbrace{1, \ldots, 1}_{i-1}).$$

– Let $\boldsymbol{x} \in \mathbb{F}_q^n$, $k_i \in \text{Sup}(\boldsymbol{x})$ and $0 \leq \delta \leq x_{k_i}$. The vector obtained by decreasing the k_i-th coordinate of $\boldsymbol{x}$ by δ without wraparound is denoted by

$$\boldsymbol{x}_{(k_i^\delta)} = \boldsymbol{x} - \delta \boldsymbol{e}_{k_i}.$$

The projection of $\boldsymbol{x}$ onto the coordinate hyperplane $x_{k_i} = 0$ is denoted by

$$\boldsymbol{x}^{(k_i)} = \boldsymbol{x} \odot \overline{\boldsymbol{e}_{k_i}}.$$

For convenience, we set $\boldsymbol{x}^{(0)} = \boldsymbol{x}$. More generally, for $\boldsymbol{K}_i = (k_1, \ldots, k_i)$, the simultaneous projection of $\boldsymbol{x}$ onto $x_{k_1} = \cdots = x_{k_i} = 0$ is denoted by

$$\boldsymbol{x}^{(0,k_1,\ldots,k_i)} = \boldsymbol{x} \odot \overline{\boldsymbol{e}_{k_1}} \odot \cdots \odot \overline{\boldsymbol{e}_{k_i}}.$$

In particular, $\boldsymbol{x}_{(k_i^{x_{k_i}})} = \boldsymbol{x}^{(k_i)}$, and for $j > i$, $\boldsymbol{x}^{(0,k_1,\ldots,k_i)}_{(k_j^\delta)} = (\boldsymbol{x} - \delta \boldsymbol{e}_{k_j}) \odot \overline{\boldsymbol{e}_{k_1}} \odot \cdots \odot \overline{\boldsymbol{e}_{k_i}}$.

3.2 Background on Fast Polynomial Evaluation

Several algorithms have been proposed to evaluate a multivariate polynomial f with different trade-offs between time and memory.

– **Standard Möbius Transform (MT).** The standard MT evaluates f on all inputs using an array of size 2^n. It consists of an initialization phase that stores all coefficient values of f, followed by n update steps. The time complexity is $\mathcal{O}(n \cdot 2^n)$ and the memory complexity is 2^n bits. For a degree-d polynomial, Dinur and Shamir [14] showed that time complexity can be reduced to $\mathcal{O}(d \cdot 2^n)$, while memory complexity remains 2^n.
– **Fast Exhaustive Search (FES) [10].** Proposed by Bouillaguet et al., this algorithm traverses $\mathbb{F}_2^n$ using the Gray code order and exploits derivatives of f up to order d. The time complexity is $\binom{n}{\leq d}^2 + d \cdot 2^n$ and the memory complexity is $\binom{n}{\leq d}$.
– **Memory-efficient Möbius Transform [12].** Dinur proposed a fix-and-update strategy in which $n - d$ variables are fixed, and the standard MT is applied to the resulting d-variable polynomial. Intermediate polynomials are stored to speed up updates. The memory complexity is less than $n \cdot \binom{n}{\leq d}$ and the time complexity is smaller than $n \cdot 2^n$ but larger than $d \cdot 2^n + 2^{n-1}$.

- **Improved Memory-efficient MT [12].** By fixing $n' = n - \log\binom{n}{\leq d}$ variables and avoiding the storage of intermediate polynomials, the memory complexity is reduced to $2 \cdot \binom{n}{\leq d}$, while the time complexity remains below $n \cdot 2^n$.
- **Optimized Memory-efficient MT [9].** Bouillaguet further optimized the polynomial update step after fixing $n - d$ variables. Memory complexity is $\binom{n}{\leq d}$ and time complexity is $\mathcal{O}(d \cdot 2^n)$.
- **Fast Enumeration Algorithm (FEA) [18].** Introduced by Furue and Takagi, this algorithm generalizes FES to a finite field $\mathbb{F}_q^n$ using lexicographic order. The time complexity and the memory complexity are $\binom{n+d}{d}^2 + d \cdot q^n$ and $\log(q \cdot n) \cdot n \cdot \binom{n+d}{d}$, respectively.
- **Polynomial Evaluation over the Structured Subset of $\mathbb{F}_2^n$ [26].** Liu et al. reformulated the FES framework in a more abstract manner by introducing a non-recursive evaluation scheme. They proposed an efficient algorithm to evaluate a Boolean polynomial f of degree d in n variables over a product Hamming ball $B(n_s, w_s) \times \cdots \times B(n_1, w_1) \subseteq \mathbb{F}_2^n$.

Since our work establishes a conceptual connection between the evaluation set considered in this paper and the structured MME framework of van der Hoeven and Schost, we first briefly review their framework. We then recall the derivative-based evaluation framework of Liu et al. [26], which forms the algorithmic basis of our work. This background provides the necessary context for extending derivative-based evaluation from the Boolean setting to structured subsets over arbitrary finite fields $\mathbb{F}_q$.

3.3 Structured MME over downward-closed evaluation spaces.

The framework of van der Hoeven and Schost provides a general setting for multipoint evaluation and interpolation of multivariate polynomials over structured evaluation domains. The key combinatorial object in this framework is an *downward-closed set* of $\mathbb{N}^n$, which simultaneously indexes the monomial support of the polynomial and the corresponding evaluation domain. Since our contribution establishes a conceptual connection between the Cartesian products of Hamming balls considered in this paper and this framework, we briefly recall the notions that are relevant to our subsequent development.

Definition 1 Let $\preceq$ denote the componentwise partial order on $\mathbb{N}^n$, i.e., for $(a_1, \ldots, a_n), (b_1, \ldots, b_n) \in \mathbb{N}^n$

$$(a_1, \ldots, a_n) \preceq (b_1, \ldots, b_n) \iff a_i \leq b_i, \qquad \forall i.$$

A finite subset $I \subseteq \mathbb{N}^n$ is called an *downward-closed set* if

$$\mathbf{b} \in I,\ \mathbf{a} \preceq \mathbf{b} \Longrightarrow \mathbf{a} \in I.$$

Intuitively, whenever an index belongs to the set, all indices obtained by decreasing one or more coordinates also belong to the set. This closure property enables recursive decomposition of the evaluation problem and forms the central combinatorial object in the framework of van der Hoeven and Schost.

Let $d_1, \ldots, d_n$ are non-negative integers and $I \subseteq \{0, \ldots d_1 - 1\} \times \cdots \times \{0, \ldots, d_n - 1\}$ is the downward-closed set. Let $\boldsymbol{u} = \{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n\}$ be the collection of tuples of field elements, where each $\boldsymbol{u}_i = (u_{i,0}, \ldots, u_{i,d_i-1})$ is an ordered sequence of field elements. For every $\alpha = (\alpha_1, \ldots, \alpha_n) \in I$, associate the evaluation point $(u_{1,\alpha_1}, \ldots, u_{n,\alpha_n})$. Following the framework of van der Hoeven and Schost [29], the corresponding evaluation space is

$$U(I, \boldsymbol{u}) = \left\{(u_{1,\alpha_1}, \ldots, u_{n,\alpha_n}) : \alpha \in I\right\}.$$

For convenience, we refer to $U(I, \boldsymbol{u})$ as a *downward-closed evaluation space* whenever its indexing set I is downward closed. The same set I also indexes the monomial support of the polynomial, which is assumed to be of the form

$$f(\boldsymbol{x}) = \sum_{\alpha \in I} a_\alpha \boldsymbol{x}^\alpha,$$

Thus, in the framework of van der Hoeven and Schost, the same downward-closed index set I determines both the monomial support and the corresponding evaluation space $U(I, \boldsymbol{u})$.

Given a downward-closed index set I and the associated evaluation space $U(I, \boldsymbol{u})$, van der Hoeven and Schost [29] developed a recursive multipoint-evaluation algorithm whose arithmetic complexity is quasi-linear in the cardinality of the evaluation space, up to polylogarithmic factors. Their approach relies on recursive decomposition, multivariate Newton representations, and fast basis transformations. Since these algorithmic ingredients are not used in the present work, we restrict our discussion to the downward-closed index sets and their associated evaluation spaces required for establishing the connection with our setting.

As shown in Section 4.1, every product Hamming ball considered in this work can be represented as a downward-closed evaluation space $U(I, \boldsymbol{u})$ in the framework of van der Hoeven and Schost. This correspondence is purely structural: our algorithm does not employ their recursive multipoint-evaluation procedure, but instead follows the derivative-based incremental evaluation strategy introduced in the next subsection. We therefore refer the reader to [29] for further details of their evaluation algorithm.

3.4 Derivative-Based Evaluation over Product Hamming Balls

In [26], the authors proposed an efficient algorithm to evaluate a Boolean polynomial f over a product Hamming ball $B(\boldsymbol{n}, \boldsymbol{w}) \in \mathbb{F}_2^n$, referred to therein as a *structured space*. We first introduce the derivative notation used in this setting.

Definition 2 Let f be a Boolean polynomial in the variables $\boldsymbol{x}$. The derivative of f with respect to the variable x_k is represented by $D_k f$ and is defined as

$$D_k f(\boldsymbol{x}) = f(\boldsymbol{x} \oplus e_k) \oplus f(\boldsymbol{x}).$$

For a multi-index $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \{0,1\}^n$, define the corresponding higher-order derivative operator by

$$D^{\boldsymbol{\alpha}} f = D_1^{\alpha_1} \cdots D_n^{\alpha_n} f.$$

In particular, if $\boldsymbol{K}_l = (k_1, \ldots, k_l)$ denotes the indices of the nonzero coordinates of $\boldsymbol{\alpha}$, then we write

$$D^{\boldsymbol{\alpha}} f = D_{\boldsymbol{K}_l} f = D_{k_1,\ldots,k_l} f = D_{k_l} \cdots D_{k_1} f.$$

To evaluate a Boolean polynomial $f(x_n, \ldots, x_1)$ of degree d at $\boldsymbol{x}$, the following recursive relation can be derived from above definition as

$$\forall k \in [1, n]: \quad f(\boldsymbol{x}) = f(\boldsymbol{x}_{(k)}) \oplus D_k f(\boldsymbol{x}_{(k)}). \tag{1}$$

It follows from (1) that, provided the value of $f(\boldsymbol{x}_{(k)})$ is already available, the evaluation of $f(\boldsymbol{x})$ reduces to computing $D_k f(\boldsymbol{x}_{(k)})$. Since $D_k f$ is itself a polynomial of degree at most $d-1$, the same argument can be applied recursively to the derivative polynomial. If $T(n, d)$ denotes the evaluation cost for a degree-d polynomial f, i.e, evaluating $f(\boldsymbol{x})$ polynomial under above assumption then

$$T(n, d) = \mathcal{O}(1) + T(n, d-1).$$

Repeating this recursion yields $T(n, d) = \mathcal{O}(d) + T(n, 0)$. Since $T(n, 0)$ is the time required to evaluate a constant polynomial, we have $T(n, 0) = \mathcal{O}(1)$. Therefore, $T(n, d) = \mathcal{O}(d)$. Rather than evaluating derivatives recursively, the authors transformed the recursion into an equivalent non-recursive formulation (2), shown below, in which each derivative is computed from a previously evaluated derivative of the same order together with one derivative of higher order.

Suppose there exists $\ell \in [1, d]$ and a set of indices $\{k_1, \ldots, k_\ell\} \subseteq \{1, \ldots, n\}$ and $k_0 = 0$ such that

$$\begin{cases} D_{\boldsymbol{K}_\ell} f(\boldsymbol{x}^{\boldsymbol{K}_\ell}) \text{ can be efficiently computed,} \\ D_{\boldsymbol{K}_{\ell-1}} f(\boldsymbol{x}^{\boldsymbol{K}_{\ell-1}}) = D_{\boldsymbol{K}_{\ell-1}} f(\boldsymbol{x}^{\boldsymbol{K}_{\ell-1}}_{(k_\ell)}) \oplus D_{\boldsymbol{K}_\ell} f(\boldsymbol{x}^{\boldsymbol{K}_{\ell-1}}_{(k_\ell)}), \\ \quad \vdots \\ D_{\boldsymbol{K}_i} f(\boldsymbol{x}^{\boldsymbol{K}_i}) = D_{\boldsymbol{K}_i} f(\boldsymbol{x}^{\boldsymbol{K}_i}_{(k_{i+1})}) \oplus D_{\boldsymbol{K}_{i+1}} f(\boldsymbol{x}^{\boldsymbol{K}_i}_{(k_{i+1})}), \\ \quad \vdots \\ D_{\boldsymbol{K}_1} f(\boldsymbol{x}^{\boldsymbol{K}_1}) = D_{\boldsymbol{K}_1} f(\boldsymbol{x}^{\boldsymbol{K}_1}_{(k_2)}) \oplus D_{\boldsymbol{K}_2} f(\boldsymbol{x}^{\boldsymbol{K}_1}_{(k_2)}), \\ D_{\boldsymbol{K}_0} f(\boldsymbol{x}^{\boldsymbol{K}_0}) = D_{\boldsymbol{K}_0} f(\boldsymbol{x}^{\boldsymbol{K}_0}_{(k_1)}) \oplus D_{\boldsymbol{K}_1} f(\boldsymbol{x}^{\boldsymbol{K}_0}_{(k_1)}) \end{cases} \tag{2}$$

where $\boldsymbol{K}_i = (k_0, k_1, \ldots, k_i)$ for $i \geq 0$. Note here that, the vectors $\boldsymbol{x}^{\boldsymbol{K}_i}_{(k_{i+1})}$ and $\boldsymbol{x}^{\boldsymbol{K}_{i+1}}$ for $i \geq 0$ are same. Starting from the highest-order derivative $D_{\boldsymbol{K}_\ell} f(\boldsymbol{x}^{\boldsymbol{K}_\ell})$, which is available to use, the value of $D_{\boldsymbol{K}_{\ell-1}} f(\boldsymbol{x}^{\boldsymbol{K}_{\ell-1}})$ can be obtained using one XOR operation, provided that $D_{\boldsymbol{K}_{\ell-1}} f(\boldsymbol{x}^{\boldsymbol{K}_{\ell-1}}_{(k_\ell)})$ is already known. Proceeding similarly through the remaining identities in (2), and assuming at each step that the first term after equality is known beforehand, the required derivatives are recovered successively. Finally, this procedure yields $f(\boldsymbol{x}) = D_{\boldsymbol{K}_0} f(\boldsymbol{x}^{(\boldsymbol{K}_0)})$. Consequently, $f(\boldsymbol{x})$ can be recovered in $\mathcal{O}(d)$ operations.

Based on this non-recursive formulation, the authors introduced two key abstract notions, namely *evaluation rule* and *evaluation order*, together with a sufficient condition for efficient evaluation of f over any ordered space.

Evaluation Rule: Let $g(x_n, \ldots, x_1)$ be a polynomial evaluated over an ordered set S. An *evaluation rule* $\mathcal{R}$ assigns to every $\boldsymbol{x} \in S$ with $\boldsymbol{x} \neq S[0]$ a unique $\boldsymbol{x}' \in S$ such that

$$g(\boldsymbol{x}) = g(\boldsymbol{x}') \oplus D_k(g)(\boldsymbol{x}'), \quad \text{where } \boldsymbol{x} \oplus \boldsymbol{x}' = \boldsymbol{e}_k.$$

This relation is denoted by $\boldsymbol{x}' \xrightarrow{\mathcal{R}} \boldsymbol{x}$.

Evaluation Order: For a polynomial g evaluated over an ordered set S, we write $\boldsymbol{x}' <_g \boldsymbol{x}$ if the evaluation of $g(\boldsymbol{x}')$ is performed before that of $g(\boldsymbol{x})$.

If $\boldsymbol{x}' <_g \boldsymbol{x}$ and $\boldsymbol{x}' \xrightarrow{\mathcal{R}} \boldsymbol{x}$, we call $\boldsymbol{x}'$ a *predecessor* of $\boldsymbol{x}$ under the evaluation rule $\mathcal{R}$.

As already discussed, every derivative appearing in Equation 2 is a polynomial; therefore, the above concepts of the evaluation rule and the evaluation order can be applied to them. We denote by $S_{\boldsymbol{K}_i}$ the set of input points for the derivative $D_{\boldsymbol{K}_i} f$, equipped with its corresponding evaluation order, and by $\mathcal{R}_{\boldsymbol{K}_i}$ the corresponding evaluation rule. The non-recursive formulation further yields the following sufficient condition for a given evaluation order and rule to make the above framework work efficiently.

Theorem 1 (Sufficient condition for derivative-based evaluation [26]) *Let $f(x_n, \ldots, x_1)$ be a polynomial of degree d evaluated over an ordered set S_0. Consider any triple $(g, S, \mathcal{R})$ belonging to*

$$\left\{(D_{\boldsymbol{K}_i} f, S_{\boldsymbol{K}_i}, \mathcal{R}_{\boldsymbol{K}_i}) \mid i = 0, \ldots, d-1\right\},$$

where g is a polynomial arising during the non-recursive evaluation process, S is the ordered input set associated with g, and $\mathcal{R}$ is the corresponding evaluation rule. If for every such $(g, S, \mathcal{R})$ and for all $\boldsymbol{x} \in S$ with $\boldsymbol{x} \neq S[0]$, there exists a predecessor $\boldsymbol{x}'$ then the evaluation of f over S_0 can be carried out using $\mathcal{O}(d \cdot |S_0|)$ operations, in addition to the cost of computing all initial values $g(S[0])$ beforehand.

If we choose the evaluation rule $\mathcal{R}$ as described in Equation (1) for all derivative polynomial $D_{\mathbf{K}_i} f$, the above theorem guarantees that every evaluation depends only on previously computed values and at most one higher-order derivative. As a result, the total evaluation complexity is linear in $|S_0|$, up to a multiplicative factor d.

Using the lexicographic order as the evaluation order, the evaluation rule for all derivative polynomial as defined in Equation (1), and the sufficient condition in Theorem 1, the authors of [26] showed that the evaluation of f over a Hamming ball S can be performed in $d \cdot |S| + \binom{n}{\leq d}$ operations, with a memory requirement of $2 \cdot \binom{n}{\leq d}$ bits. They also presented an explicit algorithm (Listing 1.2: Polynomial evaluation in [26]) realizing this framework. Since our approach builds directly upon this algorithm, we next review it. The algorithm consist of two phase: *Initialization phase*, in which the value of all possible derivatives at 0^n is computed and stored. These values can be seen as $g(S[0])$ as per theorem 1. The next phase is the *evaluation phase* in which the values of $f(\boldsymbol{x})$ is computed in the lexicographic order using non-recursive formulation (2). At each step, the algorithm moves to the predecessor input by decrementing the last nonzero coordinate by 1. Consequently, every recursive step corresponds to advancing one level deeper in the derivative chain defined by (2). This process continues until either a derivative of order d is reached or the input becomes 0^n. In the former case, the d-th order derivative is a constant polynomial. Consequently, the required value is noting but the value at 0^n. Thus in either case, the required value is $g(S[0])$ for some g and corresponding S, which is already available from the initialization phase. There are two of key components of the algorithm: *memory allocation and access*, which determine where the required derivative values are stored so that they can be efficiently retrieved during evaluation, and *memory update*, which allows the allocated memory to be reused and thereby reduces the overall memory requirement.

To understand these components, recall that each step of the non-recursive formulation in (2) requires certain previously computed derivative values to remain available. A straightforward implementation that stores all such values simultaneously would therefore incur a prohibitively large memory requirement. To reduce the resulting memory requirement, the authors introduced a reuse-and-release strategy. Specifically, a family of arrays $A_{\mathbf{K}_i}$, for $0 \leq i \leq d-1$, are created which store evaluations of the corresponding derivative polynomials $D_{\mathbf{K}_i} f$. These arrays are managed using reuse-and-release strategy , whereby entries that are no longer required are overwritten by newly computed derivative values. Figure 1 illustrates the organization of a few such arrays.

We first describe the memory allocation and access scheme used for these arrays. In particular, the following lemma specifies the form of the inputs at which each derivative polynomial is evaluated and identifies the array entries used to store the corresponding evaluations.

Lemma 1 *For every derivative polynomial $D_{\mathbf{K}_i} f$, the input is of the form $(\mathbf{a}, \underbrace{0, \ldots, 0}_{k_i})$ where $\mathbf{a} \in \mathbb{F}_2^{n-k_i}$. Moreover, the associated array $A_{\mathbf{K}_i}$ of length $n - k_i + 1$ is used to store these values. In particular, $A_{\mathbf{K}_i}[0]$ stores $D_{\mathbf{K}_i} f(0^n)$, whereas $A_{\mathbf{K}_i}[j]$ for $j \geq 1$, is used to store $D_{\mathbf{K}_i} f(*, 1, \underbrace{0, \ldots, 0}_{j-1}, \underbrace{0, \ldots, 0}_{k_i})$, where $*$ denotes an arbitrary binary string of suitable length.*

By Lemma 1, the inputs to $D_{\mathbf{K}_i} f$ are of the form $(\mathbf{a}, \underbrace{0, \ldots, 0}_{k_i})$. A straightforward implementation would therefore require storing all 2^{n-k_i} evaluations simultaneously which requires huge memory. Instead, using the reuse-and-release strategy we only requires an array of length only $n - k_i + 1$ to stores these values. It was shown that once $D_{\mathbf{K}_i} f(\mathbf{a}, 1, \underbrace{0, \ldots, 0}_{j+k_i-1})$ has been computed, all previously stored values $D_{\mathbf{K}_i} f(\mathbf{a}', 1, \underbrace{0, \ldots, 0}_{j+k_i-1})$ with $(\mathbf{a}', 1, \underbrace{0, \ldots, 0}_{j+k_i-1}) <_{\text{lex}} (\mathbf{a}, 1, \underbrace{0, \ldots, 0}_{j+k_i-1})$ become obsolete and can be safely overwritten. Consequently, each array entry (except the first one, which permanently stores the value at 0^n) can be reused throughout the evaluation.

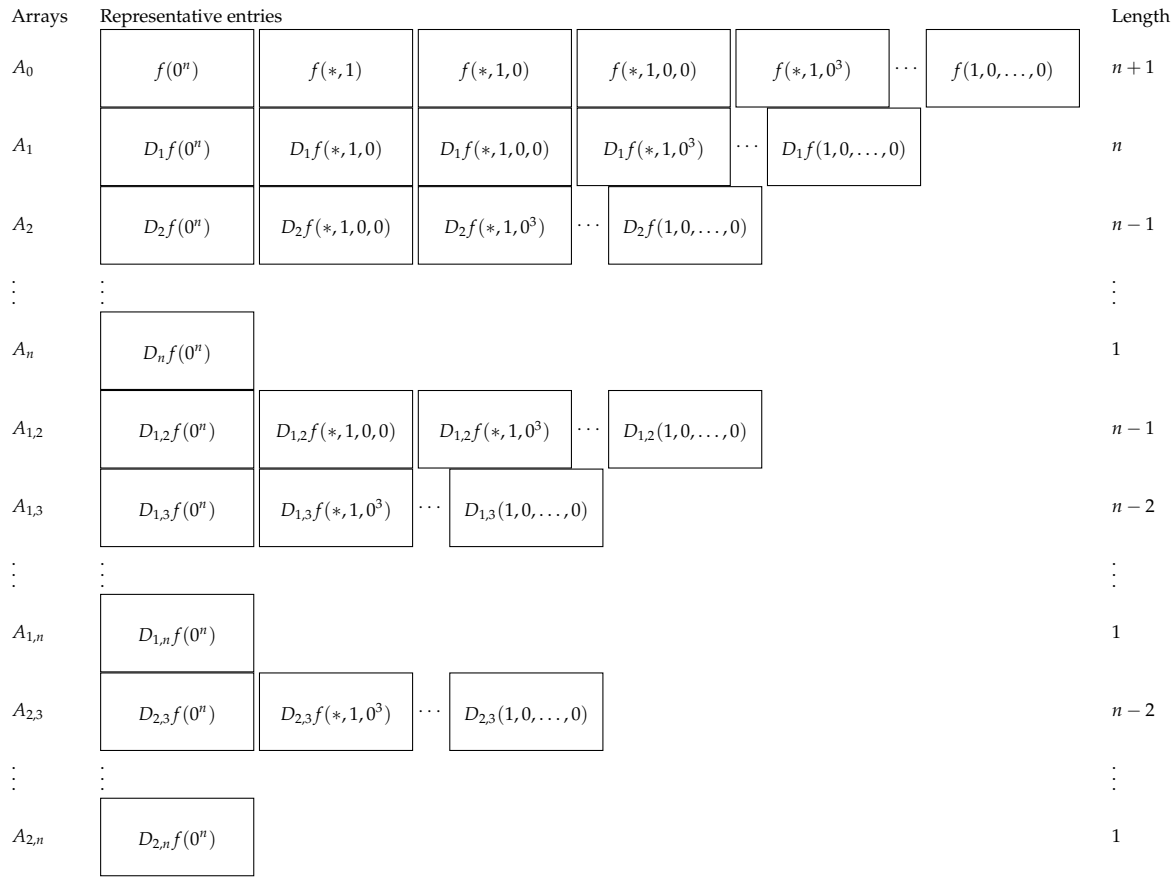

Fig. 1: Illustration of the organization of different types of arrays. '*' represents a random binary string of suitable length.

Memory update rule. Consider the i-th step of non-recusive formulation (2),

$$D_{\boldsymbol{K}_i} f(\boldsymbol{x}^{\boldsymbol{K}_i}) = D_{\boldsymbol{K}_i} f(\boldsymbol{x}^{\boldsymbol{K}_i}_{(k_{i+1})}) \oplus D_{\boldsymbol{K}_{i+1}} f(\boldsymbol{x}^{\boldsymbol{K}_i}_{(k_{i+1})}) = D_{\boldsymbol{K}_i} f(\boldsymbol{x}^{\boldsymbol{K}_{i+1}}) \oplus D_{\boldsymbol{K}_{i+1}} f(\boldsymbol{x}^{\boldsymbol{K}_{i+1}}). \tag{3}$$

According to Lemma 1, the values of derivatives $D_{\boldsymbol{K}_i} f$ and $D_{\boldsymbol{K}_{i+1}} f$ are stored in arrays $A_{\boldsymbol{K}_i}$ and $A_{\boldsymbol{K}_{i+1}}$, respectively. Moreover, the choice of the index of these arrays in which these values are stored is governed by the form of their inputs. In particular,

$$\boldsymbol{x}^{(\boldsymbol{K}_i)} = (\mathbf{a}, \underbrace{0, \ldots, 0}_{k_i}) \quad \text{and} \quad \boldsymbol{x}^{(\boldsymbol{K}_{i+1})} = (\mathbf{b}, \underbrace{0, \ldots, 0}_{k_{i+1}}),$$

where $\boldsymbol{x}^{(\boldsymbol{K}_{i+1})}$ being the predecessor of $\boldsymbol{x}^{(\boldsymbol{K}_i)}$ follow

$$\boldsymbol{x}^{(\boldsymbol{K}_i)} \oplus \boldsymbol{x}^{(\boldsymbol{K}_{i+1})} = e_{k_{i+1}}. \tag{4}$$

Accordingly, the memory update naturally splits into the following two cases.

Case 1: The predecessor is the origin. Suppose $\mathbf{b} = \mathbf{0}$. This occurs precisely when $i = h - 1$, where $h = \mathrm{Hw}(\boldsymbol{x})$. Importantly, this case can occur before the order of the derivative reaches d, i.e., $h \leq d$. If $h > d$, the derivative order d is reaches before the predecessor becomes the origin, which we discuss as a boundary case later.

From Equation (4),

$$\boldsymbol{x}^{(\boldsymbol{K}_i)} = \boldsymbol{x}^{(\boldsymbol{K}_{h-1})} = e_{k_h}, \qquad \boldsymbol{x}^{(\boldsymbol{K}_{i+1})} = \boldsymbol{x}^{(\boldsymbol{K}_h)} = \mathbf{0},$$

and therefore Equation (3) can be seen as

$$D_{\boldsymbol{K}_{h-1}} f(e_{k_h}) = D_{\boldsymbol{K}_{h-1}} f(\mathbf{0}) \oplus D_{\boldsymbol{K}_h} f(\mathbf{0}).$$

Since both values on the right-hand side are permanently stored in the first entries of their corresponding arrays, i.e., in $A_{\boldsymbol{K}_{h-1}}[0]$ and $A_{\boldsymbol{K}_h}[0]$, respectively during the initialization phase, the array update is simply

$$A_{\boldsymbol{K}_{h-1}}[k_h - k_{h-1}] = A_{\boldsymbol{K}_{h-1}}[0] \oplus A_{\boldsymbol{K}_h}[0].$$

Case 2: The predecessor is not the origin. Now suppose $\mathbf{b} \neq \mathbf{0}$. Then $\mathbf{b}$ must be of the form

$$(\mathbf{b}', 1, \underbrace{0, \ldots, 0}_{j-1}, \underbrace{0, \ldots, 0}_{k_{i+1}}),$$

where $j = k_{i+2} - k_{i+1}$. Consequently, from Equation (4)

$$\boldsymbol{x}^{(\boldsymbol{K}_i)} = (\mathbf{b}', 1, \underbrace{0, \ldots, 0}_{j-1}, 1, \underbrace{0, \ldots, 0}_{k_{i+1}-1}).$$

Thus, Equation (3) can be seen as

$$D_{\boldsymbol{K}_i} f(\mathbf{b}', 1, \underbrace{0, \ldots, 0}_{j-1}, 1, \underbrace{0, \ldots, 0}_{k_{i+1}-1}) = D_{\boldsymbol{K}_i} f(\mathbf{b}', 1, \underbrace{0, \ldots, 0}_{j-1}, \underbrace{0, \ldots, 0}_{k_{i+1}}) \oplus D_{\boldsymbol{K}_{i+1}} f(\mathbf{b}', 1, \underbrace{0, \ldots, 0}_{j-1}, \underbrace{0, \ldots, 0}_{k_{i+1}}).$$

Hence, the corresponding memory update is

$$A_{\boldsymbol{K}_i}[k_{i+1} - k_i] = A_{\boldsymbol{K}_i}[k_{i+2} - k_i] \oplus A_{\boldsymbol{K}_{i+1}}[k_{i+2} - k_{i+1}].$$

Boundary case: $i = d - 1$. When the order of the derivative at the i-th step reaches to the d but input of the corresponding derivative is still nonzero, i.e., $h > d$. In this case, the degree of such derivative becomes zero and therefore the derivative is constant. Hence,

$$D_{\boldsymbol{K}_d} f(\mathbf{b}', 1, \underbrace{0, \ldots, 0}_{j-1}, \underbrace{0, \ldots, 0}_{k_d}) = D_{\boldsymbol{K}_d} f(\mathbf{0}),$$

which is already available from the initialization phase. Consequently, the update rule simplifies to

$$A_{\boldsymbol{K}_{d-1}}[k_d - k_{d-1}] = A_{\boldsymbol{K}_{d-1}}[k_{d+1} - k_{d-1}] \oplus A_{\boldsymbol{K}_d}[0].$$

The preceding cases describe the complete update process. For an input $\boldsymbol{x}$ of Hamming weight h, the evaluation is initialized from a derivative value already stored during the initialization phase. If $h \leq d$, this value is $D_{\boldsymbol{K}_h} f(0^n)$, corresponding to Case 1, whereas if $h > d$, the algorithm initialized with $D_{\boldsymbol{K}_d} f(\mathbf{0})$. Starting from this available value, the algorithm successively applies the update of Case 2 in decreasing derivative order until the order-zero value $f(\boldsymbol{x})$ is obtained. Consequently, at most $\min\{h, d\} \leq d$ XOR operations are required for each evaluation.

4 Framework over $\mathbb{F}_q^n$

We consider the problem of evaluating a polynomial $f(x_n, \ldots, x_1)$ in n variables of degree d on a product Hamming ball as an evaluation set. For a positive integer n_i and an integer $0 \leq w_i \leq n_i$, let

$$B(n_i, w_i) = \{\boldsymbol{x} \in \mathbb{F}_q^{n_i} \mid \mathrm{Hw}(\boldsymbol{x}) \leq w_i\}$$

denote the Hamming ball centered at the origin, containing all vectors of weight at most w_i. More generally, for a tuple $\boldsymbol{n} = (n_s, \ldots, n_1)$, of non-negative integers satisfying $n = n_1 + \cdots + n_s$ with corresponding weights $\boldsymbol{w} = (w_s, \ldots, w_1)$, we define the *product Hamming ball* as

$$B(\boldsymbol{n}, \boldsymbol{w}) = B(n_s, w_s) \times \cdots \times B(n_1, w_1) \subseteq \mathbb{F}_q^n.$$

Before developing our derivative-based evaluation framework, we relate these evaluation domains to structured MME framework of van der Hoeven and Schost for downward-closed index sets [29].

4.1 Product Hamming Balls as Downward-Closed Evaluation Spaces

Recall from Section 3.3 that, for a downward-closed index set I and coordinate evaluation sequences $\boldsymbol{u} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n)$, the associated evaluation space is denoted by $U(I, \boldsymbol{u})$. We first show that every product Hamming ball over $\mathbb{F}_q$ admits such a representation.

Theorem 2 *For every product Hamming ball $B(\boldsymbol{n}, \boldsymbol{w})$ over $\mathbb{F}_q$, there exist integers $d_1, \ldots, d_n$, coordinate evaluation sequences $\boldsymbol{u} = \{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n\}$, and a downward-closed index set $I_B \subseteq \{0, \ldots d_1 - 1\} \times \cdots \times \{0, \ldots, d_n - 1\}$ such that*

$$U(I_B, \boldsymbol{u}) = B(\boldsymbol{n}, \boldsymbol{w}).$$

Moreover, every polynomial over $\mathbb{F}_q$ has an equivalent representative, on the evaluation space $B(\boldsymbol{n}, \boldsymbol{w})$, whose monomial support is contained in I_B.

Proof We first construct a downward-closed index set whose associated evaluation space is exactly the product Hamming ball. Then we show that every polynomial admits an equivalent representative supported on the constructed downward-closed index set.

Construction of the downward-closed set. For the i-th block, let

$$J_i = \left\{ \sum_{r=1}^{i-1} n_r + 1, \ldots, \sum_{r=1}^{i} n_r \right\}$$

denote its set of coordinate indices. For each $j \in J_i$, set d_j and choose the coordinate evaluation sequence as follows:

$$d_j = \begin{cases} 1, & w_i = 0, \\ q, & w_i > 0, \end{cases} \quad \text{and} \quad \boldsymbol{u}_j = \begin{cases} (0), & w_i = 0, \\ (0, 1, \ldots, q-1), & w_i > 0. \end{cases}$$

Define

$$I_B = \left\{ \boldsymbol{\alpha} \in \prod_{j=1}^{n} \{0, \ldots, d_j - 1\} : \left|\mathrm{Sup}(\boldsymbol{\alpha}) \cap J_i\right| \leq w_i \text{ for every } i = 1, \ldots, s \right\}.$$

Let $\boldsymbol{\alpha} \in I_B$ and $\boldsymbol{\beta} \preceq \boldsymbol{\alpha}$. If $\alpha_j = 0$, then necessarily $\beta_j = 0$, and hence $\mathrm{Sup}(\boldsymbol{\beta}) \subseteq \mathrm{Sup}(\boldsymbol{\alpha})$. Consequently, for every block J_i,

$$\left|\mathrm{Sup}(\boldsymbol{\beta}) \cap J_i\right| \leq \left|\mathrm{Sup}(\boldsymbol{\alpha}) \cap J_i\right| \leq w_i.$$

Thus $\boldsymbol{\beta} \in I_B$, proving that I_B is downward closed.

Equality of the evaluation domains. Consider the associated evaluation space

$$U(I_B, \boldsymbol{u}) = \left\{ (u_{1,\alpha_1}, \ldots, u_{n,\alpha_n}) : \boldsymbol{\alpha} \in I_B \right\}.$$

By the above choice of the coordinate evaluation sequences, for every $\boldsymbol{\alpha} \in I_B$, the associated evaluation point is $(u_{1,\alpha_1}, \ldots, u_{n,\alpha_n}) = (\alpha_1, \ldots, \alpha_n)$. Moreover, the defining condition of I_B is precisely that the number of nonzero coordinates in the i-th block is at most w_i. Therefore, $U(I_B, \boldsymbol{u}) = B(\boldsymbol{n}, \boldsymbol{w})$.

Reduction of the polynomial support. First replace f by its canonical polynomial-function representative modulo $x_1^q - x_1, \ldots, x_n^q - x_n$. Thus every exponent lies in $\{0, \ldots, q-1\}^n$. Let

$$f(\boldsymbol{x}) = \sum_{\boldsymbol{\beta} \in J \subseteq \{0,\ldots,q-1\}^n} a_{\boldsymbol{\beta}} \boldsymbol{x}^{\boldsymbol{\beta}}$$

be a polynomial over $\mathbb{F}_q$. Consider a monomial $a_{\boldsymbol{\beta}} \boldsymbol{x}^{\boldsymbol{\beta}}$ with $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_n) \notin I_B$. By construction of I_B, there exists a block J_i such that $|\mathrm{Sup}(\boldsymbol{\beta}) \cap J_i| > w_i$. On the other hand, every point $\boldsymbol{x} \in B(\boldsymbol{n}, \boldsymbol{w})$ has at most w_i nonzero coordinates in the block J_i. Hence, at least one coordinate $j \in \mathrm{Sup}(\boldsymbol{\beta}) \cap J_i$ satisfies $x_j = 0$ and $\beta_j > 0$. Consequently, and the monomial $a_{\boldsymbol{\beta}} \boldsymbol{x}^{\boldsymbol{\beta}}$ does not contribute to the value of f on the evaluation domain, i.e.,

$$\boldsymbol{x}^{\boldsymbol{\beta}} = 0$$

at each evaluation input.

Applying the same argument to every $\boldsymbol{\beta}$ outside I_B yields the polynomial

$$\tilde{f}(\boldsymbol{x}) = \sum_{\boldsymbol{\beta} \in J \cap I_B} a_{\boldsymbol{\beta}} \boldsymbol{x}^{\boldsymbol{\beta}},$$

which agrees with f at every point of $B(\boldsymbol{n}, \boldsymbol{w})$. Hence every polynomial over $\mathbb{F}_q$ admits an equivalent representative on $B(\boldsymbol{n}, \boldsymbol{w})$ whose monomial support is contained in I_B.

Theorem 2 shows that every product Hamming ball considered in this work can be represented as a downward-closed evaluation space $U(I_B, \boldsymbol{u})$ in the framework of van der Hoeven and Schost. Therefore, their multipoint-evaluation complexity bound applies directly to these evaluation domains. We recall this result below, which will later serve as a reference for comparing their recursive multipoint-evaluation method with the derivative-based framework developed in this work.

Theorem 3 *Let*

$$I \subseteq \{0, \ldots, d_1 - 1\} \times \cdots \times \{0, \ldots, d_n - 1\}$$

be a downward-closed index set, and let $U(I, \boldsymbol{u})$ be the corresponding evaluation space. A polynomial whose monomial support is contained in I can be evaluated at all points of $U(I, \boldsymbol{u})$ in

$$\mathcal{O}\left(|I| \sum_{j=1}^{n} \frac{M_{\mathrm{mul}}(d_j) \log d_j}{d_j} \right)$$

field operations, where $M_{\mathrm{mul}}(t)$ denotes the multiplication time function and can be taken as $M_{\mathrm{mul}}(t) = \mathcal{O}(t \log t \log\log t)$.

4.2 Derivative based framework over the Finite Fields of prime order $\mathbb{F}_q$

Having reviewed the derivative-based framework of Liu et al. in the Boolean setting, we now extend its constituent components to finite fields $\mathbb{F}_q$ of prime order. In particular, we derive a non-recursive evaluation formula that identifies the derivative polynomials and intermediate inputs arising during the evaluation process. Building upon this formulation, we define evaluation rules and evaluation orders over $\mathbb{F}_q$, establish a sufficient condition for their applicability, and finally develop the corresponding memory allocation and update rules, leading to an algorithm for evaluating f over product Hamming ball.

Let $f : \mathbb{F}_q^n \to \mathbb{F}_q$ is a polynomial of total degree d, represented in its q-ary algebraic normal form (ANF) as

$$f(\boldsymbol{x}) = \sum_{\substack{\boldsymbol{\beta}\in\{0,\ldots,q-1\}^n \\ |\boldsymbol{\beta}|\le d}} c_{\boldsymbol{\beta}} \boldsymbol{x}^{\boldsymbol{\beta}}, \tag{5}$$

where $\boldsymbol{x}^{\boldsymbol{\beta}} = x_1^{\beta_1} \cdots x_n^{\beta_n}$, $c_{\boldsymbol{\beta}} \in \mathbb{F}_q$. Recall that the componentwise partial order $\preceq$ on $\mathbb{N}^n$ was introduced in Section 3.3. Since the elements of $\mathbb{F}_q$ are represented throughout this work by their canonical integer representatives $0, 1, \ldots, q-1$, the same order naturally induces a componentwise partial order on $\mathbb{F}_q^n$.

Definition 3 For $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^n$, we write $\boldsymbol{x} \preceq \boldsymbol{y}$ if and only if $x_i \le y_i$, for all $i \in \{1, \ldots, n\}$, where the inequalities are understood between the corresponding canonical integer representatives.

A subset $S \subseteq \mathbb{F}_q^n$ is called *downward-closed* if

$$\boldsymbol{y} \in S, \qquad \boldsymbol{x} \preceq \boldsymbol{y} \quad \Longrightarrow \quad \boldsymbol{x} \in S.$$

Notice that the componentwise partial order $\preceq$ is distinct from the lexicographic order $<_{\text{lex}}$. The former describes the closure property of a set, whereas the latter specifies the order in which its elements are arranged.

Definition 4 (i) For $k \in \{1, \ldots, n\}$, the derivative of f with respect to the variable x_k is defined as

$$D_k f(\boldsymbol{x}) := f(\boldsymbol{x} + \boldsymbol{e}_k) - f(\boldsymbol{x})$$

where $\boldsymbol{x} \in \mathbb{F}_q^n$.

(ii) For $\alpha \ge 2$, the α-th order derivative of f with respect to x_k is recursively defined by

$$D_k^\alpha f(\boldsymbol{x}) := D_k^{\alpha-1} f(\boldsymbol{x} + \boldsymbol{e}_k) - D_k^{\alpha-1} f(\boldsymbol{x}),$$

with $D_k^1 f := D_k f$. For a multi-index $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$, define the corresponding higher-order derivative operator by

$$D^{\boldsymbol{\alpha}} f = D_1^{\alpha_1} \cdots D_n^{\alpha_n} f.$$

In particular, if $\boldsymbol{K}_l = (k_1, \ldots, k_l)$ denotes the indices of the nonzero coordinates of $\boldsymbol{\alpha}$ and $\beta_i = \alpha_{k_i}$ be corresponding value of the coordinate then we write

$$D^{\boldsymbol{\alpha}} f = D_{\boldsymbol{K}_l}^{\boldsymbol{\beta}_l} f = D_{k_l}^{\beta_l} \cdots D_{k_1}^{\beta_1} f,$$

where $\boldsymbol{\beta}_l = (\beta_1, \ldots, \beta_l)$.

Our construction builds upon the recursive evaluation formula introduced in the fast enumeration algorithm of [18]. For every $k \in [1, n]$,

$$f(\boldsymbol{x}) = f(\boldsymbol{x}_{(k)}) + D_k f(\boldsymbol{x}_{(k)}). \tag{6}$$

As in the Boolean setting discussed in the subsection 3.4, repeated application of this relation recursively reduces the evaluation of $f(\boldsymbol{x})$ to that of a derivative polynomial whose value can be efficiently computed. Our objective is to explicitly characterize the derivative polynomials and their corresponding evaluation points generated during the evaluation process.While deriving a non-recursive formulation analogous to the Boolean case, an important distinction arises from the possible coordinate values. Over $\mathbb{F}_2$, a nonzero coordinate is necessarily equal to 1 and hence reaches 0 after a single evaluation step. Over $\mathbb{F}_q$, a coordinate may take any canonical value in $0, \ldots, q-1$ and is successively reduced as $x_k, x_k - 1, \ldots, 1, 0$. Consequently, repeated differentiation with respect to the same variable may be required, with relevant derivative orders ranging up to $q-1$ for a reduced polynomial. To describe the intermediate derivative order with respect to the currently processed variable, we introduce the parameter δ. Importantly, the recursion exhausts the least-index nonzero coordinate before moving to the next support coordinate; consequently, earlier support coordinates have already contributed their full derivative orders when a later coordinate becomes active.

Taking these intermediate transitions into account, the recursive evaluation process admits the following non-recursive description. To evaluate $f(\boldsymbol{x})$, there exists an integer $\epsilon \in \{1, \cdots, d\}$, distinct indices $\{k_1, \ldots, k_\epsilon\} \subseteq \{1, \ldots, n\}$, $k_0 = 0$, and an integer $0 \leq \delta \leq x_{k_\epsilon}$ such that $\sum_{i<\epsilon} x_{k_i} + \delta < d$ and the evaluation proceeds through the sequence

$$\begin{cases}
D_{\boldsymbol{K}_\epsilon}^{(0,x_{k_1},\ldots,x_{k_{\epsilon-1}},\delta+1)} f\left(\boldsymbol{x}_{(k_\epsilon^{\delta+1})}^{\boldsymbol{K}_{\epsilon-1}}\right) & \text{can be efficiently computed,} \\
D_{\boldsymbol{K}_\epsilon}^{(0,x_{k_1},\ldots,x_{k_{\epsilon-1}},\delta)} f\left(\boldsymbol{x}_{(k_\epsilon^{\delta})}^{\boldsymbol{K}_{\epsilon-1}}\right) = D_{\boldsymbol{K}_\epsilon}^{(0,x_{k_1},\ldots,x_{k_{\epsilon-1}},\delta)} f\left(\boldsymbol{x}_{(k_\epsilon^{\delta+1})}^{\boldsymbol{K}_{\epsilon-1}}\right) + D_{\boldsymbol{K}_\epsilon}^{(0,x_{k_1},\ldots,x_{k_{\epsilon-1}},\delta+1)} f\left(\boldsymbol{x}_{(k_\epsilon^{\delta+1})}^{\boldsymbol{K}_{\epsilon-1}}\right) \\
\vdots \\
D_{\boldsymbol{K}_{\epsilon-1}}^{(0,x_{k_1},\ldots,x_{k_{\epsilon-1}})} f\left(\boldsymbol{x}^{\boldsymbol{K}_{\epsilon-1}}\right) = D_{\boldsymbol{K}_{\epsilon-1}}^{(0,x_{k_1},\ldots,x_{k_{\epsilon-1}})} f\left(\boldsymbol{x}_{(k_\epsilon)}^{\boldsymbol{K}_{\epsilon-1}}\right) + D_{\boldsymbol{K}_\epsilon}^{(0,x_{k_1},\ldots,x_{k_{\epsilon-1}},1)} f\left(\boldsymbol{x}_{(k_\epsilon)}^{\boldsymbol{K}_{\epsilon-1}}\right) \\
\vdots \\
D_{\boldsymbol{K}_1}^{(0,x_{k_1})} f\left(\boldsymbol{x}^{\boldsymbol{K}_1}\right) = D_{\boldsymbol{K}_1}^{(0,x_{k_1})} f\left(\boldsymbol{x}_{(k_2)}^{\boldsymbol{K}_1}\right) + D_{\boldsymbol{K}_2}^{(0,x_{k_1},1)} f\left(\boldsymbol{x}_{(k_2)}^{\boldsymbol{K}_1}\right) \\
D_{\boldsymbol{K}_1}^{(0,x_{k_1}-1)} f\left(\boldsymbol{x}_{\left(k_1^{x_{k_1}-1}\right)}^{\boldsymbol{K}_0}\right) = D_{\boldsymbol{K}_1}^{(0,x_{k_1}-1)} f\left(\boldsymbol{x}_{\left(k_1^{x_{k_1}}\right)}^{\boldsymbol{K}_0}\right) + D_{\boldsymbol{K}_1}^{(0,x_{k_1})} f\left(\boldsymbol{x}_{\left(k_1^{x_{k_1}}\right)}^{\boldsymbol{K}_0}\right) \\
\vdots \\
D_{\boldsymbol{K}_1}^{(0,1)} f\left(\boldsymbol{x}_{(k_1)}^{\boldsymbol{K}_0}\right) = D_{\boldsymbol{K}_1}^{(0,1)} f\left(\boldsymbol{x}_{(k_1^2)}^{\boldsymbol{K}_0}\right) + D_{\boldsymbol{K}_1}^{(0,2)} f\left(\boldsymbol{x}_{(k_1^2)}^{\boldsymbol{K}_0}\right) \\
D_{\boldsymbol{K}_0}^{(0)} f\left(\boldsymbol{x}^{\boldsymbol{K}_0}\right) = D_{\boldsymbol{K}_0}^{(0)} f\left(\boldsymbol{x}_{(k_1)}^{\boldsymbol{K}_0}\right) + D_{\boldsymbol{K}_1}^{(0,1)} f\left(\boldsymbol{x}_{(k_1)}^{\boldsymbol{K}_0}\right).
\end{cases} \tag{7}$$

Equation (7) therefore expresses the recursive evaluation procedure as a sequence of derivative evaluations, making every intermediate derivative polynomial and evaluation point explicit. For brevity, we write $D_{\boldsymbol{K}_\epsilon}^{\delta} f$[1] in place of $D_{\boldsymbol{K}_\epsilon}^{(0,x_{k_1},\ldots,x_{k_{\epsilon-1}},\delta)} f$. Its total order is $\sum_{r<\epsilon} x_{k_r} + \delta$: the coordinates $x_{k_1}, \ldots, x_{k_{\epsilon-1}}$ have already contributed their respective full orders, i.e., $x_{k_1}, \ldots, x_{k_{\epsilon-1}}$ respectively, while the current coordinate x_{k_ϵ} contributes order δ.

[1]The following two expressions $D_{\boldsymbol{K}_\epsilon}^{x_{k_\epsilon}} f$, $D_{\boldsymbol{K}_{\epsilon+1}}^{0} f$ denote the same derivative polynomial.

Remark 1 The non-recursive formulation in (7) preserves the order in which coordinates are processed in the relation (6). In particular, if

$$(k_1, \ldots, k_h) = \mathrm{Sup}(x_n, \ldots, x_1),$$

then differentiation is first performed repeatedly with respect to x_{k_1} until its order reaches x_{k_1}. Only then can differentiation with respect to x_{k_2} occur. More generally, differentiation with respect to x_{k_ϵ} occurs only after the derivatives with respect to $x_{k_1}, \ldots, x_{k_{\epsilon-1}}$ have attained the respective orders $x_{k_1}, \ldots, x_{k_{\epsilon-1}}$. At each application of the relation (6), the derivative order with respect to the active coordinate increases by one while the corresponding input coordinate decreases by one. Consequently, the coordinates with indices smaller than k_ϵ are zero when derivative with respect to x_{k_ϵ} is being processed.

Similar to the Boolean case, it is assumed that at each step the value of the term just after the equality is known beforehand, whereas the value of the other term can be obtained from the previous step. For efficient evaluation over an evaluation space, this assumption must hold for every input in the space and at every step of its evaluation. This requirement is captured by the following sufficient condition, which generalizes the corresponding condition of Liu et al. from the Boolean setting to arbitrary prime fields.

Theorem 4 *Let $f(x_n, \ldots, x_1)$ be a polynomial of degree d evaluated over an ordered input set S_0 according to Equation (7). For any non-constant polynomial g appearing in Equation (7), let S be the ordered input set associated with g, and let $\mathcal{R}$ denote the evaluation rule used to evaluate g over S. Suppose that, for every triple*

$$(g, S, \mathcal{R}) \in \left\{ (D^{\delta}_{\mathbf{K}_\epsilon} f, S^{\delta}_{\mathbf{K}_\epsilon}, \mathcal{R}^{\delta}_{\mathbf{K}_\epsilon}) \right\},$$

the condition

$$\forall \boldsymbol{x} \in S,\ \boldsymbol{x} \neq S[0],\ \exists\, \boldsymbol{x}' \in S \text{ such that } \boldsymbol{x}' <_g \boldsymbol{x} \quad \text{and} \quad \boldsymbol{x}' \xrightarrow{\mathcal{R}} \boldsymbol{x}$$

holds. Then evaluating f over S_0 requires at most $d \cdot |S_0|$ operations, in addition to the efficient cost of computing all initial values $g(S[0])$ beforehand, to initialize the evaluation process as needed at the last step in Equation (7).

Proof To evaluate a polynomial over an ordered input set S_0 using Equation (7), we present an algorithm consisting of two phases: the initialization phase and the evaluation phase. During the evaluation phase, an evaluation table TAB^{g}_{eva} is associated with each non-constant derivative polynomial g arising in the non-recursive formulation. Whenever $g(\boldsymbol{y})$ is computed for an input $\boldsymbol{y}$ in its associated ordered set S, the pair $(\boldsymbol{y}, g(\boldsymbol{y}))$ is stored in TAB^{g}_{eva}. These tables therefore contain the previously computed derivative values required by subsequent evaluations and collectively constitute the working memory of the evaluation phase. These tables are progressively updated as the evaluation proceeds and collectively constitute the working memory of the algorithm.

Consider an arbitrary point $\boldsymbol{x} \neq S_0[0]$. By the sufficient condition, there exists a predecessor $\boldsymbol{x}' \in S_0$ satisfying $\boldsymbol{x}' <_f \boldsymbol{x}$, together with the corresponding evaluation rule. Specifically,

$$f(\boldsymbol{x}) = D^0_{\mathbf{K}_0} f(\boldsymbol{x}) = D^0_{\mathbf{K}_0} f(\boldsymbol{x}') + D^1_{\mathbf{K}_1} f(\boldsymbol{x}') = f(\boldsymbol{x}') + D^1_{\mathbf{K}_1} f(\boldsymbol{x}')$$

Since $\boldsymbol{x}' <_f \boldsymbol{x}$, the value $f(\boldsymbol{x}')$ has already been computed and stored in TAB^{f}_{eva}. Consequently, one application of the non-recursive evaluation formula reduces the computation of $f(\boldsymbol{x})$ to the evaluation of a first-order derivative polynomial at the predecessor input. Since the sufficient condition is assumed to hold for every triple $(g, S, \mathcal{R})$ arising in the non-recursive framework, the same reasoning applies to the derivative polynomial $D^1_{\mathbf{K}_1} f$ and, more generally, to every higher-order derivatives appearing in Equation (7). Due to remark 1 and condition $\sum_{i<\epsilon} x_{k_i} + \delta < d$, at most order d derivative can appear in Equation (7). The resulting dependency chain eventually reaches a derivative value

that is efficiently computed and available from TAB_{init}. Starting from this stored value, the intermediate derivative values are then recovered successively in decreasing derivative order until $f(\boldsymbol{x})$ is obtained.

Then the evaluation of f over S_0 requires at most $\mathcal{O}(d|S_0|)$ field operations in the evaluation phase, excluding the cost of precomputing the initial derivative values $g(S[0])$ that serve as the base values for the evaluation process.

The memory requirement during the evaluation phase is given by the total size of the evaluation tables $\{TAB^{g}_{\mathbf{eva}}\}$ associated with the derivative polynomials arising in the non-recursive framework. The evaluation of each input of S_0 generates at most d intermediate derivative values and each values is stored in the corresponding table. Therefore, the total memory required is $\mathcal{O}(d \cdot |S_0|)$.

5 Admissible Enumeration of Product Hamming Balls

In the preceding section, we extended the derivative-based evaluation framework from $\mathbb{F}_2^n$ to $\mathbb{F}_q^n$ and established a sufficient condition for its application over a prescribed evaluation set. To apply this framework to product Hamming balls, it remains to specify an evaluation order and an evaluation rule satisfying the condition of Theorem 4 for every derivative polynomial arising during the evaluation.

Evaluation Order and Evaluation Rule: Let $S_0 = B(\boldsymbol{n}, \boldsymbol{w}) \subseteq \mathbb{F}_q^n$. We arrange the elements of S_0 in ascending lexicographic order, denoted by $<_{\text{lex}}$, so that

$$S_0[j-1] <_{\text{lex}} S_0[j], \qquad j \in \{1, \ldots, |S_0| - 1\}.$$

Furthermore, we use Equation (6) as the evaluation rule, and denote it by $\mathcal{R}_{\text{lex}}$. Specifically, for any nonzero

$$\boldsymbol{x} = (x_n, \ldots, x_{j+1}, x_j, \underbrace{0, \ldots, 0}_{j-1}) \in S_0,$$

where j is the smallest index such that $x_j \neq 0$, the value of polynomial f at $\boldsymbol{x}$ is obtained from the value of $f(\boldsymbol{x}')$ where

$$\boldsymbol{x}' = (x_n, \ldots, x_{j+1}, x_j - 1, \underbrace{0, \ldots, 0}_{j-1})$$

by using Equation (6). We represent this predecessor relation by

$$(x_n, \ldots, x_{j+1}, x_j - 1, \underbrace{0, \ldots, 0}_{j-1}) \xrightarrow{\mathcal{R}_{\text{lex}}} (x_n, \ldots, x_{j+1}, x_j, \underbrace{0, \ldots, 0}_{j-1}).$$

Furthermore, for every derivative polynomial of the form $D^{\delta}_{\boldsymbol{K}_\epsilon} f$ arising during the evaluation, we use the same evaluation rule $\mathcal{R}_{\text{lex}}$ and equip its associated input set $S^{\delta}_{\boldsymbol{K}_\epsilon}$ with the lexicographic order induced by $<_{\text{lex}}$ on $\mathbb{F}_q^n$.

Thus, to verify the sufficient condition of Theorem 4, it remains to show that for each derivative polynomial every noninitial input has its predecessor in the corresponding input set. The following lemma establishes precisely these properties for downward-closed sets.

5.1 Verification of the Sufficient Condition for downward-closed sets

Lemma 2 *Let $S \subseteq \mathbb{F}_q^n$ be a nonempty downward-closed set, ordered lexicographically, and let a polynomial g be evaluated over S using the evaluation rule $\mathcal{R}_{\text{lex}}$. Then every noninitial input $\boldsymbol{x} \in S$ has its predecessor in S under $\mathcal{R}_{\text{lex}}$.*

Moreover, for any derivative $D^{\delta}_{\boldsymbol{K}_\epsilon} g$ arising during the evaluation, let $S^{\delta}_{\boldsymbol{K}_\epsilon}$ denote the associated input set. Then $S^{\delta}_{\boldsymbol{K}_\epsilon}$ is itself downward closed and inherits the lexicographic order from S.

Proof Since S is nonempty and downward closed, it contains 0^n, which is its first element under the lexicographic order. Let

$$\boldsymbol{x} = (x_n, \ldots, x_{k_1}, \underbrace{0, \ldots, 0}_{k_1 - 1}) \in S,$$

be a noninitial element, where k_1 is the least index of a nonzero coordinate. Under $\mathcal{R}_{\text{lex}}$, its predecessor is

$$\boldsymbol{x}' = (x_n, \ldots, x_{k_1} - 1, \underbrace{0, \ldots, 0}_{k_1 - 1}).$$

Since only the k_1-th coordinate has been decreased, $\boldsymbol{x}' \preceq \boldsymbol{x}$. The downward closure of S therefore implies that $\boldsymbol{x}' \in S$. Hence every noninitial element of S has the required predecessor in S.

We next consider the input set associated with the first order derivative $D_{\boldsymbol{K}_1} g$. Under $\mathcal{R}_{\text{lex}}$, $D_{\boldsymbol{K}_1} g$ is evaluated at predecessors of the form $\boldsymbol{x}'$ as given above. More precisely, the input set of $D_{\boldsymbol{K}_1} g$ is given as

$$S_{\boldsymbol{K}_1} = \left\{ \boldsymbol{y} \in \mathbb{F}_q^n : y_1 = \cdots = y_{k_1 - 1} = 0,\ y_{k_1} \leq q - 2,\ \boldsymbol{y} + \boldsymbol{e}_{k_1} \in S \right\}.$$

Here $y_{k_1} \leq q - 2$ ensures that $\boldsymbol{y} + \boldsymbol{e}_{k_1}$ corresponds to increasing the canonical value of the k_1-th coordinate by one, without wraparound modulo q.

Let $\boldsymbol{y} \in S_{\boldsymbol{K}_1}$ be a nonzero element and suppose that $\boldsymbol{z} \preceq \boldsymbol{y}$. Since $y_1 = \cdots = y_{k_1 - 1} = 0$, therefore $z_1 = \cdots = z_{k_1 - 1} = 0$. Furthermore, $z_{k_1} \leq y_{k_1} \leq q - 2$ and

$$\boldsymbol{z} + \boldsymbol{e}_{k_1} \preceq \boldsymbol{y} + \boldsymbol{e}_{k_1} \quad \text{and} \quad \boldsymbol{y} + \boldsymbol{e}_{k_1} \in S \implies \boldsymbol{z} + \boldsymbol{e}_{k_1} \in S,$$

make sure that $\boldsymbol{z}$ satisfies all conditions defining $S_{\boldsymbol{K}_1}$. Therefore, $\boldsymbol{z} \in S_{\boldsymbol{K}_1}$ and thus, $S_{\boldsymbol{K}_1}$ is downward closed.

Since every point in $S_{\boldsymbol{K}_1}$ arises as a predecessor of some point in S, and since S is downward closed, that predecessor also belongs to S; hence $S^{\delta}_{\boldsymbol{K}_\epsilon} \subseteq S$. Therefore, $S_{\boldsymbol{K}_1}$ follow lexicographic order. Moreover, the above argument shows that downward closure of an input set is preserved when passing to the input set of its derivative, the result follows inductively for all higher-order derivatives. Hence every $S^{\delta}_{\boldsymbol{K}_\epsilon}$ is downward closed and inherits the lexicographic order induced by S.

Corollary 1 *Let $S_0 = B(\boldsymbol{n}, \boldsymbol{w}) = B(n_s, w_s) \times \cdots \times B(n_1, w_1) \subseteq \mathbb{F}_q^n$ be ordered lexicographically. Since S_0 is downward closed under partial order $\preceq$, Lemma 2 implies that every input set $S^{\delta}_{\boldsymbol{K}_\epsilon}$ associated with a derivative polynomial $D^{\delta}_{\boldsymbol{K}_\epsilon} f$ arising during the non-recursive evaluation under $\mathcal{R}_{\text{lex}}$ is downward closed. Hence the sufficient condition of Theorem 4 is satisfied.*

Having established that the framework is applicable, we now turn to its efficient implementation. The algorithm used in the proof of Theorem 4 stores every value generated during the evaluation. Although this procedure is sufficient for proving the existence and correctness of the required evaluation sequence, retaining all previously computed values leads to unnecessary memory consumption. Indeed, as the evaluation advances in lexicographic order, many earlier values cease to participate in any subsequent computation. It is therefore natural to ask whether such values can be discarded and their memory locations reused without affecting the dependencies required by the evaluation rule.

The following lemma answers this question affirmatively. It identifies the values that are no longer needed after completing the evaluation at a given input and thereby provides the key observation underlying the memory-reuse strategy employed later in the algorithmic implementation.

Lemma 3 *Let f be evaluated over a lexicographically ordered set $S \subseteq \mathbb{F}_q^n$ under the evaluation rule $\mathcal{R}_{\text{lex}}$ using Equation (6). After evaluating $f(\boldsymbol{x})$ at*

$$\boldsymbol{x} = (x_n, \ldots, x_{i+1}, x_i, \underbrace{0, \ldots, 0}_{i-1}) \in S, \qquad x_i > 0,$$

the values $f(\boldsymbol{y})$ corresponding to all

$$\boldsymbol{y} = (y_n, \ldots, y_{i+1}, y_i, \underbrace{0, \ldots, 0}_{i-1}) \in S, \qquad y_i > 0,$$

satisfying $\boldsymbol{y} <_{lex} \boldsymbol{x}$ are not required in the evaluation of any subsequent inputs.

Proof Let the set of the inputs preceding $\boldsymbol{x}$ in the lexicographic order whose i-th coordinate is nonzero and whose last $i-1$ coordinates are zero is represented as

$$S_{<_{\text{lex}}\boldsymbol{x}} = \left\{ \boldsymbol{y} \in S : \boldsymbol{y} = (y_n, \ldots, y_{i+1}, y_i, \underbrace{0, \ldots, 0}_{i-1}),\ y_i > 0,\ \boldsymbol{y} <_{\text{lex}} \boldsymbol{x} \right\}.$$

These are precisely the inputs whose stored values are claimed to be unnecessary after evaluating $f(\boldsymbol{x})$. Let

$$S_{\boldsymbol{x}<_{\text{lex}}} = \{\boldsymbol{z} \in S : \boldsymbol{x} <_{\text{lex}} \boldsymbol{z}\}$$

be the set of inputs that remain to be evaluated after $\boldsymbol{x}$. Under the evaluation rule $\mathcal{R}_{\text{lex}}$, evaluating $f(\boldsymbol{z})$ for any $\boldsymbol{z} \in S_{\boldsymbol{x}<_{\text{lex}}}$ requires the value of f at the predecessor $\boldsymbol{z}'$ of $\boldsymbol{z}$. Therefore, it is sufficient to prove that

$$\forall\, \boldsymbol{z} \in S_{\boldsymbol{x}<_{\text{lex}}} \text{ such that } \boldsymbol{z}' \xrightarrow{\mathcal{R}_{\text{lex}}} \boldsymbol{z} \quad \Longrightarrow \quad \boldsymbol{z}' \notin S_{<_{\text{lex}}\boldsymbol{x}}.$$

Suppose for $j \geq 1$, $\boldsymbol{z}$ is of the form

$$\boldsymbol{z} = (z_n, \ldots, z_{j+1}, z_j, \underbrace{0, \ldots, 0}_{j-1}) \in S_{\boldsymbol{x}<_{\text{lex}}}, \qquad z_j > 0.$$

According to $\mathcal{R}_{\text{lex}}$, the predecessor of $\boldsymbol{z}$ is

$$\boldsymbol{z}' = (z'_n, \ldots, z'_{j+1}, z'_j, \underbrace{0, \ldots, 0}_{j-1}) = (z_n, \ldots, z_{j+1}, z_j - 1, \underbrace{0, \ldots, 0}_{j-1}),$$

which satisfies

$$\text{INT}(\boldsymbol{z}) = \text{INT}(\boldsymbol{z}') + q^{j-1}. \tag{8}$$

We distinguish the following two cases according to the form of $\boldsymbol{z}'$.

Case 1: Suppose that $\boldsymbol{z}'$ is *not* of the form

$$(z'_n, \ldots, z'_{i+1}, z'_i, \underbrace{0, \ldots, 0}_{i-1})$$

for any $z'_i > 0$. Then $\boldsymbol{z}'$ does not satisfy the defining form of the elements of $S_{<_{\text{lex}}\boldsymbol{x}}$. Therefore,

$$\boldsymbol{z}' \notin S_{<_{\text{lex}}\boldsymbol{x}}.$$

Case 2: Suppose that

$$\boldsymbol{z}' = (z'_n, \ldots, z'_{i+1}, z'_i, \underbrace{0, \ldots, 0}_{i-1})$$

for some $z'_i > 0$. In this case $\boldsymbol{z}$ must be of either of the following form:

$$(z'_n, \ldots, z'_{i+1}, z'_i + 1, \underbrace{0, \ldots, 0}_{i-1}), \ (z'_n, \ldots, z'_{i+1}, z'_i, \underbrace{0, \ldots, 0}_{i-j-1}, 1, \underbrace{0, \ldots, 0}_{j-1}).$$

Since, $\boldsymbol{z} \in S_{\boldsymbol{x}<_{\text{lex}}}$, therefore, the inequality $j \leq i$ must satisfy.
Suppose, for contradiction, that $\boldsymbol{z}' <_{\text{lex}} \boldsymbol{x}$. Since, both $\boldsymbol{x}$ and $\boldsymbol{z}'$ have their first nonzero coordinate at i-th location. Hence, their integer representations are multiples of q^{i-1}. Consequently,

$$\text{INT}(\boldsymbol{x}) - \text{INT}(\boldsymbol{z}') \geq q^{i-1}.$$

Therefore, from Equation (8) we have the following

$$\text{INT}(\boldsymbol{z}) = \text{INT}(\boldsymbol{z}') + q^{j-1} \leq \text{INT}(\boldsymbol{z}') + q^{i-1} \leq \text{INT}(\boldsymbol{x}),$$

which contradicts the assumption that $\boldsymbol{x} <_{\text{lex}} \boldsymbol{z}$. Consequently, $\boldsymbol{x} \leq_{\text{lex}} \boldsymbol{z}'$, and hence $\boldsymbol{z}' \notin S_{<_{\text{lex}}\boldsymbol{x}}$.

Thus, the predecessor of every input $\boldsymbol{z}$ evaluated after $\boldsymbol{x}$ lies outside $S_{<_{\text{lex}}\boldsymbol{x}}$. Therefore, none of the values from the set $S_{<_{\text{lex}}\boldsymbol{x}}$ are required in any subsequent evaluation. These values may consequently be discarded, and their memory locations may be reused.

6 Algorithms and Complexity

We now develop an explicit algorithm for evaluating f over the product Hamming ball $B(\boldsymbol{n}, \boldsymbol{w})$ using the generalized non-recursive framework. Similar to the derivative-based polynomial evaluation methods [10, 18, 26], our algorithm is organized into two phases:

- During the *initialization phase*, the derivative values required to initialize the evaluation process, in particular the values of derivatives at origin are computed and stored.
- During the *evaluation phase*, the values of the polynomial and its derivatives nonzero inputs are computed in lexicographic order by successively updating and combining previously stored values according to Equation (7) while the inputs of $B(\boldsymbol{n}, \boldsymbol{w})$ are traversed in lexicographic order.

Before describing these two phases, we first specify the memory organization required for their efficient implementation. This organization enables the reuse property of Lemma 3 to be incorporated directly into the evaluation procedure. In particular, we determine which derivative evaluations need to be retained during the computation, where these values are stored, and how much memory is sufficient to support their successive reuse.

6.1 Derivative Inputs and Memory Representation

We begin by identifying the form of the inputs at which each derivative polynomial is evaluated.

Lemma 4 *Let $D^{\delta}_{\boldsymbol{K}_\epsilon} f$, with $0 \leq \delta \leq x_{k_\epsilon}$, be a derivative polynomial arising in the non-recursive evaluation of $f(\boldsymbol{x})$. Then every evaluation of $D^{\delta}_{\boldsymbol{K}_\epsilon} f$ is performed at an input of the form*

$$\left(\mathbf{a}, \ x_{k_\epsilon} - \delta, \ \underbrace{0, \ldots, 0}_{k_\epsilon - 1}\right), \qquad \mathbf{a} \in \mathbb{F}_q^{n-k_\epsilon}.$$

Consequently, if $\delta < x_{k_\epsilon}$, the least-index nonzero coordinate is k_ϵ. If $\delta = x_{k_\epsilon}$, then the input is either 0^n or its least-index nonzero coordinate lies in $\{k_\epsilon + 1, \ldots, n\}$.

Proof By Remark 1, before differentiation with respect to x_{k_ϵ} begins, all coordinates with indices smaller than k_ϵ have already been reduced to zero. Each subsequent differentiation with respect to x_{k_ϵ} increases its derivative order by one and decreases the corresponding input coordinate by one. Therefore, after δ such differentiations, the k_ϵ-th coordinate is $x_{k_\epsilon} - \delta$, while all coordinates with indices smaller than k_ϵ remain zero. The coordinates with indices larger than k_ϵ are unrestricted and may be represented by $\mathbf{a} \in \mathbb{F}_q^{n-k_\epsilon}$. Hence, every such input has the form

$$\left(\mathbf{a}, x_{k_\epsilon} - \delta, \underbrace{0, \ldots, 0}_{k_\epsilon - 1}\right).$$

If $\delta < x_{k_\epsilon}$, then $x_{k_\epsilon} - \delta > 0$, and therefore the least-index nonzero coordinate is exactly k_ϵ. If $\delta = x_{k_\epsilon}$, then the k_ϵ-th coordinate is also zero, so the last k_ϵ coordinates are all zero. In this case, either $\mathbf{a} = \mathbf{0}$, giving the input 0^n, or the least-index nonzero coordinate lies at some position in $\{k_\epsilon + 1, \ldots, n\}$.

Lemma 4 shows that a derivative polynomial arising in the non-recursive expansion is not evaluated at arbitrary points of $\mathbb{F}_q^n$. Its evaluations occur only along inputs having a prescribed trailing-zero structure. For the memory analysis, we consider $D^\delta_{\mathbf{K}_\epsilon} f$ with $0 < \delta \leq x_{k_\epsilon}$, so that x_{k_ϵ} is the most recent variable with respect to which differentiation has been performed. By Lemma 4, over all nonzero inputs at which $D^\delta_{\mathbf{K}_\epsilon} f$ is evaluated, the least-index nonzero coordinate can occur only at a position

$$j \in \{k_\epsilon, \ldots, n\}.$$

Storing the value of the derivative at every such input would still require memory proportional to the size of the evaluation domain. This is precisely where Lemma 3 becomes useful. Specifically, for each fixed j, Lemma 3 implies that, as the inputs are evaluated in lexicographic order, a previously stored value having least-index nonzero coordinate j is no longer required once a lexicographically larger input of the same form has been evaluated. Hence, a single memory location can be reused for all evaluations corresponding to that j. Therefore, $n - k_\epsilon + 1$ memory locations are sufficient for the nonzero inputs of $D^\delta_{\mathbf{K}_\epsilon} f$, one for each possible $j \in \{k_\epsilon, \ldots, n\}$. One additional location is retained for the case when the input is 0^n, i.e., to store $D^\delta_{\mathbf{K}_\epsilon} f(0^n)$. Thus, $n - k_\epsilon + 2$ memory locations are sufficient for every derivative polynomial of total order strictly between 0 and d with x_{k_ϵ} as the most recent variable with respect to which differentiation has been performed.

The two boundary cases follow from the same principle. For the original polynomial f, the least-index nonzero coordinate may occur at any position in $1, \ldots, n$. Hence, n reusable locations are sufficient for the nonzero inputs, together with one additional location for $f(0^n)$, giving a total of $n + 1$ locations. On the other hand, every derivative polynomial of total order d is constant, so its value is independent of the input and only one memory location is required.

To organize these memory locations, we associate an array with each according to the total order of the corresponding derivative. The resulting memory representation is summarized in Table 1.

Table 1: Memory representation for derivative polynomials of different orders.

Array type	**Representation**	**Size of the array**
Type-0	$A^0_{\mathbf{K}_0}$	$n+1$
Type-i, $0 < i < d$	$A^\delta_{\mathbf{K}_\epsilon}, \sum_{r=1}^{\epsilon-1} x_{k_r} + \delta = i$	$n - k_\epsilon + 2$
Type-d	$A^\delta_{\mathbf{K}_\epsilon}, \sum_{r=1}^{\epsilon-1} x_{k_r} + \delta = d$	1

Total Memory Count: We next determine the total memory occupied by all reusable arrays used throughout the evaluation process. A Type-i array corresponds to an admissible derivative multi-index $\boldsymbol{\alpha} \in \{0, \dots, q-1\}^n$, $|\boldsymbol{\alpha}| = i$. The size of such an array depends on the last occurring variable of the corresponding derivative. We therefore count the Type-i arrays according to this variable.

For $k \geq 1$, define

$$m_i^{(k)} = \left|\left\{\boldsymbol{\alpha} \in \{0, \dots, q-1\}^n : |\boldsymbol{\alpha}| = i, \alpha_j = 0 \; \forall \; k < j \leq n\right\}\right|$$

Thus, $m_i^{(k)}$ counts the admissible derivatives of total order i involving variables only from $\{x_1, \dots, x_k\}$. Equivalently, for $i > 0$, it counts those whose last occurring variable has index at most k. We also set $m_i^{(0)} = 0$ for $i > 0$. In particular, the number of Type-i arrays is $m_i^{(n)}$, and hence the total number of derivative arrays over all orders is

$$\sum_{i=0}^{d} m_i^{(n)} = |\mathcal{I}_{n,d,q}| = M.$$

Let c_{i,k_ϵ} denote the number of derivatives of order i whose last occurring variable is exactly x_{k_ϵ}. Since $m_i^{(k_\epsilon)}$ counts all such derivatives whose last occurring variable has index at most k_ϵ, while $m_i^{(k_\epsilon - 1)}$ counts those whose last occurring variable has index at most $k_\epsilon - 1$, we obtain

$$c_{i,k_\epsilon} = m_i^{(k_\epsilon)} - m_i^{(k_\epsilon - 1)}.$$

For $1 \leq i < d$, every derivative whose last occurring variable is x_{k_ϵ} is stored in an array of size $n - k_\epsilon + 2$. Therefore, the total number of field-element entries occupied by all Type-i arrays is

$$R_i = \sum_{k_\epsilon=1}^{n} (n - k_\epsilon + 2) c_{i,k_\epsilon} = \sum_{k_\epsilon=1}^{n} (n - k_\epsilon + 2) \left(m_i^{(k_\epsilon)} - m_i^{(k_\epsilon - 1)}\right) = 2m_i^{(n)} + \sum_{k_\epsilon=1}^{n-1} m_i^{(k_\epsilon)}.$$

For Type-0, we have $m_0^{(n)} = 1$, corresponding to the single array $A^0_{\mathbf{K}_0}$ of size $n + 1$. For Type-d, there are $m_d^{(n)}$ arrays, each requiring only one memory location. Therefore, together with the Type-i arrays for $1 \leq i < d$, the total number of field-element entries occupied by all reusable arrays is

$$R_{\text{total}} = (n+1) + \sum_{i=1}^{d-1} \left(2m_i^{(n)} + \sum_{k_\epsilon=1}^{n-1} m_i^{(k_\epsilon)}\right) + m_d^{(n)}.$$

To express this quantity in terms of the parameter M, define

$$M_{k,D,q} = \left|\left\{\boldsymbol{\alpha} \in \{0, \dots, q-1\}^n : |\boldsymbol{\alpha}| \leq D, \alpha_j = 0 \; \forall \; k < j \leq n\right\}\right| = \sum_{i=0}^{D} m_i^{(k)}.$$

In particular, $M = M_{n,d,q}$. Using this notation and collecting the contributions of the different derivative orders gives

$$\begin{aligned} R_{\text{total}} &= (n+1) + 2\sum_{i=1}^{d-1} m_i^{(n)} + \sum_{k_\epsilon=1}^{n-1}\sum_{i=1}^{d-1} m_i^{(k_\epsilon)} + m_d^{(n)} \\ &= (n+1) + 2\left(M_{n,d-1,q} - 1\right) + \sum_{k_\epsilon=1}^{n-1}\left(M_{k_\epsilon,d-1,q} - 1\right) + m_d^{(n)} \\ &= (n+1) + 2M_{n,d-1,q} - 2 + \sum_{k_\epsilon=1}^{n-1} M_{k_\epsilon,d-1,q} - (n-1) + m_d^{(n)} \\ &= 2M_{n,d-1,q} + \sum_{k_\epsilon=1}^{n-1} M_{k_\epsilon,d-1,q} + m_d^{(n)} \\ &= M + \sum_{k_\epsilon=1}^{n} M_{k_\epsilon,d-1,q}. \end{aligned}$$

We now compare the remaining terms with M. Clearly, $M_{n,d-1,q} \leq M$, since every admissible multi-index of total order at most $d-1$ is also admissible at total order at most d. Moreover,

$$\sum_{k=1}^{n-1} M_{k,d-1,q} \leq M.$$[2]

Consequently, $M \leq R_{\text{total}} \leq 3M$. Hence, $R_{\text{total}} = \Theta(M)$. Since every entry stores one element of $\mathbb{F}_q$ and therefore requires $\mathcal{O}(\log q)$ bits, the total memory occupied by all reusable arrays is

$$R_{\text{total}} = \Theta(M \log q) \tag{9}$$

bits. Thus, although the individual array sizes vary according to the last variable occurring in the derivative, their combined storage remains linear in the total number M of admissible derivative multi-indices.

6.2 Initialization Phase

In the proof of Theorem 4, $TAB_{\mathbf{init}}$ denotes the collection of values $g(S[0])$, where g is a derivative polynomial appearing in Equation (7) and S is its associated input set. Since the input sets arising from $B(\boldsymbol{n}, \boldsymbol{w})$ are downward closed and ordered lexicographically, their initial element is $S[0] = 0^n$. Thus, the initialization phase requires the computation of the values $D^{\delta}_{\boldsymbol{K}_\epsilon} f(0^n)$ for the derivative polynomials arising in the non-recursive evaluation. We refer to these as the *initial derivative values*, while the values of the derivatives at nonzero inputs are referred to as *intermediate values*. Each initial derivative value is stored in the first position of its corresponding array $A^{\delta}_{\boldsymbol{K}_\epsilon}$.

We present two methods for computing these initial derivative values. Method 1 computes them directly from the coefficients of f, whereas Method 2 recursively constructs the required derivative polynomials and evaluates them at the origin.

[2] A detailed derivation of this inequality, together with the simplification of R_{total} from its order-wise expression, is given in Appendix A.

6.2.1 Method 1: Direct Coefficient-Based Computation

Let

$$\mathcal{I}_{n,d,q} = \{\boldsymbol{\beta} \in \{0,\dots,q-1\}^n : |\boldsymbol{\beta}| \le d\}.$$

denote the admissible exponent set indexing the coefficients $c_{\boldsymbol{\beta}}$ in the ANF representation (5). We use the multi-index notation $D^{\boldsymbol{\alpha}} = D_1^{\alpha_1} \cdots D_n^{\alpha_n}$ introduced earlier to index the required derivative values uniformly. In particular, $D^{\delta}_{\mathbf{K}_\epsilon} f$ corresponds to $D^{\boldsymbol{\alpha}}$ with $\alpha_{k_r} = x_{k_r}$ for $1 \le r < \epsilon$, $\alpha_{k_\epsilon} = \delta$, and $\alpha_j = 0$ for all remaining coordinates. Hence, computing $D^{\boldsymbol{\alpha}} f(0^n)$ for all admissible $\boldsymbol{\alpha}$ includes every initial derivative value required by the non-recursive framework. Furthermore, It is sufficient to consider derivative multi-indices $\boldsymbol{\alpha} \in \mathcal{I}_{n,d,q}$. Indeed, since the ANF of f satisfies $\deg_{x_i} f \le q-1$ for every i, any derivative with $\alpha_i \ge q$ for some i vanishes. Moreover, since f has total degree at most d, every derivative of total order $|\boldsymbol{\alpha}| > d$ also vanishes. Hence, $\boldsymbol{\alpha} \notin \mathcal{I}_{n,d,q}$ implies $D^{\boldsymbol{\alpha}} f(0^n) = 0$. Therefore, all potentially nonzero derivative values at the origin are indexed by $\mathcal{I}_{n,d,q}$. The cardinality of $\mathcal{I}_{n,d,q}$ therefore determines both the number of admissible coefficient positions in the ANF representation and the number of admissible derivative values to be computed. Denoting this cardinality by M, inclusion-exclusion gives

$$M = |\mathcal{I}_{n,d,q}| = \sum_{j=0}^{\min\{n,\lfloor d/q \rfloor\}} (-1)^j \binom{n}{j} \binom{n+d-jq}{n}. \tag{10}$$

For $d < q$, the coordinate-wise restriction i.e., $\deg_{x_i} f \le q-1$ is inactive, giving $M = \binom{n+d}{d}$ whereas for $d = n(q-1)$, every vector in $\{0,\dots,q-1\}^n$ is admissible, giving $M = q^n$. Consequently,

$$M \le \min\left\{ \binom{n+d}{d}, q^n \right\}. \tag{11}$$

Thus, Method 1 computes M derivative values at the origin, including the zeroth-order value $D^{\mathbf{0}} f(\mathbf{0}) = f(\mathbf{0})$, and storing these outputs requires $\Theta(M)$ field elements.

Let $S(b,a)$ denote the Stirling number of the second kind. Theorem 5 of Sălăgean et al. [27] yields the following expression for the higher-order discrete derivative of a multivariate monomial evaluated at the origin. For $\boldsymbol{x}^{\boldsymbol{\beta}} = x_1^{\beta_1} \cdots x_n^{\beta_n}$,

$$D^{\boldsymbol{\alpha}} \boldsymbol{x}^{\boldsymbol{\beta}}(\mathbf{0}) = \prod_{i=1}^{n} \alpha_i! S(\beta_i, \alpha_i).$$

By linearity of the operator $D^{\boldsymbol{\alpha}}$, the derivative of polynomial f at $\mathbf{0}$ is

$$D^{\boldsymbol{\alpha}} f(\mathbf{0}) = \sum_{\boldsymbol{\beta} \in \mathcal{I}_{n,d,q}} c_{\boldsymbol{\beta}} \prod_{i=1}^{n} \alpha_i! S(\beta_i, \alpha_i). \tag{12}$$

Equation (12) is the basic coefficient-only formula used throughout the remainder of this section.

Remark 2 There are some properties of the Stirling numbers of the second kind which we used in this section. For $a \ge 0$ and $b \ge 0$

$$S(b,b) = 1,\ S(b,a) = 0\ (a > b).$$

Their standard recurrence is

$$S(b,a) = S(b-1,a-1) + aS(b-1,a). \tag{13}$$

For convenience, define

$$W(a,b) = a!S(b,a).$$

Then equation (12) becomes

$$D^{\alpha} f(\mathbf{0}) = \sum_{\boldsymbol{\beta} \in \mathcal{I}_{n,d,q}} c_{\boldsymbol{\beta}} \prod_{i=1}^{n} W(\alpha_i, \beta_i). \tag{14}$$

Similar to Stirling number, the basic properties of W with $a \geq 0$ and $b \geq 0$ are

$$W(a,a) = a!,\ W(a,b) = 0\ (a > b).$$

Furthermore, the recursive relation of $W(a,b)$ can be derived using Equation (13) as follows

$$\begin{aligned} W(a,b) &= a!\left[S(b-1,a-1) + aS(b-1,a)\right] \\ &= aW(a-1,b-1) + aW(a,b-1). \end{aligned}$$

Therefore

$$W(a,b) = a\left(W(a-1,b-1) + W(a,b-1)\right). \tag{15}$$

Let $r = \min\{d, q-1\}$, an immediate consequence of $W(\alpha_i, \beta_i) = 0$ for $\alpha_i > \beta_i$ and $0 \leq \alpha_i, \beta_i \leq r$ is that only values of $W(\alpha_i, \beta_i)$ for $0 \leq \alpha_i \leq \beta_i \leq r$ are required to evaluate all required derivatives given in Equation (14). Using recurrence (15), all such values can be obtained from previously computed values using a constant number of field operations. Hence the time complexity of calculating all require $W(\alpha_i, \beta_i)$ is $T_W = \Theta(r^2)$ field operations and, if the all values are retained, the memory complexity is $S_W = \Theta(r^2)$ field elements.

With the required values of $W(\alpha_i, \beta_i)$ in hand, Equation (14) suggests an immediate algorithm: for every admissible derivative index $\boldsymbol{\alpha}$, enumerate all coefficient indices $\boldsymbol{\beta}$ that can contribute, i.e., $\prod_i W(\alpha_i, \beta_i) \neq 0$ and form

$$D^{\alpha} f(\mathbf{0}) = \sum_{\boldsymbol{\beta}} c_{\boldsymbol{\beta}} \prod_{i} W(\alpha_i, \beta_i).$$

However, this computes every derivative independently, which consequence in recompute of some partial sums. For example, two outputs such as

$$D^{(\alpha_1, \alpha_2, \ldots)} f(\mathbf{0}) = \sum_{\beta_2, \ldots, \beta_n} \prod_{i \geq 2} W(\alpha_i, \beta_i) \sum_{\beta_1} c_{\boldsymbol{\beta}} W(\alpha_1, \beta_1)$$

and

$$D^{(\alpha_1, \alpha_2', \ldots)} f(\mathbf{0}) \sum_{\beta_2, \ldots, \beta_n} \prod_{i \geq 2} W(\alpha_i', \beta_i) \sum_{\beta_1} c_{\boldsymbol{\beta}} W(\alpha_1, \beta_1)$$

share the same transformation with respect to the first variable, yet the direct method recomputes partial sum $\sum_{\beta_1} c_{\boldsymbol{\beta}} W(a_1, \beta_1)$ independently.

An efficient algorithm where instead of calculating every derivative independently, the coefficient array can be transformed one coordinate at a time, while reusing intermediate partial sums, can be constructed.

Using the coefficient formula of Sălăgean et al. [27], the contribution here is organizing the simultaneous computation of the admissible values $D^{\alpha} f(0)$ as an in-place sequence of one-dimensional coordinate transformations over the admissible index set $\mathcal{I}_{n,d,q}$. In particular, we find the derivative with respect to one variable at a time and store them for further evaluation. After n stages we find the derivative with respect to all n variable of suitable order. The algorithm is as follow. Create an array $\Lambda^{(0)}$ indexed by the admissible exponent vectors $\boldsymbol{\beta} \in \mathcal{I}_{n,d,q}$ and initialize

$$\Lambda^{(0)}_{\beta_1, \ldots, \beta_n} = c_{\beta_1, \ldots, \beta_n}.$$

Initially every coordinate of the index of array $\Lambda^{(0)}$ represents a monomial exponent β_i. The algorithm performs n stages. At stage j, coordinate j is converted from a coefficient exponent β_j to a derivative exponent α_j. Indeed the range of α_j is same as the range of β_j. Thus the interpretation of the array evolves as

$$(\beta_1, \beta_2, \ldots, \beta_n) \longrightarrow (\alpha_1, \beta_2, \ldots, \beta_n) \longrightarrow (\alpha_1, \alpha_2, \beta_3, \ldots, \beta_n) \longrightarrow \cdots \longrightarrow (\alpha_1, \ldots, \alpha_n).$$

It is important to emphasis that the number of stored array entries does not increase during this process. Only the interpretation of an index coordinate changes. In particular, at stage j, fix every coordinate except coordinate j. The entries obtained by varying coordinate j are referred to as *a one-dimensional fiber* in direction j.

After coordinates $1, \ldots, j-1$ have already been transformed, such a fiber has the form

$$\Lambda^{(j-1)}_{\alpha_1,\ldots,\alpha_{j-1},b,\beta_{j+1},\ldots,\beta_n},$$

where only b varies. If $s = \sum_{i<j} \alpha_i + \sum_{i>j} \beta_i$, then to avoid total degree exceed to d, the value of b varies from 0 to L, where $L = \min\{q-1, d-s\}$. For brevity, we write

$$\Gamma_b = \Lambda^{(j-1)}_{\alpha_1,\ldots,\alpha_{j-1},b,\beta_{j+1},\ldots,\beta_n}.$$

Thus the fiber consists of $\Gamma_0, \Gamma_1, \ldots, \Gamma_L$. The one-dimensional transformation along j-th coordinate replaces these values by $y_0, y_1, \ldots, y_L$, respectively, where

$$y_a = \sum_{b=0}^{L} W(a,b)\Gamma_b = \sum_{b=a}^{L} W(a,b)\Gamma_b, \qquad 0 \le a \le L. \tag{16}$$

The lower limit $b = a$ follows from the triangular property $W(a,b) = 0$ $(b < a)$. After performing (16), the same fiber locations are interpreted as

$$\Lambda^{(j)}_{\alpha_1,\ldots,\alpha_{j-1},a,\beta_{j+1},\ldots,\beta_n},$$

with corresponding values as y_a. Thus coordinate j has been converted from a β_j-coordinate into an α_j-coordinate. Note that from Equation (16) the value of y_a depends only on $\Gamma_a, \Gamma_{a+1}, \ldots, \Gamma_L$. Consequently, while processing the outputs in increasing order i.e., $a = 0, 1, \ldots, L$, after y_a has been computed, the old value Γ_a is no longer required for any later output. Indeed, if $a' > a$, then

$$y_{a'} = \sum_{b=a'}^{L} W(a',b)\Gamma_b$$

uses only indices $b \ge a' > a$. Thus Γ_a may safely be overwritten immediately by y_a. In simple words, the progression within a fiber is therefore

$$(\Gamma_0, \Gamma_1, \ldots, \Gamma_L) \longrightarrow (y_0, \Gamma_1, \ldots, \Gamma_L) \longrightarrow (y_0, y_1, \Gamma_2, \ldots, \Gamma_L) \longrightarrow \cdots \longrightarrow (y_0, \ldots, y_L).$$

Remark 3 For each fixed value of α_i for $1 \le i \le j-1$ and β_i for $j+1 \le i \le n$, there is one fiber of length $\ell = L+1$ consists of $\Gamma_0, \Gamma_1, \ldots, \Gamma_L$ where $L = \min\{q-1, d-s\}$, as described above. Once the value of α_i and β_i is changed, the value of s is changed and we get a new fiber of different length. If there are total k fibers corresponding to coordinate j of length $\ell_1, \ldots, \ell_k$ then $\sum_i \ell_i = M$.

After all fibers in coordinate j have been transformed, the same global array represents $\Lambda^{(j)}$, in particular the array $\Lambda^{(j-1)}$ is represented as $\Lambda^{(j)}$ after the j-th coordination transformation. Thus the coefficient array is successively overwritten:

$$\Lambda^{(0)} \to \Lambda^{(1)} \to \cdots \to \Lambda^{(n)}.$$

Proof of Correctness We prove that if the algorithm runs with the global array as $\Lambda^{(0)}$ initialized with the coefficient $c_{\beta_1,\ldots,\beta_n}$ of the polynomial then after j stage the element of the global array are given as

$$\Lambda^{(j)}_{\alpha_1,\ldots,\alpha_j,\beta_{j+1},\ldots,\beta_n} = \sum_{\beta_1,\ldots,\beta_j} c_{\boldsymbol{\beta}} \prod_{i=1}^{j} W(\alpha_i,\beta_i), \tag{17}$$

where the sum ranges over the admissible values of the transformed coordinates. For $j = 0$, the claim follows directly from the initialization of array, i.e., $\Lambda^{(0)} = c_{\beta_1,\ldots,\beta_n}$. Assume now that Equation (17) holds after stage $j-1$. Applying transformation (16) along j-th coordinate gives

$$\begin{aligned}
\Lambda^{(j)}_{\alpha_1,\ldots,\alpha_j,\beta_{j+1},\ldots,\beta_n} &= \sum_{\beta_j} W(\alpha_j,\beta_j)\Lambda^{(j-1)}_{\alpha_1,\ldots,\alpha_{j-1},\beta_j,\beta_{j+1},\ldots,\beta_n} \\
&= \sum_{\beta_j} W(\alpha_j,\beta_j) \sum_{\beta_1,\ldots,\beta_{j-1}} c_{\boldsymbol{\beta}} \prod_{i=1}^{j-1} W(\alpha_i,\beta_i) \\
&= \sum_{\beta_1,\ldots,\beta_j} c_{\boldsymbol{\beta}} \prod_{i=1}^{j} W(\alpha_i,\beta_i).
\end{aligned}$$

After the final stage $j = n$, the global array is

$$\Lambda^{(n)}_{\alpha_1,\ldots,\alpha_n} = \sum_{\boldsymbol{\beta}} c_{\boldsymbol{\beta}} \prod_{i=1}^{n} W(\alpha_i,\beta_i),$$

By Equation (14), this is precisely $D^{\boldsymbol{\alpha}} f(\mathbf{0})$. Hence, after the n-th stage, the global array contains all admissible derivative values at the origin and, in particular, all initial derivative values required by the non-recursive framework.

To illustrate the coordinate-wise transformation, consider $n = 2, q = 3$, and $d = 3$, with $f(x_1,x_2) = c_{00} + c_{10}x_1 + c_{20}x_1^2 + c_{01}x_2 + c_{11}x_1x_2 + c_{21}x_1^2x_2 + c_{02}x_2^2 + c_{12}x_1x_2^2$. Here, $\mathcal{I}_{2,3,3}$ contains $M = 8$ admissible indices, and

$$W = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

over $\mathbb{F}_3$. Figure 2 illustrates the evolution of the same M array locations during the two coordinate transformations. Since the example is two-dimensional, the one-dimensional fibers in directions 1 and 2 appear as rows and columns, respectively; these are referred to as *row fibers* and *column fibers* only for this illustration. Initially, $\Lambda^{(0)}_{\beta_1,\beta_2} = c_{\beta_1,\beta_2}$. After the first stage, the entries are interpreted as $\Lambda^{(1)}_{\alpha_1,\beta_2}$, obtained by transforming the row fibers with respect to the first coordinate. The second stage transforms the column fibers with respect to the second coordinate, yielding $\Lambda^{(2)}_{\alpha_1,\alpha_2} = D_1^{\alpha_1} D_2^{\alpha_2} f(\mathbf{0})$. Thus, for example, the entry indexed by $(1,1)$ in the final array stores $D_1D_2f(\mathbf{0}) = c_{11} + c_{21} + c_{12}$.

Complexity Analysis of Method 1

Time complexity: Consider the j-th stage of Algorithm where the j-th coordinate is transformed. Consider one fiber of length $\ell = L+1$ consists of $\Gamma_0, \Gamma_1, \ldots, \Gamma_L$, where $L \leq \min\{d, q-1\}$. In algorithm these values are replaced with $y_0, \ldots, y_L$ respectively using Equation (16). To replace Γ_a with y_a, $L-a+1$ multiplications and $L-a$ addition field operations are required. Therefore total $2(L-a)+1$ operations are required. Hence to replace all Γ_i we need $(L+1)^2 = \ell^2$ operations, in

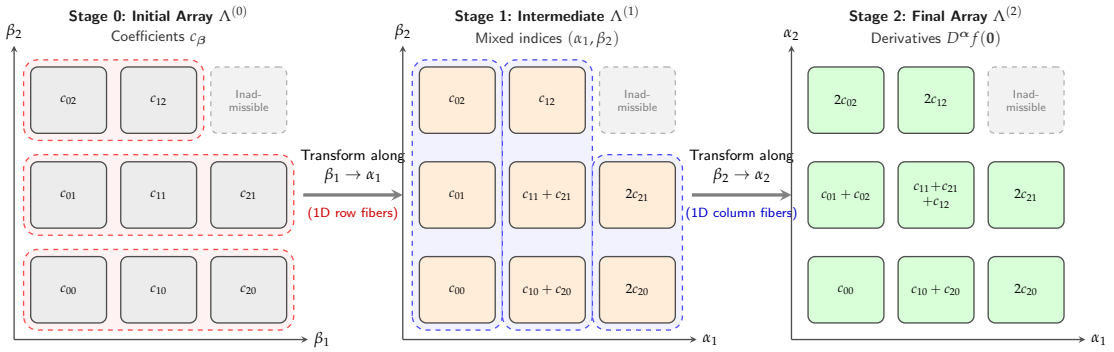

Fig. 2: Coordinate-wise transformation of the array Λ from polynomial coefficients to discrete derivatives for $n = 2$, $q = 3$, and $d = 3$.

Algorithm 1: Coefficient-Based Computation of Terminal Derivatives

Input: Coefficients $c_{\boldsymbol{\beta}}$ for $\boldsymbol{\beta} \in \mathcal{I}_{n,d,q}$
Output: $\Lambda[\boldsymbol{\alpha}] = D^{\boldsymbol{\alpha}} f(\mathbf{0})$ for all $\boldsymbol{\alpha} \in \mathcal{I}_{n,d,q}$

$r \leftarrow \min\{d, q-1\}$;
// Precompute the one-dimensional transformation weights
Compute $W(a,b) = a!S(b,a)$ for $0 \le a \le b \le r$ using Equation (15);
// Initialize the global array with the ANF coefficients
foreach $\boldsymbol{\beta} \in \mathcal{I}_{n,d,q}$ **do**
    $\Lambda[\boldsymbol{\beta}] \leftarrow c_{\boldsymbol{\beta}}$;
// Transform one coordinate at a time
for $j \leftarrow 1$ **to** n **do**
    // Fix all coordinates except the j-th one
    foreach *admissible* $(\alpha_1, \ldots, \alpha_{j-1}, \beta_{j+1}, \ldots, \beta_n)$ **do**
        $s \leftarrow \sum_{i<j} \alpha_i + \sum_{i>j} \beta_i$;
        $L \leftarrow \min\{q-1, d-s\}$;
        // Transform the corresponding one-dimensional fiber in place
        for $a \leftarrow 0$ **to** L **do**
            $z \leftarrow 0$;
            for $b \leftarrow a$ **to** L **do**
                $z \leftarrow z + W(a,b)\,\Lambda[\alpha_1, \ldots, \alpha_{j-1}, b, \beta_{j+1}, \ldots, \beta_n]$;
            $\Lambda[\alpha_1, \ldots, \alpha_{j-1}, a, \beta_{j+1}, \ldots, \beta_n] \leftarrow z$;
return Λ;

particular $\mathcal{O}(\ell^2)$ field operations. Consequently, for k fibers of length $\ell_1, \ldots, \ell_k$, $\sum_i \mathcal{O}(\ell_i^2)$ field operations are required. Recall that $r = \min\{d, q-1\}$, we have $\ell_i \le r+1$. Hence cost of one coordinate transformation is given as

$$\sum_{i=1}^{k} \mathcal{O}(\ell_i^2) \le \mathcal{O}\left((r+1)\sum_{i=1}^{k} \ell_i\right) = \mathcal{O}((r+1)M).$$

The initialization phase requires to compute all required values of $W(\alpha_i, \beta_i)$ and the transformation of all n coordinates. Thus the time complexity of the initialization phase is

$$T_{\text{Method 1}} = O\left(r^2 + n(r+1)M\right), \qquad r = \min\{d, q-1\}.$$

Using the simple bound (11),

$$T_{\text{Method 1}} = O\left(r^2 + n(r+1)\min\left\{\binom{n+d}{d}, q^n\right\}\right). \tag{18a}$$

Memory complexity: The overall memory required in the method is storing all required values of $W(\alpha_i, \beta_i)$ which need $\mathcal{O}(r^2)$ field elements and global array $\Lambda^{(0)}$ of size M used to store the coefficient of the polynomial and is reused throughout the coordinates transformations. Therefore, the memory complexity of the initialization phase is

$$M_{\text{Method 1}} = \mathcal{O}(r^2 + M)$$

field elements. In bit complexity and with bound (11), we have

$$M_{\text{Method 1}} = \mathcal{O}\left(\left[r^2 + \min\left\{\binom{n+d}{d}, q^n\right\}\right]\log q\right) \tag{18b}$$

6.2.2 Method 2: Recursive Construction of Derivative Polynomials

As an alternative to the coefficient-based computation of Method 1, the required derivative values at the origin can be obtained by recursively constructing the corresponding derivative polynomials and extracting their constant terms. A similar construction was used by Furue and Takagi [18]. However, in our framework, the recursion is restricted to derivative indices belonging to $\mathcal{I}_{n,d,q}$.

To explain this method, we consider

$$\mathbf{a} = (0, \underbrace{k_1, \ldots, k_1}_{x_{k_1}}, \underbrace{k_2, \ldots, k_2}_{x_{k_2}} \ldots, \underbrace{k_i, \ldots, k_i}_{\delta})$$

to represent $(\text{len}(\mathbf{a}) - 1)$-th order derivative $D^{\delta}_{\mathbf{K}_i} f$, while $\mathbf{a} = (0)$ represents f. For example, $(0, 1, 1, 3)$ represents third order derivative $D^{2,1}_{1,3} f$.

The recursive construction can be viewed as a rooted tree whose root is the original polynomial f. The first level consists of the first-order derivatives $D_k f$, for $1 \le k \le n$. More generally, a node represented by $\mathbf{a}$ generates a child $(\mathbf{a}, k)$ by differentiating $D^{\delta}_{\mathbf{K}_i} f$ with respect to an index $k \ge k_i$. The restriction $k \ge k_i$ preserves the nondecreasing order of the sequence. Since the difference operators commute, this canonical ordering ensures that every mixed derivative is generated only once. Note that the algorithm do not generates every such child. If $\boldsymbol{\alpha} \in \mathbb{N}^n$ is the multi-index corresponding to the current node $\mathbf{a}$, i.e., $\alpha_{k_j} = x_{k_j}$ for $1 \le j < i$, $\alpha_{k_i} = \delta$, and $\alpha_\ell = 0 \quad$ for every $\ell \notin \{k_1, \ldots, k_i\}$, a derivative with respect to x_k is generated only when $|\boldsymbol{\alpha}| < d$ and $\alpha_k < q - 1$. The first condition restricts the total derivative order to d, whereas the second prevents differentiation with respect to any individual variable beyond order $q - 1$. Moreover, reaching derivative order $q - 1$ with respect to one variable does not necessarily terminate the entire branch; it only prevents that variable from being selected again. Differentiation may continue with a larger admissible index. A node becomes a leaf when its total order is d, or when no index k satisfying the above conditions remains available. Consequently, the resulting tree contains exactly one node for every $\boldsymbol{\alpha} \in \mathcal{I}_{n,d,q}$.

Algorithm 2 traverses this tree in depth-first order. Starting from f, it repeatedly follows an admissible child until a leaf is reached. After processing that leaf, the recursion returns to its parent and continues with the next admissible child. Thus, at any stage of the traversal, only the derivative

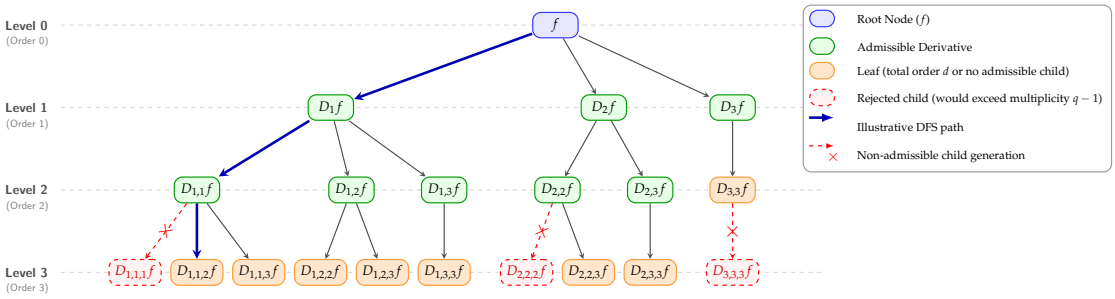

Fig. 3: Recursive construction of derivative polynomials traversing the admissible multi-indices $\mathcal{I}_{n,d,q}$ for $n = 3, q = 3, d = 3$.

polynomials lying on the current path from the root to the active node need to be retained. An example of this derivative tree is shown in Figure 3 for $n = 3$, $q = 3$, and $d = 3$. The admissible derivative polynomials form a rooted tree, while non-admissible children are excluded from the recursion. The highlighted edges indicate a typical depth-first traversal path, illustrating that only the derivative polynomials along the current branch need to be stored at any stage.

At each visited node, the constant term of the corresponding derivative polynomial gives its value at 0^n and is stored as the associated initialization value. Upon completion of the traversal, the values $D^{\boldsymbol{\alpha}} f(0^n)$ have been obtained for all $\boldsymbol{\alpha} \in \mathcal{I}_{n,d,q}$.

Algorithm 2: Recursive Construction of Derivative Polynomials

Input: Polynomial $f \in \mathbb{F}_q[x_1, \ldots, x_n]$ of total degree at most d
Output: Initialization values $D^{\boldsymbol{\alpha}} f(\mathbf{0})$ for all $\boldsymbol{\alpha} \in \mathcal{I}_{n,d,q}$

$\mathsf{TAB}_{\text{init}} \leftarrow \emptyset$;
// Start from the root corresponding to the original polynomial
$\mathbf{a} \leftarrow (0)$;
Explore$(f, \mathbf{a}, 1)$;
return $\mathsf{TAB}_{\text{init}}$;
Procedure Explore$(g, \mathbf{a}, s)$**:**
  $i \leftarrow \text{len}(\mathbf{a}) - 1$;
  // The constant term gives the derivative value at the origin
  $\mathsf{TAB}_{\text{init}}[\mathbf{a}] \leftarrow g(\mathbf{0})$;
  // No derivative of total order greater than d is required
  if $i = d$ **then**
    return;
  // Generate admissible children in nondecreasing index order
  for $k \leftarrow s$ **to** n **do**
    $\delta_k \leftarrow$ number of occurrences of k in $\mathbf{a}$;
    // Do not exceed derivative order $q - 1$ in any coordinate
    if $\delta_k < q - 1$ **then**
      $h \leftarrow D_k g = g(\mathbf{x} + \mathbf{e}_k) - g(\mathbf{x})$;
      $\mathbf{a}' \leftarrow (\mathbf{a}, k)$;
      Explore$(h, \mathbf{a}', k)$;

Complexity Analysis of Method 2

Time complexity: By construction, the recursion generates one derivative polynomial for each admissible multi-index $\boldsymbol{\alpha} \in \mathcal{I}_{n,d,q}$. Hence, including the original polynomial corresponding to $\boldsymbol{\alpha} = \mathbf{0}$, the total number of derivative polynomials generated is $M = |\mathcal{I}_{n,d,q}|$, where M is given in Equation (10). Moreover, each derivative polynomial contains at most M coefficient positions. Let $r = \min\{d, q-1\}$. To construct $D_k(g)$, each monomial of g is processed using the expansion of $g(\boldsymbol{x} + \boldsymbol{e}_k) - g(\boldsymbol{x})$. Since the exponent of x_k is at most r, differentiating one monomial produces at most r terms. Assuming the required binomial coefficients have been precomputed, one derivative polynomial can therefore be constructed using $\mathcal{O}(rM)$ field operations.

Remark 4 The computation of $D_k(g)$ repeatedly uses the binomial coefficients $\binom{a}{b}, 0 \le b \le a \le r$. These values can be precomputed modulo q using Pascal's recurrence

$$\binom{a}{b} = \binom{a-1}{b} + \binom{a-1}{b-1}.$$

The complete table can therefore be computed using $\mathcal{O}(r^2)$ arithmetic operations and stored using $\mathcal{O}(r^2 \log q)$ bits.

The recursion generates M derivative polynomials, of which $M-1$ are obtained by differentiation from a parent polynomial. Hence, together with the $\mathcal{O}(r^2)$ cost of precomputing the required binomial coefficients, the time complexity of Method 2 is

$$T_{\text{Method 2}} = \mathcal{O}\left(rM^2 + r^2\right). \tag{19a}$$

Memory complexity: Algorithm 2 performs a depth-first traversal of the derivative tree. Therefore, it is not necessary to store all derivative polynomials generated during the complete execution. At any stage, only the polynomials lying on the current recursion path must be retained, because each parent polynomial is required again when the recursion returns to generate its remaining derivatives. At depth i, the current derivative polynomial has total degree at most $d-i$ and therefore contains at most $|\mathcal{I}_{n,d-i,q}|$ coefficient positions. Thus, the memory required for the active recursion path is bounded by $\sum_{i=0}^{d} |\mathcal{I}_{n,d-i,q}|$. Since $|\mathcal{I}_{n,t,q}| \le M$ for every $t \le d$, this is at most $(d+1)M$ field elements.

Including the $\mathcal{O}(r^2)$ field elements required for the precomputed binomial coefficients, the auxiliary memory complexity (in bits) of Method 2 is

$$M_{\text{Method 2}} = \mathcal{O}\left(((d+1)M + r^2)\log q\right) \tag{19b}$$

We now compare the two initialization methods using the bounds of complexities derived above, with $r = \min\{d, q-1\}$ and $M = |\mathcal{I}_{n,d,q}|$ given in Equation (10).
From Equations (18b) and (19b), the memory complexities are

$$\mathcal{O}\big((M + r^2)\log q\big) \qquad \text{and} \qquad \mathcal{O}\big(((d+1)M + r^2)\log q\big)$$

for Methods 1 and 2, respectively. Thus, Method 1 is linear in M, whereas Method 2 has an additional factor of d in its dominant M-dependent term. For fixed q, these bounds reduce to

$$\mathcal{O}(M \log q) \qquad \text{and} \qquad \mathcal{O}(dM \log q).$$

Hence, for fixed d they have the same asymptotic dependence on n, whereas if d is allowed to grow, Method 2 requires asymptotically more memory.

Similarly, from Equations (18a) and (19a), the time complexities are

$$\mathcal{O}(nrM + r^2) \qquad \text{and} \qquad \mathcal{O}(rM^2 + r^2),$$

respectively. Method 1 is linear in M for each coordinate transformation, whereas Method 2 is quadratic in M because it explicitly constructs the derivative polynomials. For fixed q, these become $\mathcal{O}(nM)$ and $\mathcal{O}(M^2)$, respectively. Hence, Method 1 gives a stronger asymptotic bound whenever M grows faster than n, while the two bounds may have the same order when $M = \Theta(n)$.

Method 2 provides a direct recursive realization of the derivative tree and may therefore be conceptually simpler to implement. Method 1, however, avoids explicitly constructing the intermediate derivative polynomials and consequently yields smaller derived bounds for both time and memory in the general parameterized comparison.

Once the initialization values have been obtained, the subsequent evaluation phase is independent of the chosen initialization method.

6.3 Evaluation Phase

In the evaluation phase, the polynomial values are obtained using the non-recursive formulation (6). To store the intermediate derivative values of order i obtained during the evaluation process, Type-i reusable arrays $A^{\delta}_{\mathbf{K}_\epsilon}$ were created. However, it remains to describe how such derivative values are located within its associated array and how these entries are updated during the evaluation.

Array-Indexing Convention: Consider a derivative value evaluated at a nonzero input. Let k denote the index of the most recently differentiated variable present in the derivative polynomial, and let r denote the least-index nonzero coordinate of the input. Its value is stored at position

$$r - k + 1$$

of the associated reusable array. The zero-input value is always stored in the first entry of the corresponding array.

As established earlier, the boundary case $\delta = x_{k_\epsilon}$ is represented canonically at the next stage by $\delta' = 0$ whenever a subsequent nonzero coordinate exists, i.e, $D^{x_{k_\epsilon}}_{\mathbf{K}_\epsilon} f = D^{0}_{\mathbf{K}_{\epsilon+1}}$; otherwise, the corresponding input is 0^n. Hence, it is sufficient to consider

$$\delta = 0 \qquad \text{and} \qquad 0 < \delta < x_{k_\epsilon},$$

together with the separate convention for the zero input.

Under the above convention, the array positions for the different derivative types are determined as follows.

1. For the Type-0 array corresponding to the original polynomial f, set $k = k_0 = 0$. If $\boldsymbol{x} \neq 0^n$ has least-index nonzero coordinate r, then $f(\boldsymbol{x})$ is stored in $A^0_{\mathbf{K}_0}[r+1]$, while $A^0_{\mathbf{K}_0}[1]$ stores $f(0^n)$. Thus, the Type-0 array has $n+1$ entries indexed from 1 to $n+1$.
2. For a Type-i array associated with $D^{\delta}_{\mathbf{K}_\epsilon} f$, where $0 < \delta < x_{k_\epsilon}$, Lemma 4 gives $k = r = k_\epsilon$. Hence, the corresponding derivative value is stored in $A^{\delta}_{\mathbf{K}_\epsilon}[1]$. The value of $D^{\delta}_{\mathbf{K}_\epsilon} f(0^n)$ is stored at $A^{\delta}_{\mathbf{K}_\epsilon}[0]$.
3. For a Type-i array represented with $\delta = 0$, the most recently differentiated variable is $x_{k_{\epsilon-1}}$, i.e., $k = k_{\epsilon-1}$ and Lemma 4 gives $r = k_\epsilon$. Hence, $D^{0}_{\mathbf{K}_\epsilon} f$ is stored in $A^0_{\mathbf{K}_\epsilon}[\,k_\epsilon - k_{\epsilon-1} + 1\,]$. The value of $D^{0}_{\mathbf{K}_\epsilon} f(0^n)$ is stored at $A^0_{\mathbf{K}_\epsilon} f[0]$.

Memory Update Rule: Having established the correspondence between derivative inputs and array positions, we now describe how these entries are updated during the non-recursive evaluation. A typical evaluation step is

$$D^{\delta}_{\mathbf{K}_\epsilon} f\big(\mathbf{a}, \zeta, \underbrace{0, \ldots, 0}_{k_\epsilon - 1}\big) = D^{\delta}_{\mathbf{K}_\epsilon} f\big(\mathbf{a}, \zeta - 1, \underbrace{0, \ldots, 0}_{k_\epsilon - 1}\big) + D^{\delta+1}_{\mathbf{K}_\epsilon} f\big(\mathbf{a}, \zeta - 1, \underbrace{0, \ldots, 0}_{k_\epsilon - 1}\big), \tag{20}$$

where $\zeta = x_{k_\epsilon} - \delta > 0$.

We first consider the case in which the predecessor input is nonzero and $D^{\delta+1}_{\mathbf{K}_\epsilon} f$ has total order strictly smaller than d. The four update rules are determined by whether the current derivative contains a positive-order differentiation with respect to x_{k_ϵ}, i.e., $\delta = 0$ or $\delta > 0$, and whether the k_ϵ-th coordinate of the predecessor remains nonzero, i.e., $\zeta - 1 > 0$ or $\zeta - 1 = 0$.

- **Case 1:** $\delta > 0$**.** When the most recently differentiated variable is x_{k_ϵ} in all terms of Equation (20).
  - **Rule 1:** $\zeta > 1$**.** The k_ϵ-th coordinate remains nonzero in all respective inputs. Thus, all derivative values are stored at position 1 of their associated arrays, and the update rule is
  $$A^{\delta}_{\mathbf{K}_\epsilon}[1] \leftarrow A^{\delta}_{\mathbf{K}_\epsilon}[1] + A^{\delta+1}_{\mathbf{K}_\epsilon}[1].$$
  - **Rule 2:** $\zeta = 1$**.** Since the predecessor is assumed to be nonzero, let $k_{\epsilon+1}$ denote the index of its least-index nonzero coordinate. In this case the array position $A^{\delta}_{\mathbf{K}_\epsilon}[1]$ overwrites as follows
  $$A^{\delta}_{\mathbf{K}_\epsilon}[1] \leftarrow A^{\delta}_{\mathbf{K}_\epsilon}[k_{\epsilon+1} - k_\epsilon + 1] + A^{\delta+1}_{\mathbf{K}_\epsilon}[k_{\epsilon+1} - k_\epsilon + 1].$$
- **Case 2:** $\delta = 0$**.** When all derivatives except the last in (20) have most recently differentiated variable as $x_{k_{\epsilon-1}}$, whereas the last one has x_{k_ϵ} as most recently differentiated variable.
  - **Rule 3:** $\zeta > 1$**.** The least-index nonzero coordinate remains x_{k_ϵ} in all derivative inputs. Therefore,
  $$A^{\delta}_{\mathbf{K}_\epsilon}[k_\epsilon - k_{\epsilon-1} + 1] \leftarrow A^{\delta}_{\mathbf{K}_\epsilon}[k_\epsilon - k_{\epsilon-1} + 1] + A^{\delta+1}_{\mathbf{K}_\epsilon}[1].$$
  - **Rule 4:** $\zeta = 1$**.** Again, let $k_{\epsilon+1}$ denote the index of the least-index nonzero coordinate of the nonzero predecessor. Consequently, the update rule changes as
  $$A^{\delta}_{\mathbf{K}_\epsilon}[k_\epsilon - k_{\epsilon-1} + 1] \leftarrow A^{\delta}_{\mathbf{K}_\epsilon}[k_{\epsilon+1} - k_{\epsilon-1} + 1] + A^{\delta+1}_{\mathbf{K}_\epsilon}[k_{\epsilon+1} - k_\epsilon + 1].$$

The above four rules assume that the predecessor in Equation (20) is nonzero and that the newly appearing derivative has order smaller than d. We now describe the two boundary cases.

Boundary 1: If predecessor is origin. The first boundary case occurs when the predecessor is 0^n which happens when $\zeta = 1$ and $\mathbf{a} = 0^{n-k_\epsilon}$. The required derivative values are then the initial derivative values already computed in the initialization phase and are therefore read directly from the corresponding first entries of the arrays.

Accordingly, for $\delta > 0$, Rule 2 becomes

$$A^{\delta}_{\mathbf{K}_\epsilon}[1] \leftarrow A^{\delta}_{\mathbf{K}_\epsilon}[0] + A^{\delta+1}_{\mathbf{K}_\epsilon}[0].$$

For $\delta = 0$, Rule 4 becomes

$$A^{\delta}_{\mathbf{K}_\epsilon}[k_\epsilon - k_{\epsilon-1} + 1] \leftarrow A^{\delta}_{\mathbf{K}_\epsilon}[0] + A^{\delta+1}_{\mathbf{K}_\epsilon}[0],$$

provided that $D^{\delta}_{\mathbf{K}_\epsilon} f$ is of positive order, i.e., $\epsilon > 1$.

The case $\epsilon = 1$ is treated separately because for $\epsilon = 1$ and $\delta = 0$, the derivative $D^{\delta}_{\mathbf{K}_\epsilon} f = f$, i.e., the original polynomial f, whose corresponding array of storage $A^0_{\mathbf{K}_0}$ is indexed from 1 to $n+1$. In particular, the value of the polynomial at the origin is stored at $A^0_{\mathbf{K}_0}[1]$. This situation occurs only when the evaluation at a unit vector $\boldsymbol{e}_{k_1}$ is takes place, since its predecessor under $\mathcal{R}_{\text{lex}}$ is 0^n. Thus, in computing $f(\boldsymbol{e}_{k_1})$, the corresponding update is

$$A^0_{\mathbf{K}_0}[k_1 + 1] \leftarrow A^0_{\mathbf{K}_0}[1] + A^1_{\mathbf{K}_1}[0].$$

Boundary 2: When the derivative order equal to d. Suppose that the order of the derivative appearing in the (20) reaches to d. In particular, the derivative

$$D_{\boldsymbol{K}_\epsilon}^{\delta+1} f$$

has total order d. Such a derivative is constant, and hence its value at every input is equal to its value at 0^n. Since a Type-d array contains only this value, the second term in the applicable update rule is always read from entry 0, independently of the predecessor input. Accordingly, Rules 1–4 become, respectively,

$$\begin{aligned} A_{\boldsymbol{K}_\epsilon}^{\delta}[1] &\leftarrow A_{\boldsymbol{K}_\epsilon}^{\delta}[1] + A_{\boldsymbol{K}_\epsilon}^{\delta+1}[0], \\ A_{\boldsymbol{K}_\epsilon}^{\delta}[1] &\leftarrow A_{\boldsymbol{K}_\epsilon}^{\delta}[k_{\epsilon+1} - k_\epsilon + 1] + A_{\boldsymbol{K}_\epsilon}^{\delta+1}[0], \\ A_{\boldsymbol{K}_\epsilon}^{\delta}[k_\epsilon - k_{\epsilon-1} + 1] &\leftarrow A_{\boldsymbol{K}_\epsilon}^{\delta}[k_\epsilon - k_{\epsilon-1} + 1] + A_{\boldsymbol{K}_\epsilon}^{\delta+1}[0], \\ \text{and} \quad A_{\boldsymbol{K}_\epsilon}^{\delta}[k_\epsilon - k_{\epsilon-1} + 1] &\leftarrow A_{\boldsymbol{K}_\epsilon}^{\delta}[k_{\epsilon+1} - k_{\epsilon-1} + 1] + A_{\boldsymbol{K}_\epsilon}^{\delta+1}[0]. \end{aligned}$$

The two boundary cases identify the derivative value from which the evaluation of the current input can be initialized. If the predecessor is 0^n, the required value is obtained directly from the initial derivative values stored at the origin; if the derivative order reaches d, the corresponding derivative is constant and its value is likewise obtained from its Type-d array. Starting from the applicable stored value, the preceding derivative values are recovered successively through the corresponding update rules until the value of f at the current input is obtained.

Together, the four ordinary update rules and the two boundary conventions completely specify how the reusable arrays are accessed and overwritten during the evaluation. We next incorporate these rules into an explicit algorithm for evaluating f under the lexicographic evaluation rule $\mathcal{R}_{\text{lex}}$.

Evaluation at a given input: We first consider the evaluation of f at a fixed nonzero input $\boldsymbol{x} = (x_n, \ldots, x_1)$ with $\mathrm{Sup}(\boldsymbol{x}) = (k_1, \ldots, k_h)$. Let an array K stores the the support of $\boldsymbol{x}$ and an array V store the corresponding values, i.e.,

$$K[1:h] = (k_1, \ldots, k_h), \qquad V[r] = x_{K[r]}, \quad 1 \le r \le h,$$

We also maintain the total coordinate sum as

$$T = \sum_{r=1}^{h} V[r] = \sum_{r=1}^{h} x_{k_r},$$

where the sum is taken over the canonical integer representatives, not in $\mathbb{F}_q$. The quantity T identifies the boundary from which the evaluation of $\boldsymbol{x}$ can be initialized. Since each step of the non-recursive formulation decreases the least-index non zero coordinate by one and simultaneously increases the total derivative order by one. Hence exactly T such steps are required to reduce $\boldsymbol{x}$ to 0^n. Consequently, if $T < d$, the origin is reached before derivative order d; if $T > d$, derivative order d is reached first; and if $T = d$, the two situations coincide. Thus, T determines which of the two boundary situations supplies the initial derivative value for the evaluation of $\boldsymbol{x}$.

For describing the accumulated derivative order, define

$$H_0 = 0, \qquad H_\epsilon = \sum_{j=1}^{\epsilon} x_{k_j}, \qquad 1 \le \epsilon \le h,$$

so that $H_h = T$. These partial sums are used only to describe the algorithm and need not be supplied explicitly. Indeed, Algorithm reconstructs these required partial sums directly from (K, V). In the

notation of the non-recursive framework, the derivative $D^{\delta}_{\boldsymbol{k}_\epsilon}$ has total order $H_{\epsilon-1}+\delta$. We propose an algorithm `EVA` which requires

$$(T, K, V, h, d).$$

for the given input $\boldsymbol{x}$ and returns $f(\boldsymbol{x})$. For the moment, we assume that this information is available. After describing `EVA` (Algorithm 3), we show how it can be maintained directly while the elements of the evaluation set are traversed in lexicographic order.

Working of `EVA`: We first consider the simplest case $T = 1$ for which algorithm deals separately. For this is case evaluation of unit vectors $\boldsymbol{e}_{k_1}$ with $h = 1$. This is precisely the exceptional Type-0 case discussed in Boundary 1. Therefore, the evaluation of $f(\boldsymbol{x})$ for $\boldsymbol{x} = \boldsymbol{e}_{k_1}$ can be done with the update rule

$$A^0_{\boldsymbol{K}_0}[k_1+1] \leftarrow A^0_{\boldsymbol{K}_0}[1] + A^1_{\boldsymbol{K}_1}[0].$$

Now suppose $T > 1$. The algorithm first identifies the step of the non-recursive formulation from which the evaluation of $\boldsymbol{x}$ can be initialized. More precisely, it seeks the highest relevant derivative order for which the corresponding evaluation step involves, on its right-hand side, a higher-order derivative value that is an initial derivative value already available from the initialization phase. For this purpose, the algorithm initializes $S = 0$ and $p = 1$, and scans the support from the least-index nonzero coordinate upward. As p moves forward through K, the values $V[1], V[2], \ldots$ are accumulated into S. Thus, after processing $k_1, \ldots, k_\epsilon$,

$$S = H_\epsilon = \sum_{p=1}^{\epsilon} x_{k_p}.$$

The scan stops when S first reaches or exceeds d, or after the complete support has been traversed. Accordingly, if if $T \geq d$, it identifies the first $p = \epsilon$ satisfying

$$H_{p-1} < d \leq H_p,$$

whereas if $T < d$, it reaches the final support coordinate and $p = h$. At this stage,

$$\epsilon = p, \qquad k_\epsilon = K[p], \qquad x_{k_\epsilon} = V[p], \qquad S = H_\epsilon.$$

Removing the contribution of the currently active coordinate therefore gives

$$\eta = S - x_{k_\epsilon} = H_{\epsilon-1}.$$

This quantity has a direct interpretation: it is the derivative order already accumulated before differentiation with respect to x_{k_ϵ} begins and before the order of the derivative reached to d. If an additional δ differentiations are taken with respect to x_{k_ϵ}, the current total derivative order becomes $i = \eta + \delta$. The value of δ from which the evaluation process should begin is therefore chosen as $\delta = \delta_{\max}$ where

$$\delta_{\max} = \min\{x_{k_\epsilon} - 1,\ d - 1 - \eta\}.$$

This choice identifies the first update in the non-recursive formulation for which the required higher-order derivative value can be obtained directly from the initialization phase.

In particular, If $T \geq d$, the initial scan stops at $p = \epsilon$ for which

$$H_{\epsilon-1} < d \leq H_\epsilon.$$

Hence, $d - 1 - H_{\epsilon-1} \leq H_\epsilon - H_{\epsilon-1} - 1 = x_{k_\epsilon} - 1$, and therefore

$$\delta_{\max} = d - 1 - H_{\epsilon-1} = d - 1 - \eta.$$

The first derivative processed consequently has order

$$i = H_{\epsilon-1} + \delta_{\max} = d - 1,$$

so the higher-order derivative appearing in the corresponding update has total order d. Therefore, the initial derivative value is obtained from a Type-d array.
If $T < d$, the scan traverses the entire support giving $\epsilon = h$. In this case,

$$\delta_{\max} = x_{k_h} - 1 \qquad \text{and} \qquad i = H_{h-1} + x_{k_h} - 1 = T - 1.$$

Moreover, the active coordinate has value

$$\zeta = x_{k_h} - \delta_{\max} = 1.$$

Since k_h is the largest support index of $\boldsymbol{x}$ and Lemma 4 ensures that all lower-index coordinates have already become zero, decreasing this coordinate by one makes the predecessor 0^n and the corresponding array provides the initial derivative value. Hence, $\delta_{\max}$ is the correct initial choice of δ.

Starting from the appropriate boundary, the inner loop processes

$$\delta = \delta_{\max}, \delta_{\max} - 1, \ldots, 0$$

thereby completing all updates associated with the current active coordinate x_{k_ϵ} before moving to $x_{k_{\epsilon-1}}$. For each δ, the quantaties $i = \eta + \delta$, $\zeta = x_{k_\epsilon} - \delta$ give, respectively, the total order of the current derivative and the current value of the active coordinate. These quantities are sufficient to determine whether one of the boundary cases applies or which of Rules 1–4 must be used.

After all required updates for x_{k_ϵ} have been performed, the assignment $S \leftarrow \eta$ removes the contribution of this coordinate from the accumulated sum and replaces H_ϵ by $H_{\epsilon-1}$, while $p \leftarrow p - 1$ proceeds the algorithm to the preceding support coordinate, i.e., to $x_{k_{\epsilon-1}}$. Repeating this process reconstructs successively lower-order derivatives until the Type-0 entry containing $f(\boldsymbol{x})$ is obtained.

Algorithm 3: Evaluation of f from a support-value representation (Part 1)

Input: T, K, V, h, d
// $K[1:h] = (k_1, \ldots, k_h)$ and $V[r] = x_{K[r]}$
if $T = 1$ **then**
  // necessarily $\boldsymbol{x} = \boldsymbol{e}_{k_1}$ and $h = 1$
  $k_1 \leftarrow K[1]$;
  $A^0_{\boldsymbol{K}_0}[k_1 + 1] \leftarrow A^0_{\boldsymbol{K}_0}[1] + A^1_{\boldsymbol{K}_1}[0]$;
  return $A^0_{\boldsymbol{K}_0}[k_1 + 1]$;
$S \leftarrow 0$; $p \leftarrow 1$;
// locate the support coordinate at which order d is reached, if any
while $p \leq h$ ***and*** $S < d$ **do**
  $S \leftarrow S + V[p]$;
  $p \leftarrow p + 1$;
$p \leftarrow p - 1$;
// $S = H_\epsilon$ for the first block to be processed
// Algorithm continues in Part 2...

Algorithm 3: Evaluation of f from a support-value representation (Part 2)

while $p \geq 1$ **do**
  $\epsilon \leftarrow p$;
  $k_\epsilon \leftarrow K[p]$;
  $x_{k_\epsilon} \leftarrow V[p]$;
  if $p > 1$ **then**
    $k_{\epsilon-1} \leftarrow K[p-1]$;
  else
    $k_{\epsilon-1} \leftarrow 0$;
    // $k_0 = 0$
  $\eta \leftarrow S - x_{k_\epsilon}$;
  // $\eta = H_{\epsilon-1}$
  $\delta_{\max} \leftarrow \min\{x_{k_\epsilon} - 1,\ d - 1 - \eta\}$;
  for $\delta = \delta_{\max}, \delta_{\max} - 1, \ldots, 0$ **do**
    $i \leftarrow \eta + \delta$;
    // order of the current derivative
    $\zeta \leftarrow x_{k_\epsilon} - \delta$;
    // current value of the active coordinate
    if $i = T - 1$ **then**
      // Boundary 1: the predecessor is 0^n
      if $\delta > 0$ **then**
        $A^{\delta}_{\boldsymbol{K}_\epsilon}[1] \leftarrow A^{\delta}_{\boldsymbol{K}_\epsilon}[0] + A^{\delta+1}_{\boldsymbol{K}_\epsilon}[0]$;
      else
        // $T > 1$ here, so the current derivative has positive order
        $A^{\delta}_{\boldsymbol{K}_\epsilon}[k_\epsilon - k_{\epsilon-1} + 1] \leftarrow A^{\delta}_{\boldsymbol{K}_\epsilon}[0] + A^{\delta+1}_{\boldsymbol{K}_\epsilon}[0]$;
    else
      // the predecessor is nonzero
      $\mu \leftarrow \begin{cases} 0, & i = d-1, \\ 1, & i < d-1 \end{cases}$;
      // Boundary 2: the second derivative is Type-d if $\mu = 0$
      if $\delta > 0$ **then**
        if $\zeta > 1$ **then**
          // Rule 1
          $A^{\delta}_{\boldsymbol{K}_\epsilon}[1] \leftarrow A^{\delta}_{\boldsymbol{K}_\epsilon}[1] + A^{\delta+1}_{\boldsymbol{K}_\epsilon}[\mu]$;
        else
          // Rule 2: $\zeta = 1$
          $k_{\epsilon+1} \leftarrow K[p+1]$;
          // $p < h$ since the predecessor is nonzero
          $A^{\delta}_{\boldsymbol{K}_\epsilon}[1] \leftarrow A^{\delta}_{\boldsymbol{K}_\epsilon}[k_{\epsilon+1} - k_\epsilon + 1] + A^{\delta+1}_{\boldsymbol{K}_\epsilon}[\mu(k_{\epsilon+1} - k_\epsilon + 1)]$;
      else
        // $\delta = 0$
        if $\zeta > 1$ **then**
          // Rule 3
          $A^{\delta}_{\boldsymbol{K}_\epsilon}[k_\epsilon - k_{\epsilon-1} + 1] \leftarrow A^{\delta}_{\boldsymbol{K}_\epsilon}[k_\epsilon - k_{\epsilon-1} + 1] + A^{\delta+1}_{\boldsymbol{K}_\epsilon}[\mu]$;
        else
          // Rule 4: $\zeta = 1$
          $k_{\epsilon+1} \leftarrow K[p+1]$;
          // $p < h$ since the predecessor is nonzero
          $A^{\delta}_{\boldsymbol{K}_\epsilon}[k_\epsilon - k_{\epsilon-1} + 1] \leftarrow A^{\delta}_{\boldsymbol{K}_\epsilon}[k_{\epsilon+1} - k_{\epsilon-1} + 1] + A^{\delta+1}_{\boldsymbol{K}_\epsilon}[\mu(k_{\epsilon+1} - k_\epsilon + 1)]$;
  $S \leftarrow \eta$;
  // now $S = H_{\epsilon-1}$
  $p \leftarrow p - 1$;
return $A_{\boldsymbol{K}_0}[K[1] + 1]$;
// $K[1] = k_1$

Complexity of Algorithm 3 We now determine the number field operations equivalently the number of memory update steps performed for a fixed nonzero input $\boldsymbol{x}$. For a given active coordinate x_{k_ϵ}, in the absence of the degree restriction, the derivative orders processed by the inner loop are

$$i = H_{\epsilon-1} + \delta, \qquad 0 \le \delta \le x_{k_\epsilon} - 1.$$

Thus, in the absence of the degree restriction, the coordinate x_{k_ϵ} corresponds to the interval $[H_{\epsilon-1}, H_\epsilon - 1]$. Over all support coordinates, these intervals are disjoint and consecutive:

$$[H_0, H_1 - 1],\ [H_1, H_2 - 1], \ldots, [H_{h-1}, H_h - 1],$$

and their union is $\{0, 1, \ldots, T-1\}$.

If $T < d$, then $H_\epsilon \le T < d$ for every ϵ, so the degree bound never restricts the inner loop. Hence $\delta_{\max} = x_{k_\epsilon} - 1$ for every support coordinate, and the ϵ-th coordinate contributes exactly x_{k_ϵ} updates. Therefore, the total number of updates is

$$\sum_{\epsilon=1}^{h} x_{k_\epsilon} = T.$$

Now suppose that $T \ge d$, and let

$$\epsilon^* = \min\{\epsilon : H_\epsilon \ge d\}.$$

is the first value encountered by the outer loop for which $H_{\epsilon^*-1} < d \le H_{\epsilon^*}$. The first active coordinate therefore has

$$\delta_{\max} = d - 1 - H_{\epsilon^*-1},$$

and therefore contributes $d - H_{\epsilon^*-1}$ updates. For every subsequent coordinate $\epsilon < \epsilon^*$, the degree restriction is inactive, and the corresponding number of updates is x_{k_ϵ}. Their total contribution is

$$\sum_{\epsilon=1}^{\epsilon^*-1} x_{k_\epsilon} = H_{\epsilon^*-1}.$$

Hence the total number of updates is

$$\left(d - H_{\epsilon^*-1}\right) + H_{\epsilon^*-1} = d.$$

Combining the two cases, Algorithm 3 performs exactly

$$\min\{T, d\}$$

memory updates for a nonzero input $\boldsymbol{x}$. In particular, the nested loops do not incur an $\mathcal{O}(hd)$ cost, because the inner loops process disjoint consecutive ranges of derivative orders whose total length is $\min T, d$.

It remains to account for the initial scan over the support array K. If $T < d$, the scan examines all h nonzero coordinates. Since each nonzero coordinate contributes at least 1 to T, we have

$$h \le T.$$

If $T \ge d$, the scan stops once the accumulated sum first reaches d. Again, since every scanned coordinate contributes at least 1, at most d coordinates are examined. Thus, in either case, the scan requires

$$\mathcal{O}(\min\{T, d\})$$

operations.

Each memory update performs only a constant number of comparisons, index computations, array accesses, and one addition in $\mathbb{F}_q$. Therefore, including both the initial scan and the memory updates, the running time for evaluating a fixed nonzero input is

$$T_{\texttt{EVA}}(\boldsymbol{x}) = \mathcal{O}\left(\min\{T, d\}\right) \leq \mathcal{O}(d). \tag{21}$$

Algorithm EVA updates the reusable derivative arrays in place and does not allocate additional copies of them. The support positions and their corresponding values are supplied through K and V. Apart from these persistent arrays and the input representation, the algorithm uses only a constant number of scalar variables, such as $S, p, \epsilon, \eta, \delta, i, \zeta$, and μ. Hence its additional working memory is $\mathcal{O}(1)$.

Having described how Algorithm 3 evaluates f at a given input once its support-value representation is available, it remains to generate all inputs of the product Hamming ball in the required lexicographic order while maintaining this information efficiently. We propose an Algorithm ENU (Algorithm 4) which performs this task. Instead of explicitly constructing each vector in $B(\boldsymbol{n}, \boldsymbol{w})$, it maintains only the nonzero coordinate positions, their corresponding values, and the componentwise support counts needed to enforce the Hamming-weight constraints. The resulting representation is supplied directly to Algorithm 3 for evaluation.

Working of ENU: Consider the product Hamming ball

$$B(\boldsymbol{n}, \boldsymbol{w}) = B(n_s, w_s) \times \cdots \times B(n_1, w_1).$$

Define the cumulative block boundaries by

$$N[j] = \sum_{\ell=1}^{j} n_\ell, \qquad N[0] = 0$$

so that the coordinates of the j-th block are $N[j-1]+1, \ldots, N[j]$. For algorithmic purposes, the array W stores the corresponding Hamming-weight bounds, while $I[i]$ records the block containing coordinate i. Hence,

$$W[1:s] = (w_1, \ldots, w_s) \qquad \text{and} \qquad I[i] = j \text{ whenever } N[j-1] < i \leq N[j].$$

The array $C[1:s]$ maintains the number of nonzero coordinates currently selected from each block. Consequently, a coordinate i belonging to block j may be selected only when $C[j] < W[j]$. The arrays N, W, I, together with the dynamically maintained counts C, are therefore sufficient to enforce membership in $B(\boldsymbol{n}, \boldsymbol{w})$ without explicitly constructing the evaluation set.

We next describe the invariant maintained by each recursive call. Suppose at the current level of recursion, the current support has size h and is stored right-aligned in the preallocated arrays K and V as

$$K[n-h+1:n] = (k_1, \ldots, k_h), \quad k_1 < \cdots < k_h, \qquad \text{with} \qquad V[n-h+r] = x_{k_r}, \quad 1 \leq r \leq h.$$

The variable $T = \sum_{r=1}^{h} x_{k_r}$ stores the total coordinate sum, and $C[j]$ stores the number of currently selected support positions belonging to block j. The parameter end denotes the largest coordinate index that may be selected in the current recursive call. Initially, end $= n$, while every recursive extension restricts it to an index smaller than the coordinate selected in the preceding call.

When a new coordinate position i is selected, the algorithm find the block $j = I[i]$ containing i. If $C[j] < W[j]$, the coordinate may be included in the support. Since the current support occupies the suffix $K[n-h+1:n]$, setting $b = n-h$ gives the position immediately to its left. For each nonzero field value $a \in \{1, \ldots, q-1\}$, the assignments

$$K[b] \leftarrow i, \qquad V[b] \leftarrow a$$

extend the active support to $K[b:n] = (i, k_1, \ldots, k_h)$.

Since each recursive call restricts the next choice to indices smaller than the coordinate selected in the preceding call, the coordinates are selected along a branch in strictly decreasing order. Thus, if the current support is $(k_1, \ldots, k_h)$, any newly selected coordinate satisfies $i < k_1$, and the enlarged support $K[b:n]$ remains in the increasing order required by `EVA`. Importantly, $K[b:n]$ and $V[b:n]$ are passed as views of the preallocated arrays, rather than as newly allocated copies. After selecting (i, a), the state variables are updated by

$$C[j] \leftarrow C[j] + 1, \qquad T \leftarrow T + a,$$

so that $C[j]$ records the number of nonzero coordinates currently selected from block j, while T remains the total coordinate sum of the enlarged support. Hence, for the support of size $h+1$,

$$T, \quad K[b:n], \quad V[b:n], \quad h+1, \quad d$$

contain exactly the information required by Algorithm 3. Therefore, `EVA` is called immediately and the resulting value of f is output. No explicit construction of the corresponding vector $\boldsymbol{x}$ or recomputation of its support is required.

It remains to determine how the newly generated support-value representation may be further extended. If $C[j] < W[j]$, the Hamming-weight bound of block j has not yet been reached, so additional support positions may be selected from any coordinate below i. The recursive call therefore uses $\texttt{end} = i - 1$.

If the weight bound of block j has been reached, i.e., $C[j] = W[j]$, no further coordinate from that block may be selected. Since future recursive choices must also have smaller indices than the current coordinate, all remaining lower-index positions of block j can be skipped. When $j > 1$, the largest coordinate available in the preceding block is $N[j-1]$, so the recursion continues from this position. For $j = 1$, no preceding block exists and no further extension is possible.

The recursive traversal also produces the required ascending lexicographic order. At every recursion level, candidate coordinate positions are considered in increasing order, and for each fixed position the nonzero values $1, \ldots, q-1$ are considered in increasing order. For a fixed pair (i, a), the recursive call is completed before the value loop advances to the next a, and the value loop itself is completed before the position loop advances to the next i. Hence all admissible extensions using lower coordinate indices are exhausted before either the value at i or the current coordinate position is changed. This is enforced by the recursive calls in Lines 14 and 16, which are executed before the backtracking step and before either loop advances. Since vectors are written as $(x_n, \ldots, x_1)$, coordinates with smaller indices are less significant in the lexicographic order. Consequently, after fixing a coordinate i and its value, all admissible choices involving coordinates $i-1, \ldots, 1$ must be exhausted before the algorithm proceeds to the next value at i or to a larger coordinate index. Hence the resulting traversal is precisely the ascending lexicographic order required by the evaluation framework.

Each admissible support is generated exactly once. Along a recursive branch, selected coordinate indices are strictly decreasing; therefore, a support $(k_1, \ldots, k_h)$, with $k_1 < \cdots < k_h$, can arise only through the unique sequence of recursive selections

$$k_h \longrightarrow k_{h-1} \longrightarrow \cdots \longrightarrow k_1.$$

For a fixed support, the nested value loops similarly generate each assignment of nonzero field values exactly once. Thus, `ENU` generates every nonzero element of $B(\boldsymbol{n}, \boldsymbol{w})$ once and only once.

After all admissible extensions of a selected pair (i, a) have been processed, the assignments

$$T \leftarrow T - a, \qquad C[j] \leftarrow C[j] - 1$$

restore the state of the parent recursive call. No explicit restoration of K or V is required. At depth h, the parent's support occupies $K[n-h+1:n]$ and $V[n-h+1:n]$, whereas every descendant writes

only to positions further to the left. Hence the parent's active suffix remains unchanged throughout the recursive call.

Finally, the origin 0^n has empty support and is therefore not generated by ENU. It is handled separately in MAIN, where $f(0^n) = A^0_{\boldsymbol{K}_0}[1]$ is already available from the initialization phase. The initial call $\texttt{ENU}(K, V, C, I, W, N, 0, n, 0, d)$ then generates all remaining admissible inputs in ascending lexicographic order and invokes EVA once for each of them.

Complexity of ENU Algorithm 4: We first consider the work performed by Algorithm 4 itself, excluding the calls to Algorithm 3. This separation is useful because the purpose of ENU is only to traverse the evaluation set and maintain the support-value representation, whereas the arithmetic required to evaluate f is performed by EVA.

Before the recursion begins, MAIN constructs the block-boundary array N and the auxiliary array I. Computing $N[j] = \sum_{\ell=1}^{j} n_\ell$ requires $\mathcal{O}(s)$ operations, while assigning the component index $I[i]$ to each of the n coordinate positions requires $\mathcal{O}(n)$ operations. The remaining arrays are initialized in $\mathcal{O}(n+s)$ time. Hence the one-time preprocessing cost is $\mathcal{O}(n+s) = O(n)$, since $s \le n$.

We next consider the recursive enumeration. At recursion depth h, selecting a pair (i, a) with $a \in \{1, \dots, q-1\}$, creates one new admissible support-value representation. Under the right-aligned storage convention, the assignments

$$K[n-h] \leftarrow i, \qquad V[n-h] \leftarrow a, \qquad C[j] \leftarrow C[j] + 1, \qquad T \leftarrow T + a$$

require constant time, as do the corresponding backtracking operations. Moreover, the active portions of K and V are passed to EVA as views of the preallocated arrays and therefore require no copying.

The pruning imposed by $C[j]$ and $W[j]$ prevents the recursion from exploring further coordinates of a block once its weight bound has been reached. Since every admissible nonzero support-value representation is generated exactly once Hence, apart from the one-time preprocessing, the recursive traversal performs only constant work for each admissible support-value representation.

For this output-sensitive bound, any block with $w_j = 0$ is skipped by the recursive search, since all of its coordinates are necessarily zero. Such blocks can be identified during preprocessing while retaining the original coordinate indices used by K. Thus, only blocks with positive Hamming-weight bounds participate in the recursion.

Since there are $|B(\boldsymbol{n}, \boldsymbol{w})| - 1$ nonzero elements in the evaluation set, the enumeration cost is therefore

$$\mathcal{O}\left(|B(\boldsymbol{n}, \boldsymbol{w})|\right).$$

Including the one-time preprocessing performed by MAIN, we obtain

$$T_{\texttt{ENU}} = \mathcal{O}\left(n + |B(\boldsymbol{n}, \boldsymbol{w})|\right).$$

We now include the calls to Algorithm 3.The origin is output directly in MAIN, while EVA is called exactly once for every nonzero input $\boldsymbol{x} \in B(\boldsymbol{n}, \boldsymbol{w})$. By Equation (21), its cost at $\boldsymbol{x}$ is $\mathcal{O}(\min\{T(\boldsymbol{x}), d\}) = \mathcal{O}(d)$. Consequently, for $d \ge 1$, time completxity of the complete enumeration-and-evaluation phase is

$$T_{\texttt{ENU+EVA}} = \mathcal{O}\left(n + d\,|B(\boldsymbol{n}, \boldsymbol{w})|\right) \tag{22}$$

Memory complexity: Algorithm 4 maintains only the information required to represent the current branch of the enumeration. The preallocated arrays K and V each contain at most n entries: K stores support positions and V stores their corresponding nonzero field values, with the active support-value representation occupying a right-aligned suffix. The array I contains one block index for each coordinate and therefore also has length n. The arrays C, W, and N contain $\mathcal{O}(s)$ entries each.

Algorithm 4: Enumeration of $B(\boldsymbol{n}, \boldsymbol{w})$ and evaluation of f

Procedure *ENU*$(K, V, C, I, W, N, h, end, T, d)$
  for $i = 1, 2, \ldots, end$ **do**
    $j \leftarrow I[i]$;
    // coordinate i belongs to component j
    if $C[j] < W[j]$ **then**
      $b \leftarrow n - h$;
      // position immediately to the left of the current active support
      for $a = 1, 2, \ldots, q - 1$ **do**
        // choose the nonzero value of coordinate x_i
        $K[b] \leftarrow i$;
        $V[b] \leftarrow a$;
        // $K[b : n]$ now stores the support in increasing order
        $C[j] \leftarrow C[j] + 1$;
        // increase the Hamming weight of component j
        $T \leftarrow T + a$;
        // maintain $T = \sum_\ell x_\ell$
        // $K[b : n]$ and $V[b : n]$ are passed as 1-indexed views
        $y \leftarrow \texttt{EVA}(T, K[b : n], V[b : n], h + 1, d)$;
        Output y;
        // $y = f(\boldsymbol{x})$ for the current support-value representation
        if $C[j] < W[j]$ **then**
          ENU $(K, V, C, I, W, N, h + 1, i - 1, T, d)$;
          // continue with lower coordinate positions
        else if $C[j] = W[j]$ ***and*** $j > 1$ **then**
          ENU $(K, V, C, I, W, N, h + 1, N[j - 1], T, d)$;
          // skip the remaining coordinates of component j
        $T \leftarrow T - a$;
        $C[j] \leftarrow C[j] - 1$;
        // backtrack

Procedure *MAIN*$()$
  $W[1, \ldots, s] \leftarrow (w_1, \ldots, w_s)$;
  $N[0] \leftarrow 0$;
  for $j = 1, 2, \ldots, s$ **do**
    $N[j] \leftarrow N[j - 1] + n_j$;
  $C[1, \ldots, s] \leftarrow (0, \ldots, 0)$;
  Initialize $K[1, \ldots, n]$ and $V[1, \ldots, n]$;
  for $j = 1, 2, \ldots, s$ **do**
    for $i = N[j - 1] + 1, \ldots, N[j]$ **do**
      $I[i] \leftarrow j$;
  Output $A^0_{K_0}[1]$;
  // $f(0^n)$
  ENU$(K, V, C, I, W, N, 0, n, 0, d)$;

The recursion depth is at most $\sum_{j=1}^{s} w_j \leq n$, because every recursive level adds one new nonzero coordinate to the current support. The arrays are shared among all recursive calls, and only a constant amount of local information is maintained at each level. Hence, the recursion stack requires at most $\mathcal{O}(n)$ additional memory locations.

Consequently, the total auxiliary storage required by Algorithm 4 is

$$M_{\texttt{ENU}} = \mathcal{O}(n+s) = \mathcal{O}(n) \tag{23}$$

memory locations. This bound excludes the reusable derivative arrays maintained by the evaluation framework. Importantly, the memory required by `ENU` is independent of $|B(\boldsymbol{n}, \boldsymbol{w})|$, since the current support-value representation is maintained in place and previously enumerated inputs need not be stored.

6.4 Overall Complexity

We are now in a position to combine the two phases of the proposed framework. The initialization phase is performed only once and constructs the reusable derivative arrays, whereas the subsequent evaluation phase traverses $B(\boldsymbol{n}, \boldsymbol{w})$ and repeatedly updates these arrays to obtain the polynomial values. Since the initialization may be carried out using either Method 1 or Method 2, we consider the two resulting complexities separately.

Time complexity: We first recall the initialization costs derived earlier. By Equation (18a), Method 1 requires $\mathcal{O}\left(nrM + r^2\right)$ operations, whereas Equation (19a) gives $\mathcal{O}\left(rM^2 + r^2\right)$ for Method 2.

Once initialization has been completed, Algorithm 4 traverses the product Hamming ball and calls Algorithm 3 once for every nonzero element. As established in Equation (22), the complete enumeration-and-evaluation phase requires $\mathcal{O}\left(n + d\,|B(\boldsymbol{n}, \boldsymbol{w})|\right)$ operations. The $\mathcal{O}(n)$ term is the one-time preprocessing required to set up the component information, while the second term accounts for the evaluation of all points of the product Hamming ball.

Combining this with Equation (18a), the overall running time when Method 1 is used for initialization is $\mathcal{O}\left(nrM + r^2 + n + d\,|B(\boldsymbol{n}, \boldsymbol{w})|\right)$. For $d \geq 1$, we have $r \geq 1$ and $M \geq \max\{n+1, r+1\}$. Consequently, the term nrM dominates the terms r^2 and n. Hence, the overall running time with Method 1 simplifies to

$$T_{\text{overall with Method 1}} = \mathcal{O}\left(nrM + d\,|B(\boldsymbol{n}, \boldsymbol{w})|\right). \tag{24}$$

Similarly, combining the evaluation cost with Equation (19a) gives $\mathcal{O}\left(rM^2 + r^2 + n + d\,|B(\boldsymbol{n}, \boldsymbol{w})|\right)$ when Method 2 is employed. Again, the term rM^2 dominates the r^2 and n, and hence the overall running time becomes

$$T_{\text{overall with Method 2}} = \mathcal{O}\left(rM^2 + d\,|B(\boldsymbol{n}, \boldsymbol{w})|\right). \tag{25}$$

Thus, irrespective of the initialization method, the contribution of the evaluation phase is linear in both the degree parameter d and the number of points actually evaluated. The difference between the two versions of the framework lies entirely in the one-time cost of constructing the initial derivative information.

Memory complexity: The same separation applies to memory. The derivative arrays requiring M storage constructed during initialization are retained and reused throughout the complete lexicographic traversal; no separate copy of these arrays is created for an individual evaluation point. Algorithm 3 itself uses only $\mathcal{O}(1)$ additional working words, while Algorithm 4 maintains the current support-value representation and the recursion state using $\mathcal{O}(n)$ words.

To combine this with the initialization bounds, it is useful to express the enumeration storage in bits. The support indices stored in K require $\mathcal{O}(\log n)$ bits each, while the corresponding field values

in V require $\mathcal{O}(\log q)$ bits each. Together with the remaining auxiliary arrays and the recursion state, the enumeration contributes $\mathcal{O}\left(n \log(nq)\right)$ bits.

Using Method 1, with Equation (18b), the overall working-memory requirement is

$$M_{\text{overall with Method 1}} = \mathcal{O}\left((M + r^2)\log q + n\log(nq)\right). \tag{26}$$

For Method 2, combining the enumeration storage with Equation (19b) yields

$$M_{\text{overall with Method 2}} = \mathcal{O}\left(((d+1)M + r^2)\log q + n\log(nq)\right) = \mathcal{O}((d+1)M\log q + n\log(nq)). \tag{27}$$

The above equality holds because $r \le d$, and $M \ge r+1$ implies $r^2 = \mathcal{O}((d+1)M)$.

6.5 Comparison with Existing Evaluation Algorithms

We now compare the proposed framework with two existing approaches for multivariate polynomial evaluation: the fast enumeration algorithm (FEA) over $\mathbb{F}_q$ proposed in [18], and structured MME over downward-closed evaluation spaces of van der Hoeven and Schost [29]. The comparison is made over the complete space $\mathbb{F}_q^n$, which provides a common evaluation domain for all three approaches.

By choosing $s = 1$ and $w_1 = n$, the product Hamming ball becomes the full space, i.e., $B(n,n) = \mathbb{F}_q^n$ and $|B(n,n)| = q^n$. Hence, the complexity bounds derived for our framework can be specialized to the complete space simply by substituting $|B(\boldsymbol{n},\boldsymbol{w})| = q^n$.

We first specialize the complexity bound of van der Hoeven and Schost. For a downward-closed index set I, and the associated evaluation space $U(I,\boldsymbol{u})$, the complexity given in [29] is

$$\mathcal{O}\left(|I|\sum_{j=1}^{n}\frac{M_{\text{mul}}(d_j)\log d_j}{d_j}\right),$$

where d_j denotes the size of the evaluation set associated with the j-th coordinate and $M_{\text{mul}}(d_j)$ denotes the cost of multiplying univariate polynomials of degree $\mathcal{O}(d_j)$. Recall from Theorem 2 that every product Hamming ball can be represented as a downward-closed evaluation space. Furthermore, for the complete space $\mathbb{F}_q^n$, Theorem 2 gives

$$|I| = q^n, \qquad \text{and} \qquad d_j = q, \quad j = 1,\dots,n.$$

Substituting these values yields

$$\begin{aligned}\mathcal{O}\left(|I|\sum_{j=1}^{n}\frac{M_{\text{mul}}(d_j)\log d_j}{d_j}\right) &= \mathcal{O}\left(q^n\sum_{j=1}^{n}\frac{M_{\text{mul}}(q)\log q}{q}\right)\\ &= \mathcal{O}\left(nq^n\frac{M_{\text{mul}}(q)\log q}{q}\right).\end{aligned}$$

Using the standard multiplication bound $M_{\text{mul}}(q) = \mathcal{O}\left(q\log q\log\log q\right)$, we obtain

$$\begin{aligned}\mathcal{O}\left(nq^n\frac{M_{\text{mul}}(q)\log q}{q}\right) &= \mathcal{O}\left(nq^n\frac{q\log q\log\log q\cdot\log q}{q}\right)\\ &= \mathcal{O}\left(nq^n(\log q)^2\log\log q\right).\end{aligned} \tag{28}$$

Thus, over the complete space $\mathbb{F}_q^n$, the MME algorithm has time complexity $\mathcal{O}\left(nq^n(\log q)^2\log\log q\right)$.

We next specialize the complexities of our framework. From Equation (24), Method 1 requires

$$\mathcal{O}\left(nrM + dq^n\right), \tag{29}$$

whereas Equation (25) gives

$$\mathcal{O}\left(rM^2 + dq^n\right) \tag{30}$$

for Method 2.

It is also useful to compare the initialization costs of the proposed methods with that of FEA. The initialization term appearing in the FEA bound is

$$\binom{n+d}{d}^2,$$

whereas the corresponding terms for Method 1 and Method 2 are nrM and rM^2 respectively. Moreover, $r \leq d$ and $M \leq \binom{n+d}{d}$, gives

$$nrM \leq ndM \; < \binom{n+d}{d} M \; \leq \binom{n+d}{d}^2$$

Thus, at the level of the derived initialization bounds, Method 1 is never asymptotically worse than the initialization bound appearing for FEA. The two may nevertheless have the same asymptotic order in special regimes; for example, when $d = 1$, both are of $\mathcal{O}(n^2)$.

The comparison between Method 2 and FEA is parameter-dependent. Their initialization terms satisfy

$$rM^2 < \binom{n+d}{d}^2$$

precisely when

$$M < \frac{1}{\sqrt{r}} \binom{n+d}{d}.$$

Thus, method 2 may have smaller initialization bound than the FEA only for $d \geq q$ and only for some specific values of n, d and q, i.e., no uniform ordering between these two initialization bounds therefore holds for all choices of n, d, and q.

We next compare evaluation using Method 1 with the structured MME, we write $\mathcal{O}(nrM + dq^n)$ as $\mathcal{O}(ndq^n)$, since $r \leq d$. From Equation (28), Method 1 is asymptotically smaller whenever

$$d = o\left((\log q)^2 \log\log q\right).$$

Note that this condition is sufficient rather than necessary, since the sharper bound $\mathcal{O}(nrM + dq^n)$ may be smaller even when this condition does not hold, particularly when M is substantially smaller than q^n. No analogous uniform comparison follows for Method 2 because its initialization term rM^2 depends more strongly on the parameters.

We finally compare the memory complexity of our method with FEA. The FEA algorithm requires

$$\mathcal{O}\left(n \log(nq) \binom{n+d}{d}\right)$$

bits, whereas the proposed framework requires

$$\mathcal{O}\left((M + r^2)\log q + n\log(nq)\right) \qquad \text{and} \qquad \mathcal{O}\left(((d+1)M + r^2)\log q + n\log(nq)\right),$$

with Method 1 and Method 2, respectively. In the fixed-field regime, i.e., with the fixed value of q, $r = \mathcal{O}(1)$, and $d = \mathcal{O}(n)$ satisfied. Therefore, Method 1 and Method 2 requires

$$\mathcal{O}(M + n\log n) \qquad \text{and} \qquad \mathcal{O}(nM + n\log n)$$

bits, respectively. Since $M \leq \binom{n+d}{d}$, both bounds are asymptotically smaller than the FEA memory bound $\mathcal{O}\left(n\log n\binom{n+d}{d}\right)$ in this regime. Thus, over a fixed finite field, both variants of the proposed framework provide an asymptotic memory advantage over FEA.

The updated complexity comparison is summarized in Table 2. For our two methods, we report the complete memory bounds, including the auxiliary storage required by the enumeration procedure. The memory bound for structured MME is left unspecified because the theorem used above states a time bound but no auxiliary memory bound.

Table 2: Complexity comparison for evaluating a degree-d polynomial in n variables over the complete space $\mathbb{F}_q^n$

Framework	Time	Memory
Fast enumeration algorithm [18]	$\mathcal{O}\left(\binom{n+d}{d}^2 + dq^n\right)$	$\mathcal{O}\left(n\log(qn)\binom{n+d}{d}\right)$
Structured MME [29]	$\mathcal{O}\left(nq^n(\log q)^2\log\log q\right)$	–
Our algorithm with Method 1	$\mathcal{O}\left(nrM + dq^n\right)$	$\mathcal{O}\left((M+r^2)\log q + n\log(nq)\right)$
Our algorithm with Method 2	$\mathcal{O}\left(rM^2 + dq^n\right)$	$\mathcal{O}\left(((d+1)M+r^2)\log q + n\log(nq)\right)$

6.6 Implication for Hamming-weighed Polynomial-System Evaluation

The proposed framework can be used directly as an evaluation primitive in exhaustive zero searches over product Hamming balls. Let

$$f_1, \ldots, f_m \in \mathbb{F}_q[x_n, \ldots, x_1]$$

be polynomials of degrees at most d, and suppose the objective is to determine the common zeros

$$\mathcal{Z} = \{\boldsymbol{x} \in B(\boldsymbol{n}, \boldsymbol{w}) : f_1(\boldsymbol{x}) = \cdots = f_m(\boldsymbol{x}) = 0\}.$$

After independently initializing the derivative information required for each polynomial, Algorithm `ENU` may be shared among all m polynomials, since the enumeration of $B(\boldsymbol{n}, \boldsymbol{w})$ is independent of the polynomial being evaluated. At every generated input, Algorithm `EVA` is applied to each polynomial and the input is retained whenever all resulting values are zero.

Consequently, apart from the one-time initialization costs, the evaluation part of such a search requires

$$\mathcal{O}(md\,|B(\boldsymbol{n}, \boldsymbol{w})|)$$

field operations when all polynomial degrees are bounded by d. More generally, for degrees $d_1, \ldots, d_m$, the corresponding evaluation cost is

$$\mathcal{O}\left(|B(\boldsymbol{n}, \boldsymbol{w})| \sum_{j=1}^{m} d_j\right).$$

Thus, the advantage over full-space exhaustive evaluation is determined directly by the size of the structured search domain.

A concrete example of such a domain arises in the Regular Multivariate Quadratic problem of Joux and Mora [21]. If the coordinates are divided into blocks of length ℓ, their at-most-ℓ-regular vectors contain at most one nonzero coordinate in every block. Hence, for w blocks, this domain is precisely

$$\underbrace{B(\ell,1) \times \cdots \times B(\ell,1)}_{w \text{ blocks}},$$

and therefore falls directly within the class of product Hamming balls considered in this work. The proposed framework can consequently evaluate the relevant polynomial systems over this domain without enumerating the surrounding complete space. Whether this evaluation primitive leads to an improved end-to-end complexity for the RMQ algorithms depends on the remaining interpolation and reconstruction steps and is left for future investigation.

7 Extension to Arbitrary Finite Fields $\mathbb{F}_{q^\rho}$

The proposed framework has so far been described over a prime field $\mathbb{F}_q$. We now explain its extension to a finite field $\mathbb{F}_{q^\rho}$, where q is prime and $\rho \geq 1$.

7.1 Finite Field Extension and Basis Representation

Every finite field of characteristic q contains $\mathbb{F}_q$ as its prime subfield, and $\mathbb{F}_{q^\rho}$ is a ρ-dimensional vector space over $\mathbb{F}_q$; see, for example, [25]. Thus, after fixing an ordered basis $\Theta = \{\theta_1, \ldots, \theta_\rho\}$ of $\mathbb{F}_{q^\rho}$ over $\mathbb{F}_q$, every element $a \in \mathbb{F}_{q^\rho}$ admits a unique representation

$$a = \sum_{t=1}^{\rho} a_t \theta_t, \qquad a_t \in \mathbb{F}_q.$$

Define the coordinate map

$$\phi_\Theta : \mathbb{F}_{q^\rho} \longrightarrow \mathbb{F}_q^\rho \quad \text{by} \quad \phi_\Theta(a) = (a_\rho, \ldots, a_1)$$

whenever $a = \sum_{t=1}^{\rho} a_t \theta_t$. The uniqueness of the basis representation implies that ϕ_Θ is a bijection. Applying this map coordinatewise gives a bijection

$$\Phi_\Theta : \mathbb{F}_{q^\rho}^n \longrightarrow \mathbb{F}_q^{n\rho}$$

defined by

$$\Phi_\Theta(x_n, \ldots, x_1) = \big(\phi_\Theta(x_n), \ldots, \phi_\Theta(x_1)\big).$$

More explicitly, if $x_i = \sum_{t=1}^{\rho} x_{i,t}\theta_t$, then $\Phi_\Theta(\boldsymbol{x}) = \big(x_{n,\rho}, \ldots, x_{n,1}, \ldots, x_{1,\rho}, \ldots, x_{1,1}\big)$. Thus, each original coordinate $x_i \in \mathbb{F}_{q^\rho}$ is replaced by a block $\boldsymbol{v}_i = (x_{i,\rho}, \ldots, x_{i,1}) \in \mathbb{F}_q^\rho$.

7.2 Polynomial Coordinate Expansion

Let

$$f(x_n,\dots,x_1)=\sum_{\substack{\boldsymbol{\beta}\in\mathbb{N}^n\\ |\boldsymbol{\beta}|\le d}} c_{\boldsymbol{\beta}}x_n^{\beta_n}\cdots x_1^{\beta_1}\in\mathbb{F}_{q^\rho}[x_n,\dots,x_1]$$

be a polynomial of total degree at most d, where $c_{\boldsymbol{\beta}}\in\mathbb{F}_{q^\rho}$. Each coefficient has a unique basis expansion

$$c_{\boldsymbol{\beta}}=\sum_{\ell=1}^{\rho}c_{\boldsymbol{\beta},\ell}\theta_\ell,\qquad c_{\boldsymbol{\beta},\ell}\in\mathbb{F}_q.$$

Substituting $x_i=\sum_{t=1}^{\rho}x_{i,t}\theta_t$ expresses every monomial of f in terms of the scalar basis coordinates. Since $\theta_a,\theta_b\in\mathbb{F}_{q^\rho}$ and $\mathbb{F}_{q^\rho}$ is a field, their product also belongs to $\mathbb{F}_{q^\rho}$ and therefore admits a unique expansion in the basis Θ. Hence there exist structure constants $\mu_{a,b,\ell}\in\mathbb{F}_q$ such that

$$\theta_a\theta_b=\sum_{\ell=1}^{\rho}\mu_{a,b,\ell}\theta_\ell.$$

Repeatedly applying these relations to products of basis elements yields a unique representation

$$f(x_n,\dots,x_1)=\sum_{\ell=1}^{\rho}f_\ell\big(\Phi_\Theta(\boldsymbol{x})\big)\theta_\ell, \tag{31}$$

where $f_\ell\in\mathbb{F}_q\left[x_{n,\rho},\dots,x_{n,1},\dots,x_{1,\rho},\dots,x_{1,1}\right]$.

Since the substitution of every x_i is linear in the basis coordinates, each coordinate polynomial f_ℓ has total degree at most d. Therefore, evaluating one polynomial over $\mathbb{F}_{q^\rho}$ is equivalent to evaluating ρ polynomials in $n\rho$ variables over $\mathbb{F}_q$ and then reconstructing the result through Equation (31).

Before applying the prime-field framework, each coordinate polynomial f_ℓ must be replaced by its canonical polynomial-function representative modulo $x_{i,t}^q-x_{i,t}$, for $1\le i\le n$ and $1\le t\le\rho$. This guarantees an individual degree strictly smaller than q, as required in Equation (5), without increasing the total degree.

7.3 Evaluation Sets and Preservation of the Predecessor Condition

For an arbitrary evaluation set $S\subseteq\mathbb{F}_{q^\rho}^n$, define its basis-coordinate representation by $S_\Theta=\Phi_\Theta(S)\subseteq\mathbb{F}_q^{n\rho}$. Since Φ_Θ is bijective, $|S_\Theta|=|S|$. The basis expansion alone does not imply that an arbitrary set S_Θ satisfies the sufficient condition of Theorem 4. The predecessor condition must still be verified for the chosen evaluation set. We next show that this condition is preserved when S is a product Hamming ball over $\mathbb{F}_{q^\rho}$.

Recall that the Hamming weight of $\boldsymbol{x}=(x_n,\dots,x_1)\in\mathbb{F}_{q^\rho}^n$ is

$$\mathrm{Hw}(\boldsymbol{x})=\left|\{i\in\{1,\dots,n\}:x_i\neq 0\}\right|.$$

Since $x_i=0$ if and only if $\phi_\Theta(x_i)=\mathbf{0}^\rho$, the Hamming weight of $\boldsymbol{x}$ is equal to the number of nonzero blocks in $\Phi_\Theta(\boldsymbol{x})$. Accordingly, for $\boldsymbol{v}=(\boldsymbol{v}_n,\dots,\boldsymbol{v}_1)\in(\mathbb{F}_q^\rho)^n$, define its basis-block support and basis-block Hamming weight by

$$\mathrm{Sup}_{\mathrm{blk}}(\boldsymbol{v})=\{i\in\{1,\dots,n\}:\boldsymbol{v}_i\neq\mathbf{0}^\rho\}$$

and

$$\mathrm{Hw}_{\mathrm{blk}}(\boldsymbol{v})=\left|\mathrm{Sup}_{\mathrm{blk}}(\boldsymbol{v})\right|.$$

We call

$$B_{\Theta}^{\mathrm{blk}}(n,w)=\left\{\boldsymbol{v}\in(\mathbb{F}_q^{\rho})^n:\mathrm{Hw}_{\mathrm{blk}}(\boldsymbol{v})\le w\right\}$$

the basis-block Hamming ball of block length n, block size ρ, and radius w. With this notation, $\Phi_{\Theta}\big(B(n,w)\big)=B_{\Theta}^{\mathrm{blk}}(n,w)$.

More generally, let

$$B(\boldsymbol{n},\boldsymbol{w})=B(n_s,w_s)\times\cdots\times B(n_1,w_1)\subseteq\mathbb{F}_{q^{\rho}}^{n}.$$

Its image under Φ_{Θ} is

$$\Phi_{\Theta}\big(B(\boldsymbol{n},\boldsymbol{w})\big)=B_{\Theta}^{\mathrm{blk}}(n_s,w_s)\times\cdots\times B_{\Theta}^{\mathrm{blk}}(n_1,w_1).$$

Since Φ_{Θ} is bijective, the cardinality of the evaluation set is preserved. In particular,

$$|B(\boldsymbol{n},\boldsymbol{w})|=\prod_{j=1}^{s}\left(\sum_{\ell=0}^{w_j}\binom{n_j}{\ell}(q^{\rho}-1)^{\ell}\right).$$

Indeed, in the j-th component, choosing ℓ nonzero coordinates can be done in $\binom{n_j}{\ell}$ ways, and each such coordinate can take any of the $q^{\rho}-1$ nonzero values of $\mathbb{F}_{q^{\rho}}$. Thus, the image of a product Hamming ball over $\mathbb{F}_{q^{\rho}}$ is generally not an ordinary product Hamming ball in $\mathbb{F}_q^{n\rho}$. An element $x_i\neq 0$ may have several nonzero basis coordinates, but all these coordinates belong to the same block $\boldsymbol{v}_i$. Hence, the relevant weight counts nonzero blocks rather than individual nonzero coordinates.

We impose on $\mathbb{F}_{q^{\rho}}^{n}$ the order induced by the lexicographic order of its basis-coordinate representation, namely,

$$\boldsymbol{x}<_{\Theta,\mathrm{lex}}\boldsymbol{y}\quad\Longleftrightarrow\quad\Phi_{\Theta}(\boldsymbol{x})<_{\mathrm{lex}}\Phi_{\Theta}(\boldsymbol{y}).$$

The corresponding evaluation rule is applied to the $n\rho$ scalar coordinates over $\mathbb{F}_q$. Let $\boldsymbol{v}=\Phi_{\Theta}(\boldsymbol{x})$ be nonzero, and suppose that $u_{i,t}=\zeta>0$ is its least-index nonzero scalar coordinate. The predecessor of $\boldsymbol{v}$ is obtained by replacing ζ with $\zeta-1$, that is, $\boldsymbol{v}_{(i,t)}=\boldsymbol{v}-\boldsymbol{e}_{i,t}$, where $\boldsymbol{e}_{i,t}$ denotes the vector whose (i,t)-th scalar coordinate is 1 and whose remaining coordinates are zero. In terms of the original extension-field input, this operation replaces x_i with $x_i-\theta_t$. Therefore, the evaluation step is taken in the basis direction θ_t; it is not necessarily subtraction by the field element 1 unless $\theta_t=1$.

We now verify that the basis-block Hamming weight cannot increase under this predecessor operation. Every block $\boldsymbol{v}_j$ with $j\neq i$ remains unchanged, so $\boldsymbol{v}'_j=\boldsymbol{v}_j$ for every $j\neq i$. Only the block $\boldsymbol{v}_i$ containing the modified scalar coordinate can change. There are two possibilities. If $\zeta>1$, or if another scalar basis coordinate of $\boldsymbol{v}_i$ is nonzero, then $\boldsymbol{v}_i-\boldsymbol{e}_{i,t}\neq\boldsymbol{0}^{\rho}$. In this case, the block remains nonzero, and its contribution to the basis-block Hamming weight remains exactly one. On the other hand, if $\boldsymbol{v}_i=\boldsymbol{e}_{i,t}$ (meaning its only nonzero coordinate was exactly 1), then decreasing it to 0 gives $\boldsymbol{v}_i-\boldsymbol{e}_{i,t}=\boldsymbol{0}^{\rho}$. In this case, the block becomes identically zero, and the total number of nonzero blocks decreases by one. Consequently, the number of nonzero blocks either remains unchanged or decreases, so

$$\mathrm{Hw}_{\mathrm{blk}}\big(\boldsymbol{v}_{(i,t)}\big)\le\mathrm{Hw}_{\mathrm{blk}}(\boldsymbol{v}).$$

The same argument applies separately to every component of a product basis-block Hamming ball. If the block $\boldsymbol{v}_i$ belongs to the component corresponding to $B_{\Theta}^{\mathrm{blk}}(n_j,w_j)$, then the number of nonzero blocks in that component either remains unchanged or decreases by one, while the other components are unaffected. Therefore, $\boldsymbol{v}\in B_{\Theta}^{\mathrm{blk}}(n_s,w_s)\times\cdots\times B_{\Theta}^{\mathrm{blk}}(n_1,w_1)$ implies $\boldsymbol{v}_{(i,t)}\in B_{\Theta}^{\mathrm{blk}}(n_s,w_s)\times\cdots\times B_{\Theta}^{\mathrm{blk}}(n_1,w_1)$. Hence, every noninitial input has a predecessor in the same evaluation set. We present the above explanation in Fig. 4.

The preceding argument shows that the product basis-block Hamming ball is downward closed with respect to the componentwise partial order on the $n\rho$ scalar basis coordinates. Therefore, from Lemma 2, every input set associated with a higher-order basis-direction derivative is also downward

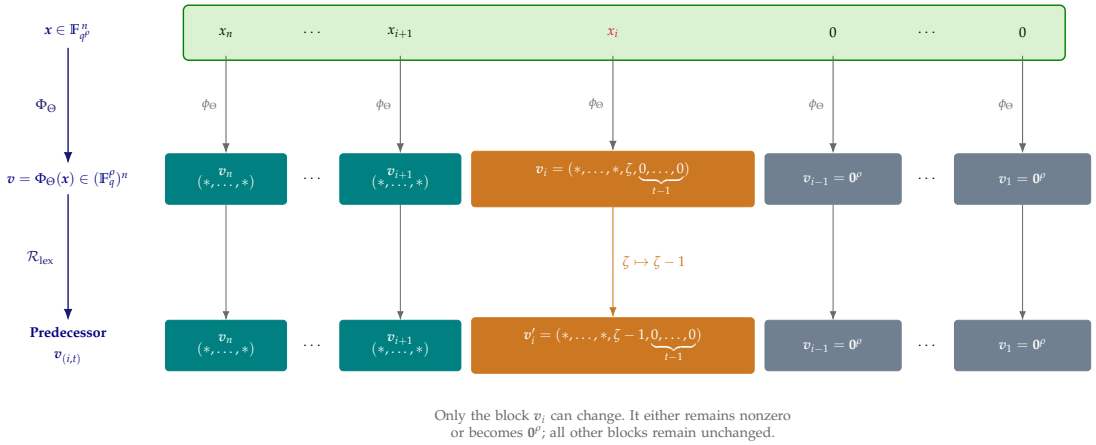

Fig. 4: Basis-coordinate interpretation of the predecessor operation over $\mathbb{F}_{q^\rho}$. The predecessor acts on one least-index nonzero scalar coordinate, while admissibility is determined by the number of nonzero blocks.

closed and every noninitial input has its predecessor in the same set under $\mathcal{R}_{\text{lex}}$. Consequently, the sufficient condition of Theorem 4 is satisfied after basis expansion.

The main modification concerns the enumeration procedure, i.e., Algorithm 4. Over $\mathbb{F}_q$, selecting one nonzero coordinate contributes exactly one position to the support and increases the corresponding Hamming-weight counter by one; in particular, the support size changes as $h \leftarrow h + 1$. Over $\mathbb{F}_{q^\rho}$, however, a single nonzero coordinate x_i may have several nonzero basis coordinates. Hence, `ENU` must enforce the Hamming-weight restriction at the level of the original n extension-field coordinates, while maintaining the support-value representation required by `EVA` at the level of the $n\rho$ scalar coordinates over $\mathbb{F}_q$.

More precisely, suppose that a nonzero value

$$x_i = \sum_{t=1}^{\rho} x_{i,t}\theta_t$$

is selected and that exactly the basis coordinates $x_{i,t_1}, x_{i,t_2}, \dots, x_{i,t_v}$ are nonzero. Then the Hamming-weight counter of the component containing x_i increases only once, because x_i represents a single nonzero extension-field coordinate. In contrast, its basis representation contributes v nonzero scalar coordinates to $\Phi_\Theta(\boldsymbol{x})$. Thus, if introducing x_i increases the extension-field Hamming weight by one, the scalar support size may increase by more than one, such as by the number v of nonzero basis coordinates of x_i.

Accordingly, these v nonzero scalar positions and their corresponding values are inserted into the right-aligned arrays K and V, the scalar support size supplied to `EVA` and the scalar coordinate sum are updated as

$$h \leftarrow h + v, \qquad T \leftarrow T + \sum_{j=1}^{v} x_{i,t_j},$$

respectively.

Thus, although the enumeration is governed by the n extension-field coordinates and their block Hamming-weight restrictions, the support information passed to `EVA` is maintained over the $n\rho$ scalar basis coordinates. The nonzero values of each x_i are traversed according to the lexicographic order of their basis representations, so that the resulting traversal agrees with the order induced by Φ_Θ.

Apart from this block-to-scalar conversion of the support information, Algorithm 3 applies without structural change.

7.4 Complexity and Memory Analysis

We now determine how the complexity bounds derived over the prime field $\mathbb{F}_q$ change after the basis expansion. Two effects must be taken into account. First, each input in $\mathbb{F}_{q^\rho}^n$ is represented by $n\rho$ scalar coordinates over $\mathbb{F}_q$. Second, the polynomial f is represented by the ρ coordinate polynomials $f_1, \ldots, f_\rho$ over $\mathbb{F}_q$, each involving $n\rho$ variables and having total degree at most d. On the other hand, the basis map Φ_Θ is bijective, so the number of evaluation inputs is unchanged, i.e., $|\Phi_\Theta(S)| = |S|$.

Evaluation phase. For a fixed scalar input $\Phi_\Theta(\boldsymbol{x})$, Algorithm 3 still performs at most d derivative-update steps for each coordinate polynomial. Therefore, evaluating all ρ coordinate polynomials at one input requires

$$\mathcal{O}(\rho d)$$

operations over $\mathbb{F}_q$.

The block-aware enumeration introduces an additional distinction between the original extension-field support and the scalar support used by `EVA`. When a nonzero extension-field coordinate is selected, its basis representation may contribute as many as ρ nonzero scalar positions to K and V. Exposing these positions, updating the scalar support size, and maintaining T therefore require at most $\mathcal{O}(\rho)$ work per generated input. For $d \geq 1$, this is absorbed by the $\mathcal{O}(\rho d)$ cost of evaluating the ρ coordinate polynomials.

As in the prime-field case, initializing the support and enumeration data over the $n\rho$ scalar coordinates requires $\mathcal{O}(n\rho)$ one-time work. Consequently, the complete enumeration-and-evaluation phase requires

$$O\left(n\rho + \rho d|S|\right) \tag{32}$$

operations over $\mathbb{F}_q$.

Time complexity. After basis expansion, each coordinate polynomial f_ℓ is a polynomial in $n\rho$ variables over $\mathbb{F}_q$. Although the derivative-index structure is common to all ρ coordinate polynomials, their coefficients are different, so the initialization arithmetic must be performed for each of them. The auxiliary combinatorial tables, however, are constructed only once.

Replacing n by $n\rho$ in Equations (18a) and (19a), the initialization costs for one coordinate polynomial are

$$\mathcal{O}\left(n\rho\, r\, M_{n\rho,d,q}\right) \qquad \text{and} \qquad \mathcal{O}\left(rM_{n\rho,d,q}^2 + r^2\right),$$

for Method 1 and Method 2, respectively. Hence, initializing all ρ coordinate polynomials, together with the common $\mathcal{O}(r^2)$ preprocessing required by the respective initialization schemes, requires

$$\mathcal{O}\left(n\rho^2 r M_{n\rho,d,q} + r^2\right) \qquad \text{and} \qquad \mathcal{O}\left(\rho r M_{n\rho,d,q}^2 + r^2\right)$$

respectively.

For $d \geq 1$, the $\mathcal{O}(r^2)$ term is absorbed by the principal initialization term. Combining these bounds with Equation (32), and noting that the initialization cost also absorbs the one-time $\mathcal{O}(n\rho)$ enumeration setup, the overall running times are

$$\begin{aligned} &\text{for Method 1: } \mathcal{O}\left(n\rho^2 r M_{n\rho,d,q} + \rho d|S|\right) \\ &\text{for Method 2: } \mathcal{O}\left(\rho r M_{n\rho,d,q}^2 + \rho d|S|\right). \end{aligned} \tag{33}$$

Memory complexity. After basis expansion, the reusable derivative arrays are indexed with respect to the $n\rho$ scalar variables. For each derivative array, the corresponding values of $f_1, \ldots, f_\rho$ require ρ elements of $\mathbb{F}_q$, and hence $\rho \log q = \log(q^\rho)$ bits per stored position.

Let $M_{n\rho,d,q}$ denote the number of admissible derivative multi-indices in the $n\rho$ scalar variables. Using the updated prime-field memory bounds, Method 1 and Method 2 require

$$\mathcal{O}\left(\left(\rho M_{n\rho,d,q} + r^2\right)\log q\right) \qquad \text{and} \qquad \mathcal{O}\left(\left((d+\rho)M_{n\rho,d,q} + r^2\right)\log q\right),$$

respectively, for the initialization and reusable derivative storage. Here, the initialization workspace can be reused across the ρ coordinate polynomials, whereas their derivative values must all be retained for the subsequent evaluation phase.

The block-aware enumeration maintains the support-value representation over at most $n\rho$ scalar coordinates. Since a support index K requires $\mathcal{O}(\log(n\rho))$ bits and a scalar value T requires $\mathcal{O}(\log(n\rho q))$ bits, therefore an additional $O\left(n\rho \log(n\rho q)\right)$ bits is required. Consequently, the overall memory complexities are

$$\begin{aligned} &\text{for Method 1: } \mathcal{O}\left(\left(\rho M_{n\rho,d,q} + r^2\right)\log q + n\rho\log(n\rho q)\right), \\ &\text{for Method 2: } \mathcal{O}\left(\left((d+\rho)M_{n\rho,d,q} + r^2\right)\log q + n\rho\log(n\rho q)\right) \end{aligned} \tag{34}$$

As in the prime-field case, the derivative arrays are updated in place and reused throughout the traversal. Hence, neither memory bound contains a term proportional to $|S|$. Setting $\rho = 1$ reduces Equations (33) and (34) to the corresponding prime-field bounds.

These bounds assume that the reduced coordinate polynomials $f_1, \ldots, f_\rho$ are already available with respect to the chosen basis Θ. If f is initially given in $\mathbb{F}_{q^\rho}[x_n, \ldots, x_1]$, the one-time cost of constructing and reducing these coordinate polynomials must be added separately; this cost depends on the representation and monomial support of the input polynomial and is not included above.

8 Conclusion

In this work, we developed a derivative-based framework for evaluating multivariate polynomials over finite fields on product Hamming balls. By exploiting the predecessor structure induced by ascending lexicographic order, polynomial and derivative values are reused and updated in place. In particular, we showed that the framework proposed by Liu et al. over the Boolean field can be extended first to arbitrary prime fields $\mathbb{F}_q$ and subsequently to general finite fields.

The framework consists of an initialization phase followed by an evaluation phase. The initialization phase involves computing the derivative values at the origin required by the evaluation phase, which then reuses them to evaluate the polynomial at each admissible input. We presented two initialization strategies: Method 1 transforms the coefficient representation directly into derivative values at the origin, while Method 2 recursively constructs the admissible derivative polynomials. With $r = \min\{d, q-1\}$ and $M \le \min\left\{\binom{n+d}{d}, q^n\right\}$, their respective overall time complexities (including evaluation) are $\mathcal{O}\left(nrM + d|B(\boldsymbol{n},\boldsymbol{w})|\right)$ and $\mathcal{O}\left(rM^2 + d|B(\boldsymbol{n},\boldsymbol{w})|\right)$. For the evaluation phase itself, Algorithm `ENU` enumerates the Hamming ball using a right-aligned support representation that permits $\mathcal{O}(1)$ backtracking, while Algorithm `EVA` computes at most $\mathcal{O}(d)$ derivative updates per input. The memory required by the framework is primarily dedicated to storing the derivative values that appear during evaluation. A central feature of the framework is that the reusable derivative arrays are overwritten throughout the traversal rather than storing values separately for each evaluated input. Consequently, excluding storage of the output values, the memory requirement is independent of the cardinality of the evaluation set.

Compared with the full-space FEA bound, Method 1 yields a no-larger initialization-time bound, with a strict asymptotic improvement in several regimes, while both proposed variants provide

smaller memory upper bounds in the fixed-field regime analyzed above. Furthermore, we naturally extended the framework to arbitrary finite fields $\mathbb{F}_{q^\rho}$ via basis expansion. By mapping extension-field inputs to $n\rho$ scalar coordinates, admissibility is governed by a basis-block Hamming weight. We prove that the predecessor operation preserves the necessary downward closure; after the block-aware modification of `ENU` described above, the prime-field evaluation procedure can be applied to the scalar basis coordinates.

9 Future Directions

A natural direction for future work is to determine whether the evaluation improvement on at-most-regular domains can be translated into an improved end-to-end complexity for the RMQ algorithms of Joux and Mora [21]. This requires integrating the proposed evaluator with their interpolation and reconstruction procedures rather than considering the evaluation step in isolation. It would also be useful to extend Algorithm `ENU` to the additional global-weight restrictions appearing in some intermediate RMQ domains and to domains requiring exactly one nonzero coordinate per block, corresponding to regular rather than at-most-regular vectors.

A second direction is to consider evaluation domains defined by a coordinate-sum constraint rather than by the number of nonzero coordinates. For a prime field $\mathbb{F}_q$, represented by $0, 1, \ldots, q-1$, define the Manhattan weight of $\boldsymbol{x} = (x_n, \ldots, x_1)$ by

$$\mathrm{wt}_{\mathrm{M}}(\boldsymbol{x}) = \sum_{i=1}^{n} x_i,$$

where the sum is taken over the integers rather than in $\mathbb{F}_q$. The corresponding Manhattan ball is

$$B_{\mathrm{M}}(n, w) = \left\{ \boldsymbol{x} \in \mathbb{F}_q^n : \mathrm{wt}_{\mathrm{M}}(\boldsymbol{x}) \le w \right\}.$$

Such coordinate-sum weights have been studied in coding theory [1]. Moreover, the predecessor operation used in the present framework decreases one nonzero coordinate by one and therefore also decreases the Manhattan weight by one. Hence, these sets satisfy the same basic downward-closure property that motivates our evaluation method. Developing an output-sensitive enumeration procedure for Manhattan balls, and subsequently for products of such balls, is therefore a natural direction for future work.

Author contributions V.D. wrote the main manuscript. All authors reviewed the manuscript.

Data availability No datasets were generated or analysed during the current study.

Declarations The authors declare no competing interests.

A Detailed derivation of the reusable-array memory bound

We provide here the algebraic details underlying the simplification of the reusable-array memory expression and the bound used in the main text.
Recall that

$$R_{\mathrm{total}} = (n+1) + \sum_{i=1}^{d-1} \left(2m_i^{(n)} + \sum_{k=1}^{n-1} m_i^{(k)} \right) + m_d^{(n)}.$$

For $k \ge 1$, we defined

$$M_{k,D,q} = \sum_{i=0}^{D} m_i^{(k)}.$$

Since there is exactly one multi-index of total order zero, namely the zero multi-index, we have $m_0^{(k)} = 1$. Therefore,

$$\sum_{i=1}^{d-1} m_i^{(k)} = M_{k,d-1,q} - 1.$$

Applying this identity first to the terms involving n variables and then to the remaining lower-dimensional terms gives

$$R_{\text{total}} = 2M_{n,d-1,q} + \sum_{k=1}^{n-1} M_{k,d-1,q} + m_d^{(n)}. \tag{35}$$

Since $M_{n,d,q} = M_{n,d-1,q} + m_d^{(n)}$ and $M = M_{n,d,q}$, the upper bound of R_{total} using Equation (35) is given as

$$R_{\text{total}} \leq 2M + \sum_{k=1}^{n-1} M_{k,d-1,q}. \tag{36}$$

For each $1 \leq k < n$, consider an admissible multi-index $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_k) \in \mathcal{I}_{k,d-1,q}$. Associate with it the n-dimensional multi-index

$$\psi_k(\boldsymbol{\alpha}) = (\alpha_1, \ldots, \alpha_k, 1, 0, \ldots, 0).$$

Consequently,

$$\boldsymbol{\alpha} \in \mathcal{I}_{k,d-1,q} \implies |\boldsymbol{\alpha}| \leq d-1 \implies |\psi_k(\boldsymbol{\alpha})| = |\boldsymbol{\alpha}| + 1 \leq d.$$

Moreover, all coordinates of $\psi_k(\boldsymbol{\alpha})$ lie in $\{0, \ldots, q-1\}$, since $q \geq 2$. Hence, $\psi_k(\boldsymbol{\alpha}) \in \mathcal{I}_{n,d,q}$. Therefore, ψ_k form an injective map from $I_{k,d-1,q}$ to $I_{n,d,q}$.
Since the $(k+1)$-th coordinate of $\psi_k(\boldsymbol{\alpha})$ for any $\boldsymbol{\alpha} \in \mathcal{I}_{k,d-1,q}$. is 1, the image corresponding to distinct values of k are disjoint. That is, if $1 \leq k_1 < k_2 \leq n$ holds then $\psi_{k_1}(\mathcal{I}_{k_1,d-1,q}) \cap \psi_{k_2}(\mathcal{I}_{k_2,d-1,q}) = \Phi$. Therefore, the sets

$$\psi_k(\mathcal{I}_{k,d-1,q}), \qquad 1 \leq k < n,$$

form pairwise disjoint subsets of $\mathcal{I}_{n,d,q}$. It follows that

$$\sum_{k=1}^{n-1} |\mathcal{I}_{k,d-1,q}| \leq |\mathcal{I}_{n,d,q}|,$$

and hence $\sum_{k=1}^{n-1} M_{k,d-1,q} \leq M$. Therefore, Equation (36) yields

$$R_{\text{total}} \leq 3M.$$

Since the lower bound of R_{total} is M, therefore, $R_{\text{total}} = \Theta(M)$.

B Examples of the Proposed Evaluation Framework

We illustrate the proposed framework through two concrete examples. The first considers evaluation over the complete space $\mathbb{F}_3^2$, which is the Hamming ball $B(2,2)$, while the second considers evaluation over a proper product Hamming ball contained in $\mathbb{F}_3^3$. In each example, `ENU` traverses the admissible inputs in ascending lexicographic order and constructs the corresponding quantities T, K, V, and the current support size required by `EVA`. Algorithm 3 then uses this state to update the reusable derivative arrays and obtain the polynomial value. The examples therefore illustrate both the enumeration mechanism and the subsequent memory-reuse evaluation.

Evaluation over the Complete Space: Consider the quadratic polynomial

$$f(x_2, x_1) = x_1^2 + 2x_1x_2 + x_2 + 1$$

over $\mathbb{F}_3$. Since

$$B(2,2) = \mathbb{F}_3^2,$$

this example corresponds to evaluation over a single Hamming ball with no restriction on the Hamming weight of the inputs.
During the initialization phase, the following derivative values are computed at the origin:

$$A_0[1] = 1, \qquad A_{0,1}[0] = 1, \qquad A_{0,2}[0] = 1,$$

and

$$A_{0,1,1}[0] = 2, \qquad A_{0,1,2}[0] = 2, \qquad A_{0,2,2}[0] = 0.$$

Table 3 presents the complete interaction between `ENU` and `EVA`. To make their interface explicit, the table displays the complete preallocated arrays $K[1:n]$ and $V[1:n]$ at every generated input, together with the values of T and b. If a new support position is written at $b = n - h$, where h is the support size before the current extension, then only the right-aligned suffixes

$$K[b:n] \qquad \text{and} \qquad V[b:n]$$

represent the current input and are passed to `EVA`. In particular,

$$K[b:n] = (k_1, \ldots, k_{h_{\mathrm{EVA}}}), \qquad V[b:n] = (x_{k_1}, \ldots, x_{k_{h_{\mathrm{EVA}}}}),$$

where the support indices satisfy $k_1 < \cdots < k_{h_{\mathrm{EVA}}}$ and

$$h_{\mathrm{EVA}} = n - b + 1 = h + 1.$$

Entries preceding position b are inactive and need not be restored during backtracking; hence, they may retain values written during previously processed recursive branches. In the table, the active entries are shown in bold, while '$-$' denotes an entry that has not yet been initialized. The quantity T is the sum of the canonical representatives of the current nonzero coordinates and is therefore not reduced modulo 3. The subsequent columns show the memory updates performed by `EVA` and the resulting polynomial value.

Table 3: Enumeration and evaluation of f over the complete space $\mathbb{F}_3^2 = B(2,2)$

	`Enu`: complete state and active suffix passed to `Eva`						`Eva`	
(x_2, x_1)	T	b	$K[1:2]$	$V[1:2]$	$(K[b:2], V[b:2])$	h_{EVA}	Memory update	$f(x_2, x_1)$
$(0,0)$	0	–	–	–	–	0	Handled directly in `Main`	$A_0[1] = 1$
$(0,1)$	1	2	$(-, \mathbf{1})$	$(-, \mathbf{1})$	$((1),(1))$	1	$A_0[2] = A_0[1] + A_{0,1}[0] = 1+1 = 2$	$A_0[2] = 2$
$(0,2)$	2	2	$(-, \mathbf{1})$	$(-, \mathbf{2})$	$((1),(2))$	1	$A_{0,1}[1] = A_{0,1}[0] + A_{0,1,1}[0] = 1+2 = 0$ $A_0[2] = A_0[2] + A_{0,1}[1] = 2+0 = 2$	$A_0[2] = 2$
$(1,0)$	1	2	$(-, \mathbf{2})$	$(-, \mathbf{1})$	$((2),(1))$	1	$A_0[3] = A_0[1] + A_{0,2}[0] = 1+1 = 2$	$A_0[3] = 2$
$(1,1)$	2	1	$(\mathbf{1}, \mathbf{2})$	$(\mathbf{1}, \mathbf{1})$	$((1,2),(1,1))$	2	$A_{0,1}[2] = A_{0,1}[0] + A_{0,1,2}[0] = 1+2 = 0$ $A_0[2] = A_0[3] + A_{0,1}[2] = 2+0 = 2$	$A_0[2] = 2$
$(1,2)$	3	1	$(\mathbf{1}, \mathbf{2})$	$(\mathbf{2}, \mathbf{1})$	$((1,2),(2,1))$	2	$A_{0,1}[1] = A_{0,1}[2] + A_{0,1,1}[0] = 0+2 = 2$ $A_0[2] = A_0[2] + A_{0,1}[1] = 2+2 = 1$	$A_0[2] = 1$
$(2,0)$	2	2	$(1, \mathbf{2})$	$(2, \mathbf{2})$	$((2),(2))$	1	$A_{0,2}[1] = A_{0,2}[0] + A_{0,2,2}[0] = 1+0 = 1$ $A_0[3] = A_0[3] + A_{0,2}[1] = 2+1 = 0$	$A_0[3] = 0$
$(2,1)$	3	1	$(\mathbf{1}, \mathbf{2})$	$(\mathbf{1}, \mathbf{2})$	$((1,2),(1,2))$	2	$A_{0,1}[2] = A_{0,1}[2] + A_{0,1,2}[0] = 0+2 = 2$ $A_0[2] = A_0[3] + A_{0,1}[2] = 0+2 = 2$	$A_0[2] = 2$
$(2,2)$	4	1	$(\mathbf{1}, \mathbf{2})$	$(\mathbf{2}, \mathbf{2})$	$((1,2),(2,2))$	2	$A_{0,1}[1] = A_{0,1}[2] + A_{0,1,1}[0] = 2+2 = 1$ $A_0[2] = A_0[2] + A_{0,1}[1] = 2+1 = 0$	$A_0[2] = 0$

Evaluation over a Product Hamming Ball: Consider the quadratic polynomial

$$g(x_3, x_2, x_1) = 2x_3^2 + x_3x_1 + 2x_1x_2 + x_1^2 + x_2 + 2$$

over $\mathbb{F}_3$, and let the evaluation set be

$$B(\boldsymbol{n}, \boldsymbol{w}) = B(2,1) \times B(1,1) \subseteq \mathbb{F}_3^3,$$

where

$$\boldsymbol{n} = (2,1) \qquad \text{and} \qquad \boldsymbol{w} = (1,1).$$

Here, the factor $B(2,1)$ corresponds to the coordinates (x_3, x_2), while $B(1,1)$ corresponds to the coordinate x_1. Thus, at most one of x_3 and x_2 can be nonzero, whereas x_1 may take any value in $\mathbb{F}_3$. Consequently,

$$B(2,1) \times B(1,1) = \left\{ (x_3, x_2, x_1) \in \mathbb{F}_3^3 : \mathrm{Hw}(x_3, x_2) \le 1 \right\}.$$

During the initialization phase, the following values are computed:

$$A_0[1] = 2, \quad A_{0,1}[0] = 1, \quad A_{0,2}[0] = 1, \quad A_{0,3}[0] = 2,$$
$$A_{0,1,1}[0] = 2, \quad A_{0,1,2}[0] = 2, \quad A_{0,1,3}[0] = 1,$$
$$A_{0,2,2}[0] = 0, \quad A_{0,2,3}[0] = 0, \quad A_{0,3,3}[0] = 1.$$

Table 4 illustrates the same `ENU`–`EVA` interaction over the proper product Hamming ball $B(2,1) \times B(1,1)$. The notation and array conventions are the same as in Table 3. Here, however, the Hamming-weight restriction of the factor $B(2,1)$ prevents x_3 and x_2 from being simultaneously nonzero. Consequently, such inputs are pruned by `ENU` and never passed to `EVA`.

Table 4: Enumeration and evaluation of g over $B(2,1) \times B(1,1) \subseteq \mathbb{F}_3^3$

	`Enu`: complete state and active suffix passed to `Eva`						`Eva`	
(x_3, x_2, x_1)	T	b	$K[1:3]$	$V[1:3]$	$(K[b:3], V[b:3])$	h_{EVA}	Memory update	$g(x_3, x_2, x_1)$
$(0,0,0)$	0	–	–	–	–	0	Handled directly in `Main`	$A_0[1] = 2$
$(0,0,1)$	1	3	$(-,-,\mathbf{1})$	$(-,-,\mathbf{1})$	$((1),(1))$	1	$A_0[2] = A_0[1] + A_{0,1}[0] = 2+1 = 0$	$A_0[2] = 0$
$(0,0,2)$	2	3	$(-,-,\mathbf{1})$	$(-,-,\mathbf{2})$	$((1),(2))$	1	$A_{0,1}[1] = A_{0,1}[0] + A_{0,1,1}[0] = 1+2 = 0$ $A_0[2] = A_0[2] + A_{0,1}[1] = 0+0 = 0$	$A_0[2] = 0$
$(0,1,0)$	1	3	$(-,-,\mathbf{2})$	$(-,-,\mathbf{1})$	$((2),(1))$	1	$A_0[3] = A_0[1] + A_{0,2}[0] = 2+1 = 0$	$A_0[3] = 0$
$(0,1,1)$	2	2	$(-,\mathbf{1},\mathbf{2})$	$(-,\mathbf{1},\mathbf{1})$	$((1,2),(1,1))$	2	$A_{0,1}[2] = A_{0,1}[0] + A_{0,1,2}[0] = 1+2 = 0$ $A_0[2] = A_0[3] + A_{0,1}[2] = 0+0 = 0$	$A_0[2] = 0$
$(0,1,2)$	3	2	$(-,\mathbf{1},\mathbf{2})$	$(-,\mathbf{2},\mathbf{1})$	$((1,2),(2,1))$	2	$A_{0,1}[1] = A_{0,1}[2] + A_{0,1,1}[0] = 0+2 = 2$ $A_0[2] = A_0[2] + A_{0,1}[1] = 0+2 = 2$	$A_0[2] = 2$
$(0,2,0)$	2	3	$(-,1,\mathbf{2})$	$(-,2,\mathbf{2})$	$((2),(2))$	1	$A_{0,2}[1] = A_{0,2}[0] + A_{0,2,2}[0] = 1+0 = 1$ $A_0[3] = A_0[3] + A_{0,2}[1] = 0+1 = 1$	$A_0[3] = 1$
$(0,2,1)$	3	2	$(-,\mathbf{1},\mathbf{2})$	$(-,\mathbf{1},\mathbf{2})$	$((1,2),(1,2))$	2	$A_{0,1}[2] = A_{0,1}[2] + A_{0,1,2}[0] = 0+2 = 2$ $A_0[2] = A_0[3] + A_{0,1}[2] = 1+2 = 0$	$A_0[2] = 0$
$(0,2,2)$	4	2	$(-,\mathbf{1},\mathbf{2})$	$(-,\mathbf{2},\mathbf{2})$	$((1,2),(2,2))$	2	$A_{0,1}[1] = A_{0,1}[2] + A_{0,1,1}[0] = 2+2 = 1$ $A_0[2] = A_0[2] + A_{0,1}[1] = 0+1 = 1$	$A_0[2] = 1$
$(1,0,0)$	1	3	$(-,1,\mathbf{3})$	$(-,2,\mathbf{1})$	$((3),(1))$	1	$A_0[4] = A_0[1] + A_{0,3}[0] = 2+2 = 1$	$A_0[4] = 1$
$(1,0,1)$	2	2	$(-,\mathbf{1},\mathbf{3})$	$(-,\mathbf{1},\mathbf{1})$	$((1,3),(1,1))$	2	$A_{0,1}[3] = A_{0,1}[0] + A_{0,1,3}[0] = 1+1 = 2$ $A_0[2] = A_0[4] + A_{0,1}[3] = 1+2 = 0$	$A_0[2] = 0$
$(1,0,2)$	3	2	$(-,\mathbf{1},\mathbf{3})$	$(-,\mathbf{2},\mathbf{1})$	$((1,3),(2,1))$	2	$A_{0,1}[1] = A_{0,1}[3] + A_{0,1,1}[0] = 2+2 = 1$ $A_0[2] = A_0[2] + A_{0,1}[1] = 0+1 = 1$	$A_0[2] = 1$
$(2,0,0)$	2	3	$(-,1,\mathbf{3})$	$(-,2,\mathbf{2})$	$((3),(2))$	1	$A_{0,3}[1] = A_{0,3}[0] + A_{0,3,3}[0] = 2+1 = 0$ $A_0[4] = A_0[4] + A_{0,3}[1] = 1+0 = 1$	$A_0[4] = 1$
$(2,0,1)$	3	2	$(-,\mathbf{1},\mathbf{3})$	$(-,\mathbf{1},\mathbf{2})$	$((1,3),(1,2))$	2	$A_{0,1}[3] = A_{0,1}[3] + A_{0,1,3}[0] = 2+1 = 0$ $A_0[2] = A_0[4] + A_{0,1}[3] = 1+0 = 1$	$A_0[2] = 1$
$(2,0,2)$	4	2	$(-,\mathbf{1},\mathbf{3})$	$(-,\mathbf{2},\mathbf{2})$	$((1,3),(2,2))$	2	$A_{0,1}[1] = A_{0,1}[3] + A_{0,1,1}[0] = 0+2 = 2$ $A_0[2] = A_0[2] + A_{0,1}[1] = 1+2 = 0$	$A_0[2] = 0$

References

1. Alderson, T.L., Morine, B.: Mws and fws codes for coordinate-wise weight functions. Applicable Algebra in Engineering, Communication and Computing **36**, 367–385 (2025). DOI 10.1007/s00200-023-00608-4
2. Benadjila, R., Feneuil, T., Rivain, M.: MQ on my Mind: Post-quantum signatures from the non-structured multivariate quadratic problem. In: 2024 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 468–485. IEEE (2024). DOI 10.1109/EuroSP60621.2024.00032
3. Berlekamp, E.R., McEliece, R.J., van Tilborg, H.C.A.: On the inherent intractability of certain coding problems (corresp.). IEEE Trans. Inf. Theory **24**(3), 384–386 (1978). DOI 10.1109/TIT.1978.1055873. URL https://doi.org/10.1109/TIT.1978.1055873

4. Beullens, W.: Improved cryptanalysis of UOV and rainbow. IACR Cryptol. ePrint Arch. p. 1343 (2020). URL https://eprint.iacr.org/2020/1343
5. Beullens, W.: MAYO: Practical post-quantum signatures from oil-and-vinegar maps. In: Selected Areas in Cryptography—SAC 2021, *Lecture Notes in Computer Science*, vol. 13203, pp. 355–376. Springer (2022). DOI 10.1007/978-3-030-99277-4_17
6. Bhargava, V., Ghosh, S., Guo, Z., Kumar, M., Umans, C.: Fast multivariate multipoint evaluation over all finite fields. In: 2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS), pp. 221–232 (2022). DOI 10.1109/FOCS54457.2022.00028
7. Bhargava, V., Ghosh, S., Kumar, M., Mohapatra, C.K.: Fast, algebraic multivariate multipoint evaluation in small characteristic and applications. In: Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2022, p. 403–415. Association for Computing Machinery, New York, NY, USA (2022). DOI 10.1145/3519935.3519968. URL https://doi.org/10.1145/3519935.3519968
8. Billet, O., Gilbert, H.: Cryptanalysis of rainbow. In: R.D. Prisco, M. Yung (eds.) Security and Cryptography for Networks, 5th International Conference, SCN 2006, Maiori, Italy, September 6-8, 2006, Proceedings, *Lecture Notes in Computer Science*, vol. 4116, pp. 336–347. Springer (2006). DOI 10.1007/11832072_23. URL https://doi.org/10.1007/11832072_23
9. Bouillaguet, C.: Algorithm 1052: Evaluating a boolean polynomial on all possible inputs. ACM Trans. Math. Softw. **50**(4), 28:1–28:37 (2024)
10. Bouillaguet, C., Chen, H.C., Cheng, C.M., Chou, T., Niederhagen, R., Shamir, A., Yang, B.Y.: Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In: S. Mangard, F.X. Standaert (eds.) Cryptographic Hardware and Embedded Systems, CHES 2010, pp. 203–218. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
11. Debris-Alazard, T., Sendrier, N., Tillich, J.: Wave: A new code-based signature scheme. CoRR **abs/1810.07554** (2018). URL http://arxiv.org/abs/1810.07554
12. Dinur, I.: Cryptanalytic applications of the polynomial method for solving multivariate equation systems over GF(2). In: EUROCRYPT (1), *Lecture Notes in Computer Science*, vol. 12696, pp. 374–403. Springer (2021)
13. Dinur, I., Shamir, A.: Cube Attacks on Tweakable Black Box Polynomials. In: EUROCRYPT, *Lecture Notes in Computer Science*, vol. 5479, pp. 278–299. Springer (2009)
14. Dinur, I., Shamir, A.: An Improved Algebraic Attack on Hamsi-256. In: FSE, *Lecture Notes in Computer Science*, vol. 6733, pp. 88–106. Springer (2011)
15. Fischer, N., Kallmayer, M., Wennmann, L.: A simple algorithm for trimmed multipoint evaluation. In: A. Benoit, H. Kaplan, S. Wild, G. Herman (eds.) 33rd Annual European Symposium on Algorithms, ESA 2025, Warsaw, Poland, September 15-17, 2025, *LIPIcs*, vol. 351, pp. 89:1–89:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025). DOI 10.4230/LIPICS.ESA.2025.89. URL https://doi.org/10.4230/LIPIcs.ESA.2025.89
16. Furue, H., Ikematsu, Y., Kiyomura, Y., Takagi, T.: A new variant of unbalanced oil and vinegar using quotient ring: QR-UOV. In: Advances in Cryptology—ASIACRYPT 2021, *Lecture Notes in Computer Science*, vol. 13093, pp. 187–217. Springer (2021). DOI 10.1007/978-3-030-92068-5_7
17. Furue, H., Kudo, M.: Polynomial XL: A variant of the XL algorithm using macaulay matrices over polynomial rings. CoRR **abs/2112.05023** (2021). URL https://arxiv.org/abs/2112.05023
18. Furue, H., Takagi, T.: Fast enumeration algorithm for multivariate polynomials over general finite fields. In: T. Johansson, D. Smith-Tone (eds.) Post-Quantum Cryptography, pp. 357–378. Springer Nature Switzerland, Cham (2023)
19. Ghosh, S., Harsha, P., Herdade, S., Kumar, M., Saptharishi, R.: Fast numerical multivariate multipoint evaluation. In: 2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS), pp. 1426–1439 (2023). DOI 10.1109/FOCS57990.2023.00088
20. van der Hoeven, J., Lecerf, G.: Faster multi-point evaluation over any field. Applicable Algebra in Engineering, Communication and Computing pp. 1–32 (2025)
21. Joux, A., Mora, R.: The regular multivariate quadratic problem. Designs, Codes and Cryptography **93**, 5179–5229 (2025). DOI 10.1007/s10623-025-01717-6
22. Joux, A., Vitse, V.: A crossbred algorithm for solving boolean polynomial systems. In: J. Kaczorowski, J. Pieprzyk, J. Pomykala (eds.) Number-Theoretic Methods in Cryptology - First International Conference, NuTMiC 2017, Warsaw, Poland, September 11-13, 2017, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 10737, pp. 3–21. Springer (2017). DOI 10.1007/978-3-319-76620-1_1. URL https://doi.org/10.1007/978-3-319-76620-1_1
23. Kedlaya, K.S., Umans, C.: Fast modular composition in any characteristic. In: 2008 49th Annual IEEE Symposium on Foundations of Computer Science, pp. 146–155 (2008)
24. Kipnis, A., Patarin, J., Goubin, L.: Unbalanced oil and vinegar signature schemes. In: Advances in Cryptology—EUROCRYPT '99, *Lecture Notes in Computer Science*, vol. 1592, pp. 206–222. Springer (1999). DOI 10.1007/3-540-48910-X_15
25. Lidl, R., Niederreiter, H.: Finite Fields, 2 edn. Encyclopedia of Mathematics and its Applications. Cambridge University Press (1996)
26. Liu, F., Dixit, V., Yamamoto, D., Ogata, W., Sarkar, S., Meier, W.: Efficient polynomial evaluation over structured space and application to polynomial method. Cryptology ePrint Archive, Paper 2026/040 (2026). URL https://eprint.iacr.org/2026/040

27. Salagean, A., Winter, R., Mandache-Salagean, M., Phan, R.C.: Higher order differentiation over finite fields with applications to generalising the cube attack. Des. Codes Cryptogr. **84**(3), 425–449 (2017). DOI 10.1007/S10623-016-0277-5. URL https://doi.org/10.1007/s10623-016-0277-5
28. Umans, C.: Fast polynomial factorization and modular composition in small characteristic. In: Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing, STOC '08, p. 481–490. Association for Computing Machinery, New York, NY, USA (2008). DOI 10.1145/1374376.1374445. URL https://doi.org/10.1145/1374376.1374445
29. Van Der Hoeven, J., Schost, É.: Multi-point evaluation in higher dimensions. Applicable Algebra in Engineering, Communication and Computing **24**(1), 37–52 (2013)