Polynomial Evaluation on Many Inputs with Bounded Hamming Weight over GF(2)

Fukang Liu[1], Vaibhav Dixit[2], Muzhou Li[3], Daisuke Yamamoto[1], Wakaha Ogata[1], Santanu Sarkar[2], Willi Meier[4]

[1] Institute of Science Tokyo, Tokyo, Japan
{liu.f.ad,ogata.w.aa}@m.titech.ac.jp
[2] IIT Madras, India
vaibhavrdrk@gmail.com, sarkar.santanu.bir1@gmail.com
[3] Shandong University, Jinan, China
muzhouli@mail.sdu.edu.cn
[4] FHNW, Windisch, Switzerland
willimeier48@gmail.com

Abstract. We propose a new polynomial evaluation algorithm to evaluate a degree-d Boolean polynomial $f(x_n, \ldots, x_1)$ on all elements in some structured sets $S \subseteq \mathbb{F}_2^n$. This problem has been well-studied for $S = \mathbb{F}_2^n$ and there are efficient polynomial evaluation algorithms like standard Möbius transform, memory-efficient Möbius transform (EUROCRYPT 2021, TOMS 2024) and fast exhaustive search (CHES 2010, PQCrypt 2023) for the case $S = \mathbb{F}_2^n$. In addition, the standard Möbius transform can also be used to evaluate a polynomial over the set $S = P_n^w$ of all $\boldsymbol{x} = (x_n, \ldots, x_1) \in \mathbb{F}_2^n$ whose Hamming weight is upper bounded by w, and its memory complexity is the same as the size of S. In Dinur's algorithm for polynomial method proposed at EUROCRYPT 2021, there is a critical step to efficiently evaluate f over a more general input set $P_{n-n_1}^w \times P_{n_1}^{n_1} \subseteq \mathbb{F}_2^n$. To our knowledge, in the literature, no memory-efficient polynomial evaluation algorithms are designed for such a special input set. Dinur proposed a tweaked fast exhaustive search algorithm for this problem, but it has not been proved nor implemented. This leaves us with a natural question: can we design an efficient polynomial evaluation algorithm tailored for a more general input set $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1} = P_{n_s}^{w_s} \times \cdots \times P_{n_1}^{w_1} \subseteq \mathbb{F}_2^n$ where $\sum_{i=1}^{s} n_i = n$ and $w_i \leq n_i$? We answer this question by proposing a new polynomial evaluation algorithm named **FESG**, i.e., **F**ast **E**xhaustive **S**earch over a more **G**eneral input set, to handle the input set $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$. This algorithm is based on the derivative-based polynomial evaluation framework proposed at CHES 2010. In addition to extending the application of derivative-based framework to a more general input set, we also successfully address a major issue in existing derivative-based algorithms over $\mathbb{F}_2$ proposed at CHES 2010 and PQCrypt 2023, reducing the time complexity of the initialization phase from $\mathcal{O}\left(\binom{n}{\leq d}^2\right)$ to $\mathcal{O}\left(\binom{n}{\leq d}\right)$ where $\binom{n}{\leq d} = \sum_{i=0}^{d} \binom{n}{i}$. As a result, FESG is also efficient for large d. Our algorithm is almost optimal, whose theoretic time and memory complexity are $\binom{n}{\leq d} + d \cdot |P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}|$ bit operations and $2 \cdot \binom{n}{\leq d}$ bits, respectively. Here, $|P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}|$ is the size

of the set $P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$. An efficient implementation of the FESG algorithm for any $(d, n_s, \ldots, n_1, w_s, \ldots, w_1)$ is also given in this work. In particular, the FESG algorithm also provides a proven method for a critical step in Dinur's algorithm for polynomial method without affecting its overall complexity.

Keywords: Boolean function, polynomial evaluation, Möbius transform, polynomial method

1 Introduction

We study the problem to efficiently evaluate a Boolean polynomial $f(x_n, \ldots, x_1)$ of degree d over a given input set S. It is well-known that when $S = \mathbb{F}_2^n$, a standard Möbius transform can solve this problem with time complexity of $n \cdot 2^n$ bit operations and memory complexity of 2^n bits. A slightly modified variant by Dinur and Shamir can achieve time complexity of $d \cdot 2^n$ bit operations and memory complexity of still 2^n bits [7].

To address the issue of the huge memory consumption in standard Möbius transform, 5 different memory-efficient algorithms have been developed in the literature, and they can be classified into two types. The first type was first proposed by Bouillaguet et al. at CHES 2010 [4], which is called the fast exhaustive search (FES) algorithm. It uses a Gray code to enumerate all inputs in $\mathbb{F}_2^n$ and exploits the derivatives of f to achieve efficient polynomial evaluations. We call this type the derivative-based polynomial evaluations. This FES algorithm takes $\binom{n}{\leq d}^2 + d \cdot 2^n$ bit operations and $\binom{n}{\leq d}$ bits of memory. The term $\binom{n}{\leq d}^2$ is the cost of the initialization phase which computes all derivatives of f up to the d-th order. Indeed, the lower bound of the initialization phase is $\sum_{i=0}^{d} \binom{n}{i} \cdot \binom{n-i}{d-i}$ since there are $\binom{n}{i}$ different polynomials of degree $d-i$ and in $n-i$ variables for $0 \leq i \leq d$ to compute. The authors of [4] use $\binom{n}{\leq d}^2$ as a crude upper bound, which has been widely used in the literature. Building on the FES framework, Furue and Takagi proposed at PQCrypt 2023 a different order to enumerate all inputs in $\mathbb{F}_2^n$, and also achieved time complexity of $\binom{n}{\leq d}^2 + d \cdot 2^n$ bit operations [8], though the memory complexity is increased to $\log n \cdot n \cdot \binom{n}{\leq d}$ bits. Moreover, Furue-Takagi's technique also works on a more general vector space $\mathbb{F}_p^n$ where p is a prime number. Both derivative-based polynomial evaluation algorithms [4, 8] have a costly initialization phase, i.e., in order to compute $\binom{n}{\leq d}$ different derivatives of f with time complexity upper bounded by $\binom{n}{\leq d}^2$.

For large d, both derivative-based polynomial evaluation algorithms [4, 8] become inefficient due to the costly initialization phase, even though $\binom{n}{\leq d}^2$ is a crude upper bound. This has motivated Dinur to develop a memory-efficient Möbius transform at EUROCRYPT 2021 [6] where he proposed his algorithm for the polynomial method [12] (we call it Dinur's polynomial method for simplicity). Specifically, the last step in Dinur's polynomial method is to perform polynomial evaluations for a polynomial of a relatively high degree $d \approx n/3$ over $\mathbb{F}_2^n$. The

idea of the memory-efficient Möbius transform is to fix some variables in f first, and then update the polynomial accordingly to obtain an updated polynomial in fewer variables, and finally perform standard Möbius transform on this updated polynomial. Although the cost to update the polynomial for all possible values of the to-be-fixed variables is non-negligible, by properly choosing the number of variables to fix first, a time complexity below $n \cdot 2^n$ can be reached and the memory complexity is about $2 \cdot \binom{n}{\leq d}$ bits. Bouillaguet further explored this method at TOMS [3], and achieved the time complexity of $2d \cdot 2^n$ bit operations and memory complexity of $\binom{n}{\leq d}$ bits, where the additional cost $d \cdot 2^n$ is consumed by updating the polynomial.

Motivation. First, there is a critical step in Dinur's polynomial method [6], which is to perform polynomial evaluations over the set

$$\{(x_n, \ldots, x_{n_1+1}) \in \mathbb{F}_2^{n-n_1} \mid \mathrm{wt}(x_n, \ldots, x_{n_1+1}) \leq w\} \times \{0,1\}^{n_1}$$

rather than all inputs in $\mathbb{F}_2^n$, where $\mathrm{wt}(x_n, \ldots, x_1)$ is the Hamming weight of $(x_n, \ldots, x_1) \in \mathbb{F}_2^n$ and $0 \leq w \leq n - n_1$. Dinur claimed that a slightly tweaked version of the FES algorithm [4] with Gray code can solve this step with time complexity $d \cdot \binom{n-n_1}{\leq w} \cdot 2^{n_1}$ as long as n is much smaller than 2^{n_1}, i.e., $n << 2^{n_1}$ holds in Dinur's polynomial method. The details can be found on Page 9 of [6]. It is also claimed that it is nontrivial to achieve this time complexity for any n_1. On the other hand, the correctness of this slightly modified FES algorithm was not rigorously proved, and details of how this algorithm actually runs were not given. We thus started this research in order to address these issues. This is important since Dinur's polynomial method to solve Boolean equations has a significant advantage over known solving techniques and has been applied to cryptanalysis of several popular ciphers including SHA-3 [6], LowMC [1, 6, 11] and Rain [10].

After a preliminary study on Dinur's memory-efficient Möbius transform for the full input set $\mathbb{F}_2^n$, we find that it becomes less efficient when the input set is a subset of $\mathbb{F}_2^n$. Therefore, we are motivated to study the derivative-based framework. However, derivative-based polynomial evaluation algorithms [4, 8] have a strong condition on the order of the inputs to evaluate and they are originally designed for the full input set $\mathbb{F}_2^n$. Whether it is possible to extend them to a subset of $\mathbb{F}_2^n$ is unclear since it is equivalent to asking whether such algorithms can still work if some inputs are skipped. Dinur also asked whether a monotone Gray code can be combined with the FES algorithm [4] to achieve an efficient polynomial evaluation algorithm for the input set $\{(x_n, \ldots, x_1) \in \mathbb{F}_2^n \mid \mathrm{wt}(x_n, \ldots, x_1) \leq w\}$ in [6]. Moreover, these algorithms [4, 8] require a costly and complicated initialization phase to compute many derivatives. This is not friendly to a practical efficient implementation for any d. Indeed, the FES algorithm in [4] was only implemented for $d = 2$, and [8] also only gave a naive SAGE implementation without optimization. Therefore, we are also motivated to address this issue caused by the initialization phase.

Our contribution. We dive into the derivative-based framework proposed by Bouillaguet et al. in [4], and rephrase it into a simpler (non-recursive) form. Based on new concepts called evaluation rule and evaluation order derived from this new form, we propose a sufficient condition for this framework.

Therefore, checking whether the application of this framework can be extended to a more general input set $P_{n_s}^{w_s} \times \cdots \times P_{n_1}^{w_1} \subseteq \mathbb{F}_2^n$ where $P_{n_i}^{w_i} = \{\boldsymbol{x} \in \mathbb{F}_2^{n_i} \mid \mathrm{wt}(\boldsymbol{x}) \leq w_i\}$ is reduced to checking whether it is possible to find a proper evaluation rule and evaluation order that can satisfy the sufficient condition. We then prove that the evaluation order and evaluation rule in Furue-Takagi's algorithm [8] can be used for this purpose. Despite the usage of the same evaluation order and rule, our algorithm reduces the memory usage by a factor of about $n\log n$, and replace the costly phase to compute many derivatives with an efficient phase to simply read all coefficients of the polynomial. Thanks to this improvement, the derivative-based polynomial evaluation algorithm can be comparable with Möbius transform (or its variants) even for large degree.

We call the new polynomial evaluation algorithm **FESG**, i.e., **F**ast **E**xhaustive **S**earch over a more **G**eneral input set, to handle the input set $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$. The time complexity and memory complexity of FESG are $\binom{n}{\leq d} + d \cdot \Pi_{i=1}^{s} \binom{n_i}{\leq w_i}$ bit operations and $2 \cdot \binom{n}{\leq d}$ bits, respectively. If the input set is $\mathbb{F}_2^n$, we get the time complexity $\binom{n}{\leq d} + d \cdot 2^n$ and memory complexity $2 \cdot \binom{n}{\leq d}$. Directly applying our algorithm to Dinur's polynomial method, each step is now solved with a proven method and the overall complexity is not affected. Thus, existing algebraic attacks based on the polynomial method remain valid and need not be revised.

We also implement our algorithm in Python. It can run for any d and $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$ as long as there is no memory overflow. It turns out that the code is very simple, since both the initialization phase and evaluation phase of our FESG algorithm are very simple. The code is available at https://github.com/LFKOKAMI/polynomialEvaluation.

Organization. In Section 2, we introduce the notation and existing polynomial evaluation algorithms. In Section 3, we present our new algorithm, and analyze its complexity. In Section 4, it is detailed how to implement our algorithm. In Section 5, we rephrase the FES framework and give a sufficient condition that possibly allows to extend it to a more general input set. In Section 6, we prove the correctness of our algorithm.

2 Preliminaries

2.1 Notation

The following notation will be used throughout this paper. The size of a set S is denoted by $|S|$, and $S = \{S[0], \ldots, S[|S|-1]\}$, i.e., $S[i]$ is the i-th element in S. $\mathbf{0}^n$ represents the all-zero vector $(0, \ldots, 0) \in \mathbb{F}_2^n$. The sum $\sum_{i=0}^{d} \binom{n}{i}$ is denoted by $\binom{n}{\leq d}$. The set of integers $\{i, i+1, \ldots, j\}$ is denoted by $[i, j]$. The

bold character (e.g. $\boldsymbol{x}, \boldsymbol{y}$) is used to represent a Boolean vector. In particular, we set $\boldsymbol{x} = (x_n, \ldots, x_1) \in \mathbb{F}_2^n$ (here the index is from 1). In addition, we set $\boldsymbol{e}_i = (0, \ldots, 0, 1, \underbrace{0, \ldots, 0}_{i-1}) \in \mathbb{F}_2^n$ and $\overline{\boldsymbol{e}_i} = (1, \ldots, 1, 0, \underbrace{1, \ldots, 1}_{i-1}) \in \mathbb{F}_2^n$ where $1 \leq i \leq n$. The Hamming weight of the Boolean vector $\boldsymbol{x}$ is denoted by $\mathrm{wt}(\boldsymbol{x})$ and it is defined as the number of nonzero elements in $\boldsymbol{x}$, i.e.,

$$\mathrm{wt}(\boldsymbol{x}) = \#\{i \mid x_i = 1 \text{ for } i \in [1, n]\}.$$

We also define the function $INT(\boldsymbol{x})$ as $INT(\boldsymbol{x}) = \sum_{i=1}^{n} 2^{i-1} x_i$, i.e., $\boldsymbol{x}$ is the n-bit binary representation of the integer $INT(\boldsymbol{x})$.

In this paper, we focus on Boolean polynomials. As is well-known, for a Boolean polynomial $f(x_n, \ldots, x_1)$, its algebraic normal form (ANF) can be expressed as

$$f(x_n, \ldots, x_1) = \sum_{(b_n, \ldots, b_1) \in \mathbb{F}_2^n} \mu(\boldsymbol{b}) \prod_{i=1}^{n} x_i^{b_i} \tag{1}$$

where $\boldsymbol{b} = (b_n, \ldots, b_1) \in \mathbb{F}_2^n$ and $\mu(\boldsymbol{b}) \in \mathbb{F}_2$ denotes the coefficient of the monomial $x_1^{b_1} x_2^{b_2} \ldots x_n^{b_n}$. If the polynomial is of degree d, we have $\mu(\boldsymbol{b}) = 0$ for $\mathrm{wt}(\boldsymbol{b}) > d$. Throughout this paper, the polynomial $f(x_n, \ldots, x_1)$ will be written as either f or $f(\boldsymbol{x})$, which should be clear from the contexts.

In addition, we need to understand the lexicographic order (we simply call it lex order) for binary strings $\boldsymbol{x} \in \mathbb{F}_2^n$. Given $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{F}_2^n$, we write

$$\boldsymbol{x} <_{\mathrm{lex}} \boldsymbol{x}' \text{ iff } INT(\boldsymbol{x}) < INT(\boldsymbol{x}').$$

In other words, $\boldsymbol{x}$ is smaller than $\boldsymbol{x}'$ with respect to the lex order.

2.2 Memory-Efficient Möbius Transform

As standard (also called in-place) Möbius transform is well-known, we refer readers to Appendix A for more details. Specifically, we describe at Appendix A how standard Möbius transform is used to evaluate a Boolean polynomial $f(x_n, \ldots, x_1)$ of degree d over the set P_n^w of all $\boldsymbol{x} \in \mathbb{F}_2^n$ with Hamming weight of at most w by P_n^w, i.e.,

$$P_n^w = \{\boldsymbol{x} \in \mathbb{F}_2^n \mid \mathrm{wt}(\boldsymbol{x}) \leq w\}. \tag{2}$$

The theoretic time and memory complexity are $d \cdot \binom{n}{\leq w}$ bit operations and $\binom{n}{\leq w}$ bits, respectively.

Indeed, it is also well-known that the in-place Möbius transform is an efficient implementation of the recursive Möbius transform, which evaluates $f(\boldsymbol{x})$ based on the following recursive relation:

$$f(x_n, \ldots, x_1) = x_1 \cdot f_1(x_n, \ldots, x_2) \oplus f_0(x_n, \ldots, x_2),$$

where f_0 and f_1 are 2 polynomials in $n-1$ variables and of degree d and $d-1$, respectively. Hence, computing $f(\boldsymbol{x})$ is reduced to computing f_0 and f_1, which are polynomials in 1 fewer variables. f_0 and f_1 are again evaluated based on the recursive relation. Note that once $f_0(x_n, \ldots, x_2)$ and $f_1(x_n, \ldots, x_2)$ are known, $f(x_n, \ldots, x_2, 0)$ and $f(x_n, \ldots, x_2, 1)$ can be computed as follows:

$$\begin{aligned} f(x_n, \ldots, x_2, 0) &= f_0(x_n, \ldots, x_2), \\ f(x_n, \ldots, x_2, 1) &= f_1(x_n, \ldots, x_2) \oplus f_0(x_n, \ldots, x_2) \\ &= f_1(x_n, \ldots, x_2) \oplus f(x_n, \ldots, x_2, 0). \end{aligned}$$

Dinur observed that the above recursive Möbius transform can be implemented in a memory-efficient style at EUROCRYPT 2021 [6] and we call it memory-efficient Möbius transform. Its original algorithmic description is for evaluating a degree-d polynomial f over the full space $P_n^n = \mathbb{F}_2^n$. Specifically, in the memory-efficient variant, $f(x_n, \ldots, x_2, 0)$ and $f(x_n, \ldots, x_2, 1)$ are not simultaneously computed, which otherwise requires to store $f_0(x_n, \ldots, x_2)$ and $f_1(x_n, \ldots, x_2)$ in the stack. Instead, it first computes $f(x_n, \ldots, x_2, 0)$, which is reduced to computing $f_0(x_n, \ldots, x_2)$. Then, it computes $f(x_n, \ldots, x_2, 1) = f(x_n, \ldots, x_2, 0) \oplus f_1(x_n, \ldots, x_2)$, which is reduced to computing $f_1(x_n, \ldots, x_2)$. When the polynomial to compute is only in d variables (i.e., at depth $n-d$), switch to the in-place Möbius transform to evaluate this polynomial in d variables on the input set P_d^d.

We can follow a similar idea to apply the memory-efficient Möbius transform to a subset $P_n^w \subseteq \mathbb{F}_2^n$. Specifically, the Hamming weight of $(x_{n-d}, \ldots, x_1)$ denoted by w' has to satisfy $w' \leq \min(w, n-d)$. For any such fixed value of $(x_{n-d}, \ldots, x_1)$ of Hamming weight w', use the in-place Möbius transform to evaluate the polynomial in d variables on the input set $P_d^{w-w'}$. There is an equivalent and simpler description of the memory-efficient Möbius transform:

Step 1: Enumerate $\boldsymbol{y} = (y_{n-d}, \ldots, y_1) \in P_{n-d}^{\min(w,n-d)} \subseteq \mathbb{F}_2^{n-d}$ in lex order. For each $\boldsymbol{y}$, compute $n-d$ polynomials sequentially:

$$f(x_n, \ldots, x_2, y_1), \ldots, f(x_n, \ldots, x_{n-d+1}, y_{n-d}, \ldots, y_1)$$

Step 2: Evaluate $f(x_n, \ldots, x_{n-d+1}, y_{n-d}, \ldots, y_1)$ on the input set $P_d^{w-\mathrm{wt}(\boldsymbol{y})}$.

The case $w = n$ [6]. For the case $w = n$, the cost to compute the $n-d$ polynomials for 2^{n-d} inputs are estimated as

$$T_{\text{update}} = \binom{n}{\leq d} + 2 \cdot \binom{n-1}{\leq d} + 2^2 \cdot \binom{n-2}{\leq d} + \cdots + 2^{n-d-1} \cdot \binom{d+1}{\leq d}.$$

The total cost is then

$$T_{\text{total}} = T_{\text{update}} + 2^{n-d} \cdot d \cdot 2^d = T_{\text{update}} + d \cdot 2^n.$$

Since $2^{n-d-1} \cdot \binom{d+1}{\leq d} > 2^{n-d-1} \cdot \binom{d}{\leq d} = 2^{n-1}$, we have $T_{\text{update}} > 2^{n-1}$. Dinur showed that T_{total} can be safely estimated as $\mathcal{O}(d \cdot 2^n)$.

The case $w < n$. In this case, it becomes cumbersome to compute the cost, even for Step 1. Specifically, T_{update} becomes

$$T'_{\text{update}} = \binom{n}{\leq d} + \alpha_1 \cdot \binom{n-1}{\leq d} + \alpha_2 \cdot \binom{n-2}{\leq d} + \cdots + \alpha_{n-d-1} \cdot \binom{d+1}{\leq d}.$$

where

$$\alpha_j = 2^j \text{ for } j \leq w, \ \alpha_j = \binom{j}{\leq w} \text{ for } j > w.$$

Hence, we have

$$T'_{\text{update}} = \binom{n}{\leq d} + \sum_{j=1}^{w} 2^j \cdot \binom{n-j}{\leq d} + \sum_{j=w+1}^{n-d-1} \binom{j}{\leq w} \cdot \binom{n-j}{\leq d}.$$

The cost of Step 2 depends on the Hamming weight of $\boldsymbol{y} \in \mathbb{F}_2^{n-d}$ and we have

$$T'_{\text{total}} = T'_{\text{update}} + d \cdot \sum_{j=0}^{\min(w,n-d)} \binom{n-d}{j} \cdot \binom{d}{\leq \beta_j},$$

where

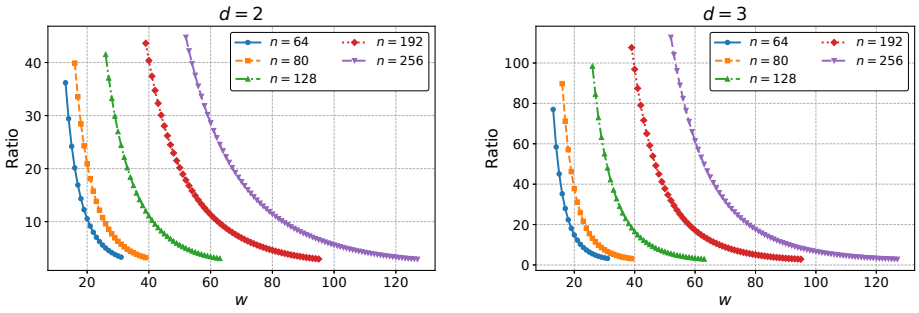
$$\beta_j = d \text{ if } w - j \geq d; \ \beta_j = w - j \text{ if } w - j < d.$$

According to our experiments on various parameters (n, d, w), Ratio $= \frac{T'_{\text{total}}}{d \cdot \binom{n}{\leq w}}$ grows much faster as w decreases under the same (n, d). The experimental data for $d \in \{2, 3, 4, 5\}$, $n \in \{64, 80, 128, 192, 256\}$ and $\lfloor \frac{n}{5} \rfloor w \leq \lfloor \frac{n}{2} \rfloor$ are shown in Figure 1.

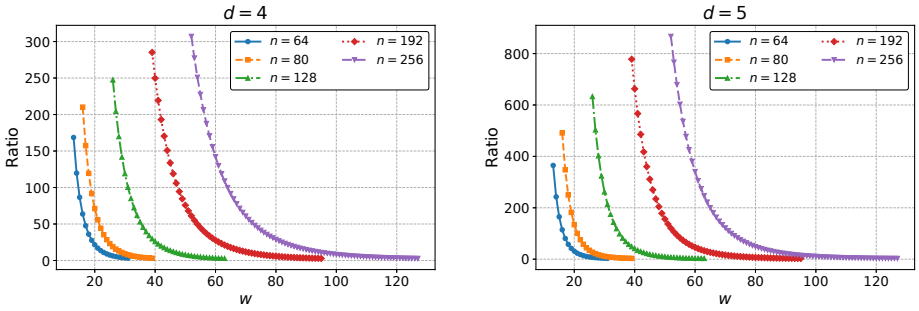
Another variant of memory-efficient Möbius transform for P_n^n [6]. In the above, it is necessary to keep the $n - d$ polynomials in stack and hence the memory complexity is in $\mathcal{O}(n \cdot \binom{n}{\leq d})$. To further optimize the memory complexity by increasing the time complexity, Dinur also proposed another variant of memory-efficient Möbius transform. The idea is to fix $k \approx n - \log\binom{n}{\leq d}$ variables in f, and independently recursively compute the polynomial $f(x_n, \ldots, x_1)$ on all possible $(x_k, \ldots, x_1)$. Finally, for each such polynomial $f(x_n, \ldots, x_{k+1}, x_k, \ldots, x_1)$ with a fixed $(x_k, \ldots, x_1)$, perform the in-place Möbius transform to evaluate it on all inputs in P_{n-k}^{n-k}. The time complexity is upper bounded by $2^k \cdot \sum_{j=0}^{k} \binom{n-j}{\leq d} + (n-k) \cdot 2^n < n \cdot 2^n$. The memory complexity is upper bounded by $3 \cdot \binom{n}{\leq d}$ bits [6]. If we apply it to P_n^w, there is a similar problem to compute its time complexity. In addition, in terms of time complexity, this cannot beat the first variant of memory-efficient Möbius transform described above for P_n^n. This indicates that the first variant is more efficient in terms of time consumption.

2.3 Derivative-based Method for Polynomial Evaluation

At CHES 2010, the fast exhaustive search (FES) algorithm was proposed to evaluate any Boolean polynomial $f(x_n, \ldots, x_1)$ over the full space $\mathbb{F}_2^n$. If the



(a) $d = 2$ (b) $d = 3$



(c) $d = 4$ (d) $d = 5$

Fig. 1: Ratio as a function of w for different values of n, shown for $d = 2, 3, 4, 5$.

degree of f is d, after a preprocessing phase to compute all derivatives of f up to the d-th order, it takes $\mathcal{O}(d \cdot 2^n)$ bit operations to evaluate f over $\mathbb{F}_2^n$ and the memory complexity is $\mathcal{O}(\binom{n}{\leq d})$. To understand this algorithm, we first recall the derivative of f with respect to i different variables $(x_{k_i}, \ldots, x_{k_1})$.

Definition 1. *Let $k[:i]$ be a set of indices $\{0, k_1, \ldots, k_i\}$ where $\{k_1, \ldots, k_i\} \subseteq \{1, \ldots, n\}$ and $k[0] = 0, k[j] = k_j$ for $1 \leq j \leq i$. The i-th order derivative of a Boolean polynomial $f(x_n, \ldots, x_1)$ with respect to i distinct variables $x_{k_1}, \ldots, x_{k_i}$ is defined as*

$$D_{k[:i]} f = \sum_{(x_{k_i}, \ldots, x_{k_1}) \in \mathbb{F}_2^i} f(x_n, \ldots, x_1).$$

In particular, we define $D_0 f = f$[5]*.*

Based on the derivative of f, the authors of [4] recursively used the following relation to fast evaluate f over all inputs in $\mathbb{F}_2^n$:

$$f(\boldsymbol{x}) = f(\boldsymbol{x} \oplus \boldsymbol{e}_{k_1}) \oplus D_{k[:1]} f(\boldsymbol{x}).$$

[5]Throughout this paper, when $k[:i]$ is clear and it is at the index, we usually omit $\{\}$. For example, $D_0 f = D_{\{0\}} f$, $S_{0,1,3} = S_{\{0,1,3\}}$.

Note that the variable x_{k_1} is missing in $D_{k[:1]}f$ and hence

$$\forall \boldsymbol{x} \in \mathbb{F}_2^n : \ D_{k[:1]}f(\boldsymbol{x}) = D_{k[:1]}f(\boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_1}})$$

In this way, we have

$$f(\boldsymbol{x}) = f(\boldsymbol{x} \oplus \boldsymbol{e}_{k_1}) \oplus D_{k[:1]}f(\boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_1}}). \tag{3}$$

Recursively, there is

$$D_{k[:i]}f(\boldsymbol{x}) = D_{k[:i]}f(\boldsymbol{x} \oplus \boldsymbol{e}_{k_{i+1}}) \oplus D_{k[:i+1]}f(\boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_{i+1}}}). \tag{4}$$

Let recursiveEva$(f, \boldsymbol{x}, k_1)$ be a procedure to evaluate f on the input $\boldsymbol{x}$ using the index k_1. Based on Equation 3 to evaluate $f(\boldsymbol{x})$, it is required that $f(\boldsymbol{x} \oplus \boldsymbol{e}_{k_1})$ has been computed and stored. Then, the problem is reduced to evaluating $D_{k[:1]}f$ on the input $\boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_1}}$ with another index $k_2 \neq k_1$, i.e., the input to $D_{k[:1]}f$ is simply setting the k_1-th bit of $\boldsymbol{x}$ as 0. If $D_{k[:1]}f(\boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_1}})$ is unknown, based on Equation 4, call the recursive procedure recursiveEva$(D_{k[:1]}f, \boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_1}}, k_2)$ until the input polynomial f' of the procedure recursiveEva$(f', \boldsymbol{x}')$ is constant or $f'(\boldsymbol{x}')$ has been computed. The recursion must end after at most d calls since the degree of the input polynomial of the procedure recursiveEva$(f, \boldsymbol{x}, k_1)$ is reduced by 1 after each call.

Ordered input set of $D_{k[:i]}f$. Based on the above description, the input set of f denoted by S_0 is an ordered set traversing all elements in $\mathbb{F}_2^n$, and the input set of $D_{k[:i]}$ denoted by $S_{k[:i]}$ is also an ordered set and $S_{k[:i]} \subseteq \mathbb{F}_2^n$.

The ordering chosen in [4]. When the derivative-based framework was first proposed in [4], the authors proposed to use (binary-reflected) Gray code to traverse $\mathbb{F}_2^n$, which is an ordering of the set $\mathbb{F}_2^n$. Based on Equation 4 to evaluate f on all $\boldsymbol{x} \in \mathbb{F}_2^n$ where $\boldsymbol{x}$ is enumerated in Gray code, the Gray code trivially ensures

$$\mathrm{wt}(S_0[j] \oplus S_0[j-1]) = 1, \ \text{for } 1 \leq j \leq 2^n - 1.$$

In addition, the authors of [4] prove that

$$\forall \{k_1, \ldots, k_i\} \subseteq \{1, \ldots, n\} : \ \mathrm{wt}(S_{k[:i]}[j] \oplus S_{k[:i]}[j-1]) = 1, \ \text{for } 1 \leq j \leq |S_{k[:i]}| - 1.$$

This implies that for any two consecutive inputs of the same polynomial $D_{k[:i]}f$, they only differ in 1 bit. In other words, this does not only hold for the original polynomial f, but also for any derivative of f if the polynomial evaluation is based on Equation 4. Therefore, the corresponding index k_i always exists, which is exactly where the two consecutive input differs. The full details can be referred to [4].

The ordering chosen in [8]. Furue and Takagi also designed a derivative-based polynomial evaluation algorithm at PQCrypt 2023 [8], but using the ascending lex order to traverse all elements in $\mathbb{F}_2^n$. In other words, the input set $S_0 = \mathbb{F}_2^n$ satisfies $INT(S_0[j]) = j$ for $0 \leq j \leq 2^n - 1$. They show that by this order, it is always possible to identify an index k_1 when calling the recursive procedure recursiveEva$(f', \boldsymbol{x}', k_1)$, where k_1 is the smallest index such that $\boldsymbol{x}_{k_1} = 1$. In this way, $S_{k[:i]}$ is also an ordered set which satisfies

$$INT(S_{k[:i]}[j]) = j \times 2^{k_i}, \text{ for } 0 \leq j \leq |S_{k[:i]}| - 1 = 2^{n-k_i} - 1.$$

Disadvantage of derivative-based algorithms [4, 8] Both derivative-based polynomial evaluation algorithms [4,8] have a costly initialization phase, which is to compute all possible derivatives of f up to the d-th order. The time complexity is lower bounded by $\sum_{i=0}^{d} \binom{n}{i} \cdot \binom{n-i}{\leq d-i}$. The authors of [4, 8] used the same crude upper bound $\binom{n}{\leq d}^2$ for the cost of the initialization phase. Dinur also used this crude upper bound to claim the advantage of his memory-efficient Möbius transform [6] over [4].

Challenge to extend FES algorithm to a subset of $\mathbb{F}_2^n$. It is clear that the derivative-based framework used in [4,8] has a strong condition on the ordering in the input set. Indeed, the condition is not only on the ordering in the input set of f, but also on the ordering in any input set of the derivative of f up to the d-th order. Therefore, if the problem is to evaluate f over a subset of $\mathbb{F}_2^n$, whether the existing 2 types of ordering, i.e., Gray code [4] and lex order [8], can still work is unclear. It is equivalent to checking whether such algorithms [4, 8] can still work if some inputs of f are skipped, which requires nontrivial proof.

2.4 Revisiting Dinur's Algorithm for Polynomial Method

To save space, we only give a high-level description of Dinur's algorithm for the polynomial method proposed at EUROCRYPT 2021 [6]. The full details can be referred to the original paper [6]. Before moving to the description, we generalize the definition in Equation 2 of the set of inputs with bounded Hamming weight as follows:

$$P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1} = P_{n_s}^{w_s} \times \cdots P_{n_2}^{w_2} \times P_{n_1}^{w_1} \subseteq \mathbb{F}_2^n, \ n = \sum_{i=1}^{s} n_i, \tag{5}$$
$$\forall i \in [1, s] : P_{n_i}^{w_i} = \{\boldsymbol{x} \in \mathbb{F}_2^{n_i} \mid \mathrm{wt}(\boldsymbol{x}) \leq w_i\}.$$

Then, the general procedure of Dinur's algorithm can be described as follows:

Step 1: Enumerate all solutions of a system of $n_1 + 1$ equations

$$f_1(x_n, \ldots, x_1) = \cdots = f_{n_1+1}(x_n, \ldots, x_1) = 0$$

of degree d over the set P_{n-n_1,n_1}^{w,n_1} where $w = d \cdot (n_1 + 1) - n_1 + 1$.

Step 2: Based on the solutions obtained at Step 1, interpolate n_1+1 polynomials in $n-n_1$ variables whose degree is upper bounded by w.

Step 3: Based on the n_1+1 polynomials obtained at Step 2, evaluate them on all inputs in the full space $P_{n-n_1}^{n-n_1} = \{0,1\}^{n-n_1}$.

Dinur's method to solve Step 1. For Step 1, Dinur observed that the least significant n_1 bits can be traversed using a Gray code. Hence, to enumerate all vectors $\boldsymbol{x} \in P_{n-n_1,n_1}^{w,n_1}$, $(x_{n_1},\ldots,x_1)$ is traversed using Gray code under the same prefix $(x_n,\ldots,x_{n_1+1})$. Once all 2^{n_1} vectors are enumerated, update the prefix $(x_n,\ldots,x_{n_1+1})$ and repeat enumerating $(x_{n_1},\ldots,x_1)$ using Gray code until all elements are enumerated. Therefore, the Gray code enumeration can be tweaked in such a way that in every 2^{n_1} steps, it updates to the next binary vector $\boldsymbol{x}'$ that no longer satisfies the property of the Gray code that any two consecutive enumerated vectors differ in only 1 bit. As $\boldsymbol{x}'$ differs in at most n bits from any previously enumerated vector $\boldsymbol{x} \in \mathbb{F}_2^n$, Dinur claimed that evaluating $f_i(\boldsymbol{x}')$ takes at most n steps of the FES algorithm, and each step of the algorithm takes at most d XOR operations. Hence, computing $f_i(\boldsymbol{x}')$ takes at most $n\cdot d$ bit operations. Such a procedure is executed every 2^{n_1} steps. Hence, if $n << 2^{n_1}$, i.e., $n\cdot d\cdot\binom{n}{\leq w} << 2^{n_1}\cdot d\cdot\binom{n}{\leq w}$, evaluating $f_i(\boldsymbol{x})$ over P_{n-n_1,n_1}^{w,n_1} takes $d\cdot\binom{n-n_1}{\leq w}\cdot 2^{n_1}$ bit operations.

Our interpretation. No additional descriptions or proof were given for the above method in [6]. Here, we give our interpretation of Dinur's method. In every 2^{n_1} steps, it updates to the next binary vector $\boldsymbol{x}'$ that breaks the property of the Gray code. Assume there is an input vector $\boldsymbol{x}$ enumerated just before $\boldsymbol{x}'$. Then, $h = \mathrm{wt}(\boldsymbol{x}' \oplus \boldsymbol{x}) \leq n$. We can set

$$\boldsymbol{x}' = \boldsymbol{x} \oplus \boldsymbol{e}_{i_1} \oplus \cdots \oplus \boldsymbol{e}_{i_h}$$

where $\{i_1,\ldots,i_h\} \subseteq \{1,\ldots,n\}$. In Dinur's method, it is assumed that $f(\boldsymbol{x}\oplus\boldsymbol{e}_{i_1})$ can be computed from $f(\boldsymbol{x})$ and it takes at most d XOR operations according to the derivative-based method [4]. Then, repeat this process and compute $f(\boldsymbol{x} \oplus \sum_{j=1}^{t+1}\boldsymbol{e}_{i_j})$ from $f(\boldsymbol{x} \oplus \sum_{j=1}^{t}\boldsymbol{e}_{i_j})$ until $t+1=h$. Hence, Dinur claimed that it took at most $h\cdot d \leq n\cdot d$ bit operations to compute $f(\boldsymbol{x}')$ from $f(\boldsymbol{x})$. Obviously, in such a process, h inputs to f are evaluated and any two consecutive inputs of f differ in 1 bit. However, as explained above, the fast polynomial evaluation algorithm [4] is not just based on a single relation in f as in Equation 3, but on multiple relations in a series of derivatives of f as shown in Equation 4. In such a tweaked algorithm, it was not proved that any two consecutive inputs for the corresponding derivatives of f also only differ in 1 bit. However, this condition has been rigorously proved in the original FES algorithm [4] if the input set is the full space $\mathbb{F}_2^n$. To our knowledge, this tweaked FES algorithm for the input set P_{n-n_1,n_1}^{w,n_1} has not been implemented either. It is conceivable that such a method cannot work at all.

Complexity analysis of Dinur algorithm for polynomial method. The overall time complexity is dominated by Step 1 and Step 3. Dinur estimated that

the tweaked algorithm for Step 1 to evaluate a Boolean polynomial over the set $P^{w,n_1}_{n-n_1,n_1}$ takes $d \cdot \binom{n-n_1}{\leq w} \cdot 2^{n_1}$ bit operations, and thus he derived

$$T_{\text{step1}} = 2d \cdot \log n \cdot 2^{n_1} \cdot \binom{n-n_1}{\leq w}, \; T_{\text{step3}} = (n_1+1) \cdot (n-n_1) \cdot 2^{n-n_1}.$$

The optimal time complexity is achieved by choosing n_1 that minimizing $T_{\text{step1}} + T_{\text{step3}}$. The memory complexity of Step 1 is $(n_1+1) \cdot \binom{n}{\leq d}$ which is to use the original n_1+1 degree-d polynomials. This is always well below the total memory complexity dominated by Step 3, i.e., $(n_1+1) \cdot 2 \cdot \binom{n-n_1}{\leq w}$.

3 New Memory-efficient Polynomial Evaluation

We design a new memory-efficient polynomial evaluation algorithm tailored to the input subset $P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$ defined in Equation 5. The reason to consider this general input set is that our algorithm can handle it when we discover the algorithm. For $s = 2$, it directly provides a reliable approach to solve Step 1 in Dinur's polynomial method without affecting the overall complexity. The case $s = 1$ can be applied to both polynomial interpolation and evaluation over P^w_n. We leave the application of the case $s > 2$ as an open problem.

3.1 Overview of the Main Idea

Let $f(\boldsymbol{x})$ be a degree-d polynomial in n Boolean variables. The goal is to evaluate f on an input $\boldsymbol{x}$ in an efficient way. Obviously, the input $\boldsymbol{x}$ either satisfies $\text{wt}(\boldsymbol{x}) \leq d$ or $\text{wt}(\boldsymbol{x}) > d$. Let $h = \text{wt}(\boldsymbol{x})$ and $\boldsymbol{x} = \sum_{i=1}^{h} \boldsymbol{e}_{k_i}$. We expand the recursive procedure in Equation 4 to show the main idea of our algorithm.

If $h \leq d$, the recursive procedure to compute $f(\boldsymbol{x})$ is as follows:

$$D_{k[:h-1]} f(\boldsymbol{e}_{k_h}) = \underbrace{D_{k[:h-1]} f(0^n)}_{\text{coefficient of } x_{k_1} \cdots x_{k_{h-1}}} \oplus \underbrace{D_{k[:h]} f(0^n)}_{\text{coefficient of } x_{k_1} \cdots x_{k_h}}$$

$$D_{k[:j]} f(\sum_{i=j+1}^{h} \boldsymbol{e}_{k_i}) = D_{k[:j]} f(\sum_{i=j+2}^{h} \boldsymbol{e}_{k_i}) \oplus D_{k[:j+1]} f(\sum_{i=j+2}^{h} \boldsymbol{e}_{k_i}), \; h-2 \geq j \geq 0.$$

If $h > d$, the recursive procedure to compute $f(\boldsymbol{x})$ is as follows:

$$D_{k[:d-1]} f(\sum_{i=d}^{h} \boldsymbol{e}_{k_i}) = D_{k[:d-1]} f(\sum_{i=d+1}^{h} \boldsymbol{e}_{k_i}) \oplus \underbrace{D_{k[:d]} f(0^n)}_{\text{coefficient of } x_{k_1} \cdots x_{k_d}}$$

$$D_{k[:j]} f(\sum_{i=j+1}^{h} \boldsymbol{e}_{k_i}) = D_{k[:j]} f(\sum_{i=j+2}^{h} \boldsymbol{e}_{k_i}) \oplus D_{k[:j+1]} f(\sum_{i=j+2}^{h} \boldsymbol{e}_{k_i}), \; d-2 \geq j \geq 0.$$

Note that we have not yet specified the order in the indices $k_1, \ldots, k_h$, as this recursion holds for any order in them. In our practical algorithm, as in [8] where the lex order is used to enumerate all $\boldsymbol{x}$ in $\mathbb{F}_2^n$, we choose the order in the indices as $k_1 < k_2 < \cdots < k_h$.

3.2 New Polynomial Evaluation Algorithm

The new algorithm is named **FESG**, i.e., **F**ast **E**xhaustive **S**earch over a **G**eneral input set, since it extends the fast exhaustive search framework [4] to a more general input set. The FESG algorithm consists of an initialization phase and evaluation phase, as described below. A concrete example is given at Appendix B.

Initialization phase. Assign $\binom{n}{\leq d}$ different arrays, which are classified into $d+1$ different types:

- Type-0: The array is denoted by $A_0 = A_{k[:0]}$. The size of $A_{k[:0]}$ is $n+1$.
- Type-i ($1 \leq i \leq d-1$): The array is denoted by $A_{k[:i]}$ where $1 \leq k_1 < \ldots < k_i \leq n$. The size of $A_{k[:i]}$ is $n+1-k_i$.
- Type-d: The array is denoted by $A_{k[:d]}$ where $1 \leq k_1 < \ldots < k_d \leq n$. The size of $A_{k[:d]}$ is 1.

Only initialize the first entry of each of these arrays as follows, which is equivalent to reading the coefficients $\mu(b_n, \ldots, b_1)$ of f, i.e., coefficients of all monomials $x_n^{b_n} \cdots x_1^{b_1}$ of degree at most d: $A_{k[:0]}[0] = \mu(\mathbf{0}^n)$ and $A_{k[:i]}[0] = \mu(\sum_{j=1}^{i} \boldsymbol{e}_{k_j})$ for all non-empty subset $\{k_1, \ldots, k_i\} \subseteq \{1, \ldots, n\}$ of size at most d where $k_1 < \cdots < k_i$.

Evaluation phase. After initialization, start evaluating f on all $x \in P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$. Before describing the evaluation phase, we introduce the following concept.

Definition 2. *The support of a binary vector $(x_n, \ldots, x_1)$ denoted by*

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

is defined as the set of nonzero positions counted from 1. Specifically,

$$\forall i \in \{k_h, \ldots, k_1\} : \ x_i = 1; \ \forall i \in \{n, \ldots, 1\} \setminus \{k_h, \ldots, k_1\} : \ x_i = 0.$$

The procedure of the FESG algorithm can be described as follows:

Step 1: Get $f(\mathbf{0}^n)$ from $A_{k[:0]}[0]$.

Step 2: Enumerate $\boldsymbol{x} \in P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$ in ascending lex order and we start from the second element. For each enumerated input $\boldsymbol{x}$, get

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

and use the following procedure to update the arrays, and finally output

$$f(\boldsymbol{x}) = A_{k[:0]}[k_1] = A_{k[:0]}[k_1 - k_0].$$

(a) If $h \leq d$, update the arrays as follows:

$$A_{k[:h-1]}[k_h - k_{h-1}] = A_{k[:h-1]}[0] \oplus A_{k[:h]}[0],$$

For j from $h-2$ to 0 :

$$A_{k[:j]}[k_{j+1} - k_j] = A_{k[:j]}[k_{j+2} - k_j] \oplus A_{k[:j+1]}[k_{j+2} - k_{j+1}].$$

(b) If $h > d$, update the arrays as follows:

$$A_{k[:d-1]}[k_d - k_{d-1}] = A_{k[:d-1]}[k_{d+1} - k_{d-1}] \oplus A_{k[:d]}[0],$$
For j from $d-2$ to 0 :
$$A_{k[:j]}[k_{j+1} - k_j] = A_{k[:j]}[k_{j+2} - k_j] \oplus A_{k[:j+1]}[k_{j+2} - k_{j+1}].$$

A brief explanation. The procedure to update the arrays for the input $\boldsymbol{x} = \sum_{i=1}^{h} \boldsymbol{e}_{k_i}$ exactly follows the main idea described in Section 3.1. The corresponding relation is as follows:

$$\begin{aligned}&(A_{k[:j]}[0], A_{k[:j]}[k_{j+1} - k_j], A_{k[:j]}[k_{j+2} - k_j])\\ \to\ &(D_{k[:j]}f(0^n), D_{k[:j]}f(\sum_{i=j+1}^{h} \boldsymbol{e}_{k_i}), D_{k[:j]}f(\sum_{i=j+2}^{h} \boldsymbol{e}_{k_i})).\end{aligned}$$

Why this holds will be proved in the last section of this paper.

3.3 Complexity Analysis

We study the memory and time complexity of our algorithm.

Memory complexity. At the initialization phase, there are $\binom{n}{i}$ different Type-i arrays where $0 \le i \le d$. Therefore, there are in total $\binom{n}{\le d}$ arrays. The size of A_0 is $1+n = \binom{n}{0}+\binom{n}{1}$ bits, and the total size of $A_{\{0,1\}}, \ldots, A_{\{0,n\}}$ is $n+\sum_{j=1}^{n}(n-j) = \binom{n}{1} + \binom{n}{2}$ bits. For each $2 \le i < d$, the memory required for all Type-i arrays is

$$\begin{aligned}\sum_{j=i}^{n} \binom{j-1}{i-1}(n+1-j) &= (n+1)\cdot \sum_{j=i}^{n} \binom{j-1}{i-1} - \sum_{j=i}^{n} \binom{j-1}{i-1} j\\ &= (n+1)\binom{n}{i} - i \cdot \sum_{j=i}^{n} \binom{j}{i} = (n+1)\binom{n}{i} - i \cdot \binom{n+1}{i+1}\\ &= \binom{n+1}{i+1} = \binom{n}{i} + \binom{n}{i+1}\end{aligned}$$

bits, since the number of Type-i arrays of size $n+1-j$ is $\binom{j-1}{i-1}$, i.e., there are in total $\binom{j-1}{i-1}$ possible combinations of $k_1, \ldots, k_{i-1}$ satisfying $1 \le k_1 < k_2 < \ldots < k_{i-1} < j = k_i$. The size of all Type-$d$ arrays is simply $\binom{n}{d}$ bits. Hence, the memory complexity (estimated in bits) of our algorithm is

$$\binom{n}{d} + \sum_{i=0}^{d-1} \left(\binom{n}{i} + \binom{n}{i+1} \right) = \binom{n}{\le d} + \binom{n}{\le d} - 1 < 2 \cdot \binom{n}{\le d}.$$

Time complexity. The time is mainly consumed to update the arrays for each input $(x_n, \ldots, x_1)$, which corresponds to at most d xor operations. In addition, we need to initialize $\binom{n}{\leq d}$ entries of the arrays at the initialization phase. Therefore, the number of bit operations of our algorithm is estimated as

$$\binom{n}{\leq d} + d \cdot \Pi_{i=1}^{s} \binom{n_i}{\leq w_i}.$$

When $s = 1, n_1 = w_1 = n$, we get the time complexity $\binom{n}{\leq d} + d \cdot 2^n$.

4 Efficient Implementation of FESG Algorithm

We address the following 3 minor concerns when the FESG algorithm is practically implemented:

1. How to quickly find the array $A_{k[:j]}$ among $\binom{n}{\leq d}$ arrays?
2. How to efficiently enumerate the element $\boldsymbol{x}$ in $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$ in lex order and get the corresponding $(k_h, \ldots, k_1)$ for $\mathrm{wt}(\boldsymbol{x}) = h$?
3. Will the index computation like $k_{j+2} - k_j$ and $k_{j+1} - k_j$ matter?

4.1 Efficient Data Structure to Store $\binom{n}{\leq d}$ Arrays

Definition 3. *For all non-empty subsets $\{k_1, \ldots, k_j\} \subseteq \{1, \ldots, n\}$ of size j, i.e., all j-subsets, the colexicographic (colex) order of these subsets is an ordering that compares elements starting from the largest element rather than the smallest.*

Lemma 1 (Index for j-subsets [3]). *Using the colex order to enumerate all j-subsets of $\{1, \ldots, n\}$, the subset $\{k_1, \ldots, k_j\}$ is the $\left(\sum_{i=1}^{j} \binom{k_i - 1}{i}\right)$-th enumerated subset (counted from 0), where $\binom{i_1}{i_0} = 0$ for $i_1 < i_0$.*

For example, the ascending colex order for all 2-subsets of $\{1, 2, 3, 4\}$ is

$$\{1,2\} <_{\text{colex}} \{1,3\} <_{\text{colex}} \{2,3\} <_{\text{colex}} \{1,4\} <_{\text{colex}} \{2,4\} <_{\text{colex}} \{3,4\}.$$

Then, all subsets of $\{1, \ldots, n\}$ of size at most d can be enumerated in this way: for j from 0 to d, enumerate all j-subsets according to the colex order. Here, the empty subset is represented by $\{0\}$. Based on [3], the non-empty subset $\{k_1, \ldots, k_j\} \subseteq \{1, \ldots, n\}$ is the $I_n(k_1, \ldots, k_j)$-th subset in the sequence (counted from 0), where

$$I_n(0) = 0,\ I_n(k_1, \ldots, k_j) = \underbrace{1 + \sum_{i=1}^{j-1} \binom{n}{i}}_{\#\{0,1,\ldots,j-1\}-subsets} + \sum_{i=1}^{j} \binom{k_i - 1}{i}. \tag{6}$$

For example, $I_6(0) = 0, I_6(1) = 1, I_6(2) = 2, I_6(1,2) = 7$.

Our critical observation is that

$$I_n(k_1, \ldots, k_j) = I_n(k_1, \ldots, k_{j-1}) + \binom{n}{j-1} + \binom{k_j - 1}{j}. \tag{7}$$

Efficient data structure for $\binom{n}{\leq d}$ arrays. Based on the above analysis of the colex order, we can use a dynamic two-dimensional array B to store these arrays. In particular, the array $A_{k[:j]}$ is stored at the $I_n(k_1,\ldots,k_j)$-th row of B, i.e., $B[I_n(k_1,\ldots,k_j)]$. Moreover, each row $B[I_n(k_1,\ldots,k_j)]$ is of size $n+1-k_j$ for $j<d$. If $j=d$, the size of $B[I_n(k_1,\ldots,k_j)]$ is simply 1. The array A_0 is stored at $B[0]$ whose size is $n+1$.

In the FESG algorithm, to evaluate f on each input $\boldsymbol{x}$, it is necessary to access arrays $A_{k[:j]}$ for $0\leq j\leq \min(\mathrm{wt}(\boldsymbol{x}),d)$. Based on Equation 6, we have

$$I_n(k_1)=k_1, B[k_1]=A_{k[:1]}.$$

Based on Equation 7, we have

$$B\left[I_n(k_1,\ldots,k_{j-1})+\binom{n}{j-1}+\binom{k_j-1}{j}\right]=A_{k[:j]} \text{ for } 2\leq j\leq d.$$

Therefore, the indices of these arrays $A_{k[:j]}$ can be computed with at most $2d$ integer additions: first set $I_n(0)=0, I_n(k_1)=k_1$, and then iteratively compute $I_n(k_1,\ldots,k_j)$ with Equation 7 for $2\leq j\leq \min(\mathrm{wt}(\boldsymbol{x}),d)$.

Here, all binomial coefficients $\binom{i}{j}$ for $0\leq i\leq n$ and $0\leq j\leq \min(d,i)$ are computed in time $\mathcal{O}(\binom{n}{\leq d})$ and stored in a two-dimensional array BCOF, where $\mathrm{BCOF}[i][j]=\binom{i}{j}$. The memory complexity is $\log\binom{n}{\leq d}\cdot\binom{n}{\leq d}\leq n\cdot\binom{n}{\leq d}$ in bits. Note that BCOF is constructed once and then can be used forever.

Remark 1. If not using the standard Möbius transform whose memory complexity is $O(2^n)$, it is inevitable to have an efficient method to frequently access the coefficient of the monomial $x_{k_1}\cdots x_{k_j}$ in all other polynomial evaluation algorithms. This is because the polynomial is stored in a compact array of size $\binom{n}{\leq d}$ in order to save space. In addition, in derivative-based methods [4, 8], it is also inevitable to frequently access $\binom{n}{\leq d}$ derivatives which are identified by $(k_1,\ldots,k_j)$. Hence, for fair comparison with other memory-efficient polynomial evaluation algorithms [3, 4, 6, 8], the cost to construct BCOF in practical implementations is not taken into account. We also want to emphasize that, if d is small, e.g., $d=2$ or $d=3$, we can avoid BCOF by directly using a 3- or 4-dimensional dynamic array to store all these $\binom{n}{\leq d}$ arrays, i.e., $A[k_1][k_2][\,]$ or $A[k_1][k_2][k_3][\,]$. This is indeed why the FES algorithm in [4] is only implemented for $d=2$.

4.2 Efficient Enumeration in Lex Order

Let the problem be enumerating supports of $\boldsymbol{x}\in P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$ such that $\boldsymbol{x}$ is in lex order. We show that a simple depth-first-search program can efficiently solve this as it is equivalent to traversing a tree.

In our python implementation, the list $W=[w_1,,\ldots,w_s]$ is used to store $w_s,\ldots,w_1$. In addition, we use the list H of size s to record the Hamming

weight of the current $\boldsymbol{x} \in P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$ where $H[0] = \mathrm{wt}(x_{n_1}, \ldots, x_1)$ and $H[i] = \mathrm{wt}(x_{n_{i+1}}, \ldots, x_{n_i+1})$ for $1 \le i < s$. By definition, $\mathrm{wt}(\boldsymbol{x}) = \sum_{i=0}^{s-1} H[i]$.

Moreover, we use a list N of size $s+1$ such that $N[0] = 0$ and $N[i] = N[i-1] + n_i = \sum_{t=1}^{i} n_t$ for $1 \le i \le s$. In addition, use a list G of size $n+1$ to efficiently determine the range of the integer $j \in [1, n]$ where $G[0] = 0$ and

$$G[j] = i - 1 \Leftrightarrow N[i-1] + 1 \le j \le N[i] \text{ for } 1 \le i \le s$$

The list K stores the support of the current $\boldsymbol{x}$ and $h = \mathrm{wt}(\boldsymbol{x})$, where

$$K[h] = k_1, K[h-1] = k_2, \ldots, K[1] = k_h. \tag{8}$$

In other words, $(k_h, \ldots, k_1) = \mathrm{Sup}(\boldsymbol{x})$ is stored in the reversed direction in K.

Concrete implementation. We use Listing 1.1 to recursively enumerate all supports of $\boldsymbol{x} \in P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$ such that $\boldsymbol{x}$ is in lex order.

Listing 1.1: Enumerate supports of $\boldsymbol{x} \in P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$

```
#G, N, W are global lists
def enumerationG(K, H, h, end):
    for j in range(1,end+1):# from 1 to end
        hw = h + 1
        K[hw] = j
        i = G[j]

        H[i] = H[i] + 1
        evaluation(K,hw) #polynomial evaluation (implemented later)
        if (H[i] < W[i]):
            enumerationG(K, H, hw, K[hw] - 1)
        elif (H[i] == W[i] and i > 0):
            enumerationG(K, H, hw, N[i - 1])
        else:
            H[i] = H[i] - 1

    if (h > 0):
        H[G[K[h]]] = H[G[K[h]]] - 1
```

For better understanding, an example for the search tree of $P^{2,1}_{3,2}$ is shown in Figure 2. With the depth-first search program in Algorithm 1.1, if each new node is visited, a new support is obtained (i.e., from root to this new node). The sequence is as follows:

$$[1], [2], [3], [3,1], [3,2], [4], [4,1], [4,2], [4,3], [4,3,1], [4,3,2],$$
$$[5], [5,1], [5,2], [5,3], [5,3,1], [5,3,2], [5,4], [5,4,1], [5,4,2].$$

Complexity analysis. There are in total $|P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}|$ nodes in the search tree. Visiting each new node with the depth-first search program gives a support of a new element in $P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$. The time complexity is $\mathcal{O}(P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1})|$ and the memory complexity is $\mathcal{O}(n)$.

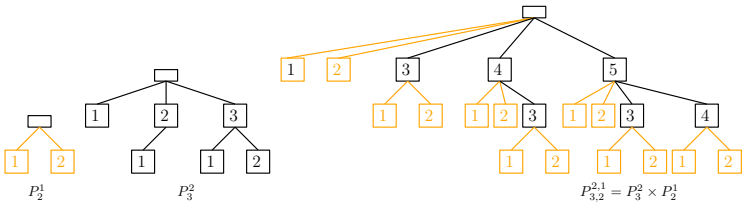


Fig. 2: Search trees to enumerate supports of elements in $P_{2,1}$, P_3^2 and $P_{3,2}^{2,1}$.

4.3 Implementation of Polynomial Evaluation

For each obtained support of the input $\boldsymbol{x}$ and its Hamming weight h, we can use the function evaluation(K, h) to compute $f(\boldsymbol{x})$ based on the algorithmic procedure in Section 3.2, as shown in Listing 1.2. In this function, it becomes clear how to efficiently compute the index of the array $A_{k[:j]}$ based on the relation in Equation 7. For better understanding, we also give a big picture of our full python implementation in Listing 1.3 in Appendix E.

Listing 1.2: Polynomial evaluation

```
#ind[j] is the index of A_{0,k_1,...,k_j}
#k[] stores k_0,k_1,..., i.e., k[j]=k_j=K[h+1-j]
def evaluation(K,h):
    size = min(h,d)
    #compute indices first
    k[0] = 0
    k[1] = K[h]
    ind[0] = 0
    ind[1] = k[1]
    for j in range(2,size+1):
        k[j] = K[h+1-j]
        ind[j] = ind[j-1] + BCOF[n][j-1] + BCOF[k[j]-1][j]
    #evaluation phase
    if (h<=d):
        B[ind[h-1]][k[h]-k[h-1]]=B[ind[h-1]][0] ^ B[ind[h]][0]
    else:
        k[d+1] = K[h-d]
        B[ind[d-1]][k[d]-k[d-1]]=B[ind[d-1]][k[d+1]-k[d-1]] ^ B[ind[d]][0]
    for j in range(size-2,-1,-1):#from size-2 to 0
        i1 = k[j+1]-k[j]; i2 = k[j+2]-k[j]; i3 = k[j+2]-k[j+1]
        B[ind[j]][i1]=B[ind[j]][i2] ^ B[ind[j+1]][i3]
    #output the evaluation result f(x)
    print(B[0][k[1]])
```

Complexity evaluation for practical implementation of FESG. Consider to evaluate a batch of $m \geq 1$ polynomials[6] in n variables and of degree d over the same input set $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$. The cost of the following operations remains static as m increases:

OP1: Enumerate the supports of elements in $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$;
OP2: Build the table storing the binomial coefficients;
OP3: Compute the index of the array $A_{k[:j]}$ in B;

[6] In this case, we use m two-dimensional arrays B.

OP4: Compute $k_{j+1}-k_j, k_{j+2}-k_j, k_{j+2}-k_{j+1}$ in evaluation on each input.

The complexity of these operations is estimated as $\mathcal{O}(d\cdot|P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}|+\binom{n}{\leq d})$, which is difficult to estimate in the number of bit operations. Taking our theoretic complexity analysis in Section 3.3 into account, we estimate the practical time complexity of the FESG algorithm for m polynomial evaluations as

$$\underbrace{\mathcal{O}\left(d\cdot|P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}|+\binom{n}{\leq d}\right)}_{\text{irrelevant to } m}+\underbrace{m\cdot\left(\binom{n}{\leq d}+d\cdot|P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}|\right)}_{\#\text{bit operations}}$$

The practical memory complexity of the FESG algorithm for m polynomial evaluations is estimated as

$$\underbrace{\log\binom{n}{\leq d}\cdot\binom{n}{\leq d}}_{\text{table for binomial coefficients}}+\underbrace{m\cdot 2\cdot\binom{n}{\leq d}}_{m \text{ different 2-dimensional arrays } B}\quad\text{bits.}$$

Comparison with known algorithms. We emphasize that in practical implementation of all memory-efficient polynomial evaluation algorithms especially for large d, it is inevitable to precompute and store these binomial coefficients for efficient access of coefficients of monomials (memory-efficient variants of Möbius transform [2, 3, 6]) or index of derivatives (derivative-based algorithms [4, 8]). However, when claiming the time and memory complexity of these memory-efficient polynomial evaluations algorithms [3, 4, 6, 8], these additional costs in practical implementation is not taken into account.

Indeed, such a cost model is reasonable in practice. This is because in most applications cases like the FES attack [4], the polynomial method [6] and the crossbred algorithm [5,9], m is large and such costs by OP1, OP2, OP3 and OP4 can be amortized.

Hence, based on the same cost model as in [2–4, 6, 8], we claim that the time and memory complexity of our FESG algorithm to evaluate a degree-d polynomial in n variables over the input set $P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$ as $\binom{n}{\leq d}+d\cdot|P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}|$ bit operations and $2\cdot\binom{n}{\leq d}$ bits, respectively.

Compared with the derivative-based algorithms [4, 8] developed for the full input space $\mathbb{F}_2^n$, our FESG algorithm can be used for a more general input set, extending the application range of the derivative-based polynomial evaluation framework. In particular, we also significantly optimize the initialization phase of this framework, removing the costly computation of all necessary derivatives, whose cost is lower bounded by $\sum_{i=0}^{d}\binom{n}{i}\cdot\binom{n-i}{\leq d-i}<\binom{n}{\leq d}^2$. In particular, in the literature [3, 4, 6, 8], the crude upper bound $\binom{n}{\leq d}^2$ is used for the complexity of the initialization phase.

Compared with the memory-efficient Möbius transform [3, 6], we also analyzed how these algorithms become inefficient when applied to the input set P^w_n for $w<n$. The underlying reason is that these algorithms are essentially based

on the idea to first fix some variables in the polynomial and then apply the standard Möbius transform.

In summary, the main advantage over known polynomial evaluation algorithms [3, 4, 6, 8] exists in its efficient application to a more general input set. If comparing them under the same cost model as described above (i.e., not considering additional costs in the practical implementation), our algorithm is almost optimized in both time and memory complexity, as shown in Table 1.

Table 1: Comparison between different algorithms for the evaluation of a degree-d Boolean polynomial in n variables on all inputs over $\mathbb{F}_2^n$

Method	Time (T)	Memory
standard Möbius transform	$n \cdot 2^n$	2^n
fast exhaustive search [4]	$\binom{n}{\leq d}^2 + d \cdot 2^n$	$\binom{n}{\leq d}$
fast enumeration algorithm [8]	$\binom{n}{\leq d}^2 + d \cdot 2^n$	$\log n \cdot n \cdot \binom{n}{\leq d}$
memory-efficient Möbius transform [6]	$2^{n-1} + d \cdot 2^n < T < n \cdot 2^n$	$n \cdot \binom{n}{\leq d}$
memory-efficient Möbius transform [6]	$n \cdot 2^n$	$2 \cdot \binom{n}{\leq d}$
memory-efficient Möbius transform [3]	$2d \cdot 2^n$	$\binom{n}{\leq d}$
FESG	$\binom{n}{\leq d} + d \cdot 2^n$	$2 \cdot \binom{n}{\leq d}$

5 Revisiting Fast Exhaustive Search Framework

From this section, we start discussing the correctness of our algorithm.

5.1 Rephrasing the Framework for Polynomial Evaluation

For a Boolean vector $\boldsymbol{x} = (x_n, \ldots, x_1)$, we define

$$\boldsymbol{x}_{(i)} = \boldsymbol{x} \oplus \boldsymbol{e}_i, \ \boldsymbol{x}^{(k[:i])} = \boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_i}} \wedge \cdots \wedge \overline{\boldsymbol{e}_{k_1}}, \ \boldsymbol{x}^{(k[:i])}_{(j)} = \boldsymbol{x}^{(k[:i])} \oplus \boldsymbol{e}_j.$$

We first convert the recursive description in Equation 4 into a non-recursive form to easily deduce some critical implications for this derivative-based framework to work.

Converting recursive form into non-recursive form. To evaluate $f(\boldsymbol{x})$, expanding Equation 4, it can be rephrased in the way that $\exists \ell : [1, d]$ and

$\{k_1, \ldots, k_\ell\} \subseteq \{1, \ldots, n\}$, and we have

$$\begin{cases} D_{k[:\ell]} f(\boldsymbol{x}^{(k[:\ell])}) \text{ can be efficiently computed} \\ D_{k[:\ell-1]} f(\boldsymbol{x}^{(k[:\ell-1])}) = D_{k[:\ell-1]} f(\boldsymbol{x}^{(k[:\ell-1])}_{(k_\ell)}) \oplus D_{k[:\ell]} f(\boldsymbol{x}^{(k[:\ell])}) \\ \cdots \\ D_{k[:i]} f(\boldsymbol{x}^{(k[:i])}) = D_{k[:i]} f(\boldsymbol{x}^{(k[:i])}_{(k_{i+1})}) \oplus D_{k[:i+1]} f(\boldsymbol{x}^{(k[:i+1])}) \\ \cdots \\ D_{k[:0]} f(\boldsymbol{x}^{(k[:0])}) = D_{k[:0]} f(\boldsymbol{x}^{(k[:0])}_{(k_1)}) \oplus D_{k[:1]} f(\boldsymbol{x}^{(k[:1])}) \end{cases} \tag{9}$$

Termination conditions. The recursion terminates if $\ell = d$ or $\boldsymbol{x}^{(k[:\ell])} = \mathbf{0}^n$, since $D_{k[:d]} f(\boldsymbol{x}^{(k[:d])}) = \mu(\sum_{i=1}^{d} \boldsymbol{e}_{k_i})$ for $\forall \boldsymbol{x} \in \mathbb{F}_2^n$ and $D_{k[:\ell]} f(\mathbf{0}^n) = \mu(\sum_{i=1}^{\ell} \boldsymbol{e}_{k_i})$.

Implication on evaluation rule. The first implication is that there should be a rule to determine the value of $(k_1, \ldots, k_\ell)$ for any given input $\boldsymbol{x}$. Specifically,

- For any $i \in [0, d-1]$, given the input $\boldsymbol{x}^{(k[:i])}$ to $D_{k[:i]} f$, there is a rule to determine a unique $\boldsymbol{x}^{(k[:i])}_{(k_{i+1})}$, i.e., k_{i+1} is uniquely determined for each input $\boldsymbol{x}^{(k[:i])}$.

Definition 4 (Evaluation Rule). *The evaluation rule $\mathcal{R}$ for evaluating a polynomial $g(x_n, \ldots, x_1)$ over an ordered input set S is defined in the way that given an input $\boldsymbol{x} \in S$ to the polynomial g, there is a unique choice of $\boldsymbol{x}' \in S$ such that we compute $g(\boldsymbol{x})$ with*

$$g(\boldsymbol{x}) = g(\boldsymbol{x}') \oplus D_i(g)(\boldsymbol{x}^{(i)}) \text{ where } \boldsymbol{x} \oplus \boldsymbol{x}' = \boldsymbol{e}_i.$$

In particular, this is denoted by $\boldsymbol{x}' \xrightarrow{\mathcal{R}} \boldsymbol{x}$.

Implication on evaluation order. The second implication is related to the order to enumerate the input $\boldsymbol{x}$ of f. According to the non-recursive description, we have the following conditions:

- For any $i \in [0, d-1]$, the input $\boldsymbol{x}^{(k[:i])}$ to $D_{k[:i]} f$ must be enumerated after the input $\boldsymbol{x}^{(k[:i])}_{(k_{i+1})}$, and $D_{k[:i]} f(\boldsymbol{x}^{(k[:i])}_{(k_{i+1})})$ must be stored somewhere at the point to compute $D_{k[:i]} f(\boldsymbol{x}^{(k[:i])})$. In this way, computing $D_{k[:i]} f(\boldsymbol{x}^{(k[:i])})$ only requires to compute $D_{k[:i+1]} f(\boldsymbol{x}^{(k[:i+1])})$.

Definition 5 (Evaluation Order). *If the evaluation of a polynomial g on the input $\boldsymbol{x}$ is performed before the evaluation of g on the input $\boldsymbol{x}'$, we say $\boldsymbol{x} <_g \boldsymbol{x}'$.*

For convenience, throughout this paper, $S_{k[:i]}$ is used to denote the ordered input set for $D_{k[:i]} f$, and the evaluation rule to evaluate $D_{k[:i]} f$ over $S_{k[:i]}$ is denoted by $\mathcal{R}_{k[:i]}$. In other words, we have

$$\forall j \in [1, |S_{k[:i]}| - 1] : \; S_{k[:i]}[j-1] <_{D_{k[:i]} f} S_{k[:i]}[j].$$

Algorithm 1 Find ordered input set based on the evaluation rule $\mathcal{R}$ to evaluate a polynomial over S

procedure $\texttt{NextSet}(S, \mathcal{R}, i)$
  $S' = \emptyset$ // empty set
  for $j \in [1, |S| - 1]$ **do**
    Find $j' \in [0, j-1]$ such that $S[j'] \xrightarrow{\mathcal{R}} S[j]$ and $\boldsymbol{e}_i = S[j'] \oplus S[j]$
    Add $S[j] \wedge \overline{\boldsymbol{e}_i}$ to the set S'
  end for
  return S'
end procedure

Moreover, based on Algorithm 1, we have

$$\forall i \in [0, d-1]: \ S_{k[:i+1]} = \texttt{NEXTSET}(S_{k[:i]}, \mathcal{R}_{k[:i]}, k_{i+1}). \tag{10}$$

In this way, we can implicitly construct the ordered input sets for all possible polynomials appearing in the procedure to evaluate f on all inputs in $S_{k[:0]}$. Some examples are given at Appendix C.

Implication on initialization phase. After determining the evaluation order and the evaluation rules to perform polynomial evaluations, we can always implicitly construct the ordered input set for each possible polynomial that will appear in Equation 9. If we reach the point to evaluate the corresponding polynomial on the first element in its input set, we can no longer use Equation 9. As a result, it is required to evaluate each polynomial on the first element in its corresponding ordered input set at the initialization phase, and store these results in a table. In other words, at the initialization phase, the following set should be computed and stored:

$$S_{\text{ini}} = \{f(S_{k[0]}[0]), D_{k[:i]}f(S_{k[:i]}[0]) \mid i \in [1, d], \{k_1, \ldots, k_i\} \subseteq \{1, \ldots, n\}\}.$$

5.2 Sufficient Condition for the Framework

We have proposed two concepts called **evaluation rule** and **evaluation order** to rephrase the framework for polynomial evaluations proposed in [4]. In this section, we further dive into the rephrased framework and give a sufficient condition to make the framework work efficiently. This sufficient condition will be extremely useful for applying this framework to the structured space $P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$.

Theorem 1 (Sufficient condition). *When evaluating a degree-d polynomial $f(x_n, \ldots, x_1)$ over an ordered set $S_{\{0\}} = S_{k[:0]}$ based on Equation 9, let g be the non-constant polynomial that will appear in Equation 9, and the ordered input set for g be S. Moreover, the evaluation rule to evaluate g over S is denoted by $\mathcal{R}$, i.e.,*

$$(g, S, \mathcal{R}) \in \{(D_{k[:i]}f, S_{k[:i]}, \mathcal{R}_{k[:i]}) \mid i \in [0, d], (k_1, \ldots, k_i) \subseteq \{1, \ldots, n\}\}.$$

For any such $(g, S, \mathcal{R})$*, if*

$$\forall \boldsymbol{x} \in S \text{ and } \boldsymbol{x} \neq S[0]: \ \exists \boldsymbol{x}' \in S, \boldsymbol{x}' <_g \boldsymbol{x}, \ \boldsymbol{x}' \xrightarrow{\mathcal{R}} \boldsymbol{x}, \tag{11}$$

evaluating f *over* $S_{k[:0]}$ *requires at most* $d \cdot |S_{k[:0]}|$ *bit operations, in addition to the cost to evaluate all possible* $g(S[0])$ *at the initialization phase.*

Proof. We show a memory-inefficient algorithm consisting of the initialization phase and evaluation phase to evaluate f over $S_{k[:0]}$. At the initialization phase, compute all possible $g(S[0])$ for all $(g, S) \in \{(D_{k[:i]}, S_{k[:i]}) \mid i \in [0, d], (k_1, \ldots, k_i) \subseteq \{1, \ldots, n\}\}$, and add the evaluation results $(k[:i], S_{k[:i]}[0], D_{k[:i]}f(S_{k[:i]}[0]))$ to the table $TAB_{\mathbf{eva}}$.

Then, compute $f(\boldsymbol{x})$ for all $\boldsymbol{x} \in S_{k[:0]}$ by calling $\texttt{FUN}(k[:0], \boldsymbol{x}, f)$ as described in Algorithm 2.

Algorithm 2 Recursively compute $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])})$

procedure $\texttt{FUN}(k[:i], \boldsymbol{x}, f)$
  if $\boldsymbol{x}^{(k[:i])} = S_{k[:i]}[0]$ **then**
    get $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])})$ from TAB_{eva}
  else
    find $\boldsymbol{x}' \in S_{k[:i]}$ and k_{i+1} such that

$$\boldsymbol{x}' <_{D_{k[:i]}f} \boldsymbol{x}^{(k[:i])}, \boldsymbol{x}' \xrightarrow{\mathcal{R}_{k[:i]}} \boldsymbol{x}^{(k[:i])}, \boldsymbol{x}' \oplus \boldsymbol{x}^{(k[:i])} = \boldsymbol{e}_{k_{i+1}}.$$

    Get $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}_{(k_{i+1})})$ from TAB_{eva}
    $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}) = D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}_{(k_{i+1})}) \oplus \texttt{FUN}(k[:i+1], \boldsymbol{x}, f)$
    add $(k[:i], \boldsymbol{x}^{(k[:i])}, D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}))$ to TAB_{eva}
  end if
  return $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])})$
end procedure

The reason is that if $\boldsymbol{x}^{(k[:i])} \neq S_{k[:i]}[0]$, according to Equation 11, there exists $\boldsymbol{x}' \in S_{k[:i]}$ such that

$$\boldsymbol{x}' <_{D_{k[:i]}f} \boldsymbol{x}^{(k[:i])}, \boldsymbol{x}' \xrightarrow{\mathcal{R}_{k[:i]}} \boldsymbol{x}^{(k[:i])}, \boldsymbol{x}' \oplus \boldsymbol{x}^{(k[:i])} = \boldsymbol{e}_{k_{i+1}}.$$

Since $\boldsymbol{x}' = \boldsymbol{x}^{(k[:i])} \oplus \boldsymbol{e}_{k_{i+1}} = \boldsymbol{x}^{(k[:i])}_{(k_{i+1})}$, the relation

$$D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}) = D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}_{(k_{i+1})}) \oplus D_{k[:i+1]}f(\boldsymbol{x}^{(k[:i+1])})$$

is used to compute $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])})$, where $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}_{(k_{i+1})}) = D_{k[:i]}f(\boldsymbol{x}')$ can be directly obtained from $TAB_{\mathbf{eva}}$ since it has been computed before $f(\boldsymbol{x})$ by definition of the evaluation order.

Based on the above procedure, computing $f(\boldsymbol{x})$ for any $\boldsymbol{x} \in S_{k[:0]}$ takes at most d bit operations (assuming that the cost to find $\boldsymbol{x}'$ is negligible). Consequently, evaluating f over $S_{k[:0]}$ takes at most $d \cdot |S_{k[:0]}|$ bit operations for the evaluation phase. □

Comparison. We emphasize that the same idea in the proof of Theorem 1 has been implicitly used in the fast exhaustive search framework [4] to evaluate a polynomial on all inputs in $\mathbb{F}_2^n$ using Gray code. Here, we make a further generalization that can possibly lead to broader applications. Specifically, neither the evaluation orders, nor the evaluation rules, nor even the input set S_0 that might not be the full space $\mathbb{F}_2^n$, are specified. Therefore, they can be customized for concrete applications as long as the sufficient condition can hold, e.g., for efficiently evaluating a Boolean polynomial over the structured space $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$.

Impact of the sufficient condition. Dinur asked whether it is possible to revise the FES algorithm [4] with monotone Gray code for the input set P_n^w. With the above sufficient condition, we claim that it is either infeasible to use monotone Gray code in the FES algorithm, or it is nontrivial to find a proper evaluation rule $\mathcal{R}_{k[:i]}$ for each polynomial $D_{k[:i]}f$ that can satisfy the sufficient condition. Details can be found at Appendix D.

6 Application to FESG Algorithm

In this section, we prove that the evaluation rule and the evaluation order proposed in Furue-Takagi's algorithm [8] can be reused to perform efficient polynomial evaluations over $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$. The problem is reduced to testing the sufficient condition specified in Theorem 1. In addition, optimizing the memory complexity requires nontrivial efforts as the algorithm in Theorem 1 is memory-inefficient.

Remark 2. Our algorithm is distinguished from Furue-Takagi's FEA algorithm [8] by the initialization phase, a better data structure to store intermediate results, resulting in improved time complexity and memory complexity under the same setting $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1} = \mathbb{F}_2^n$. In particular, we rigorously prove that it is feasible to apply such an evaluation rule and evaluation order to polynomial evaluations over a more general input set $P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1} \subseteq \mathbb{F}_2^n$. All these are made possible due to our interpretation of the fast exhaustive search framework [4], Theorem 1 developed for its sufficient condition, Lemma 2 to test the sufficient condition, and Lemma 3 to optimize the memory consumption.

The evaluation order and evaluation rule. Let $S_{k[:0]} = P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$. $S_{k[:0]}$ is an ordered set satisfying the following order:

$$\forall j \in [1, |S_{k[:0]}| - 1]: \ S_{k[:0]}[j-1] <_{\text{lex}} S_{k[:0]}[j].$$

The evaluation rule for any polynomial $D_{k[:i]}f$ over $S_{k[:i]}$ is the same, i.e., $\mathcal{R}_{k[:i]} = \mathcal{R}_{\text{lex}}$, which is defined below:

$$(x_n, \ldots, x_{j+1}, 0, \underbrace{0, \ldots, 0}_{j-1}) \xrightarrow{\mathcal{R}_{\text{lex}}} (x_n, \ldots, x_{j+1}, 1, \underbrace{0, \ldots, 0}_{j-1}) \tag{12}$$

6.1 Testing Sufficient Condition

We first show an important lemma that plays a crucial role to apply the framework to the structured space $P^{w_s,\dots,w_1}_{n_s,\dots,n_1}$.

Lemma 2. *Let $S_{k[:i-1]} = P^{w_s,\dots,w_1}_{n_s,\dots,n_1} \subseteq \mathbb{F}_2^n$ be an ordered set where $|S_{k[:i-1]}| > 1$ and*

$$\forall j \in [1, |S_{k[:i-1]}| - 1]: \ S_{k[:i-1]}[j-1] <_{lex} S_{k[:i-1]}[j].$$

If evaluating a Boolean polynomial $D_{k[:i-1]}f(x_n, \dots, x_1)$ over $S_{k[:i-1]}$ under the evaluation rule $\mathcal{R}_{lex}$, there is

$$\forall \boldsymbol{x} \in S_{k[:i-1]} \text{ and } \boldsymbol{x} \neq S_{k[:i-1]}[0]: \ \exists \boldsymbol{x}' \in S_{k[:i-1]}, \boldsymbol{x}' <_{lex} \boldsymbol{x}, \ \boldsymbol{x}' \xrightarrow{\mathcal{R}_{lex}} \boldsymbol{x}. \quad (13)$$

Moreover, let $S_{k[:i]}$ be the ordered input set for the polynomial $D_{k[:i]}f$ when evaluating $D_{k[:i-1]}f(\boldsymbol{x})$ is based on $\mathcal{R}_{lex}$. Then the input set $S_{k[:i]} \subseteq \mathbb{F}_2^n$ for $D_{k[:i]}f$ is also an ordered set satisfying

$$\forall j \in [1, |S_{k[:i]}| - 1]: \ S_{k[:i]}[j-1] <_{lex} S_{k[:i]}[j]. \quad (14)$$

In particular, we have

$$\sum_{j=1}^{t-1} n_j < k_i \leq \sum_{j=1}^{t} n_j, \ n'_t = n_t - (k_i - \sum_{j=1}^{t-1} n_j),$$
$$S_{k[:i]} = P^{w_s}_{n_s} \times \cdots \times P^{n_{t+1}}_{n_{t+1}} \times P^{min(n'_t, w_t - 1)}_{n'_t} \times P^0_{k_i}.$$

Proof. For any $\boldsymbol{x} \in S_{k[:i-1]} = P^{w_s,\dots,w_1}_{n_s,\dots,n_1}$ and $\boldsymbol{x} \neq S_{k[:i-1]}[0] = \mathbf{0}^n$, we have

$$\exists j \in [1, s]: \ \boldsymbol{x} = (\boldsymbol{a}, \boldsymbol{b}, \mathbf{0}^{n_1+\dots+n_{j-1}}), \boldsymbol{a} \in P^{w_s,\dots,w_{j+1}}_{n_s,\dots,n_{j+1}}, \boldsymbol{b} \in P^{w_j}_{n_j}, \boldsymbol{b} \neq \mathbf{0}^{n_j}.$$

Since $\boldsymbol{b} \neq \mathbf{0}^{n_j}$, we can set $\boldsymbol{b} = (\boldsymbol{c}, 1, \mathbf{0}^{u-1}) \in P^{w_j}_{n_j} \subseteq \mathbb{F}_2^{n_j}$, resulting in $\boldsymbol{b}' = (\boldsymbol{c}, \mathbf{0}^u) \in P^{w_j-1}_{n_j} \subseteq P^{w_j}_{n_j}$. Hence, it is always possible to find $\boldsymbol{x}' = (\boldsymbol{a}, \boldsymbol{b}', \mathbf{0}^{n_1+\dots+n_{j-1}}) \in S_{k[:i-1]} = P^{w_s,\dots,w_1}_{n_s,\dots,n_1}$ such that $\boldsymbol{x}' <_{\text{lex}} \boldsymbol{x}$, $\boldsymbol{x}' \xrightarrow{\mathcal{R}_{\text{lex}}} \boldsymbol{x}$. This proves Equation 13.

Based on $\mathcal{R}_{\text{lex}}$, the relation

$$D_{k[:i-1]}f(\boldsymbol{x}) = D_{k[:i-1]}f(\boldsymbol{x} \oplus \boldsymbol{e}_{k_i}) + D_{k[:i]}f(\boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_i}})$$

is only used to evaluate $D_{k[:i-1]}f$ on the input

$$\boldsymbol{x} = (\boldsymbol{a}, \boldsymbol{c}, 1, \mathbf{0}^{k_i-1}) \in P^{w_s,\dots,w_{t+1}}_{n_s,\dots,n_{t+1}} \times P^{w_t}_{n_t} \times P^0_{n_1+\dots+n_{t-1}} \subseteq P^{w_s,\dots,w_1}_{n_s,\dots,n_1},$$

where $\sum_{j=1}^{t-1} n_j < k_i \leq \sum_{j=1}^{t} n_j$. Hence, the input $\boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_i}}$ of $D_{k[:i]}f$ satisfies $\boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_i}} = (\boldsymbol{a}, \boldsymbol{c}, \mathbf{0}^{k_i})$. When $\boldsymbol{x} \in P^{w_s,\dots,w_1}_{n_s,\dots,n_1}$ is enumerated in ascending lex order, obviously the set of $(\boldsymbol{a}, \boldsymbol{c})$ is also in ascending lex order, which proves Equation 14. Moreover, since

$$\sum_{j=1}^{t-1} n_j < k_i \leq \sum_{j=1}^{t} n_j, \ (\boldsymbol{a}, \boldsymbol{c}, \mathbf{0}^{k_i}) \in P^{w_s,\dots,w_{t+1}}_{n_s,\dots,n_{t+1}} \times P^{w_t-1}_{n_t} \times P^0_{n_1+\dots+n_{t-1}},$$

if $(\boldsymbol{a}, \boldsymbol{c})$ are exhausted, the set of all $(\boldsymbol{a}, \boldsymbol{c})$ is exactly

$$P_{n_s,\ldots,n_{t+1}}^{w_s,\ldots,w_{t+1}} \times P_{n_t'}^{\min(n_t', w_t-1)} \text{ where } n_t' = n_t - (k_i - \sum_{j=1}^{t-1} n_j).$$

This proves $S_{k[:i]} = P_{n_s}^{w_s} \times \cdots \times P_{n_{t+1}}^{n_{t+1}} \times P_{n_t'}^{\min(n_t', w_t-1)} \times P_{k_i}^0$. □

Lemma 3. *Let $S_{k[:i-1]} = P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1} \subseteq \mathbb{F}_2^n$ be in ascending lex order. After evaluating $D_{k[:i-1]}f$ on the input $\boldsymbol{x} = (\boldsymbol{a}, 1, \boldsymbol{0}^{k_i-1}) \in P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}$ based on $\mathcal{R}_{lex}$ and*

$$D_{k[:i-1]}f(\boldsymbol{x}) = D_{k[:i-1]}f(\boldsymbol{x} \oplus \boldsymbol{e}_{k_i}) + D_{k[:i]}f(\boldsymbol{x} \wedge \overline{\boldsymbol{e}_{k_i}}),$$

for any element $\boldsymbol{y}$ in the set

$$S_{\boldsymbol{x}} = \{\boldsymbol{y} = (\boldsymbol{b}, 1, \boldsymbol{0}^{k_i-1}) \mid \boldsymbol{y} \in P_{n_s,\ldots,n_1}^{w_s,\ldots,w_1}, \boldsymbol{y} <_{lex} \boldsymbol{x} = (\boldsymbol{a}, 1, \boldsymbol{0}^{k_i-1})\},$$

$D_{k[:i-1]}f(\boldsymbol{y})$ is never accessed when computing $D_{k[:i-1]}f(\boldsymbol{z})$ for $\boldsymbol{x} <_{lex} \boldsymbol{z}$ and $\boldsymbol{z} \in S_{k[:i-1]}$.

Proof. To prove this, we only need to show that

$$\forall \boldsymbol{z} \in S_{k[:i-1]} \text{ and } \boldsymbol{x} <_{\text{lex}} \boldsymbol{z} : \text{ if } \boldsymbol{z}' \xrightarrow{\mathcal{R}_{\text{lex}}} \boldsymbol{z}, \text{ then } \boldsymbol{z}' \notin S_{\boldsymbol{x}}. \tag{15}$$

The reason is that only the elements in $\{\boldsymbol{z} \mid \boldsymbol{z} \in S_{k[:i-1]}, \boldsymbol{x} <_{\text{lex}} \boldsymbol{z}\}$ are left for the inputs of $D_{k[:i-1]}f$ after $D_{k[:i-1]}f(\boldsymbol{x})$ is evaluated. If evaluating these $D_{k[:i-1]}f(\boldsymbol{z})$ does not require the knowledge of $\{(\boldsymbol{y}, D_{k[:i-1]}f(\boldsymbol{y})) \mid \boldsymbol{y} \in S_{\boldsymbol{x}}\}$, the set $\{(\boldsymbol{y}, D_{k[:i-1]}f(\boldsymbol{y})) \mid \boldsymbol{y} \in S_{\boldsymbol{x}}\}$ becomes useless, which is useful to optimize the memory-inefficient algorithm in Theorem 1.

We prove Equation 15 case by case:

Case 1: When $\boldsymbol{z} = (\boldsymbol{c}, 1, \boldsymbol{0}^{j-1}) \in \{\boldsymbol{z} \mid \boldsymbol{z} \in S_{k[:i-1]}, \boldsymbol{x} <_{\text{lex}} \boldsymbol{z}\}$ for $k_i \le j \le n$, according to the evaluation rule $\mathcal{R}_{\text{lex}}$, we have $\boldsymbol{z}' = (\boldsymbol{c}, \boldsymbol{0}^j) \xrightarrow{\mathcal{R}_{\text{lex}}} \boldsymbol{z}$. For $j \ge k_i$, the last k_i elements in $\boldsymbol{z}'$ is all 0, i.e., $\boldsymbol{0}^{k_i} \ne (1, \boldsymbol{0}^{k_i-1})$, thus resulting in $\boldsymbol{z}' \notin S_{\boldsymbol{x}}$.

Case 2: When $\boldsymbol{z} = (\boldsymbol{c}, 1, \boldsymbol{0}^{j-1}) \in \{\boldsymbol{z} \mid \boldsymbol{z} \in S_{k[:i-1]}, \boldsymbol{x} <_{\text{lex}} \boldsymbol{z}\}$ for $1 \le j < k_i$, according to the evaluation rule $\mathcal{R}_{\text{lex}}$, we have $\boldsymbol{z}' = (\boldsymbol{c}, \boldsymbol{0}^j) \xrightarrow{\mathcal{R}_{\text{lex}}} \boldsymbol{z}$. Let $\boldsymbol{z}' = (z_n', \ldots, z_1')$.
If $(z_{k_i}', \ldots, z_1') \ne (1, \boldsymbol{0}^{k_i-1})$, we directly have $\boldsymbol{z}' \notin S_{\boldsymbol{x}}$.
If $(z_{k_i}', \ldots, z_1') = (1, \boldsymbol{0}^{k_i-1})$, since $j < k_i$, we have

$$\boldsymbol{z}' = (z_n, \ldots, z_{k_i+1}, 1, \boldsymbol{0}^{k_i-1}), \ \boldsymbol{z} = (z_n, \ldots, z_{k_i+1}, 1, \boldsymbol{0}^{k_i-j-1}, 1, \boldsymbol{0}^{j-1}).$$

Since $\boldsymbol{x} = (\boldsymbol{a}, 1, \boldsymbol{0}^{k_i-1}) <_{\text{lex}} \boldsymbol{z}$, we have $\boldsymbol{a} \le_{\text{lex}} (z_n, \ldots, z_{k_i+1})$, resulting in $\boldsymbol{x} \le_{\text{lex}} \boldsymbol{z}'$. Therefore, we also have $\boldsymbol{z}' \notin S_{\boldsymbol{x}}$.

□

6.2 Proving the Correctness of FESG

Consider to evaluate a polynomial $f(x_n, \ldots, x_1)$ of degree d over the ordered set $S_{k[:0]} = P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$ under $\mathcal{R}_{\text{lex}}$ based on Equation 9, where

$$\forall j \in [1, |S_{k[:0]}| - 1] : \; S_{k[:0]}[j-1] <_{\text{lex}} S_{k[:0]}[j].$$

Iteratively applying Lemma 2, the input set $S_{k[:i]} = \texttt{NEXTSET}(S_k[:i-1], \mathcal{R}_{\text{lex}}, k_i)$ for any polynomial $D_{k[:i]}f$ preserves the same structure as $P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$, where $S_{k[:i]} = P^{w'_v,\ldots,w'_2,0}_{n'_v,\ldots,n'_2,k_i}$ and the elements in $S_{k[:i]}$ are also listed in ascending lex order. In other words, any input set $S = S_{k[:i]}$ satisfies Equation 13. Consequently, the sufficient condition in Theorem 1 holds. Therefore, the derivative-based framework, i.e., Equation 9, can be applied to the input set $P^{w_s,\ldots,w_1}_{n_s,\ldots,n_1}$.

Moreover, $S_{k[:i]}[0] = \mathbf{0}^n$ for all input sets and hence the initialization phase is to compute $D_{k[:i]}f(\mathbf{0}^n)$, which is equivalent to reading all coefficients of f. This shows the correctness of the initialization phase of the FESG algorithm.

The left work is to optimize the memory complexity of the memory-inefficient algorithm in Theorem 1 based on Lemma 3.

Construction of the arrays $A_{k[:i]}$. We can view table TAB_{eva} described in Theorem 1 as a set of $\binom{n}{\leq d}$ arrays $A_{k[:i]}$. Adding $(k[:i], \boldsymbol{x}^{(k[:i])}, D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}))$ to TAB_{eva} is equivalent to storing $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])})$ in a certain entry of $A_{k[:i]}$. The construction of these arrays is described below.

- $A_{k[:i]}[0]$ stores $D_{k[:i]}f(\mathbf{0}^n) = \mu(\sum_{j=1}^{i} \boldsymbol{e}_{k_j})$ (initialization phase).
- $A_{k[:i]}[j]$ $(1 \leq j \leq n - k_i)$ stores the evaluation result of $D_{k[:i]}f$ on the latest input $\boldsymbol{x}$ where $\boldsymbol{x} = (x_n, \ldots, x_{k_i+j+1}, 1, \mathbf{0}^{j-1}, \mathbf{0}^{k_i}) \in S_{k[:i]}$.

Specifically, while $(\boldsymbol{x}^{k[:i]}, D_{k[:i]}f(\boldsymbol{x}^{k[:i]}))$ should be stored in TAB_{eva} for all $\boldsymbol{x}^{k[:i]} \in S_{k[:i]}$ in the original memory-inefficient algorithm in Theorem 1, due to Lemma 3, after $(k[:i], \boldsymbol{x}^{k[:i]}, D_{k[:i]}f(\boldsymbol{x}^{k[:i]}))$ is added to TAB_{eva}, $(k[:i], \boldsymbol{y}, D_{k[:i]}f(\boldsymbol{y}))$ where $\boldsymbol{y} = (y_n, \ldots, y_{k_i+j+1}, 1, \mathbf{0}^{j-1}, \mathbf{0}^{k_i})$ and $\boldsymbol{y} <_{\text{lex}} \boldsymbol{x}$ can be safely removed. Therefore, it is sufficient to only use $A_{k[:i]}[j]$ to store the evaluation of $D_{k[:i]}f$ on the latest input $\boldsymbol{x}$ where $(x_{j+k_i}, \ldots, x_1) = (1, \mathbf{0}^{j-1}, \mathbf{0}^{k_i})$.

Update of arrays. Evaluating $f(\boldsymbol{x})$ is based on Equation 9. Let $\boldsymbol{x} = \sum_{j=1}^{h} \boldsymbol{e}_{k_j}$ be the current input to f where $1 \leq k_1 < \ldots < k_h \leq n$. From Equation 9, we know that the following relation is used for computing $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])})$:

$$D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}) = D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}_{(k_{i+1})}) \oplus D_{k[:i+1]}f(\boldsymbol{x}^{(k[:i+1])}). \tag{16}$$

In addition, we have

$$\boldsymbol{x}^{(k[:i])} = \sum_{j=i+1}^{h} \boldsymbol{e}_{k_j}, \; \boldsymbol{x}^{(k[:i])}_{(k_{i+1})} = \sum_{j=i+2}^{h} \boldsymbol{e}_{k_j}, \; \boldsymbol{x}^{(k[:i+1])} = \sum_{j=i+2}^{h} \boldsymbol{e}_{k_j}.$$

If $i = h - 1$, Equation 16 becomes

$$D_{k[:h-1]}f(\boldsymbol{x}^{(k[:h-1])}) = D_{k[:h-1]}f(\mathbf{0}^n) \oplus D_{k[:h]}f(\mathbf{0}^n) = A_{k[:h-1]}[0] \oplus A_{k[:h]}[0].$$

If $i = d - 1$ and $i < h - 1$, Equation 16 becomes

$$D_{k[:d-1]}f(\boldsymbol{x}^{(k[:d-1])}) = D_{k[:d-1]}f(\boldsymbol{x}_{(k_d)}^{(k[:d-1])}) \oplus D_{k[:d]}f(\mathbf{0}^n). \tag{17}$$

Due to the sufficient condition, when reaching Equation 17, $D_{k[:d-1]}f(\boldsymbol{x}_{(k_d)}^{(k[:d-1])})$ must have been computed before $D_{k[:d-1]}f(\boldsymbol{x}^{(k[:d-1])})$ and stored in the array. Since it is used here, it has not been removed before this computation and it is stored in $A_{k[:d-1]}[k_{d+1} - k_{d-1}]$ by our construction of the array. Hence, Equation 17 becomes

$$D_{k[:d-1]}f(\boldsymbol{x}^{(k[:d-1])}) = A_{k[:d-1]}[k_{d+1} - k_{d-1}] \oplus A_{k[:d]}[0].$$

For other cases, due to the sufficient condition, Lemma 3, and the construction of our array, Equation 16 becomes

$$D_{k[:i]}f(\boldsymbol{x}^{(k[:i])}) = A_{k[:i]}[k_{i+2} - k_i] \oplus A_{k[:i+1]}[k_{i+2} - k_{i+1}].$$

After computing $D_{k[:i]}f(\boldsymbol{x}^{(k[:i])})$, it is used to overwrite $A_{k[:i]}[k_{i+1} - k_i]$ by the construction of our array. This gives our algorithm described in Section 3.

References

1. Banik, S., Caforio, A., Vaudenay, S.: New attacks on lowmc using partial sets in the single-data setting. IACR Commun. Cryptol. **1**(1), 22 (2024)
2. Banik, S., Regazzoni, F.: Compact Circuits for Efficient Möbius Transform. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2024**(2), 481–521 (2024)
3. Bouillaguet, C.: Algorithm 1052: Evaluating a boolean polynomial on all possible inputs. ACM Trans. Math. Softw. **50**(4), 28:1–28:37 (2024)
4. 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: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems, CHES 2010. pp. 203–218. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
5. Bouillaguet, C., Delaplace, C., Trimoska, M.: A Simple Deterministic Algorithm for Systems of Quadratic Polynomials over F_2. In: SOSA. pp. 285–296. SIAM (2022)
6. 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)
7. 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)
8. Furue, H., Takagi, T.: Fast enumeration algorithm for multivariate polynomials over general finite fields. In: Johansson, T., Smith-Tone, D. (eds.) Post-Quantum Cryptography. pp. 357–378. Springer Nature Switzerland, Cham (2023)
9. Joux, A., Vitse, V.: A Crossbred Algorithm for Solving Boolean Polynomial Systems. In: NuTMiC. Lecture Notes in Computer Science, vol. 10737, pp. 3–21. Springer (2017)

10. Liu, F., Mahzoun, M., Øygarden, M., Meier, W.: Algebraic Attacks on RAIN and AIM Using Equivalent Representations. IACR Trans. Symmetric Cryptol. **2023**(4), 166–186 (2023)
11. Liu, F., Meier, W., Sarkar, S., Isobe, T.: New Low-Memory Algebraic Attacks on LowMC in the Picnic Setting. IACR Trans. Symmetric Cryptol. **2022**(3), 102–122 (2022)
12. Lokshtanov, D., Paturi, R., Tamaki, S., Williams, R.R., Yu, H.: Beating Brute Force for Systems of Polynomial Equations over Finite Fields. In: SODA. pp. 2190–2202. SIAM (2017)
13. Savage, C.D., Winkler, P.: Monotone Gray Codes and the Middle Levels Problem. J. Comb. Theory A **70**(2), 230–248 (1995)

supplementary materials

A Standard Möbius Transform

In this section, we focus on a Boolean polynomial $f(x_n, \ldots, x_1)$ of degree d. Denote the set of all $\boldsymbol{x} \in \mathbb{F}_2^n$ with Hamming weight of at most w by P_n^w, i.e.,

$$P_n^w = \{\boldsymbol{x} \in \mathbb{F}_2^n \mid \mathrm{wt}(\boldsymbol{x}) \leq w\}. \tag{18}$$

The standard (also called in-place) Möbius transform can be used to evaluate f on all inputs in P_n^w with $\mathcal{O}(|P_n^w|)$ bits of memory and $\mathcal{O}(n \cdot |P_n^w|)$ bit operations. There are two phases which we call the initialization (or pre-processing) phase and the evaluation phase.

Initialization phase. At the initialization phase, prepare a binary array A of size $|P_n^w| = \binom{n}{\leq w}$. In addition, assume that there is an efficient bijection $\Phi_{n,w} : P_n^w \mapsto \{0, \ldots, \binom{n}{\leq w} - 1\}$, i.e.,

$$\Phi_{n,w} : (x_n, \ldots, x_1) \in P_n^w \mapsto i \in \{0, \ldots, \binom{n}{\leq w} - 1\}. \tag{19}$$

When $w = n$, $\Phi_{n,w}$ can be simply

$$\Phi_{n,n}(\boldsymbol{x}) = INT(\boldsymbol{x}).$$

However, when $w \leq n$, constructing an efficient bijection $\Phi_{n,w}$ is closely related to monomial ordering and we discuss it later. Here, we assume the existence of such a $\Phi_{n,w}$ for any w.

After having the array A, it is initialized as follows:

$$A[\Phi_{n,w}(\boldsymbol{x})] = \mu(\boldsymbol{x}) \text{ if } \mathrm{wt}(\boldsymbol{x}) \leq d,$$
$$A[\Phi_{n,w}(\boldsymbol{x})] = 0 \text{ if } \mathrm{wt}(\boldsymbol{x}) > d.$$

Note that we always assume $w \geq d$ as this is a common setting, i.e., the input size is larger than the maximal number of monomials in f of degree d.

Evaluation phase. After the initialization phase, the evaluation phase consists of the following n steps:

– Step $i \in [1, n]$: Enumerate $\boldsymbol{x}$ in P_n^w in lex order. If $x_i = 1$ in $\boldsymbol{x}$, update A as follows:

$$A[\Phi_{n,w}(\boldsymbol{x})] = A[\Phi_{n,w}(\boldsymbol{x} \oplus \boldsymbol{e}_i)] \oplus A[\Phi_{n,w}(\boldsymbol{x})].$$

Note that the entry $A[\Phi_{n,w}(\boldsymbol{x})]$ is never updated at Step i if $x_i = 0$. In addition, when $x_i = 1$, $\boldsymbol{x} \oplus \boldsymbol{e}_i$ is equal to

$$(x_n, \ldots, x_{i+1}, 0, x_{i-1}, \ldots, x_1).$$

After n steps of evaluation, we get

$$\forall \boldsymbol{x} \in P_n^w : \ f(\boldsymbol{x}) = A[\Phi_{n,w}(\boldsymbol{x})]$$

In other words, the $|P_n^w|$ evaluation results are stored exactly in A. As a result, the time and memory complexity of the in-place Möbius transform are $\mathcal{O}(n \cdot |P_n^w|)$ bit operations and $\mathcal{O}(|P_n^w|)$ bits.

Remark 3. The reason to use $\mathcal{O}$ here is that there are indeed additional costs in practical implementations. Specifically, it is necessary to also consider the practical cost to compute $\Phi_{n,w}(\boldsymbol{x})$ and to enumerate all $\boldsymbol{x}$ in lex order in P_n^w. Indeed, such costs are inevitable and difficult to estimate in bit operations. However, it seems common in the literature to estimate the time complexity of the above in-place Möbius transform as exactly $n \cdot |P_n^w|$ bit operations, i.e., a theoretic value.

Remark 4. If the polynomial f is of degree d, Dinur and Shamir have shown in [7] that the time complexity of the in-place Möbius transform can be reduced to $\mathcal{O}(d \cdot |P_n^w|)$ bit operations, but the memory complexity is still $\mathcal{O}(|P_n^w|)$ bits. The main idea is to exploit the gap between w and d as many entries in the initial array A are zero and they are changed at very late steps.

B Example for FESG

Let $n = 3$ and

$$f(x_3, x_2, x_1) = x_3x_2 \oplus x_2x_1 \oplus x_2 \oplus x_1.$$

Since $d = 2$, we need to prepare $\binom{3}{\leq 2} = 7$ different tables:

$$A_0, A_{0,1}, A_{0,2}, A_{0,3}, A_{0,1,2}, A_{0,1,3}, A_{0,2,3},$$

and initialize them as follows:

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

We consider evaluating this f on all inputs in $\mathbb{F}_2^3$ and the corresponding evaluation phase is shown in Table 2.

Table 2: Example of our algorithm for polynomial evaluations

(x_3, x_2, x_1)	$(k_h, \ldots, k_1)$	Update	$f(x_3, x_2, x_1)$
$(0,0,0)$	–	–	$A_0[0] = 0$
$(0,0,1)$	(1)	$A_0[1] = A_0[0] \oplus A_{0,1}[0] = 0 \oplus 1 = 1$	$A_0[1] = 1$
$(0,1,0)$	(2)	$A_0[2] = A_0[0] \oplus A_{0,2}[0] = 0 \oplus 1 = 1$	$A_0[2] = 1$
$(0,1,1)$	$(2,1)$	$A_{0,1}[1] = A_{0,1}[0] \oplus A_{0,1,2}[0] = 1 \oplus 1 = 0$ $A_0[1] = A_0[2] \oplus A_{0,1}[1] = 1 \oplus 0 = 1$	$A_0[1] = 1$
$(1,0,0)$	(3)	$A_0[3] = A_0[0] \oplus A_{0,3}[0] = 0 \oplus 0 = 0$	$A_0[3] = 0$
$(1,0,1)$	$(3,1)$	$A_{0,1}[2] = A_{0,1}[0] \oplus A_{0,1,3}[0] = 1 \oplus 0 = 1$ $A_0[1] = A_0[3] \oplus A_{0,1}[2] = 0 \oplus 1 = 1$	$A_0[1] = 1$
$(1,1,0)$	$(3,2)$	$A_{0,2}[1] = A_{0,2}[0] \oplus A_{0,2,3}[0] = 1 \oplus 1 = 0$ $A_0[2] = A_0[3] \oplus A_{0,2}[1] = 0 \oplus 0 = 0$	$A_0[2] = 0$
$(1,1,1)$	$(3,2,1)$	$A_{0,1}[1] = A_{0,1}[2] \oplus A_{0,1,2}[0] = 1 \oplus 1 = 0$ $A_0[1] = A_0[2] \oplus A_{0,1}[1] = 0 \oplus 0 = 0$	$A_0[1] = 0$

C Examples for Understanding Algorithm 1

Example 1. We first take the method in [4] as an instance, and consider evaluating a quadratic polynomial $f(x_3, x_2, x_1)$ in the full space $\mathbb{F}_2^3$, i.e., $d = 2$ and $n = 3$. Then, the ordered set S_0 (i.e., Gray code G_3 for $\mathbb{F}_2^3$) is

$$S_0 = \{(0,0,0), (0,0,1), (0,1,1), (0,1,0), (1,1,0), (1,1,1), (1,0,1), (1,0,0)\}.$$

It can be observed that any two consecutive elements in S_0 differs in only 1 bit. The evaluation rule $\mathcal{R}$ over a set S is defined as:

$$\forall j \in [1, |S| - 1] : \; S[j-1] \xrightarrow{\mathcal{R}} S[j].$$

With Algorithm 1, by setting $S_{0,i} = \texttt{NEXTSET}(S_0, \mathcal{R}, i)$, we can get

$$\begin{aligned} S_{0,1} &= \{(0,0,0), (0,1,0), (1,1,0), (1,0,0)\}, \\ S_{0,2} &= \{(0,0,1), (1,0,1)\}, \\ S_{0,3} &= \{(0,1,0)\}, \end{aligned}$$

which are the input sets to polynomials $D_{0,1}(f), D_{0,2}(f), D_{0,3}(f)$, respectively.

It can be observed that any two consecutive elements in $S_{0,1}, \ldots, S_{0,3}$ still differ in 1 bit. Therefore, the same evaluation as $\mathcal{R}$ rule can be reused for the sets $S_{0,1}, S_{0,2}, S_{0,3}$. With Algorithm 1 for these 3 sets, we can similarly get

$$S_{0,1,2} = \{(0,0,0), (1,0,0)\}, S_{0,1,3} = \{(0,1,0)\}, S_{0,2,3} = \{(0,0,1)\}$$

which are the input sets for polynomials $D_{0,1,2}(f), D_{0,1,3}(f), D_{0,2,3}(f)$, respectively. Note that $D_{0,1,2}(f), D_{0,1,3}(f), D_{0,2,3}(f)$ are constants since the degree of f is 2.

For the initialization phase, the following 4 outputs have to be computed and stored:

$$f(0,0,0),\ D_{0,1}(f)(0,0,0),\ D_{0,2}(f)(0,0,1),\ D_{0,3}(f)(0,1,0).$$

Example 2. We then consider the method in [8] and we still use $n = 3$ and $d = 2$. Then, the ordered set S_0 is

$$S_0 = \{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1).\}$$

The evaluation rule $\mathcal{R}$ over S is defined as

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

As a result, by setting $S_{0,i} = \texttt{NEXTSET}(S_0, \mathcal{R}, i)$, we have

$$\begin{aligned} S_{0,1} &= \{(0,0,0),(0,1,0),(1,0,0),(1,1,0)\},\\ S_{0,2} &= \{(0,0,0),(1,0,0)\},\\ S_{0,3} &= \{(0,0,0)\}. \end{aligned}$$

As the degree of f is 2, we only need to consider the non-constant polynomials $D_{0,1}(f), D_{0,2}(f), D_{0,3}(f)$, whose input sets are $S_{0,1}, S_{0,2}, S_{0,3}$, respectively.

At the initialization phase, it is required to compute

$$f(0,0,0),\ D_{0,1}(f)(0,0,0),\ D_{0,2}(f)(0,0,0),\ D_{0,3}(f)(0,0,0).$$

D Impact of the Sufficient Condition

We first recall Gray code and monotone Gray code[7].

Definition 6 (Gray Code). *The Gray code (also called reflected binary Gray code) for n-bit binary vectors denoted by G_n is an ordered set of all $\boldsymbol{x} \in \mathbb{F}_2^n$ where*

$$\forall \boldsymbol{x} \in \mathbb{F}_2^n :\ G_n[INT(\boldsymbol{x})] = \boldsymbol{x} \oplus (\boldsymbol{x} >> 1).$$

In particular, G_n has the following property:

$$\forall j \in [1, 2^n - 1] :\ \mathrm{wt}(G_n[j-1] \oplus G_n[j]) = 1.$$

Monotone Gray code was proposed in [13], which was explained from the perspective of a graph. Here, we are only interested in its construction, and give an equivalent description from the perspective of sets of elements in $\mathbb{F}_2^n$.

[7] A related lecture is also available at https://ktiml.mff.cuni.cz/~gregor/hypercube/hypercubes_lec10.pdf.

Definition 7 (Construction of Monotone Gray Code [13]). *Let* $V_{0,1}$ *be an ordered set of size* $s_{0,1} = 2$ *and*

$$V_{0,1} = \{\boldsymbol{x}, \boldsymbol{y}\} \text{ where } \mathrm{wt}(\boldsymbol{x} \oplus \boldsymbol{y}) = 1, 0 \leq \mathrm{wt}(\boldsymbol{x}), \mathrm{wt}(\boldsymbol{y}) \leq 1.$$

Let $V_{i,i+1}$ $(1 \leq i \leq n-1)$ *be an ordered set of size* $s_{i,i+1} \geq 2$*, where*

$$\begin{cases} \mathrm{wt}(V_{i-1,i}[s_{i-1,i}-1] \oplus V_{i,i+1}[0]) = 1 \\ \mathrm{wt}(V_{i,i+1}[j] \oplus V_{i,i+1}[j+1]) = 1 \text{ for } j \in [0, s_{i,i+1}-2] \\ \mathrm{wt}(V_{i,i+1}[j]) \in [i, i+1] \text{ for } j \in [0, s_{i,i+1}-1] \\ V_{i-1,i} \cap V_{i,i+1} = \emptyset, \\ \{\boldsymbol{x} \in \mathbb{F}_2^n \mid \mathrm{wt}(\boldsymbol{x}) = i\} \subseteq V_{i-1,i} \cup V_{i,i+1}. \end{cases} \tag{20}$$

Then, the ordered set MG_n *is the monotone Gray code for all* $\boldsymbol{x} \in \mathbb{F}_2^n$*, where*

$$\begin{aligned} MG_n = \{\ & V_{0,1}[0], \ldots, V_{0,1}[s_{0,1}-1], \ldots, V_{i-1,i}[0], \ldots, V_{i-1,i}[s_{i-1,i}-1], \\ & V_{i,i+1}[0], \ldots, V_{i,i+1}[s_{i,i+1}-1], \ldots, V_{n-1,n}[0], \ldots, V_{n-1,n}[s_{n-1,n}-1]\}. \end{aligned}$$

It is easy to observe that the ordered set

$$MG_{n,i} = \{V_{0,1}[0], \ldots, V_{0,1}[s_{0,1}-1], \ldots, V_{i,i+1}[0], \ldots, V_{i,i+1}[s_{i,i+1}-1]\}$$

has the property $\{\boldsymbol{x} \in \mathbb{F}_2^n \mid \mathrm{wt}(\boldsymbol{x}) \leq i\} \subseteq MG_{n,i}$.

For example, the Gray code for 4-bit binary vectors is shown as follows:

$$\begin{aligned} G_4 = \{\ & (0,0,0,0), (0,0,0,1), (0,0,1,1), (0,0,1,0), \\ & (0,1,1,0), (0,1,1,1), (0,1,0,1), (0,1,0,0), \\ & (1,1,0,0), (1,1,0,1), (1,1,1,1), (1,1,1,0), \\ & (1,0,1,0), (1,0,1,1), (1,0,0,1), (1,0,0,0)\}. \end{aligned}$$

One example of the monotone Gray code for 4-bit binary vectors is shown as follows:

$$\begin{aligned} MG_4 = \{\ & \underbrace{(0,0,0,0), (0,0,0,1)}_{V_{0,1}}, \\ & \underbrace{(0,0,1,1), (0,0,1,0), (0,1,1,0), (0,1,0,0), (1,1,0,0), (1,0,0,0)}_{V_{1,2}}, \\ & \underbrace{(1,0,1,0), (1,0,1,1), (1,0,0,1), (1,1,0,1), (0,1,0,1), (0,1,1,1)}_{V_{2,3}}, \\ & \underbrace{(1,1,1,1), (1,1,1,0)}_{V_{3,4}}\}. \end{aligned}$$

Based on the construction of monotone Gray code, we propose a useful lemma that can later be used to prove that it is either infeasible or nontrivial to adapt the FES algorithm [4] with monotone Gray code, which is stated as an interesting question by Dinur in [6].

Lemma 4. *When* $|V_{i,i+1}| = s_{i,i+1} \geq n+2$, *for any* $0 \leq j \leq s_{i,i+1} - n - 2$, *among the* $(n+1)$ *ordered pairs of two consecutive elements in* $n+2$ *consecutive elements* $V_{i,i+1}[j], \ldots, V_{i,i+1}[j+n+1]$ *in* $V_{i,i+1}$, *there are at least two ordered pairs* $(V_{i,i+1}[j_1], V_{i,i+1}[j_1+1])$ *and* $(V_{i,i+1}[j_2], V_{i,i+1}[j_2+1])$ *satisfying*

$$j_1, j_2 \in [j, j+n],\ j_1 \neq j_2,\ V_{i,i+1}[j_1] \oplus V_{i,i+1}[j_1+1] = V_{i,i+1}[j_2] \oplus V_{i,i+1}[j_2+1].$$

In particular, let $V_{i,i+1}[j_1] \oplus V_{i,i+1}[j_1+1] = \boldsymbol{e}_k$ *for each such* (j_1, j_2). *Then, we have*

$$V_{i,i+1}[j_1] \wedge \overline{\boldsymbol{e}_k} = V_{i,i+1}[j_1+1] \wedge \overline{\boldsymbol{e}_k}, V_{i,i+1}[j_2] \wedge \overline{\boldsymbol{e}_k} = V_{i,i+1}[j_2+1] \wedge \overline{\boldsymbol{e}_k},$$

and $\mathrm{wt}(V_{i,i+1}[j_1+1] \wedge \overline{\boldsymbol{e}_k}) = \mathrm{wt}(V_{i,i+1}[j_2+1] \wedge \overline{\boldsymbol{e}_k}) = i$.

Proof. Based on the constraints in Equation 20, we have that

$$\forall j \in [0, s_{i,i+1} - 2] :\ \mathrm{wt}(V_{i,i+1}[j] \oplus V_{i,i+1}[j+1]) = 1.$$

If $|V_{i,i+1}| = s_{i,i+1} \geq n+2$, for any $0 \leq j \leq s_{i,i+1} - n - 2$, there are $n+2$ consecutive elements $V_{i,i+1}[j], \ldots, V_{i,i+1}[j+n+1]$ in $V_{i,i+1}$. Hence, there will be $n+1$ integers $(k_j, \ldots, k_{j+n})$ $(1 \leq k_j, \ldots, k_{j+n} \leq n)$ such that

$$\forall j_3 \in [j, j+n] :\ V_{i,i+1}[j_3] \oplus V_{i,i+1}[j_3+1] = \boldsymbol{e}_{k_{j_3}}.$$

The $(n+1)$ ordered pairs of 2 consecutive elements are $(V_{i,i+1}[j], V_{i,i+1}[j+1]), \ldots, (V_{i,i+1}[j+n], V_{i,i+1}[j+n+1])$. Due to $1 \leq k_j, \ldots, k_{j+n} \leq n$ and $\boldsymbol{e}_{k_{j_3}} \in \{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$, there must exist $j_1, j_2 \in [j, j+n]$ and $j_1 \neq j_2$ such that $k_{j_1} = k_{j_2}$. In other words,

$$\exists j_1, j_2 \in [j, j+n], j_1 \neq j_2 :\ V_{i,i+1}[j_1] \oplus V_{i,i+1}[j_1+1] = V_{i,i+1}[j_2] \oplus V_{i,i+1}[j_2+1].$$

For such (j_1, j_2), let $V_{i,i+1}[j_1] \oplus V_{i,i+1}[j_1+1] = \boldsymbol{e}_k$. Then, we have

$$V_{i,i+1}[j_1] \oplus V_{i,i+1}[j_1+1] = V_{i,i+1}[j_2] \oplus V_{i,i+1}[j_2+1] = \boldsymbol{e}_k.$$

In other words, both pairs $(V_{i,i+1}[j_1], V_{i,i+1}[j_1+1])$ and $(V_{i,i+1}[j_2], V_{i,i+1}[j_2+1])$ differ in the k-th bit. Therefore, by setting the k-bit bit of each pair to 0, they become identical, resulting in

$$V_{i,i+1}[j_1] \wedge \overline{\boldsymbol{e}_k} = V_{i,i+1}[j_1+1] \wedge \overline{\boldsymbol{e}_k}, V_{i,i+1}[j_2] \wedge \overline{\boldsymbol{e}_k} = V_{i,i+1}[j_2+1] \wedge \overline{\boldsymbol{e}_k}.$$

Moreover, due to $\mathrm{wt}(V_{i,i+1}[j_1]), \mathrm{wt}(V_{i,i+1}[j_1+1]), \mathrm{wt}(V_{i,i+1}[j_2]), \mathrm{wt}(V_{i,i+1}[j_2+1]) \in \{i, i+1\}$, we have $\mathrm{wt}(V_{i,i+1}[j_1+1] \wedge \overline{\boldsymbol{e}_k}) = \mathrm{wt}(V_{i,i+1}[j_2+1] \wedge \overline{\boldsymbol{e}_k}) = i$. □

Based on the sufficient condition, we prove that it is either infeasible or nontrivial to use monotone Gray code in the FES algorithm [4].

Let the evaluation rule $\mathcal{R}_{\mathrm{GC}}$ over an ordered set S be defined as

$$\forall j \in [1, |S| - 1] :\ S[j-1] \xrightarrow{\mathcal{R}_{\mathrm{GC}}} S[j].$$

Lemma 5. *[4] Let $S_0 = G_n$ be an ordered set of size 2^n, where G_n is the Gray code for all $\boldsymbol{x} \in \mathbb{F}_2^n$. For any ordered set $S_{k_0,\ldots,k_i}$ obtained with*

$$S_{k_0,\ldots,k_i} = \texttt{NEXTSET}(S_{k_0,\ldots,k_{i-1}}, \mathcal{R}_{GC}, k_i),$$

it always satisfies that if $|S_{k_0,\ldots,k_i}| > 1$, then.

$$\forall j \in [1, |S_{k_0,\ldots,k_i}| - 1]: \ \mathrm{wt}(S_{k_0,\ldots,k_i}[j-1] \oplus S_{k_0,\ldots,k_i}[j]) = 1.$$

Lemma 5 indicates that by setting any evaluation rule $R_{k_0,\ldots,k_i}$ as R_{GC}, each ordered input set (of size > 1) of the polynomial $D_{k_0,\ldots,k_i}(f)$ has the property that each pair of consecutive elements in the set only differ in 1 bits, thus satisfying the sufficient condition under the evaluation rule R_{GC}. As a result, the original FES algorithm [4] can work for any degree due to this property under the same evaluation rule R_{GC}.

In the following, we show that for $d \geq 2$, this is no longer the case when changing S_0 to $MG_{n,w}$ for any $w \geq 2$. In other words, it becomes difficult to achieve the time complexity $d \cdot \binom{n}{\leq w}$ when using the FES algorithm with monotone Gray code to evaluate f over P_n^w for $w \geq 2$.

Theorem 2. *Let $S_0 = MG_{n,w}$ and $S_{0,i} = \texttt{NEXTSET}(S_0, \mathcal{R}_{GC}, i)$ for $i \in [1, n]$. When $n \geq 4$ and $w \geq 2$, there must exist $S_{0,i}$ such that*

$$\exists j \in [1, |S_{0,i}| - 1]: \ \mathrm{wt}(S_{0,i}[j-1] \oplus S_{0,i}[j]) \neq 1.$$

Proof. By definition,

$$MG_{n,w} = \{V_{0,1}[0], \ldots, V_{0,1}[s_{0,1} - 1], \ldots, V_{i,i+1}[0], \ldots, V_{w,w+1}[s_{w,w+1} - 1]\}.$$

Due to $\{\boldsymbol{x} \in \mathbb{F}_2^n \mid \mathrm{wt}(\boldsymbol{x}) = k\} \subseteq V_{k-1,k} \cup V_{k,k+1}$, we have[8] $s_{1,2} = 2(n-1) \geq n+2$ for $n \geq 4$. In addition, we also have

$$s_{0,1} = 2, s_{n-1,n} = 2, s_{k-1,k} + s_{k,k+1} \geq \binom{n}{k}, \sum_{k=0}^{n-1} s_{k,k+1} = 2^n.$$

Therefore, as n increases, there are many $k \in [1, w]$ satisfying $s_{k,k+1} \geq n+2$. Anyway, when $n \geq 4$ and $w \geq 2$ (so that $V_{2,3}$ is also included in $MG_{n,w}$), there exists $k \in [1, w]$ such that $s_{k,k+1} \geq n+2$.

Based on Lemma 4, for any $n+2$ consecutive elements in such an ordered set $V_{k,k+1}$, there must exist at least 2 different pairs of 2 consecutive elements denoted by $(\boldsymbol{x}, \boldsymbol{y}), (\boldsymbol{x}', \boldsymbol{y}') \neq (\boldsymbol{x}, \boldsymbol{y})$ where $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}', \boldsymbol{y}' \in V_{k,k+1}$ such that

$$\exists i \in [1, n]: \ \boldsymbol{x} \xrightarrow{\mathcal{R}_{\mathrm{GC}}} \boldsymbol{y}, \boldsymbol{x}' \xrightarrow{\mathcal{R}_{\mathrm{GC}}} \boldsymbol{y}', \boldsymbol{x} \oplus \boldsymbol{y} = \boldsymbol{x}' \oplus \boldsymbol{y}' = \boldsymbol{e}_i.$$

[8] Note that $s_{0,1} = 2$, resulting in $s_{1,2} = 2(n-1)$ or $s_{1,2} = 2(n-1) - 1$. In this case of $s_{1,2} = 2(n-1) - 1$, the last element in $V_{1,2}$ must have Hamming weight 1, and hence the first element in $V_{2,3}$ must have Hamming weight 2. We can thus equivalently treat the first element in $V_{2,3}$ as the last element in $V_{1,2}$, resulting in also $s_{1,2} = 2(n-1)$.

As a result, according to the function NEXTSET in Algorithm 1, $\boldsymbol{y}^{(i)}$ and $(\boldsymbol{y}')^{(i)}$ will be added to $S_{0,i}$. Moreover, $\boldsymbol{y}^{(i)} = \boldsymbol{y} \wedge \overline{\boldsymbol{e}_i}$ and $(\boldsymbol{y}')^{(i)} = \boldsymbol{y}' \wedge \overline{\boldsymbol{e}_i}$ are consecutive in $S_{0,i}$. Based on Lemma 4, $\mathrm{wt}(\boldsymbol{y} \wedge \overline{\boldsymbol{e}_i}) = \mathrm{wt}(\boldsymbol{y}' \wedge \overline{\boldsymbol{e}_i}) = k$, resulting in $\mathrm{wt}(\boldsymbol{y}^{(i)} \oplus (\boldsymbol{y}')^{(i)}) \neq 1$. □

Theorem 2 indicates that using $\mathcal{R}_{\mathrm{GC}}$ for f, there is at least one polynomial $D_{0,i}(f)$ whose ordered input set $S_{0,i}$ does not preserve the property that any 2 consecutive elements differ in only 1 bit. Hence, $S_{0,i}[j-1] \xrightarrow{\mathcal{R}_{\mathrm{GC}}} S_{0,i}[j]$ does not always hold, i.e., due to $S_{0,i}[j-1] \oplus S_{0,i}[j] \notin \{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$. Indeed, as n increases, there are many (not only 2) consecutive elements in $S_{0,i}$ such that they differ in more than 1 bit. This can also be easily observed from the proof of Lemma 4. Specifically, there are $|V_{k,k+1}|$ consecutive elements that form $|V_{k,k+1}| - 1$ consecutive pairs $(\boldsymbol{x}, \boldsymbol{y})$ of 2 consecutive elements in $V_{k,k+1}$, where each $(\boldsymbol{x}, \boldsymbol{y})$ satisfies $\mathrm{wt}(\boldsymbol{x} \oplus \boldsymbol{y}) = 1$. Hence, there will be at least $\lceil \frac{|V_{k,k+1}|-1}{n} \rceil$ consecutive elements in a set $S_{0,i}$ whose Hamming weight is always k. It is also possible that there are many such input sets $S_{0,i}$. All of these are difficult to predict without explicitly constructing each input set for $D_{0,i}(f)$ from $MG_{n,w}$ under the evaluation rule $\mathcal{R}_{\mathrm{GC}}$.

Based on Theorem 2, in order to combine the FES algorithm with monotone Gray code, the same evaluation rule $\mathcal{R}_{\mathrm{GC}}$ can no longer be used to evaluate each $D_{k_0,\ldots,k_i}(f)$. Hence, we claim that it is either infeasible to use monotone Gray code in the FES algorithm, or it is nontrivial to find a proper evaluation order $\mathcal{R}_{k_0,\ldots,k_i}$ for each polynomial $D_{k_0,\ldots,k_i}(f)$ that can satisfy the sufficient condition. This is mainly due to the fundamental structural difference between Gray code and monotone Gray code.

E Big Picture of the Full Implementation

Listing 1.3: Big picture of our implementation

```
W = [w_1,...,w_s]
li = [n_1,...,n_s]
n  = sum(li)
s  = len(li)
d  = degree_of_f
N = [0] * (s) #used in function enumerationG()
G = [0] * (n + 1) #used in function enumerationG()
k = [0] * (d + 2) #used in function evaluation()
ind = [0] * (d + 1) #used in function evaluation()

def setN():
    N[0] = li[0]
    for i in range(1,s):
        N[i] = li[i] + N[i-1]

def setG():
    i = 0
    for j in range(1, n + 1):
        while j > N[i]:
            i += 1
        G[j] = i

BCOF = [] #binomial coefficients
def build_binom(n, d):
    #check the full code: construct BCOF such that BCOF[i][j] = binom{i}{j}

B = [] #2-dimensional dynamic array
def initializationPhasef:
    #check the full code: read coefficients of f and initialize B

def evaluationPhase():
    setN()
    setG()
    K = [0] * (n+1)
    H = [0] * (s)
    h = 0
    end = n
    f(0^n)= B[0][0]
    enumerationG(K,H,h,end)
```