# A Memory-efficient Implementation of Wiedemann XL

Daniel Cabarcas[1], Tung Chou[2], Felipe Merino[1] and Javier Verbel[3]

[1] Universidad Nacional de Colombia, Colombia
dcabarc@unal.edu.co, fmerino@unal.edu.co
[2] Academia Sinica, Taiwan
blueprint@crypto.tw
[3] Technology Innovation Institute, UAE
javier.verbel@tii.ae

**Abstract.** The signature schemes MQOM, SNOVA, MAYO, and UOV are third-round candidates in the NIST standardization process for additional post-quantum signatures, and their security relies on the hardness of the multivariate quadratic (MQ) problem. XL with block Wiedemann (BW) is one of the most efficient MQ solvers in practice, but its memory demand has led to a debate about its memory access cost: generating the rows of the Macaulay matrix on the fly is widely believed to save memory at a non-negligible computational cost.
In this work, we evaluate the memory–time trade-off in practice. We provide the first efficient on-the-fly implementation of XL with BW. We build on the state-of-the-art implementation by Cheng, Chou, Niederhagen, and Yang (CCNY), which uses SSE instructions and supports the fields $\mathbb{F}_2$, $\mathbb{F}_{16}$ and $\mathbb{F}_{31}$; we remove its column-index tables and generate the rows on the fly in the first phase of BW (BW1). We amortize the cost of on-the-fly generation with a new algorithm that computes column indices with $O(n^2)$ instead of $O(n^3)$ table look-ups, where $n$ is the number of variables. In our experiments we observe that our on-the-fly implementation achieves running times comparable to CCNY's with up to five times less memory. We also port both implementations to AVX2 over $\mathbb{F}_{256}$, the main field of UOV, and observe a similar behavior: comparable BW1 running times with less memory for the on-the-fly variant.

## 1 Introduction

The MQ problem asks to find a common root of a system of $m$ quadratic polynomials in $n$ variables over a finite field $\mathbb{F}_q$ of size $q$. In cryptography, solving an MQ instance is typically the main step of algebraic attacks. In post-quantum cryptography (PQC), the hardness of MQ is the main security assumption of MQOM, SNOVA, MAYO, and UOV, which are third-round candidates in the NIST PQC standardization process for additional digital signature schemes.
The XL algorithm is one of the most efficient algorithms for solving unstructured MQ instances in practice, especially over mid-size fields such as $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$, which are the fields used by these schemes. Roughly speaking, XL first extends the input system to a degree $D$ by multiplying the input polynomials by monomials of degree up to $D-2$, and builds the coefficient matrix of the extended system, called the Macaulay matrix. Each row of this matrix represents a product $\mu f$, where $f$ is an input polynomial and $\mu$ a monomial of degree at most $D-2$. Then, it finds a vector in the kernel of a submatrix of the Macaulay matrix, from which a common root of the input system can be extracted whenever one exists. Typically, this kernel vector is found with the block Wiedemann (BW) algorithm [Cop94], which works with a square submatrix $M$ of the Macaulay matrix. In its first phase, which dominates the

running time and which we refer to as BW1, BW computes the sequence $\{M^i Z\}$, where $Z$ is a randomly chosen matrix, and the memory use of BW1 is dominated by the storage of $M$ and $M^i Z$. Only the current $M^i Z$ is stored.

Cheng, Chou, Niederhagen, and Yang [CCNY12] developed the state-of-the-art implementation of XL with BW, which we refer to as the CCNY implementation and which uses SSE instructions. This implementation currently holds the record of the Fukuoka MQ Challenge for Type III systems ($m = 2n$, over $\mathbb{F}_{31}$) [YDH+15], and it was used to break the Level I parameters of Rainbow from the second round of the NIST PQC standardization process [Beu22]. The CCNY implementation reduces the memory footprint of $M$ by storing it in a compact way. More precisely, it takes advantage of the fact that each row of $M$ corresponds to a polynomial $\mu f$, where $\mu$ is a monomial of degree at most $D-2$ and $f$ is one of the input quadratic polynomials. Let $\iota(\cdot)$ denote the map that sends a monomial to the index of its column in $M$. The CCNY implementation stores the column index $\iota(\mu \cdot \mu')$ for every monomial $\mu$ of degree at most $D-2$ and every monomial $\mu'$ of degree at most 2. For a fixed monomial $\mu$, the $\binom{n+2}{2}$ indices of the form $\iota(\mu \cdot \mu')$ give the column positions of the non-zero entries in the row corresponding to $\mu f$, and the values in these entries are the coefficients of $f$. The CCNY implementation uses less memory than Gröbner basis algorithms such as F4 and F5, which echelonize submatrices of the Macaulay matrix of increasing degrees and lose sparsity at each stage. Still, the CCNY implementation remains memory-intensive.

The large memory requirement of XL has led to a debate in the cryptographic community about its memory access cost. For instance, Beullens's MinRank attack at Eurocrypt 2021 [Beu21] uses XL to solve the Support-Minors equations, and in response the Rainbow team argued that the attack does not break the scheme once memory costs are properly accounted for [The20]. Later, at Crypto 2022, Baena et al. [BBC+22] proposed a strategy to reduce the memory access cost of this MinRank attack and gave a concrete complexity analysis that accounts for the memory access cost, the parallelization cost, and the cost of generating Macaulay matrix rows on the fly, i.e., constructing a row of $M$ only when it is needed to perform one row–matrix multiplication during the computation of $M^i Z$.

It is widely believed that generating the rows of $M$ on the fly reduces the memory demand of XL at the cost of a non-negligible computational overhead [The20, BBC+22, DGY26]. This trade-off came up again at Eurocrypt 2026 in Ran's wedge attack on UOV and MAYO [Ran25], which ultimately relies on solving MQ. Ran states that any XL algorithm based on Wiedemann's algorithm has to either store the entire sparse matrix or recompute its entries each time they are needed, which adds a computational overhead.

In this work, we aim to evaluate in practice the memory–time trade-off of solving MQ with the block Wiedemann XL algorithm. To the best of our knowledge, this trade-off has so far been estimated only through theoretical arguments [The20, BBC+22, DGY26].

## 1.1 Our Contribution

To the best of our knowledge, we provide the first efficient on-the-fly (OTF) implementation of the XL algorithm for solving the MQ problem. We observe that, for the parameter sets we tested, the computational overhead of generating the rows of $M$ on the fly can be eliminated in practice. Our efficient OTF implementation achieves running times comparable to those of the original CCNY implementation, which stores the square matrix $M$ and which we refer to as the not-on-the-fly (NOTF) implementation. Moreover, our efficient OTF implementation reduces memory use by up to a factor of five, see Table 4.

Our implementation builds upon the CCNY implementation. In a first implementation, denoted by OTF-SSE, we remove the step in CCNY that computes and stores all column indices $\iota(\mu \cdot \mu')$ corresponding to non-zero entries of $M$. Instead, OTF-SSE computes $I_\mu = (\iota(\mu \cdot \mu'_0), \ldots, \iota(\mu \cdot \mu'_{N_2-1}))$, where $N_d = \binom{n+d}{d}$ denotes the number

of monomials of degree at most $d$, and $\mu'_0, \ldots, \mu'_{N_2-1}$ are the monomials of degree at most 2, whenever a row $\mu f$ of $M$ needs to be multiplied by $M^{i-1}Z$ during the computation of $M^iZ = M(M^{i-1}Z)$. OTF-SSE reduces the number of evaluations of $\iota(\cdot)$ by computing $I_\mu$ once and reusing it for all rows of $M$ of the form $\mu f$. Moreover, this implementation uses a specialized parallelization strategy that minimizes the number of unranking operations, i.e., computations of the $k$-th monomial $\mu_k$ of degree at most $D-2$ from its rank $k$. In this strategy, the rows of $M$ are distributed in blocks of rows that share the same extension monomial $\mu$, that is, rows in a set $\{\mu f_1, \ldots, \mu f_m\}$, where $f_1, \ldots, f_m$ are the input polynomials. OTF-SSE reduces the memory demand of storing the matrix $M$ from $O(N_{D-2} \cdot N_2)$ integers of $\lceil \log_2(N_D) \rceil$ bits to $O(mn^2)$ $\mathbb{F}_q$-elements. However, it comes at a significant performance cost: OTF-SSE is up to 3.5 times slower than the CCNY implementation on the parameter sets we tested, see Table 2.

We propose a second implementation, denoted by OTF-SSE$^+$, that improves the efficiency OTF-SSE while maintaining its memory savings. The improvement comes from an asymptotically faster algorithm for computing the column indices $I_\mu$. In the CCNY implementation, computing $I_\mu$ takes $O(n^3)$ table look-ups and $O(n^3)$ integer additions. We develop a new algorithm that computes all indices in $I_\mu$ with only $O(n^2)$ table look-ups and $O(n^2)$ integer additions. We benchmark OTF-SSE$^+$ against the CCNY implementation and we observe that OTF-SSE$^+$ achieves running times comparable to those of the CCNY implementation with a memory footprint up to four times smaller, see Tables 2 to 4.

We benchmark our OTF-SSE$^+$ implementation against the CCNY one, showing that ours achieves comparable time performance with a smaller memory footprint, see Tables 2 to 4. Our OTF-SSE$^+$ implementation also achieves a speed-up for shared-memory parallelization among cores that is comparable to CCNY's, see Figure 3.

Finally, we port the NOTF implementation and the OTF-SSE$^+$ to AVX2 over $\mathbb{F}_{256}$, which we denote by NOTF-AVX2 and OTF-AVX2$^+$. To the best of our knowledge, no efficient implementation of XL supported $\mathbb{F}_{256}$ before this work, although it is the field used for all but one of the parameter sets of UOV [BCD+26]. For $m = 2n$ and $D = 6$, the OTF-AVX2$^+$ needs between 0.73 and 0.92 of the memory required by NOTF-AVX2, depending on the block size. In BW1, the running times of OTF-AVX2$^+$ and NOTF-AVX2 are comparable, see Figure 4 and Table 5.

## 1.2 Related Work

The CCNY implementation [CCNY12], which is the starting point for our work, is described in Section 3.

Mohamed et al. [MDK+10] developed a parallel Wiedemann XL implementation (PWXL) that uses the BW code of Kleinjung et al., which was used in the matrix step of the number field sieve for the factorization of RSA-768 [KAF+10]. The RSA-768 matrix was an unstructured sparse matrix over $\mathbb{F}_2$ with 192 million rows, and it occupied 105 GB; BW1 took three-fifths of the 119 calendar days spent on BW. In contrast, the Macaulay matrix in XL is highly structured, and a Macaulay matrix of similar size can be stored in less than one MB. Hence, such a generic BW code is not designed to exploit the structure of the Macaulay matrix.

Other works improve the parallelization of BW for NFS [SAD13, RDC22, RG22]. Schmidt et al. [SAD13] address the CPU–GPU communication and the storage of rows with widely varying weights, and Roy et al. [RDC22] balance the workload by column weight for the kernel $A^t(Av)$. None of these issues arises in our CPU-only setting, where every row of $M$ has at most $\binom{n+2}{2}$ non-zero entries and each iteration of BW1 requires a single product by $M$. Rawal and Gupta [RG22] broadcast the dense matrix to all MPI processes with `MPI_Bcast`, so the memory of every process grows with the full dimension of the matrix.

## 1.3 Paper Organization

The rest of the paper is organized as follows. Section 2 recalls monomial indexing, the XL algorithm, and the block Wiedemann algorithm. Section 3 describes the CCNY implementation and analyzes the memory used by BW1. Section 4 presents our on-the-fly generation of the Macaulay matrix. Section 5 reports our experiments, and Section 6 concludes.

# 2 Preliminaries

## 2.1 Monomials and Monomial Ordering

In this paper, we consider two polynomial rings $R = \mathbb{F}_q[x_1, \ldots, x_n]$ and $\mathcal{R} = R/\langle x_1^2 - x_1, \ldots, x_n^2 - x_n\rangle$. We denote by $\mathcal{M}_d$ the set of monomials of degree at most $d$, and we define $N_d \coloneqq |\mathcal{M}_d|$. We will sometimes write $N$ instead of $N_D$.
There are a few standard monomial orderings that can be used to define the order of monomials. We use the *graded reverse lexicographic order* (grevlex). In this ordering, given two monomials $\mu = x_1^{d_1} \cdots x_n^{d_n}$ and $\mu' = x_1^{d'_1} \cdots x_n^{d'_n}$, $\mu < \mu'$ if and only if one of the following two conditions holds.

1. $\deg(\mu) < \deg(\mu')$.
2. $\deg(\mu) = \deg(\mu')$, and there exists $i \in \{1, \ldots, n\}$ such that $d_j = d'_j$ for all $j > i$ and $d_i > d'_i$.

Let $\mu$ be stored as the array $(d_1, \ldots, d_n)$ of its exponents. There is a standard method, which we call *step*, that overwrites this array with the successor of $\mu$ in grevlex. It scans the array from the left until it finds a nonzero exponent, and then modifies at most three entries. Its worst-case cost is therefore $O(n)$. On average, however, it reads only about $(n + d)/(d + 1)$ entries, where $d$ is the degree of $\mu$, and it never multiplies or looks up binomial coefficients.

## 2.2 Indices of Monomials

The index of a monomial $\mu$ in $R$ (resp. in $\mathcal{R}$) is defined as the number of monomials in $R$ (resp. in $\mathcal{R}$) that are smaller than $\mu$. We denote the index of $\mu$ as $\iota(\mu)$, whose definition depends on which ring is considered.

**The case of $R$.** Let $\mu = x_1^{d_1} \cdots x_n^{d_n} \in R$ be a monomial. It is known that the index $\iota(\mu)$ can be computed as

$$\sum_{i=1}^{n} \sigma(i, \overline{d}_i), \tag{1}$$

where $\overline{d}_i \coloneqq \sum_{j=1}^{i} d_j$ and

$$\sigma(i, d) \coloneqq \begin{cases} \binom{i+d-1}{d-1} & \text{if } d > 0, \\ 0 & \text{otherwise.} \end{cases}$$

**The case of $\mathcal{R}$.** Let $\mu = x_1^{d_1} \cdots x_n^{d_n} \in \mathcal{R}$ be a monomial. For each $i$, $d_i$ is thus either 0 or 1. It is known that the index $\iota(\mu)$ can be computed as

$$\sum_{i=1}^{\deg(\mu)} \binom{n}{i} - \sum_{d_i=1} \binom{i-1}{\overline{d}_i},$$

where $\overline{d}_i \coloneqq \sum_{j=1}^{i} d_j$. This expression can also be written as

$$\sum_{i=1}^{n} \sigma(i, \overline{d}_i, d_i), \tag{2}$$

where

$$\sigma(i,d,b) \coloneqq \begin{cases} 0 & \text{if } b = 0, \\ \binom{n}{d} - \binom{i-1}{d} & \text{if } b = 1. \end{cases}$$

The definitions of $\sigma$ (and $\iota$) here are of course different from those in the case of $R$, but we use the same notation for convenience.

**Unranking** The inverse map $\iota^{-1}$, which recovers a monomial from its index, is usually called *unranking*. For $\mu = x_1^{d_1} \cdots x_n^{d_n}$ set $c_i = \overline{d}_i + i - 1$, $1 \le i \le n$. Since $\overline{d}_1 \le \cdots \le \overline{d}_n$, we have $0 \le c_1 < c_2 < \cdots < c_n$, and since $\binom{i+\overline{d}_i-1}{\overline{d}_i-1} = \binom{c_i}{i}$ (which vanishes when $\overline{d}_i = 0$), Equation (1) becomes

$$\iota(\mu) = \sum_{i=1}^{n} \binom{c_i}{i}. \tag{3}$$

This is the representation of $\iota(\mu)$ in the combinatorial number system of degree $n$ [Leh64]. Hence $\mu \mapsto \{c_1, \ldots, c_n\}$ is a bijection between the monomials of $R$ and the $n$-subsets of $\mathbb{N}$, and $\iota$ is a bijection onto $\mathbb{N}$. Given an index $k$, the $c_i$ are therefore recovered greedily: with $k_n = k$, for $i = n, n-1, \ldots, 1$,

$$c_i = \max\left\{c : \binom{c}{i} \le k_i\right\}, \qquad k_{i-1} = k_i - \binom{c_i}{i},$$

and then $\overline{d}_i = c_i - i + 1$, $d_1 = \overline{d}_1$, and $d_i = \overline{d}_i - \overline{d}_{i-1}$ for $i \ge 2$. If $\deg(\mu) \le D$, then $c_n \le n + D - 1$. Because $c_i < c_{i+1}$, each $c_i$ can be found by a linear scan going down from $c_{i+1} - 1$. The total number of binomial look-ups is then at most $n + D$, using a precomputed table of $\binom{a}{b}$ with $a \le n + D$ and $b \le D$ (using $\binom{c}{i} = \binom{c}{c-i}$). Unranking thus costs $O(n + D)$ operations. Note that, enumerating the monomials in increasing order with *step* is considerably cheaper than computing each of them with $\iota^{-1}$.

## 2.3 The Extended Linearization (XL) Algorithm

The approach underlying the XL algorithm goes back to Lazard [Laz83], who described how systems of polynomial equations can be solved by linear algebra. It was later rediscovered by Courtois, Klimov, Patarin, and Shamir, who used it in cryptanalysis for solving overdefined systems of quadratic equations [CKPS00] and named it the Extended Linearization (XL) algorithm.

For simplicity, we describe the XL algorithm over $R$ with the grevlex order. The description of the algorithm over the quotient ring $\mathcal{R}$ is analogous.

Given an integer $D \ge 2$, let $v_D$ be the vector of all monomials in $R$ of degree at most $D$ sorted in grevlex order.The length of $v_D$ is

$$N_D = \binom{n+D}{D}.$$

Let $F = (f_1, \ldots, f_m) \subset R$ be a sequence of $m$ quadratic polynomials. The degree-$D$ *Macaulay matrix* of $F$, denoted by $M_D$ is the matrix of the coefficients of $\{\mu f : \mu \in v_{D-2}, f \in F\}$, or more precisely, it is the $mN_{D-2} \times N_D$ matrix over $\mathbb{F}_q$ such that

$$v_{D-2} \otimes \begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix} = M_D \cdot v_D.$$

For any $a \in \mathbb{F}_q^n$ satisfying $f_1(a) = \cdots = f_m(a) = 0$, it holds that $M_D \cdot v_D(a) = 0$, where $v_D(a) \in \mathbb{F}_q^{N_D}$ is the *evaluation vector* of $a$ up to degree $D$, i.e. the $i$-th entry of $v_D(a)$ is $\mu(a)$, where $\mu$ is the $i$-th entry of $v_D$.

Given a quadratic polynomial sequence $F$ and an integer $D \geq 2$, XL seeks a vector $w \in \mathbb{F}_q^{N_D}$ in the right kernel of $M_D$ which has a one in its first entry. Then, it outputs the coordinates of $w$ corresponding to the monomials $x_1, \ldots, x_n$.
Assuming that the system $f_1(x) = \cdots = f_m(x) = 0$ has a unique solution and that $D$ is such that the Macaulay matrix $M_D$ has corank 1, the right kernel of $M_D$ is the one-dimensional space spanned by $v_D(a)$ and the solution can be recovered from a vector in the right kernel of $M_D$.
Let $F = (f_1, \ldots, f_m) \subset R$ be a quadratic sequence with a unique solution whose homogenized sequence $F^{(h)} \subset \mathbb{F}_q[x_1, \ldots, x_n, h]$ is semi-regular[1]. Then, the corank of $M_D$ is expected to be 1 when $D$ is the minimum integer such that the coefficient of $t^D$ in the series

$$\frac{(1-t^2)^m}{(1-t)^{n+1}}$$

is non-positive [BFS04].

## 2.4 The Block-Wiedemann (BW) Algorithm

The Block-Wiedemann (BW) algorithm [Cop94], introduced by Coppersmith as a block generalization of Wiedemann's algorithm [Wie86], computes right kernel vectors of a sparse matrix $M \in \mathbb{F}_q^{N \times N}$, i.e., nonzero vectors $w \in \mathbb{F}_q^N$ such that $Mw = 0$. The BW algorithm is parameterized by two block sizes $\tilde{m}$ and $\tilde{n}$ (Wiedemann's algorithm is the case $\tilde{m} = \tilde{n} = 1$) and proceeds in three steps, which we refer to as BW1, BW2, and BW3. Here we follow the description of Kaltofen [Kal95].
**BW1**: In the first step, one chooses a left block $X \in \mathbb{F}_q^{\tilde{m} \times N}$ and a right block $Z \in \mathbb{F}_q^{N \times \tilde{n}}$ and computes the projected block Krylov sequence

$$A_i = XM^iZ \in \mathbb{F}_q^{\tilde{m} \times \tilde{n}}, \quad i = 0, \ldots, \frac{N}{\tilde{m}} + \frac{N}{\tilde{n}} + \frac{2\tilde{n}}{\tilde{m}}.$$

That is, $M$ is applied repeatedly to the right block $Z$, and each iterate $M^iZ$ is projected by the left block $X$ onto the $\tilde{m} \times \tilde{n}$ matrix $XM^iZ$. In practice, Coppersmith sets $X = [I_{\tilde{m}}\ 0]$, where $I_{\tilde{m}}$ is the identity matrix. Moreover, efficient implementations of the BW subroutine of the XL algorithm also make this choice for $X$ [CCNY12, CN26].
**BW2**: In the second step, one computes a nonzero linear generator for this sequence. In particular, one computes vectors $\mathbf{c}_0, \ldots, \mathbf{c}_\ell \in \mathbb{F}_q^{\tilde{n}}$, not all zero, such that

$$\sum_{k=0}^{\ell} A_{j+k}\, \mathbf{c}_k = \mathbf{0} \quad \text{for all } j = 1, \ldots, \eta,$$

where $\ell = \lceil N/\tilde{n} \rceil$ and $\eta = \lceil \tilde{n}(\ell+1)/\tilde{m} \rceil$. This is a homogeneous linear system with a block Toeplitz structure, $\tilde{m}\eta$ equations and $\tilde{n}(\ell+1)$ unknowns, whose coefficient matrix consists of the sequence terms $A_1, \ldots, A_{\ell+\eta}$.
**BW3**: Let $r$ be the smallest index such that $\mathbf{c}_r \neq 0$. In the last step, one computes

$$\hat{w} = Z\mathbf{c}_r + MZ\mathbf{c}_{r+1} + \cdots + M^{\ell-r}Z\mathbf{c}_\ell,$$

which takes at most $\ell - r$ further multiplications of $M$ by a vector. Then, one determines the first $i \leq r+1$ such that $M^i\hat{w} = 0$. Finally, one returns $w = M^{i-1}\hat{w}$.

**Computational Cost.** Now we describe the computational cost of the BW algorithm. BW1 computes the sequence $A_1, \ldots, A_\kappa$ iteratively. In the $i$-th iteration, it computes $M^iZ = M \cdot (M^{i-1}Z)$ and then $XM^iZ$. Let $\omega$ be the average number of non-zero entries in each row of $M$, so that $M$ contains $N\omega$ nonzero entries. Hence, the cost of computing all the $XM^iZ$ is

$$C_{\text{bw1}} = \left(\frac{N}{\tilde{m}} + \frac{N}{\tilde{n}} + \frac{2\tilde{n}}{\tilde{m}} + 1\right) N\omega\tilde{n}.$$

[1] $F^{(h)} = (f_1^{(h)}, \ldots, f_m^{(h)})$, where $f^{(h)} = h^2 f(\frac{x_1}{h}, \ldots, \frac{x_n}{h}) \in \mathbb{F}_q[x_1, \ldots, x_n, h]$.

Assuming $2\tilde{n} \ll N\tilde{m}$, we have

$$C_{\text{bw1}} \approx \left(1 + \frac{\tilde{n}}{\tilde{m}}\right) \omega N^2.$$

Assuming that $\tilde{n} = \tilde{m}$, the cost of BW1 is approximately $2\omega N^2$.
There are several approaches to find the sequence of $\mathbf{c}_i$ in BW2. Coppersmith proposed a block generalization of the Berlekamp–Massey algorithm [Mas69], which has a cost of $O((\tilde{m} + \tilde{n})N^2)$ arithmetic operations. Under the assumption that $\tilde{m}$ and $\tilde{n}$ are of size $O(\log N)$, Thomé proposed an algorithm that computes it in only $O(N^{1+\epsilon})$ field multiplications [Tho02].
In general BW3 requires $(\frac{N}{\tilde{n}} + 2)N\omega\tilde{n} < \omega N^2\tilde{n}$ field multiplications [Kal95], which becomes $O(\omega N^2)$ for $\tilde{n} = O(1)$. Recently, Chou and Niederhagen proposed an optimized version of BW3 that is specialized for the BW subroutine in the XL algorithm solving a quadratic polynomial system of $m$ equations in $n$ variables. Under the assumption that $\tilde{n} = O(1)$, this optimization takes only $O(Nn)$ field operations [CN26].
Since BW1 is by far the most expensive step of the Block-Wiedemann algorithm when solving quadratic polynomial systems, in this paper we focus on that step.

## 2.5 Wiedemann XL

The original Macaulay matrix $M_D$ usually has more rows than columns, while block Wiedemann takes a square matrix as input. To solve the problem, the CCNY implementation [CCNY12] discards roughly the same number of rows in each row block to obtain a square matrix $M$. So far we are not aware of any paper that discusses how the rows can be discarded without enlarging the kernel space. Nevertheless, in the remainder of this paper, we assume that the matrix $M$ is obtained by discarding rows and that this does not enlarge the kernel space.

# 3 The CCNY Wiedemann XL Implementation

The implementation described in this section corresponds to the XL-Wiedemann solver proposed by Cheng, Chou, Niederhagen, and Yang in [CCNY12], which serves as the reference implementation throughout the remainder of this work.
Rather than explicitly storing the Macaulay matrix, this implementation exploits the regular structure of the XL construction to reconstruct its rows on demand during the sparse-dense matrix multiplication performed in Step BW1. More precisely, the tall rectangular Macaulay matrix $M_D$ is represented through three auxiliary data structures: (i) the original polynomial system, which stores the nonzero coefficients of each polynomial, (ii) a precomputed *column map*, which determines the column position of every nonzero entry in the Macaulay matrix, and (iii) a *row selection mask*, which determines which rows of $M_D$ are retained to form the square matrix $M$ used by the Block Wiedemann algorithm.
The following subsections describe these data structures and the parallel multiplication kernel employed during Step BW1, which together constitute the computational core of the implementation.

## 3.1 Column Map

The column map is a $\binom{n+D-2}{D-2} \times \binom{n+2}{2}$ matrix whose rows are indexed by extension monomials $\mu \in v_{D-2}$ and whose columns are indexed by system monomials $\mu' \in v_2$. The entry corresponding to $(\mu, \mu')$ is $\iota(\mu\mu')$, where $\iota$ denotes the grevlex monomial index defined in (1).
During the sparse-dense matrix multiplication of step BW1, each row of the Macaulay matrix $M_D$ corresponds to the product of one extension monomial $\mu \in v_{D-2}$ and one polynomial $f$ of the original system. For each system monomial $\mu' \in v_2$ of $f$, the implementation requires the position of $\mu'$ in that row, namely $\iota(\mu\mu')$. Rather than

computing this mapping each time, the implementation precomputes all possible combinations once during initialization and stores the resulting column indices in a contiguous lookup table named `col_idx`.
This representation allows one to recover the column index of any nonzero entry with one constant-time memory access, at the cost of storing $N_{D-2}N_2$ indices.

## 3.2 Row Selection Mask

The Macaulay matrix $M_D$ contains one row for every pair consisting of an extension monomial $\mu \in v_{D-2}$ and a polynomial of the original system. It therefore has $mN_{D-2}$ rows and $N_D$ columns. Since the Block-Wiedemann algorithm operates on a square matrix, the implementation discards $mN_{D-2} - N_D$ rows of $M_D$ and obtains a square matrix $M \in \mathbb{F}_q^{N_D \times N_D}$.
The row-selection mask records the correspondence between the rows of $M$ and the retained rows of $M_D$. The implementation constructs this selection using a heuristic that removes approximately the same number of rows from each extension-monomial block, that is, from each set of $m$ rows obtained by multiplying the original polynomials by the same extension monomial.
Each block retains $\lfloor N_D/N_{D-2} \rfloor$ rows, and $N_D \mod N_{D-2}$ blocks retain one additional row. Thus, the numbers of retained rows in any two blocks differ by at most one. In this implementation, the number of retained rows in block $b$, for $0 \le b < N_{D-2}$, is

$$\texttt{block_cnt}[b] = \left\lfloor \frac{(b+1)N_D}{N_{D-2}} \right\rfloor - \left\lfloor \frac{bN_D}{N_{D-2}} \right\rfloor .$$

The selected row indices and the corresponding block information are computed during initialization and stored explicitly in the row-selection mask `row_mask`, which consists of three arrays:

- `L`, of length $N$, stores the index in $M_D$ of each retained row. Using zero-based indices, if row $r$ of $M$ corresponds to polynomial $j$ in extension-monomial block $b$, then $\texttt{L}[r] = bm + j$. Thus, the block index and the polynomial index can be recovered as $\lfloor \texttt{L}[r]/m \rfloor$ and $\texttt{L}[r] \bmod m$, respectively.
- `block_cnt`, of length $N_{D-2}$, stores the number of retained rows in each extension-monomial block.
- `block_start`, of length $N_{D-2}$, stores the starting position in `L` of the retained rows of each block.

## 3.3 Parallel Multiplication Kernel

The main computational kernel of step BW1 is the repeated multiplication of $M \in \mathbb{F}_q^{N_D \times N_D}$ by a dense matrix $Z \in \mathbb{F}_q^{N_D \times \tilde{n}}$. For an output row $r$, the row-selection mask first determines the corresponding extension-monomial block and the polynomial of the original system associated with that row. The coefficients of the row are obtained directly from the corresponding polynomial, while the column positions of its entries are retrieved from `col_idx`. More precisely, consider one multiplication $C = MZ$, and let $\mathbf{z}_j$ and $\mathbf{c}_r$ denote row $j$ of $Z$ and row $r$ of $C$, respectively. We use zero-based row and column indices, so for $0 \le r < N_D$, the $r$-th output row is

$$\mathbf{c}_r = \sum_{j=0}^{N_D-1} M_{r,j}\mathbf{z}_j .$$

Since $M$ is sparse, only terms corresponding to nonzero coefficients of row $r$ of $M$ contribute to this sum. The column map provides the potential nonzero indices $j$ of $M_{r,j}$.
The row-block structure of the Macaulay matrix allows these column positions to be reused. For a fixed extension monomial $\mu$, all $m$ rows in its block $\mu f_0, \ldots, \mu f_{m-1}$ use the same set of candidate column positions $\{\iota(\mu\mu') : \mu' \in v_2\}$ and differ only in the

**Table 1:** Numbers of bits taken by the data used in BW1.

| type | data | size (bits) |
|---|---|---|
| input | $f^{(1)}, \ldots, f^{(m)}$ | $mN_2 \cdot \lceil \log_2 q \rceil$ |
| | column map | $N_{D-2}N_2 \cdot \lceil \log_2 N \rceil$ |
| | row mask | $N_{D-2}m$ |
| intermediate | $(M^{i-1}Z, M^iZ)$ | $2N\tilde{n} \cdot \lceil \log_2 q \rceil$ |
| output | $A_i$'s | $(N\tilde{m} + N\tilde{n}) \cdot \lceil \log_2 q \rceil$ |
| | $\overline{M^iZ}$'s | $N(n+1) \cdot \lceil \log_2 q \rceil$ |

coefficients inherited from the corresponding polynomials $f_0, \ldots, f_{m-1}$. Therefore, the required rows of $Z$ can be retrieved once for an extension-monomial block and reused while processing the selected rows within that block.

The implementation further exploits the small size of the underlying finite field to reduce the number of scalar multiplications. For a fixed output row $r$, the rows of $Z$ are grouped according to the corresponding coefficient $\alpha = M_{r,j}$. Hence,

$$\mathbf{c}_r = \sum_{\alpha \in \mathbb{F}_q \setminus \{0\}} \alpha \left( \sum_{\substack{0 \le j < N_D \\ M_{r,j} = \alpha}} \mathbf{z}_j \right)$$

A temporary buffer is maintained for each nonzero $\alpha \in \mathbb{F}_q$ and accumulates the corresponding rows of $Z$. Once the row has been processed, these buffers are combined to obtain $\mathbf{c}_r$. In this way, several scalar multiplications by the same field element are replaced by vector additions followed by a scalar multiplication of the accumulated vector.

Finally, since each output row $\mathbf{c}_r$ depends only on row $r$ of $M$ and the shared input matrix $Z$, different output rows can be computed independently. The reference implementation exploits this independence using an OpenMP `schedule(static)` loop over the output rows. Each thread maintains private accumulation buffers and a private temporary array containing the required rows of $Z$, while writing only to its assigned rows of $C$. Consequently, no synchronization is required during the computation of individual output rows.

## 3.4 Theoretical Analysis of Memory/Disk Usage of BW1

In the implementation of [CCNY12], BW1 takes some input data, maintains some intermediate values, and generates some output data. This subsection discusses how much memory/disk space these data would take in theory. "In theory" implies that our analysis is independent of some implementation choices made in the implementation of [CCNY12], such as using 32 bits to store each index in the column map and in the row mask.

The values of the non-zero entries of $M$ come from the input multivariate system, which takes $mN_2$ field elements to store, where each field element takes $\lceil \log_2 q \rceil$ bits. There are $N_{D-2}$ row blocks in the original Macaulay matrix $M_D$. For each of these row blocks, the implementation of [CCNY12] precomputes the $N_2$ column indices in the column map. Therefore, there are $N_{D-2}N_2$ indices to store, where each index takes $\lceil \log_2 N \rceil$ bits in theory. There are $mN_{D-2}$ rows in $M_D$, so on average there are $\frac{N_2}{m} = \Theta(n)$ indices per row under the reasonable assumption that $m = \Theta(n)$. $M$ has fewer rows, so the average number of indices per row can only be larger, which implies that $N_{D-2}N_2 = \Omega(nN)$. We conclude that there are $\Omega(nN \log N)$ bits to store.

The implementation of [CCNY12] also precomputes the row mask. The row mask can be stored using a binary vector of length $N_{D-2}m$. Since

$$\frac{N_{D-2}N_2}{N_{D-2}m} = \Theta(n),$$

we conclude that the row mask takes $\Omega(N)$ bits to store.
Regarding intermediate values, BW1 performs a multiplication of the form $M \cdot (M^{i-1}Z)$ in each iteration, so there needs to be memory space for storing $M^{i-1}Z$ and also the product. Both matrices are $N \times \tilde{n}$, so there are $2N\tilde{n}$ field elements to store. [2]
The output of BW1 depends on which version of BW3 is used. If the original version is used, the output is simply the matrices $A_i$'s, which take about

$$\left(\frac{N}{\tilde{m}} + \frac{N}{\tilde{n}}\right)\tilde{n}\tilde{m} = N\tilde{m} + N\tilde{n}$$

field elements to store. If the optimized version of BW3 is used, in addition to $A_i$'s, $N/\tilde{n}$ matrices of the form $\overline{M^i z}$ ($\overline{A}$ indicates the first $n+1$ rows of the matrix $A$) will be stored. These matrices take about $Nn$ field elements to store.
We summarize the size of the data used in BW1 in Table 1.

**Summary.** The column map takes $\Omega(nN \log N)$ bits. The row mask takes $\Omega(N)$ bits. $\overline{M^i Z}$'s, which are only required for the optimized version of BW3, take $\Theta(Nn)$ bits. All other data take $O(N)$ bits under the assumption that $D \geq 3$. We conclude that asymptotically the column map is the most space-consuming data. In the remainder of the paper, we discuss how to generate the column map and the row mask on the fly while achieving the same speed as the implementation of [CCNY12].

# 4 Generating the Macaulay Matrix On-the-fly

This section describes the modifications introduced to eliminate the memory-intensive auxiliary structures described in Section 3. The proposed approach replaces precomputed information with computations performed on demand during step BW1, introducing a trade-off between memory consumption and computational cost. In particular, we address three aspects of this trade-off: the parallel decomposition used to process row blocks, the on-the-fly reconstruction of the row-selection mask and the on-the-fly generation of their column indices.

## 4.1 Chunk-Based Parallelization

Algorithm 1 summarizes the on-the-fly sparse-dense multiplication used in BW1. This subsection explains how computation is organized into row blocks and chunks.
Let $\mu_0, \mu_1, \ldots, \mu_{N_{D-2}-1}$ denote the ordered extension monomials, $\mathcal{B}_k = \{\mu_k f_0, \ldots, \mu_k f_{m-1}\}$ the row block associated with $\mu_k$, where $k = \iota(\mu_k)$, and $v_2 = (\mu'_0, \ldots, \mu'_{N_2-1})$ the ordered vector of monomials of degree at most two. For each row block $\mathcal{B}_k$, define the associated column-index vector as

$$I = \left(\iota(\mu_k\mu'_0), \ldots, \iota(\mu_k\mu'_{N_2-1})\right),$$

so that $I[j] = \iota(\mu_k\mu'_j)$, with $0 \leq j < N_2$. The entries of $I$ give the column positions in the Macaulay matrix obtained by multiplying $\mu_k$ by each monomial in $v_2$. As mentioned in Section 3, $I$ is shared by all rows in $\mathcal{B}_k$, while their coefficients are inherited from the corresponding polynomials $f_0, \ldots, f_{m-1}$. It is therefore natural to compute $I$ once and reuse it while processing all selected rows of the block. Figure 1 illustrates this reuse of the column indices within a row block.
In the reference implementation, these column indices are immediately available from the precomputed column map. However, in the on-the-fly implementation, they must

[2] Although the initial $Z$ is stored separately in the original version of BW3, its space overhead is negligible because it can be generated from a short $O(1)$-bit seed [CN26].

**Algorithm 1** Block-wise multiplication of the original system by the extension monomials

**Require:** quadratic system $F = (f_0, \ldots, f_{m-1})$ with $f_i = \sum_{j=0}^{N_2-1} a_{i,j}\, x^{\alpha_j}$; degree $D$; $N_D \times \tilde{n}$ matrix $Z$; chunk size $c$.

**Ensure:** $C = MZ$, where $M$ is the $N_D \times N_D$ submatrix of the degree-$D$ Macaulay matrix of $F$ obtained by discarding $mN_{D-2} - N_D$ rows.

1: $d \leftarrow m\, N_{D-2} - N_D$ ▷ number of rows to discard
2: $\rho \leftarrow d \bmod N_{D-2}$
3: **for parallel** $s \leftarrow 0, c, 2c, \ldots$ **while** $s < N_{D-2}$ **do** ▷ process blocks in chunks
4: $\mu \leftarrow \textsc{Unrank}(D-2,\, s)$ ▷ monomial of the first block in the chunk
5: **for** $t \leftarrow N_{D-2} - 1 - s$ **downto** $\max(N_{D-2} - s - c,\, 0)$ **do** ▷ blocks of the chunk
6: $b_t \leftarrow m - \lfloor d/N_{D-2} \rfloor$ ▷ base block size
7: **if** $t < \rho$ **then**
8: $b_t \leftarrow b_t - 1$ ▷ the first $\rho$ blocks lose one extra row
9: **end if**
10: $\mathcal{S} \leftarrow \textsc{SelectedRows}(m,\, b_t,\, t)$
11: $I \leftarrow \textsc{Indices}(\mu)$ ▷ $I[j] = \iota(\mu x^{\alpha_j})$, $0 \le j < N_2$
12: **for** $\ell \leftarrow 0$ **to** $b_t - 1$ **do** ▷ rows of block $t$
13: $i \leftarrow \mathcal{S}[\ell]$
14: $r \leftarrow t\left(m - \lfloor d/N_{D-2} \rfloor\right) - \min(t, \rho) + \ell$ ▷ global row index
15: $\mathbf{c}_r \leftarrow \sum_{j=0}^{N_2-1} a_{i,j}\, \mathbf{z}_{I[j]}$
16: **end for**
17: $\mu \leftarrow \textsc{Step}(\mu)$ ▷ successor of $\mu$ in grevlex order
18: **end for**
19: **end for**

instead be generated from the extension monomial associated with the current row block. Given an arbitrary monomial rank $k, 0 \le k \le N_{D-2} - 1$, the corresponding extension monomial $\mu_k$ can be recovered by inverting the monomial-indexing map via unranking as described in Section 2.2. Thus, a straightforward parallelization strategy would be to assign row blocks independently and perform one unranking operation whenever a block is processed.

Although this strategy allows arbitrary access to the row blocks, it reconstructs each extension monomial independently. In contrast, once $\mu_k$ is known, the next extension monomial in the same ordering can be obtained using the step operation. As explained in the preliminaries, stepping is substantially cheaper than unranking. To exploit this sequential relation while retaining parallelism, we group consecutive row blocks into *chunks*. For a chunk of size $c$ beginning at block $s$, define $\mathcal{C}_{s,c} = (\mathcal{B}_s, \mathcal{B}_{s+1}, \ldots, \mathcal{B}_{s+c-1})$, with the final chunk truncated when necessary.

When a thread begins processing $\mathcal{C}_{s,c}$, it first computes $\mu_s$ by unranking and then processes the blocks in the chunk sequentially, obtaining each subsequent extension monomial by applying the step operation. Consequently, only the first block of each chunk requires an unranking operation. Chunks are distributed independently among the available threads. Figure 2 illustrates this decomposition.

For any of the original polynomials $f_i = \sum_{j=0}^{N_2-1} a_{i,j}\mu'_j$, and an extension monomial $\mu$,

$$\mu f_i = \sum_{j=0}^{N_2-1} a_{i,j}(\mu\mu'_j),$$

hence, within a row block $\mathcal{B}_k$, multiplication by $\mu_k$ preserves the coefficients $a_{i,j}$ and places them at the columns $I[j]$.

For the row block $\mathcal{B}_k$, let $t = N_{D-2} - 1 - k$ be its output-block index and let $b_t$ be the number of retained rows, computed in lines 6–8 of Algorithm 1. The reversed block numbering follows an implementation convention but is not a requirement of BW1. The procedure SelectedRows$(m, b_t, t)$ returns an ordered list $\mathcal{S}$ of $b_t$ distinct

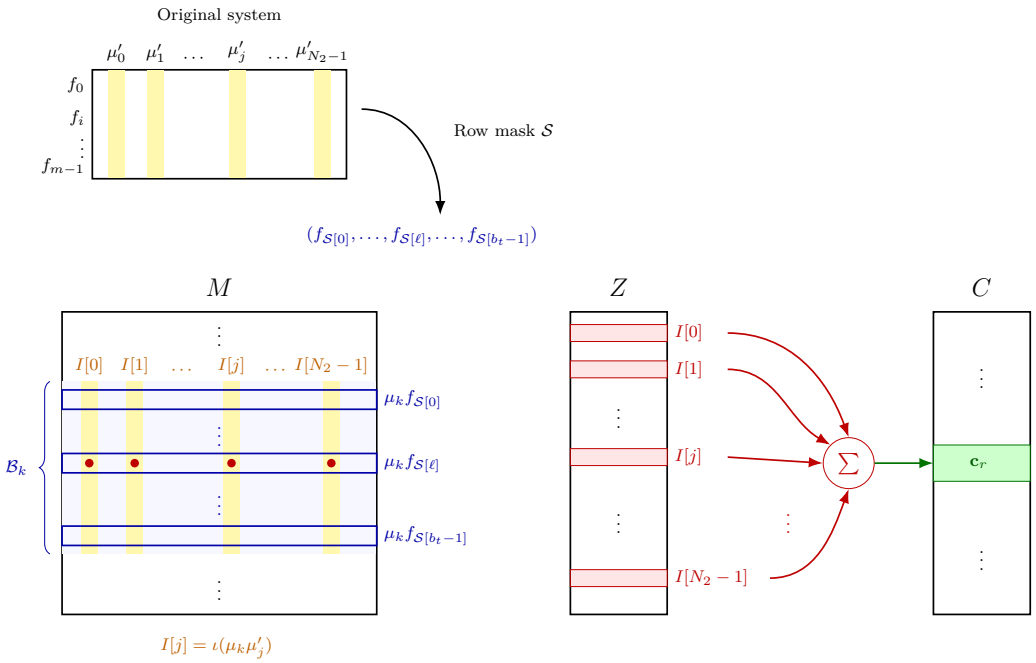



**Figure 1:** Column-index reuse within a row block of the Macaulay matrix. The row mask selects polynomials from the original system, whose coefficients weight the corresponding rows of $Z$ to produce an output row $\mathbf{c}_r$.

polynomial indices. For each position $\ell$, the polynomial index is $i = \mathcal{S}[\ell]$, and the corresponding output-row index is

$$r = t\left(m - \left\lfloor \frac{d}{N_{D-2}} \right\rfloor\right) - \min(t, \rho) + \ell, \qquad 0 \le \ell < b_t.$$

The expression before $+\ell$ counts the retained rows in the preceding output blocks. The construction of $\mathcal{S}$ is described in Section 4.2.

Note that the efficiency of this solution depends on the chunk size $c$. A small $c$ increases the number of chunks and thus the number of calls to unranking, but a large $c$ may cause an unequal division of work among processing units, resulting in some idle time. In Appendix A.1 we discuss the issue in more detail and show some experimental results.

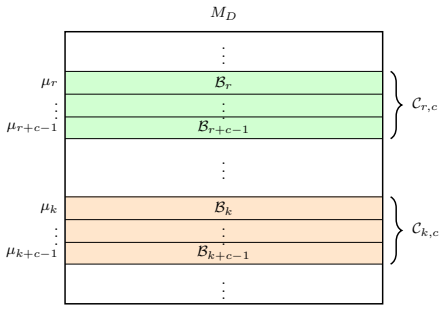



**Figure 2:** Chunk-based decomposition of the row blocks of the Macaulay matrix. Each chunk contains $c$ consecutive row blocks and can be assigned independently to a thread.

## 4.2 Generating the Row Mask On-the-fly

The second auxiliary structure eliminated by our implementation is the row-selection mask. As described in Section 3.2, the retained rows are distributed approximately uniformly among the extension-monomial blocks. Here the number $b_t$ of retained rows is computed directly from the output-block index $t$, as in lines 6–8 of Algorithm 1.

The reference implementation stores the selected row indices explicitly in the row-selection mask. Our implementation instead reconstructs the selection independently whenever a row block is processed.

For each block $\mathcal{B}_k$, the output-block index $t = N_{D-2} - 1 - k$ is used as the seed of a deterministic pseudo-random generator. Starting from the ordered sequence $(0, 1, \ldots, m-1)$, a partial Fisher–Yates shuffle is performed, and the first $b_t$ positions of the resulting permutation are selected. Initializing the array takes $O(m)$ operations, and the partial shuffle performs $b_t$ swap steps, taking $O(b_t)$ operations. The total cost is therefore $O(m + b_t)$. Thus, the procedure generates an ordered list $\mathcal{S}$ of $b_t$ polynomial indices, with $0 \le \mathcal{S}[\ell] < m$ for $0 \le \ell < b_t$. No polynomial index is selected more than once. This list is returned by SelectedRows$(m, b_t, t)$ in Algorithm 1.

Because the pseudo-random generator is initialized with the output-block index $t$, the same block always produces the same ordered list $\mathcal{S}$. Consequently, the row-selection information does not need to be stored globally: it can be regenerated deterministically whenever $\mathcal{B}_k$ is processed.

For each position $\ell$ in $\mathcal{S}$, Algorithm 1 sets $i = \mathcal{S}[\ell]$ and retains the row corresponding to $\mu_k f_i$. The procedure returns only the polynomial indices, so the output-row index $r$ is computed separately from $t$ and $\ell$ in line 14 of Algorithm 1.

Thus, the global row-selection mask of the reference implementation is replaced by a small amount of temporary per-thread state and the cost of regenerating the selection on demand.

## 4.3 Generating Column Indices for One Row Block

Given a monomial $\mu = x_1^{d_1} \cdots x_n^{d_n}$ represented as $(d_1, \ldots, d_n)$, to generate the indices in

$$I_\mu = \left\{ \iota(\mu \cdot \mu') \mid \mu' \in \mathcal{M}_2 \right\},$$

the naive approach is to use Equation (1) $\binom{n+2}{2}$ times when the ring is $R$, or to use Equation (2) $\binom{n}{2} + n + 1$ times when the ring is $\mathcal{R}$. In order to save time, in the implementation of [CCNY12], the values of $\binom{a}{b}$ for all the necessary $(a, b)$'s are stored in a precomputed table. In this way, generating the indices in $I_\mu$ takes $O(n^3)$ table look-ups and $O(n^3)$ integer additions. This complexity is not a problem for the implementation of [CCNY12], as indices for all row blocks are computed once at the beginning of the algorithm and stored for future use. However, when we tried to compute the indices on-the-fly in each iteration of BW1, it turned out that the computation became much slower.

We developed novel algorithms that allow us to use only $O(n^2)$ table look-ups and $O(n^2)$ integer additions to compute all indices in $I_\mu$ given $\mu$, for both $R$ and $\mathcal{R}$. To be more precise, from $\mu$, we first compute $\iota(\mu)$ and two vectors $v^{(1)}, v^{(2)} \in \mathbb{Z}^n$ using $O(n)$ table look-ups and $O(n)$ integer additions. These vectors can be considered as two tables or collectively as one two-dimensional table. After that, each index in $I_\mu$ can be computed using no more than 2 table look-ups and 2 integer additions.

The algorithms we developed can be easily generalized so that, given a monomial $\mu$, computing all indices in $\left\{ \iota(\mu \cdot \mu') \mid \mu' \in \mathcal{M}_\delta \right\}$ takes $O(n^\delta)$ table look-ups and $O(n^\delta)$ integer additions for any constant $\delta$.

### Computing $I_\mu$ in $R$

We first explain how $v^{(1)}, v^{(2)} \in \mathbb{Z}^n$ are defined. Let $u^{(0)}, u^{(1)}, u^{(2)} \in \mathbb{Z}^n$ be such that

$$u_i^{(\delta)} := \sum_{j=i}^{n} \sigma(j, \overline{d}_j + \delta).$$

It turns out that $\iota(\mu) = u_1^{(0)}$. Then, $v^{(1)}$ and $v^{(2)}$ are simply defined as the vectors such that

$$v_i^{(\delta)} := u_i^{(\delta)} - u_i^{(\delta-1)}.$$

It is easy to see that each $u^{(\delta)}$ and $v^{(\delta)}$ can be computed using $O(n)$ table look-ups and $O(n)$ integer additions, so we do not give a proof here. We show that given $\iota(\mu)$, $v^{(1)}$, and $v^{(2)}$, one can easily compute the index of any monomial of the form $\mu \cdot x_{i_1}$ and of any monomial of the form $\mu \cdot x_{i_1} x_{i_2}$, $i_1 \leq i_2$, using the propositions below.

**Proposition 1.** $\iota(\mu \cdot x_{i_1}) = \iota(\mu) + v_{i_1}^{(1)}$.

*Proof.* Observe that

$$\iota(\mu \cdot x_{i_1}) = \sum_{j=1}^{i_1-1} \sigma(j, \overline{d}_j) + \sum_{j=i_1}^{n} \sigma(j, \overline{d}_j + 1).$$

Hence,

$$\iota(\mu \cdot x_{i_1}) - \iota(\mu) = \sum_{j=i_1}^{n} \sigma(j, \overline{d}_j + 1) - \sum_{j=i_1}^{n} \sigma(j, \overline{d}_j) = v_{i_1}^{(1)}. \tag{4}$$

□

**Proposition 2.** $\iota(\mu \cdot x_{i_1} x_{i_2}) = \iota(\mu) + v_{i_1}^{(1)} + v_{i_2}^{(2)}$ *if* $i_1 \leq i_2$.

*Proof.* Observe that

$$\iota(\mu \cdot x_{i_1} x_{x_2}) = \sum_{j=1}^{i_1-1} \sigma(j, \overline{d}_j) + \sum_{j=i_1}^{i_2-1} \sigma(j, \overline{d}_j + 1) + \sum_{j=i_2}^{n} \sigma(j, \overline{d}_j + 2).$$

Then,

$$\begin{aligned}
\iota(\mu \cdot x_{i_1} x_{x_2}) - \iota(\mu \cdot x_{i_1}) &= \sum_{j=i_1}^{i_2-1} \sigma(j, \overline{d}_j + 1) + \sum_{j=i_2}^{n} \sigma(j, \overline{d}_j + 2) - \sum_{j=i_1}^{n} \sigma(j, \overline{d}_j + 1) \\
&= \sum_{j=i_2}^{n} \sigma(j, \overline{d}_j + 2) - \sum_{j=i_2}^{n} \sigma(j, \overline{d}_j + 1) \\
&= v_{i_2}^{(2)}.
\end{aligned}$$

□

### Computing $I_\mu$ in $\mathcal{R}$

Again, we first explain how $v^{(1)}, v^{(2)} \in \mathbb{Z}^n$ are defined. Let $u^{(0)}, u^{(1)}, u^{(2)} \in \mathbb{Z}^n$ be such that

$$u_i^{(\delta)} := \begin{cases} \sum_{j=i+1}^{n} \sigma(j, \overline{d}_j + \delta, 1) & \text{if } d_i = 0, \\ 0 & \text{if } d_i = 1. \end{cases}$$

Then, $v^{(1)}$ and $v^{(2)}$ are simply defined as the vectors such that

$$v_i^{(\delta)} := \begin{cases} u_i^{(\delta)} - u_i^{(\delta-1)} + \sigma(i, \overline{d}_i + \delta, 1) & \text{if } d_i = 0, \\ 0 & \text{if } d_i = 1. \end{cases}$$

It is easy to see that each $u^{(\delta)}$ and $v^{(\delta)}$, and also $\iota(\mu)$, can be computed using $O(n)$ table look-ups and $O(n)$ integer additions, so we do not give a proof here. We show that given $\iota(\mu)$, $v^{(1)}$, and $v^{(2)}$, one can easily compute the index of any monomial of the form $\mu \cdot x_{i_1}$ and of any monomial of the form $\mu \cdot x_{i_1} x_{i_2}$, $i_1 < i_2$, using the propositions below.

**Proposition 3.** $\iota(\mu \cdot x_{i_1}) = \iota(\mu) + v^{(1)}_{i_1}$.

*Proof.* The case $d_{i_1} = 1$ is trivial as $v^{(1)}_{i_1} = 0$. For the case $d_{i_1} = 0$, observe that

$$\iota(\mu \cdot x_{i_1}) = \sum_{j=1}^{i_1-1} \sigma(j, \overline{d}_j, d_j) + \sigma(i_1, \overline{d}_{i_1} + 1, 1) + \sum_{j=i_1+1}^{n} \sigma(j, \overline{d}_j + 1, d_j).$$

Therefore,

$$\begin{aligned}\iota(\mu \cdot x_{i_1}) - \iota(\mu) &= \sigma(i_1, \overline{d}_{i_1} + 1, 1) + \sum_{j=i_1+1}^{n} \sigma(j, \overline{d}_j + 1, d_j) - \sum_{j=i_1+1}^{n} \sigma(j, \overline{d}_j, d_j) \\ &= v^{(1)}_{i_1}.\end{aligned}$$

□

**Proposition 4.** $\iota(\mu \cdot x_{i_1} x_{i_2}) = \iota(\mu) + v^{(1)}_{i_1} + v^{(2-d_{i_1})}_{i_2}$ *if* $i_1 < i_2$.

*Proof.* Following Proposition 3, this proposition is equivalent to

$$\iota(\mu \cdot x_{i_1} x_{i_2}) = \iota(\mu \cdot x_{i_1}) + v^{(2-d_{i_1})}_{i_2},$$

if $i_1 < i_2$. The case $d_{i_2} = 1$ is trivial as $v^{(1)}_{i_2} = v^{(2)}_{i_2} = 0$. The case $d_{i_1} = 1, d_{i_2} = 0$ can be proven by using Proposition 3. For the case $d_{i_1} = d_{i_2} = 0$, observe that

$$\begin{aligned}\iota(\mu \cdot x_{i_1} x_{i_2}) = \sum_{j=1}^{i_1-1} \sigma(j, \overline{d}_j, d_j) + \sigma(i_1, \overline{d}_{i_1} + 1, 1) + \sum_{j=i_1+1}^{i_2-1} \sigma(j, \overline{d}_j + 1, d_j) \\ + \sigma(i_2, \overline{d}_{i_2} + 2, 1) + \sum_{j=i_2+1}^{n} \sigma(j, \overline{d}_j + 2, d_j).\end{aligned}$$

Therefore,

$$\begin{aligned}&\iota(\mu \cdot x_{i_1} x_{i_2}) - \iota(\mu \cdot x_{i_1}) \\ =&\iota(\mu \cdot x_{i_1} x_{i_2}) - \left( \sum_{j=1}^{i_1-1} \sigma(j, \overline{d}_j, d_j) + \sigma(i_1, \overline{d}_{i_1} + 1, 1) \right. \\ &\qquad \left. + \sum_{j=i_1+1}^{i_2-1} \sigma(j, \overline{d}_j + 1, d_j) + \sum_{j=i_2+1}^{n} \sigma(j, \overline{d}_j + 1, d_j) \right) \\ =&\sigma(i_2, \overline{d}_{i_2} + 2, 1) + \sum_{j=i_2+1}^{n} \sigma(j, \overline{d}_j + 2, d_j) - \sum_{j=i_2+1}^{n} \sigma(j, \overline{d}_j + 1, d_j) \\ =& v^{(2)}_{i_2}.\end{aligned}$$

□

In the case of $R$, since

$$\iota(\mu \cdot x_{i_1} x_{i_2}) = \iota(\mu) + v^{(1)}_{i_1} + v^{(2)}_{i_2} = \iota(\mu \cdot x_{i_1}) + v^{(2)}_{i_2},$$

computing $\iota(\mu \cdot x_{i_1} x_{i_2})$ only takes 1 table look-up and 1 integer addition, assuming that $\iota(\mu \cdot x_{i_1})$ has been computed. Similarly, in the case of $\mathcal{R}$, since

$$\iota(\mu \cdot x_{i_1} x_{i_2}) = \iota(\mu) + v_{i_1}^{(1)} + v_{i_2}^{(2-d_{i_1})} = \iota(\mu \cdot x_{i_1}) + v_{i_2}^{(2-d_{i_1})},$$

computing $\iota(\mu \cdot x_{i_1} x_{i_2})$ only takes 1 table look-up and 1 integer addition (if we do not count the small subtraction $2 - d_{i_1}$), assuming that $\iota(\mu \cdot x_{i_1})$ has been computed. Therefore, to minimize the number of operations to compute $I_\mu$, one can compute indices of the form $\iota(\mu \cdot x_{i_1})$ before computing the remaining indices. We demonstrate our algorithms for computing $I_\mu$ in the case of $R$ and $\mathcal{R}$ using the Sage scripts `index.sage` and `index_gf2.sage` included in the supplementary material.

# 5 Evaluation

In this section, we first compare peak memory usage (obtained by the value of `VmHWM` shown in `/proc/<pid>/status`) and timings for three SSE implementations, namely NOTF-SSE, OTF-SSE, and OTF-SSE$^+$. NOTF-SSE refers to the implementation available at https://github.com/IIS-ACE-lab/xl, which combines the code for BW1 and Coppersmith's BM from [CCNY12], and the code for the optimized BW3 from [CN26]. In both OTF-SSE and OTF-SSE$^+$, the column map and the row mask are generated on-the-fly. The difference is that, OTF-SSE does not make use of the optimization mentioned in Section 4.3, while OTF-SSE$^+$ does make use of the optimization. Both OTF-SSE and OTF-SSE$^+$ are adapted from NOTF-SSE. Our code for OTF-SSE and OTF-SSE$^+$ has been included in the supplementary material.

## 5.1 Experiments with SSE Implementations

Table 2 shows experimental results for solving several instances with $q = 16$ and $m = n + 2$ with 64 cores of AMD EPYC 7501 CPU. Table 3 shows experimental results for solving several instances with $q = 2$ and $m = n + 2$ with 64 cores of AMD EPYC 7763 CPU.

**Table 2:** Peak memory usage (kB) and timings (s) for solving some instances with $q = 16$ and $m = n + 2$ on 64 CPU cores. The chunk size is set to 128.

| $(m, n)$ | $D$ | $N$ | $\tilde{m} = \tilde{n}$ | $M_{\text{bw1}}$ | $M_{\text{all}}$ | $T_{\text{bw1}}$ | $T_{\text{all}}$ | implementation |
|---|---|---|---|---|---|---|---|---|
| (16,14) | 9 | 817190 | 64 | 166096 | 166096 | 191 | 313 | NOTF-SSE |
| | | | | 107368 | 107368 | 332 | 446 | OTF-SSE |
| | | | | 107500 | 109768 | 173 | 291 | OTF-SSE$^+$ |
| | | | 32 | 115084 | 116568 | 235 | 350 | NOTF-SSE |
| | | | | 56224 | 56224 | 571 | 682 | OTF-SSE |
| | | | | 56440 | 57868 | 260 | 372 | OTF-SSE$^+$ |
| (17,15) | 9 | 1307504 | 64 | 265804 | 268100 | 479 | 734 | NOTF-SSE |
| | | | | 168612 | 168612 | 955 | 1203 | OTF-SSE |
| | | | | 168880 | 168880 | 465 | 713 | OTF-SSE$^+$ |
| | | | 32 | 184032 | 185460 | 645 | 898 | NOTF-SSE |
| | | | | 86928 | 86928 | 1616 | 1840 | OTF-SSE |
| | | | | 87152 | 87152 | 657 | 875 | OTF-SSE$^+$ |
| (18,16) | 10 | 5311735 | 64 | 1135260 | 1137552 | 7800 | 11063 | NOTF-SSE |
| | | | | 669364 | 671716 | 20513 | 23811 | OTF-SSE |
| | | | | 669440 | 671784 | 7995 | 11089 | OTF-SSE$^+$ |
| | | | 32 | 803396 | 804816 | 10671 | 13246 | NOTF-SSE |
| | | | | 337296 | 339012 | 35163 | 37943 | OTF-SSE |
| | | | | 337248 | 338940 | 11244 | 13823 | OTF-SSE$^+$ |
| (19,17) | 10 | 8436285 | 64 | 1824068 | 1826160 | 21372 | 29220 | NOTF-SSE |
| | | | | 1060024 | 1062312 | 50509 | 57974 | OTF-SSE |
| | | | | 1059928 | 1062240 | 20538 | 28159 | OTF-SSE$^+$ |
| | | | 32 | 1296344 | 1297796 | 26313 | 32305 | NOTF-SSE |
| | | | | 532744 | 534392 | 87839 | 93613 | OTF-SSE |
| | | | | 532624 | 534404 | 30720 | 36387 | OTF-SSE$^+$ |

As expected, both OTF-SSE and OTF-SSE$^+$ achieve lower peak memory consumption than NOTF-SSE, and the difference is about the same as the sum of the size of the column map and the size of the row mask. Take $(m, n, \tilde{m} = \tilde{n}) = (19, 17, 64)$ for example. According to Sections 3.1 and 3.2, the column map and the row mask together would take

$$\binom{n+D-2}{D-2}\binom{n+2}{2} \cdot 4 + N \cdot 4 + \binom{n+D-2}{D-2} \cdot 8 = 782\,195\,040$$

bytes, which is about 763 862 kB.
The memory consumption for our implementations is roughly the same as the number of bytes taken by $M^{i-1}Z$, $M^iZ$, and $A_i$'s (see Section 3.4). Take $(m, n, \tilde{m} = \tilde{n}) = (19, 17, 64)$ for example, these data consist of roughly

$$2N\tilde{n} + \left(\frac{N}{\tilde{m}} + \frac{N}{\tilde{n}}\right)\tilde{m}\tilde{n} = 2\,159\,688\,960$$

field elements, which take about 1 054 536 kB. One might notice that we do not consider $\overline{M^iZ}$. This is because, with $\tilde{m} \geq n + 1$ and $\mathbf{x} = (I_{\tilde{m}}\ 0)$, $\overline{M^iZ}$ is simply the first $n + 1$ rows of $A_i$.
Thanks to our novel optimization mentioned in Section 4.3, when $\tilde{m} = \tilde{n} = 64$, OTF-SSE$^+$ is more than twice as fast as OTF-SSE when running BW1 and achieves roughly the same speed as NOTF-SSE.
The results are similar for $q = 2$ as observed in Table 3. Memory consumption is consistent with our theoretical analysis and OTF-SSE$^+$ achieves roughly the same speed as NOTF-SSE. We didn't report timings for OTF-SSE because it behaves very similar than in the case $q = 16$, it is twice slower than OTF-SSE$^+$.

**Table 3:** Peak memory usage (kB) and timings (s) for solving some instances with $q = 2$ and $m = n + 2$ on 64 cores. The chunk size is set to 128.

| $(m, n)$ | $D$ | $N_D$ | $\tilde{m} = \tilde{n}$ | $M_{\text{bw1}}$ | $M_{\text{all}}$ | $T_{\text{bw1}}$ | $T_{\text{all}}$ | implementation |
|---|---|---|---|---|---|---|---|---|
| (32,30) | 6 | 768212 | 128 | 115016 | 116331 | 122.55 | 188.73 | NOTF-SSE |
| | | | | 53349 | 55037 | 128.87 | 191.82 | OTF-SSE$^+$ |
| (33,31) | 7 | 3572224 | 128 | 645148 | 647059 | 2821.05 | 3688.16 | NOTF-SSE |
| | | | | 228581 | 231108 | 2746.53 | 3606.20 | OTF-SSE$^+$ |
| (34,32) | 7 | 4514873 | 128 | 809100 | 811105 | 5159.17 | 6515.54 | NOTF-SSE |
| | | | | 287528 | 290069 | 5277.04 | 6626.39 | OTF-SSE$^+$ |
| (35,33) | 7 | 5663890 | 128 | 1008000 | 1009957 | 9191.44 | 11320.25 | NOTF-SSE |
| | | | | 359560 | 362080 | 8550.07 | 10632.51 | OTF-SSE$^+$ |
| (36,34) | 7 | 7055732 | 128 | 1247984 | 1249832 | 15754.86 | 18991.31 | NOTF-SSE |
| | | | | 446547 | 449136 | 16094.58 | 19277.86 | OTF-SSE$^+$ |

**Scalability tests.** Figure 3 shows the strong-scaling behavior on a two-socket AMD EPYC 7763 system (2 × 64 cores). Both implementations scale well up to 128 cores. OTF-SSE$^+$ reaches a speedup of about 50.4× (parallel efficiency ≈ 39%), and NOTF-SSE reaches about 46.6× (≈ 36%).

## 5.2 Large Distributed Deployment

For running larger MQ instances, it is necessary to have a distributed deployment akin to the large experiments of the past decade that factored RSA moduli, such as [KAF+10]. In such a scenario, several nodes would be assigned a few dense columns of $Z$ to compute BW1 independently. Because of the large memory requirement and the loose connection among nodes, the resulting $A_i = XM^iZ$ can be temporarily stored on disk and later put together to continue with BW2. Although such a deployment is beyond our budget, we simulated the work of one node for a few iterations so as to evaluate the scalability of the solution. Table 4 shows the experimental results.We only run 100 iterations and report the time per iteration, which can be extrapolated for

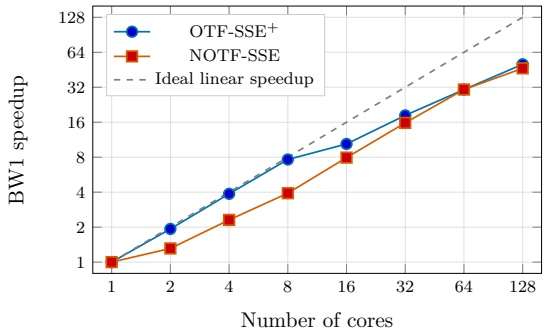



**Figure 3:** BW1 parallel speedup for $q = 16$, $(m, n) = (16, 14)$, and $\tilde{m} = \tilde{n} = 64$, with chunk size 128 for OTF-SSE$^+$. For each implementation, speedup is computed as $S(p) = \overline{T}(1)/\overline{T}(p)$, where $\overline{T}(p)$ is the mean BW1 runtime over three runs using $p$ cores. The dashed line represents ideal linear speedup, $S(p) = p$. Both axes use logarithmic scales with base 2.

the actual number of iterations required. The experiment shows that in this scenario, the OTF implementation requires less than one fourth of the memory of NOTF and it is actually faster (between 11% and 28%). Since the OTF implementation actually performs more computations than NOTF, it seems that the savings can only come from memory access cost.

**Table 4:** Peak memory usage (kB) and average time per BW1 iteration (s) for $q = 16$ and $m = n + 2$ using 128 CPU cores. Values are averaged over three runs of 100 BW1 iterations, with the sequence $A_i$ streamed to disk. The chunk size for OTF-SSE$^+$ is set to the square root of $N_{D-2}$.

| $m$ | $n$ | $D$ | $N$ | $\tilde{m} = \tilde{n}$ | $M_{\text{bw1}}$ | $T_{\text{bw1}}$/iter | Implementation |
|---|---|---|---|---|---|---|---|
| 22 | 20 | 12 | 225792840 | 32 | 35290196 | 0.894 | NOTF-SSE |
| | | | | | 7062425 | 0.640 | OTF-SSE$^+$ |
| 23 | 21 | 12 | 354817320 | 32 | 56659428 | 1.494 | NOTF-SSE |
| | | | | | 11094540 | 1.063 | OTF-SSE$^+$ |
| 24 | 22 | 13 | 1476337800 | 32 | 262077731 | 5.576 | NOTF-SSE |
| | | | | | 46142059 | 4.923 | OTF-SSE$^+$ |

## 5.3 The AVX2 Implementations

Here we test whether the on-the-fly approach of Section 4 still reduces memory without slowing down BW1 over $\mathbb{F}_{256}$ and with a more recent instruction set. To the best of our knowledge, prior to this work, no efficient implementation of XL supports $\mathbb{F}_{256}$, which is the field used for most of the parameters in the UOV signature scheme, a third-round NIST candidate.

We ported both NOTF-SSE and OTF-SSE$^+$ to AVX2 over $\mathbb{F}_{256}$, which gives NOTF-AVX2 and OTF-AVX2$^+$, and we compare them for $m = 2n$. OTF-AVX2$^+$ generates the column map and the row mask on the fly during BW1, as in Section 4, with the optimization of Section 4.3, to which the + refers. In every product by $M$, OpenMP partitions the rows statically among the threads in NOTF-AVX2. In OTF-AVX2$^+$, it assigns chunks of $c$ consecutive row blocks to the threads dynamically (Section 4.1). Figure 4 shows the ratios OTF-AVX2$^+$/NOTF-AVX2 of peak memory, BW1 time and total time for $n \in \{22, 24, 26, 28\}$, for which $D = 6$, and $\tilde{m} = \tilde{n} \in \{32, 64, 128\}$, with $c \approx N_{D-2}/(8N_{\text{cores}})$, where $N_{\text{cores}} = 128$, so that each thread receives about eight chunks, on a two-socket AMD EPYC 7763 ($2 \times 64$ cores) with 128 threads bound to the physical cores.

*Memory.* OTF-AVX2$^+$ needs about 0.73, 0.84 and 0.92 of the peak memory of NOTF-AVX2 for $\tilde{n} = 32$, 64 and 128, nearly independently of $n$. The ratio increases

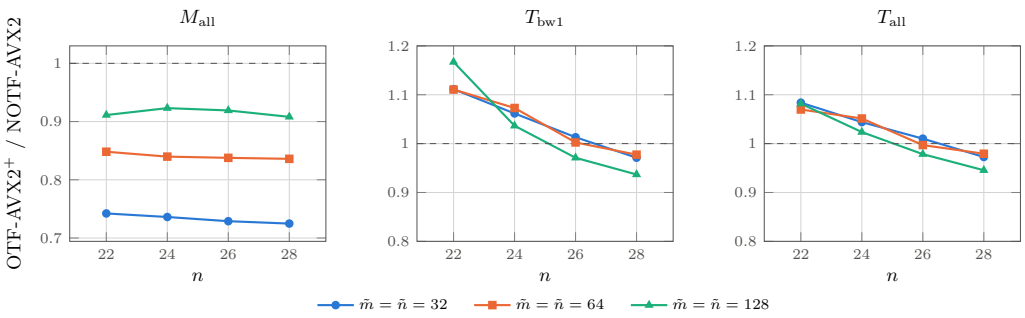



**Figure 4:** Ratios OTF-AVX2$^+$/NOTF-AVX2 of the peak memory $M_{\text{all}}$, i.e., the maximum resident set size of the whole solve (left), the BW1 time $T_{\text{bw1}}$ (center) and the total time $T_{\text{all}}$ (right; setup plus BW1, BW2 and BW3), over $\mathbb{F}_{256}$ for $m = 2n$ and $n \in \{22, 24, 26, 28\}$ ($D = 6$), with one curve per $\tilde{m} = \tilde{n}$. Each point is the ratio of the means over three runs. Points below the dashed line at 1 favor OTF-AVX2$^+$.

**Table 5:** Estimated BW1 time of NOTF-AVX2 and OTF-AVX2$^+$ (with $c = 981 = \lfloor\sqrt{N_{D-2}}\rfloor$) for $n = 38$ $m = 76$, $\mathbb{F}_{256}$, $D = 7$, $N = 45379620$, $\tilde{m} = \tilde{n}$, 128 threads. $T_{\text{bw1}}$/iter is the time in seconds of one BW1 iteration, dominated by one product by $M$, averaged over 1000 iterations; $L = 2\lfloor N/\tilde{n}\rfloor + 8$ is the number of BW1 iterations, and Estimated $T_{\text{bw1}} = L \cdot (T_{\text{bw1}}/\text{iter})$.

| | | $T_{\text{bw1}}$/iter | | Estimated $T_{\text{bw1}}$ | |
|---|---|---|---|---|---|
| $\tilde{n}$ | $L$ | NOTF-AVX2 | OTF-AVX2$^+$ | NOTF-AVX2 | OTF-AVX2$^+$ |
| 32 | 2836234 | 0.2684 | 0.2663 | 761237 | 755386 |
| 64 | 1418120 | 0.4999 | 0.4823 | 708866 | 684023 |
| 128 | 709064 | 0.9632 | 0.9341 | 682985 | 662311 |

with $\tilde{n}$ because the dense data of BW1 grows with $\tilde{n}$, while the stored indices do not (Section 3.4). The saving is smaller than in Table 2 because field elements are twice as large and $D$ is lower.

*Time.* In BW1, OTF-AVX2$^+$ is slower than NOTF-AVX2 at $n = 22$ (ratio 1.11 to 1.17) and at $n = 24$ (1.04 to 1.07) for every $\tilde{n}$, and close to parity at $n = 26$ (0.97 to 1.01), where only $\tilde{n} = 128$ is faster. At $n = 28$ it is faster for every $\tilde{n}$ (0.94 to 0.98), although only for $\tilde{n} = 128$ does the whole min–max range of the three runs lie below 1. The BW1 ratio decreases with $n$ for every $\tilde{n}$ and crosses 1 between $n = 24$ and $n = 26$ for $\tilde{n} = 128$, and between $n = 26$ and $n = 28$ for $\tilde{n} = 32$ and 64. The total time follows the same trend but stays closer to 1 (0.95 to 1.08), since BW2 and BW3 are the same code in both implementations. The only exception is $n = 26$, $\tilde{n} = 64$, where the BW1 ratio is 1.002 and the total-time ratio 0.997.

*Estimate for $n = 38$.* The value $n = 38$ matches the largest type II instance solved in the MQ challenges [YDH+15] at the time of writing. Without running the full solves, we multiply the BW1 time per iteration, $T_{\text{bw1}}$/iter, averaged over 1000 iterations, by the number $L$ of BW1 iterations, assuming that the time per iteration is constant over BW1 (Table 5). For this instance uses chunks of size $c = \lfloor\sqrt{N_{D-2}}\rfloor = 981$ OTF-AVX2$^+$ is expected to take about 0.99, 0.96 and 0.97 of the BW1 time of NOTF-AVX2 for $\tilde{n} = 32$, 64, and 128, respectively.

# 6 Conclusion

We observe that, for the parameter sets we tested, generating the rows of $M$ on the fly in BW1 has a negligible computational cost: OTF-SSE$^+$ has running times

comparable and sometimes faster than those of the CCNY implementation, and over $\mathbb{F}_{256}$ the BW1 running time of OTF-AVX2$^+$ approaches that of NOTF-AVX2 as $n$ grows, matching it for the largest instances we ran. This gives a practical answer to the question of the memory–time trade-off raised in [The20, BBC+22, DGY26, Ran25]. Since storing $M$ then requires only $O(mn^2)$ $\mathbb{F}_q$-elements, the memory of BW1 is dominated by the dense blocks $M^{i-1}Z$ and $M^iZ$, whose columns can be split among processing units that generate $M$ on the fly without sharing it; hence, we expect this approach to mitigate the memory access cost of XL.
The CCNY implementation already exploits the structure of $M$, but stores the column indices $\iota(\mu\mu')$ for all pairs $(\mu, \mu')$, that is, $O(N_{D-2} \cdot N_2)$ integers. We avoid storing them at comparable running times by amortizing two costs. We unrank once per group of row blocks and then use the cheaper *step* routine, and we compute all indices $\iota(\mu\mu')$ with $\deg \mu' \leq 2$ together, from a table of size $O(n)$ built once per $\mu$, at $O(1)$ table look-ups per index.

# A Appendix

## A.1 Choosing the chunk size

The efficiency of our OTF solution depends on the chunk size. Let $N_{cores}$ be the number of cores, $N_{D-2}$ the number of blocks, and $c$ the chunk size. Let $N_{task}, I \in \mathbb{Z}$ be such that $N_{D-2} = N_{task} \cdot c + I$, with $0 \leq I < c$. Note that $N_{task}$ is the number of tasks of size $c$ and $I$ is the size of the last incomplete task. Let $q, r$ be such that $N_{task} = qN_{cores} + r$, with $0 \leq r < N_{cores}$. Note that $q$ is the number of full parallel rounds and $r$ is the size of the last unbalanced round.
Let $T_{block}$ be the time to process one block and $T_{sync}$ the synchronization time. In the worst case, $I \approx c$ and $r = 1$, so that in the last round a single core is processing one task of size $c$ and the rest are idle. Then the total time is

$$\begin{aligned} T_{total} &\approx (q+1)c\,T_{block} + (N_{task}+1)T_{sync} \\ &= qc\,T_{block} + (N_{task}+1)T_{sync} + c\,T_{block}. \end{aligned}$$

The first term of the expression is the ideal time without synchronization cost and having no idle time. The second term is the synchronization cost, and since $N_{task} = \lceil N_{D-2}/c \rceil$, it is inversely proportional to $c$. The third term is the cost associated with the unbalanced distribution of tasks and is proportional to $c$.
Asymptotically, in the worst case, the optimal $c$ is $\sqrt{N_{D-2}}$, so that the sum of the last two terms is $O(\sqrt{N_{D-2}})$. Figure 5 shows BW1 time against $c$ for an experiment with $N_{blocks} = 116280$ and 32 cores. We observe that indeed the performance deteriorates for very small and large values of $c$. The performance fluctuates widely for larger values of $c$. Also, there seems to be a range around $\sqrt{N_{D-2}}$ where the performance is better and more stable.

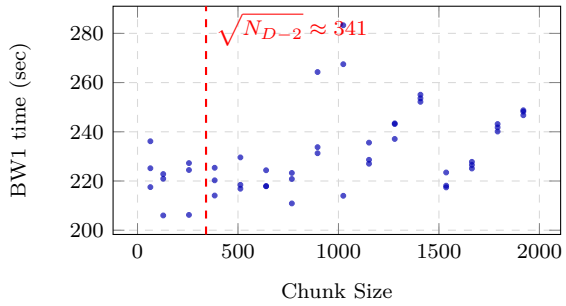



**Figure 5:** BW1 time of OTF-SSE$^+$ for various chunk sizes solving a system with $q = 16$, $n = 14$, $m = 16$ and $\tilde{n} = \tilde{m} = 128$ on an AMD613 EPYC 7763 CPU using 32 cores.

# References


[BBC+22] John Baena, Pierre Briaud, Daniel Cabarcas, Ray Perlner, Daniel Smith-Tone, and Javier Verbel. Improving support-minors rank attacks: Applications to gemss and rainbow. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 376–405, Cham, 2022. Springer Nature Switzerland.

[BCD+26] Ward Beullens, Ming-Shing Chen, Jintai Ding, Boru Gong, Matthias J. Kannwischer, Jacques Patarin, Bo-Yuan Peng, Dieter Schmidt, Cheng-Jhih Shih, Chengdong Tao, and Bo-Yin Yang. Uov: Unbalanced oil and vinegar - algorithm specifications and supporting documentation. Version 3.0, NIST PQC Standardization / uovsig.org, August 2026.

[Beu21] Ward Beullens. Improved cryptanalysis of uov and rainbow. In *Advances in Cryptology – EUROCRYPT 2021*, page 348–373, Berlin, Heidelberg, 2021. Springer-Verlag.

[Beu22] Ward Beullens. Breaking rainbow takes a weekend on a laptop. In *Advances in Cryptology – CRYPTO 2022.*, page 464–479, Berlin, Heidelberg, 2022. Springer-Verlag.

[BFS04] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proceedings of the International Conference on Polynomial System Solving (ICPSS)*, pages 71–72, 2004.

[CCNY12] Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Solving quadratic equations with xl on parallel architectures. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 356–373, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CKPS00] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, pages 392–407, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[CN26] Tung Chou and Ruben Niederhagen. A practical optimization for wiedemann XL. Cryptology ePrint Archive, Paper 2026/1787, 2026.

[Cop94] Don Coppersmith. Solving homogeneous linear equations over gf(2) via block wiedemann algorithm. *Mathematics of Computation*, 62:333–350, 1994.

[DGY26] Jintai Ding, Hao Guo, and Bo-Yin Yang. On memory effects in PWXL variants. Cryptology ePrint Archive, Paper 2026/1819, 2026.

[KAF+10] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit rsa modulus. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 333–350, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Kal95] Erich Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):777–806, 1995.

[Laz83] Daniel Lazard. Gröbner bases, Gaussian elimination and resolution of systems of algebraic equations. In *Computer Algebra – EUROCAL '83*, volume 162 of *Lecture Notes in Computer Science*, pages 146–156, Berlin, Heidelberg, 1983. Springer.

[Leh64] Derrick H. Lehmer. The machine tools of combinatorics. In Edwin F. Beckenbach, editor, *Applied Combinatorial Mathematics*, chapter 1, pages 5–31. John Wiley & Sons, New York, 1964.

[Mas69] James L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, January 1969.

[MDK+10] Wael Said Abdelmageed Mohamed, Jintai Ding, Thorsten Kleinjung, Stanislav Bulygin, and Johannes Buchmann. Pwxl: A parallel wiedemann-xl algorithm for solving polynomial equations over gf (2). In *Conference on Symbolic Computation and Cryptography*, page 89, 2010.

[Ran25] Lars Ran. Wedges, oil, and vinegar – an analysis of UOV in the exterior algebra. Cryptology ePrint Archive, Paper 2025/1143, 2025.

[RDC22] Rahul Roy, Abhijit Das, and Dipanwita Roy Chowdhury. Parallel and distributed implementations of the wiedemann and the block-wiedemann methods over gf (2). In *SECRYPT*, pages 540–547, 2022.

[RG22] Shruti Rawal and Indivar Gupta. Solution of large sparse system of linear equations over gf(2) on a multi node multi gpu platform. *Defence Science Journal*, 72(6):836–845, Dec. 2022.

[SAD13] Bertil Schmidt, Hans Aribowo, and Hoang-Vu Dang. Iterative sparse matrix–vector multiplication for accelerating the block wiedemann algorithm over gf(2) on multi-graphics processing unit systems. *Concurrency and Computation: Practice and Experience*, 25(4):586–603, 2013.

[The20] The Rainbow Team. Response to recent paper by Ward Beullens. https://troll.iis.sinica.edu.tw/by-publ/recent/response-ward.pdf, December 2020. Technical Report / NIST PQC Comment.

[Tho02] Emmanuel Thomé. Subquadratic computation of vector generating polynomials and improvement of the block wiedemann algorithm. *Journal of Symbolic Computation*, 33(5):757–775, 2002.

[Wie86] D. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, 1986.

[YDH+15] Takanori Yasuda, Xavier Dahan, Yun-Ju Huang, Tsuyoshi Takagi, and Kouichi Sakurai. Mq challenge: Hardness evaluation of solving multivariate quadratic problems. In *Proceedings of the NIST Workshop on Cybersecurity in a Post-Quantum World*, Washington, D.C., USA, April 2015.