Algorithmic techniques in algebraic cryptanalysis and their applications

Makarim, R.H.

Part III
Contributions and Main Results

5 M4GB: An Efficient Gröbner Bases Algorithm

Contents

In algebraic cryptanalysis the most crucial step is solving a system of polynomial equations. Since the classical Buchberger algorithm, the development of Gröbner bases algorithms has reached significant improvements, notably with the F_4 [Fau99] and the F_5 [Fau02] algorithm.

Even though the theoretical development of Gröbner bases algorithms has achieved significant progress, many of their efficient implementations remain as closed-source binaries such as the implementation of the F_4 algorithm in FGb [Fau10] and Magma [BCP97]. Note, that the original papers of the F_4 and the F_5 algorithm did not discuss how both algorithms should be efficiently implemented. This is in contrast to the fact that their efficiency were proven by means of computer implementation.

We believe it should be the norm to release the source code of academic implementations belonging to a paper as well. As often important implementation details are not discussed in the paper, the source code is thereby necessary to fully understand and verify the results. Moreover, importantly it facilitates scientific progress and gives a thorough understanding of the problem from practical point-of-view. Two important examples of open-source community development in the domain of lattice reduction algorithms are FPLLL [dt16] and G6K [ADH+19].

On the other hand, the rise of multivariate public-key cryptography (MPKC) as one of the candidates for post-quantum cryptography requires a better understanding of the ability of Gröbner bases algorithms to solve MQ problems in practice. Thus, there is a need to revisit and understand the internal details of Gröbner bases algorithms in order to optimize it against specific types of systems of polynomial equations, in particular systems of polynomial equations underlying MPKC, i.e., dense and overdefined.

This chapter introduces a new Gröbner bases algorithm, whose name is M4GB. The M4GB algorithm is designed to consistently maintain tail-reduced polynomials, i.e., polynomials whose tail is not reducible by the intermediate

basis. This strategy is employed to avoid redundant work in the reduction step of the improved version of the F_4 algorithm. Specifically, the M4GB algorithm can be seen as operating on submatrices of the F_4's coefficient matrices, namely over only those columns which belong to non-reducible monomials. To achieve this, we also introduce a new reduction algorithm that performs an in-place reduction on polynomials of the form $\mathtt{t} \cdot f$, where $\mathtt{t}$ is a term and f is a polynomial, without explicit construction of the polynomial $\mathtt{t} \cdot f$ itself.

The M4GB algorithm is an extension of the Buchberger algorithm and it is described more natively so that one can easily derive an implementation from the high-level description of the algorithm. We have implemented a version of the M4GB algorithm optimized for dense and overdefined quadratic systems of equations over finite fields. We compare our implementation of the M4GB algorithm against three other implementations of Gröbner bases algorithm: FGb [Fau10], Magma [BCP97], and OpenF4 [CVJ]. The benchmarks are based on multivariate polynomial equations that represents multivariate public-key and signature cryptosystems. We observed that the implementation of the M4GB algorithm uses the least total CPU time and the least memory usage among these implementations, often by a significant factor. This is also proven by the ability of the M4GB algorithm to set new records in the computational challenge related to the hardness of MQ problem [YDH+15a]. The result of this chapter is a joint work with Marc Stevens presented at International Symposium on Symbolic and Algebraic Computation (ISSAC) 2017 [MS17].

Related Work. Since the proposal of the Buchberger algorithm [Buc65, Buc06], the progress on the development of Gröbner bases algorithm reached a new direction with the introduction of the F_4 algorithm [Fau99]. The efforts to improve Gröbner bases computation before the introduction of the F_4 algorithm were mainly focused around detecting useless computation in advance [GM88, Buc79]. The F_4 algorithm improves the computation of Gröbner bases by parallelizing the reduction of multiple S-polynomials at once using efficient implementations of (sparse) linear algebra. The reduction algorithm constructs a coefficient matrix corresponding to a finite set of polynomials, followed by the computation of its (reduced-)row echelon form. Moreover the existing criteria to remove unnecessary computation, such as the Gebauer-Möller installation [GM88], can be easily integrated to the F_4 algorithm. The algorithm achieved a significant performance improvement compared to the Buchberger algorithm which was demonstrated by its ability to compute a Gröbner basis for the Cyclic-9 root problem for the first time in practice. Later, Faugère introduced the concept of polynomial *signature* and proposed a new Gröbner bases algorithm called F_5 [Fau02]. The main advantage of the F_5 algorithm over the F_4 algorithm is its ability to avoid zero reductions when the input polynomials are a regular sequence.* The efficiency of the F_5 algorithm was demonstrated by its ability to compute a Gröbner basis for the Cyclic-10 problem.

The description of both the F_4 and the F_5 algorithms leaves several degrees of freedom. For instance, the selection strategy and the choice of reductor were left unexplained. This leads to the development of numerous variants of the F_4 and the F_5 algorithms. For instance, Joux and Vitse proposed a heuris-

* An ordered set $(f_1, \ldots, f_m) \subset \mathcal{R}$ is regular if f_i is not a zero divisor in the quotient ring $\mathcal{R}/\langle f_1, \ldots, f_{i-1}\rangle$ for all $i = 1, \ldots, m$. Equivalently if there exists $g \in \mathcal{R}$ such that $gf_i \in \langle f_1, \ldots, f_{i-1}\rangle$ then g belongs to $\langle f_1, \ldots, f_{i-1}\rangle$ [BFS04, Definition 1].

tic and probabilistic variant of the F_4 algorithm in [JV11] which was claimed to be suitable for algebraic attacks. Another heuristic adaptation of the F_4 algorithm for cryptanalytic purposes was also proposed in [HB13]. The landscape of variants of the F_5 algorithm is even more diverse than for the F_4 algorithm [AP10, FJ03, GGV10, GIW16, EP10, EP11, HA10]. An extensive survey on variants of the F_5 algorithm is described by Eder and Faugère in [EF17].

The F_4 algorithm is implemented in various computer algebra software, such as Maple [Map18], Magma [BCP97], Singular [DGPS18] and SageMath [The18]. Other implementations of the F_4 algorithm are standalone implementations, such as FGb [Fau10] and OpenF4 [CVJ]. The implementations dedicated for Boolean polynomials, i.e., polynomials in $\mathbb{F}_2[x_1, \ldots, x_n]/\langle x_1^2 + x_1, \ldots, x_n^2 + x_n\rangle$, are available in Magma [BCP97] and PolyBoRi [BD09a]. These implementations are particularly suitable for algebraic cryptanalysis of bit-based symmetric-key primitives such as block ciphers and stream ciphers. An implementation of the F_4 algorithm dedicated for shared and distributed memory architecture is described in [Neu13], which focuses on reducing memory usage and avoiding communication overhead. Another implementation of the F_4 algorithm described by Pierce and Monagan in [MP15] also parallelizes the construction of coefficient matrices. Finally, the computational techniques and data structures for Gröbner bases algorithms that employ polynomial signatures is explained in [RS12].

Outline Although it is presented as a variant of the Buchberger's algorithm, the strategy to maintain tail-reduced polynomials can be seen as an improvement of the Simplify function in the reduction step of the F_4 algorithm algorithm. We first revisit the reduction step of the F_4 algorithm together with the Simplify function in Section 5.1. It will be followed by the description of a reduction algorithm in Section 5.2 that prioritizes the reduction of every newly constructed reductor in a recursive manner. We generalize this reduction algorithm to perform an in-place reduction on polynomials of the form $\mathtt{t} \cdot f$, where $\mathtt{t}$ is a term and f is a polynomial. This generalization plays a role in maintaining tail-reduced polynomials and it is employed in the reduction of S-polynomials inside the M4GB algorithm, which will be described in Section 5.3. We discuss our implementation of the M4GB algorithm tailored for dense quadratic overdefined system of equations in Section 5.6. Finally, the benchmark and performance comparison of the M4GB algorithm against several existing implementations of Gröbner bases algorithm is given in Section 5.7. The discussion on the records of MQ challenges solved by the M4GB algorithm will be given in Chapter 6.

5.1 Revisiting the Reduction Step of the F_4 Algorithm

Recall that the original paper of the F_4 algorithm describes two different variants of the algorithm. The first variant is described without any criteria to remove useless critical pairs. The second variant uses two additional routines to improve its performance: Update and Simplify (see Algorithm 4.3 and Algorithm 4.9). The Update function is integrated directly in the main iteration of the F_4 algorithm, whereas the Simplify function is incorporated as a subroutine inside the SymbolicPreprocessing.

Let $I = \langle f_1, \ldots, f_m\rangle \subset \mathcal{R}$ be an ideal of $\mathcal{R}$ generated by $f_1, \ldots, f_m \in \mathcal{R}$. The goal of the Update function is to remove unnecessary critical pairs that appear during the computation of a Gröbner basis of I using the Buchberger's criteria.

Faugerè suggested to use the Gebauer-Möller installation [GM88]. On the other hand, the task of the SIMPLIFY function is to replace polynomials of the form $\mathfrak{m} \cdot f$ by a new "equivalent" and "more reduced" polynomial of the form $\mathfrak{m}' \cdot f'$ such that $\mathfrak{m}' \leq \mathfrak{m}$ where $\mathfrak{m}, \mathfrak{m}' \in \mathsf{M}(\mathcal{R})$ and $f, f' \in I$. The notion of "equivalence" here is that the polynomials $\mathfrak{m}' \cdot f'$ satisfy $\mathsf{LM}(\mathfrak{m} \cdot f) = \mathsf{LM}(\mathfrak{m}' \cdot f')$ as stated in Lemma 4.23. The paper also remarked that in 95% of cases, the monomial $\mathfrak{m}'$ is a variable in the ring $\mathcal{R}$ and $\mathfrak{m}'$ is often equal to x_n [Fau99, Remark 2.6]. At this point, however, the advantage of incorporating the SIMPLIFY function on the performance of the improved F_4 algorithm is not obvious. Our next aim is to derive an intuitive explanation on the performance gain of the F_4 algorithm when using the SIMPLIFY function.

At iteration d, recall that the SIMPLIFY function takes as input $\mathfrak{m} \in \mathsf{M}(\mathcal{R})$, $f \in I$ and a tuple $(F_1, \ldots, F_{d-1})$ such that $F_i \subset \mathcal{R}$ for all $i = 1, \ldots, d-1$. Each F_i corresponds to the constructed coefficient matrix at the i-th iteration. If $\mathfrak{m}$ has no nontrivial divisor, then it returns $(\mathfrak{m}, f)$. Otherwise, for all divisors $\mathfrak{u}$ of $\mathfrak{m}$, it checks if there exists $j \in \{1, \ldots, d-1\}$ such that $\mathfrak{u} \cdot f \in F_j$. This implies that there exists a polynomial p in the (reduced) row echelon form $\tilde{F}_j$ of F_j such that $\mathsf{LM}(p) = \mathsf{LM}(\mathfrak{u} \cdot f)$. When $\mathfrak{u}$ is a proper divisor of $\mathfrak{m}$, then it recursively calls SIMPLIFY$(\mathfrak{m}/\mathfrak{u}, p, (F_1, \ldots, F_{d-1}))$. Otherwise, the function returns $(1, p)$. Suppose that such j exists. We then have

$$\mathsf{LM}(\mathfrak{m}/\mathfrak{u} \cdot p) = \mathsf{LM}(\mathfrak{m}/\mathfrak{u} \cdot \mathfrak{u} f) = \mathsf{LM}(\mathfrak{m} \cdot f)$$

The polynomial $p \in \tilde{F}_j$ is the result of a reduction of the polynomial $\mathfrak{u} \cdot f \in F_j$ where $\mathsf{LM}(\mathfrak{u} \cdot f) = \mathsf{LM}(p)$. More precisely, it is the result of performing tail reduction on $\mathfrak{u} \cdot f$. We can now provide a more intuitive explanation of the purpose of the SIMPLIFY function in the following remark.

Remark 5.1. *Let $I = \langle f_1, \ldots, f_m \rangle \subset \mathcal{R}$ be an ideal of $\mathcal{R}$ generated by polynomials $f_1, \ldots, f_m \in \mathcal{R}$. Given a monomial $\mathfrak{m} \in \mathsf{M}(\mathcal{R})$ and a nonzero polynomial $f \in I$, the goal of* SIMPLIFY *function is to find a nonzero polynomial $g \in I$ such that $\mathsf{LM}(g) = \mathsf{LM}(\mathfrak{m} f)$ and g is a polynomial in I that can be found with the least number of terms in its tail. Thus,* SYMBOLICPREPROCESSING *returns a sparser coefficient matrix or, if possible, a smaller coefficient matrix than* SYMBOLICPREPROCESSINGBASIC. *This effectively reduces the number of field operations when computing the (reduced)-row echelon form of the coefficient matrix.*

Remark 5.2. *For Gröbner bases computations over fields, it is unnecessary to store two distinct nonzero polynomials $f, f' \in \mathcal{R}$ such that $\mathsf{LM}(f) = \mathsf{LM}(f')$ since the reducibility of a term $\mathfrak{t}$ only depends on the divisibility of the monomial of $\mathfrak{t}$ by $\mathsf{LM}(g)$ for some $g \in G$ where G is the intermediate basis.*

However, the SIMPLIFY function is implemented at the expense of storing all previously constructed coefficient matrices $F_1, \ldots, F_{d-1}$ together with their corresponding (reduced)-row echelon form $\tilde{F}_1, \ldots, \tilde{F}_{d-1}$. Moreover, the way to find an "equivalent" representation for $\mathfrak{m} \cdot f$ is done by constructing a finite sequence $(\mathfrak{m}_k, f_k)$ such that $(\mathfrak{m}_0, f_0) = (\mathfrak{m}, f)$ and $\mathfrak{m}_{k+1} \leq \mathfrak{m}_k$. Note that in this case there is a trade-off between time and memory: On one end one can improve the performance of the F_4 algorithm by implementing the SIMPLIFY function and storing (part of) $F_j, \tilde{F}_j$. Whereas on the other end one can reduce the

memory usage of F_4 algorithm by ignoring the SIMPLIFY function and discard $F_j, \tilde{F}_j$ in each iteration, at the cost of increasing time complexity to compute Gröbner bases.

5.2 M4GB Reduction

Suppose that we want to reduce a nonzero polynomial $f \in \mathcal{R}$ by a nonempty intermediate basis $G \subset \mathcal{R}$. Let $\mathfrak{t}$ be a term of f and let $g \in G$ such that $\mathsf{LM}(g)$ divides the monomial of $\mathfrak{t}$. The standard reduction technique directly constructs a reductor $r = \mathfrak{t}/\mathsf{LT}(g) \cdot g$ and replaces f by $f - r$ to eliminate $\mathfrak{t}$ from f. It is possible that the polynomial r may be used in future iterations as a reductor. Thus in order to avoid recomputing r from g, one can store r as an element of the intermediate basis G. An approach to efficiently store polynomials in the intermediate basis G, based on Remark 5.1, is to ensure that each polynomial in G is tail-reduced w.r.t. G itself. To achieve this, we develop a new reduction algorithm that prioritizes tail-reduction of the newly constructed reductor over the reduction of f. We refer to this algorithm as the M4GBREDUCTION and it is described in Algorithm 5.1.

The algorithm iterates over each term $\mathfrak{t}$ of a polynomial f and checks the reducibility of $\mathfrak{t}$. In case $\mathfrak{t}$ is reducible then to obtain the reductor that eliminates $\mathfrak{t}$, the function GETREDUCTORBASIC is called (see Algorithm 5.2) with input parameter G and $\mathfrak{t}$. Suppose that $\mathfrak{t} = \mathfrak{c} \cdot \mathfrak{m}$ where $\mathfrak{c} \in \mathbb{F}, \mathfrak{m} \in \mathsf{M}(\mathcal{R})$ and assume there exists a $g \in G$ such that $\mathsf{LM}(g) \mid \mathfrak{m}$. The function GETREDUCTORBASIC returns a reductor for $\mathfrak{t}$ in two separate cases. The first case is when there exists a $g \in G$ such that $\mathsf{LM}(g) = \mathfrak{m}$, otherwise in the second case there is a g such that $\mathsf{LM}(g)$ strictly divides $\mathfrak{m}$. In the former, GETREDUCTORBASIC directly returns g as a reductor while in the latter case it ensures that the newly constructed reductor $\mathfrak{t}/\mathsf{LT}(g) \cdot g$ for $\mathfrak{t}$ is tail-reduced. Note that reducing the tail of $\mathfrak{t}/\mathsf{LT}(g) \cdot g$ is equal to the reducing the polynomial $\ell = \mathfrak{t}/\mathsf{LT}(g) \cdot \mathsf{Tail}(g)$. The function GETREDUCTORBASIC then calls the M4GBREDUCTION with input G and ℓ. The polynomial $\mathfrak{t} + \ell'$ is the tail-reduced reductor for $\mathfrak{t}$, where ℓ' is the polynomial output of M4GBREDUCTION(G, ℓ).

Example 5.3. *Let $\mathcal{R} = \mathbb{F}_2[x_1, x_2, x_3, x_4]$ we use the degree-reverse lexicographic (degrevlex) ordering on the monomials of $\mathcal{R}$. Suppose that we want to compute the remainder after division of $f = x_1^2x_2^3 + x_1x_2^3x_4 + x_1x_3^3 + x_1^3x_4 + x_2x_3^2 + x_4^2 \in \mathcal{R}$ by $G = \{g_1, g_2, g_3\} \subset \mathcal{R}$ where*

$$\begin{aligned} g_1 &= x_1^2x_2^3 + x_1x_3^3 + x_4^2, \\ g_2 &= x_2^3x_4 + x_2x_3 + x_3 + 1, \\ g_3 &= x_1x_2x_3 + x_1x_3. \end{aligned}$$

Note that the polynomials in G are already tail-reduced.

1. $(\mathfrak{t} = x_1^2x_2^3)$ *Since $\mathfrak{t}$ is reducible by G and $\mathsf{LM}(g_1) = x_1^2x_2^3$, then the function* GETREDUCTORBASIC$(G, \mathfrak{t})$ *is called and immediately returns (g_1, G). The polynomial h is equal to*

$$h = h - \mathfrak{t}/\mathsf{LT}(g_1) \cdot \mathsf{Tail}(g_1) = x_1x_3^3 + x_4^2.$$

2. $(\mathfrak{t} = x_1x_2^3x_4)$ *Since $\mathsf{LT}(g_2) \mid \mathfrak{t}$ then the function* GETREDUCTORBASIC$(G, \mathfrak{t})$ *is again called in this iteration. At this step, we have a case where $\mathsf{LM}(g_2)$*

Input: A set $G \subset \mathcal{R}$
Input: A polynomial $f \in \mathcal{R}$
Output: An updated basis $G' \supseteq G$ with $\langle G' \rangle = \langle G \rangle$
Output: Either 0 or a full-reduced nonzero polynomial $h = f - \sum_i g_i p_i \in \mathcal{R}$ where $g_i \in G', p_i \in \mathcal{R}$ such that $\mathsf{LM}(g_i p_i) \leq \mathsf{LM}(f)$ whenever $g_i p_i \neq 0$

```
1 h ← 0
2 G' ← G
3 forall t ∈ T(f) do
4     if ∃g ∈ G' : LT(g) | t then
          /* LM(g) = LM(t), g is tail-reduced */
5         (G', g) ← GetReductorBasic(G', t)
6         h ← h − (t/LT(g)) · Tail(g)
7     else
8         h ← h + t
9 return (G', h)
```

Algorithm 5.1: M4GBReduction

strictly divides the monomial of t. *Thus, we construct the polynomial* $\ell = \mathsf{t}/\mathsf{LT}(g_2) \cdot \mathsf{Tail}(g_2) = x_1x_2x_3 + x_1x_3 + x_1$ *and call* M4GBReduction(G, ℓ).

2.1. $(\mathsf{t} = x_1x_2x_3)$ *Since* $\mathsf{LT}(g_3) | \mathsf{t}$ *and* $\mathsf{LM}(g_3) = x_1x_2x_3$ *then the function* GetReductorBasic(G, t) *returns* (g_3, G). *Thus,*

$$h = h - (\mathsf{t}/\mathsf{LT}(g_3)) \cdot \mathsf{Tail}(g_3) = x_1x_3.$$

2.2. $(\mathsf{t} = x_1x_3)$ *Since* t *is not reducible by* G, *then* $h = h + \mathsf{t} = 0$.

2.3. $(\mathsf{t} = x_1)$ *Similarly in this case* t *is not reducible by* G, *and finally*

$$h = h + \mathsf{t} = x_1.$$

The newly constructed reductor g_4 *is equal to*

$$g_4 = \mathsf{t} + \ell' = x_1x_2^3x_4 + x_1.$$

The intermediate basis G *now consists of the polynomials* $G = \{g_1, g_2, g_3, g_4\}$ *and* GetReductorBasic *returns* (G, g_4). *The polynomial* h *is now equal to*

$$h = h - (\mathsf{t}/\mathsf{LT}(g_4)) \cdot \mathsf{Tail}(g_4) = x_1x_3^3 + x_4^2 + x_1.$$

3. At this stage, no remaining term of f *is reducible by* G. *One can verify that* M4GBReduction *eventually returns* (G, h) *where*

$$h = x_1^3x_4 + x_2x_3^2 + x_1$$

and $G = \{g_1, g_2, g_3, g_4\}$.

Input: A set $G = \{g_1, \ldots, g_s\} \subset \mathcal{R}$
Input: A term $\mathfrak{t} \in \mathsf{T}(\mathcal{R})$
Output: An updated basis $G' \supseteq G$ with $\langle G' \rangle = \langle G \rangle$
Output: A tail-reduced polynomial $h \in G'$ such that $\mathsf{LM}(h) = \mathsf{LM}(\mathfrak{t})$

```
1  G' ← G
2  if ∃g ∈ G' : LM(g) = LM(t) then
3  |  h ← g
4  else
5  |  Select g ∈ G' such that LM(g) | LM(t)
6  |  ℓ ← t/LT(g) · Tail(g)
7  |  (G', ℓ') ← M4GBReduction(G', ℓ)
8  |  g_{s+1} ← t + ℓ'
9  |  G' ← G' ∪ g_{s+1}
10 |  h ← g_{s+1}
11 return (G', h)
```

Algorithm 5.2: GetReductorBasic

5.3 M4GB Algorithm

In this section we give the description of the M4GB algorithm. It is an extension of the Buchberger's algorithm with the specific aim to store and process polynomials only in tail-reduced form. In the remaining of this chapter, the intermediate basis in M4GB algorithm is denoted by M.

5.3.1 M4GB Invariant

The main difference in the strategy of maintaining intermediate basis in the M4GB algorithm compared to the Buchberger's algorithm described in Algorithm 4.2 is that the intermediate basis M in M4GB is not kept minimal, i.e., for some $f \in M$ there exists $g \in M$ such that $\mathsf{LM}(g)$ strictly divides $\mathsf{LM}(f)$. The reason for keeping redundant elements in M is to avoid their recomputation as reductors in the subsequent iterations. In addition to that, M4GB algorithm sets a restriction that there exists no two distinct nonzero polynomials in M with equal leading monomial. The rationale of this approach is based on Remark 5.2.

However, there are two problems with such strategy. Firstly, applying Gebauer-Möller installation on M would remove the redundant polynomials from M, thus contradicting our approach. Secondly, if we let M to be non-minimal and at the same time we did not employ the Gebauer-Möller installation, then the algorithm would generate many useless critical pairs. To overcome this situation, we introduce a set of monomials $L \subset \mathsf{M}(\mathcal{R})$. The purpose of the set L is to indicate the elements of M that constitute a minimal intermediate basis of the ideal. In addition to that, the set L replaces the role of the intermediate basis M inside the Gebauer-Möller installation and the reducibility of a polynomial by the set M can be determined more efficiently using L. We summarize this strategy in the following notion.

Definition 5.4 (M4GB Invariant)**.** *The pair* (L, M) *where* $L \subset \mathsf{M}(\mathcal{R})$ *and* $M \subset \mathcal{R}$ *is said to satisfy the* M4GB *invariant if*

1. *For all* $f, g \in M$, $\mathsf{LM}(f) = \mathsf{LM}(g)$ *implies that* $f = g$. *For any monomial* $\mathfrak{m} \in \mathsf{LM}(M)$, *we define* $M[\mathfrak{m}]$ *to be the polynomial* $g \in M$ *such that* $\mathsf{LM}(g) = \mathfrak{m}$. *Moreover, for any nonempty subset* $L \subseteq \mathsf{LM}(M)$ *we also define* $M[L] = \{f \in M : \mathsf{LM}(f) \in L\}$.
2. *The set* L *is a subset of* $\mathsf{LM}(M)$,
3. *The ideal* $\langle M \rangle$ *is equal to* $\langle M[L] \rangle$,
4. *For all* $f \in M$, *there exists* $\mathfrak{m} \in L$ *such that* $\mathfrak{m} \mid \mathsf{LM}(f)$,
5. *For all* $f \in M$, *the polynomial* f *is not tail-reducible by* L.

5.3.2 Generalization of M4GB Reduction

In the previous section, we have introduced a new approach to perform polynomial reduction with the M4GBReduction. In the actual M4GB algorithm, we generalize the M4GBReduction to polynomials of the form $\mathfrak{t} \cdot f$ where $\mathfrak{t} \in \mathsf{T}(\mathcal{R})$ and $f \in \mathcal{R}$. This generalization allows reduction on $\mathfrak{t} \cdot f$ without the explicit construction of the polynomial $\mathfrak{t} \cdot f$ itself, a strategy that is not employed by both the Buchberger and the F_4 algorithm. The resulting algorithm is called the MulFullReduce and it is described in Algorithm 5.3.

Input: A pair of set (L, M) where $L \subset \mathsf{M}(\mathcal{R}), M \subset \mathcal{R}$ satisfying M4GB invariant
Input: A term $\mathfrak{t} \in \mathsf{T}(\mathcal{R})$
Input: A polynomial $f \in \mathcal{R}$
Output: An updated intermediate basis $M' \supseteq M$ with $\langle M' \rangle = \langle M \rangle$
Output: Either 0 or a full-reduced nonzero polynomial $h = \mathfrak{t}f - \sum_i g_i p_i \in \mathcal{R}$, where $g_i \in M', p_i \in \mathcal{R}$ such that $\mathsf{LM}(g_i p_i) \leq \mathsf{LM}(\mathfrak{t}f)$ whenever $g_i p_i \neq 0$

```
1  M' ← M
2  h ← 0
3  forall 𝔰 ∈ T(f) do
4      𝔯 ← 𝔱 · 𝔰
5      if ∃𝔪 ∈ L : 𝔪 | 𝔯 then
6          (M', g) ← GetReductor(L, M', 𝔯)
7          h ← h − (𝔯/LT(g)) · Tail(g)
8      else
9          h ← h + 𝔯
10 return (M', h)
```

Algorithm 5.3: MulFullReduce Algorithm.

The MulFullReduce function generalizes the M4GBReduction by performing divisibility check and the reduction on the terms $\mathfrak{t} \cdot \mathfrak{s}$ for all $\mathfrak{s} \in \mathsf{T}(f)$. Another difference with the M4GBReduction is that the reducibility test in MulFullReduce uses the set L instead of the intermediate basis M, which minimizes the number of divisibility checks. The function GetReductor is an adjustment of the GetReductorBasic by incorporating the set L and M.

Input: A pair of set (L, M) where $L \subset \mathsf{M}(\mathcal{R}), M \subset \mathcal{R}$ satisfying M4GB invariant
Input: A term $\mathfrak{t} \in \mathsf{T}(\mathcal{R})$
Output: An updated intermediate basis $M' \supseteq M$ with $\langle M' \rangle = \langle M \rangle$
Output: A tail-reduced reductor $h \in \langle M \rangle$ with $\mathsf{LM}(h) = \mathsf{LM}(\mathfrak{t})$

1 $M' \leftarrow M$
2 **if** $\mathsf{LM}(\mathfrak{t}) \in \mathsf{LM}(M')$ **then**
3 $\quad h \leftarrow M'[\mathsf{LM}(\mathfrak{t})]$
4 **else**
5 $\quad$ Select $\mathfrak{m} \in L$ such that $\mathfrak{m} \mid \mathfrak{t}$
6 $\quad r \leftarrow M'[\mathfrak{m}]$
7 $\quad (M', \ell') \leftarrow \textsc{MulFullReduce}(L, M', \mathfrak{t}/\mathsf{LT}(r), \mathsf{Tail}(r))$
8 $\quad h \leftarrow \mathfrak{t} + \ell'$
9 $\quad M' \leftarrow M' \cup \{h\}$
10 **return** (M', h)

Algorithm 5.4: GetReductor Algorithm.

Similar to the MulFullReduce function, it uses the set L to find a polynomial in M that will be used to construct the reductor.

When a term in $\mathfrak{t}f$ is reducible by M' (line 5-7 of Algorithm 5.3), GetReductor is called which would possibly followed by another call to MulFullReduce to perform tail-reduction on a newly constructed reductor (line 7 of Algorithm 5.4). One then asks if the initial call to the MulFullReduce eventually terminates. Suppose that we call $\textsc{MulFullReduce}(L, M, \mathfrak{t}, f)$ and let $\mathfrak{s} \in \mathsf{T}(f)$ such that there exists $\mathfrak{m} \in L$, $\mathfrak{m} \mid \mathfrak{r} = \mathfrak{t}\mathfrak{s}$ and $\mathsf{LM}(\mathfrak{r}) \notin \mathsf{LM}(M')$. Then GetReductor will call $\textsc{MulFullReduce}(L, M', \mathfrak{r}/\mathsf{LT}(r), \mathsf{Tail}(r))$. Assuming $\mathsf{Tail}(r) \neq 0$, the important remark here is that

$$\mathsf{LM}(\mathfrak{r}/\mathsf{LT}(r) \cdot \mathsf{Tail}(r)) = \mathsf{LM}(\mathsf{Tail}(\mathfrak{r}/\mathsf{LT}(r) \cdot r)) < \mathfrak{r} \leq \mathsf{LM}(\mathfrak{t}f).$$

Since by definition a monomial ordering $<$ is a well-ordering, (equivalently every strictly decreasing sequence of monomials eventually terminates). Therefore a call to $\textsc{MulFullReduce}(L, M, \mathfrak{t}, f)$ always terminates.

5.3.3 Computation of S-Polynomials

The function MulFullReduce can easily be used to avoid unnecessary step in the computation of S-polynomials. Recall that the S-polynomial of $f, g \in \mathcal{R} \backslash \{0\}$ is defined as

$$\mathsf{Spoly}(f, g) = \frac{u}{\mathsf{LT}(f)} \cdot f - \frac{u}{\mathsf{LT}(g)} \cdot g$$

where $u = \mathsf{LCM}(\mathsf{LM}(f), \mathsf{LM}(g))$. The notion of S-polynomial was introduced to produce cancellation of leading terms. Despite this cancellation is known in advance, Buchberger algorithm and F_4 algorithm still perform explicit computation of S-polynomials (in the F_4 algorithm, an S-polynomial is computed by substraction of two rows in the coefficient matrix that correspond to a critical pair). Recall that by Remark 4.4 the S-polynomial $\mathsf{Spoly}(f, g)$ can be expressed as

$$\mathsf{Spoly}(f, g) = \frac{u}{\mathsf{LT}(f)} \cdot \mathsf{Tail}(f) - \frac{u}{\mathsf{LT}(g)} \cdot \mathsf{Tail}(g).$$

The M4GB algorithm computes the reduction of S-polynomials by taking the advantage that the cancellation of leading terms is known in advance. It first computes the reduction of the polynomial $\frac{u}{\mathsf{LT}(f)} \cdot \mathsf{Tail}(f)$ followed by the reduction of $\frac{u}{\mathsf{LT}(g)} \cdot \mathsf{Tail}(g)$. These reductions are done by calling the function MulFullReduce with parameters $(L, M, u/\mathsf{LT}(f), \mathsf{Tail}(f))$ and $(L, M, -u/\mathsf{LT}(g), \mathsf{Tail}(g))$ respectively. Finally, the resulting polynomial outputs are added.

5.3.4 The M4GB Algorithm

The complete description of the M4GB algorithm is given in Algorithm 5.5. During the initialization phase and for every new polynomial found in the ideal as a result of the reduction of S-polynomial, the algorithm calls the function UpdateReduce. Suppose that we call it with parameters (L, M, P, f). The goal of the UpdateReduce function is to perform tail-reduction on polynomials in the intermediate basis M using f. The tail-reduction of polynomials in M is achieved in two phases. The first phase, described in line 3-7 of Algorithm 5.6, is a collection of all necessary reductors constructed from f that will be used to eliminate terms in the tail of polynomials in M. The constructed reductors are stored in the set H. In addition to the reductors of polynomials in M, the same step also constructs reductors to perform tail-reduction on polynomials in H itself. This can be seen on the set Q that keeps updated during the iteration. This will be followed by the second phase, which is the actual tail-reduction of polynomials in M and H. Finally, UpdateReduce calls the Gebauer-Möller installation as described in Algorithm 4.3.

Input: A finite nonempty subset $F \subset \mathcal{R}$
Output: A Gröbner basis G of $\langle F \rangle$

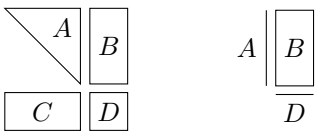
```
1  L ← {}
2  M ← {}
3  P ← {}
4  forall f ∈ F do
5  |  (M, f) ← MulFullReduce(L, M, 1, f)
6  |  (L, M, P) ← UpdateReduce(L, M, P, f)
7  while P ≠ {} do
8  |  (f_LM, g_LM) ← Select(P)
9  |  P ← P \ {(f_LM, g_LM)}
10 |  (f, g) ← (M[f_LM], M[g_LM])
11 |  u ← LCM(f_LM, g_LM)
12 |  (M, h1) ← MulFullReduce(L, M, u/LT(f), Tail(f))
13 |  (M, h2) ← MulFullReduce(L, M, u/LT(g), Tail(g))
14 |  h ← h1 − h2
15 |  if h ≠ 0 then
16 |  |  (L, M, P) ← UpdateReduce(L, M, P, h)
17 G ← M[L]
18 return G
```

Algorithm 5.5: M4GB Algorithm.

Input: A pair of set (L, M) where $L \subset \mathsf{M}(\mathcal{R}), M \subset \mathcal{R}$ satisfying M4GB invariant
Input: A set of critical pairs $P \subset \mathsf{M}(\mathcal{R}) \times \mathsf{M}(\mathcal{R})$
Input: A polynomial $f \in \langle M \rangle \setminus M$
Output: An updated L', M', satisfying M4GB invariant and $\langle M[L] \rangle = \langle M'[L'] \rangle$
Output: An updated set of critical pairs P'

$M' \leftarrow M$
$L' \leftarrow L$
// The set H contains polynomials to add to M', to maintain M4GB invariant while adding f
$H \leftarrow \{\mathsf{LC}(f)^{-1} \cdot f\}$
// The set Q contains monomials that needs to be processed
$Q \leftarrow \mathsf{M}(\mathsf{Tail}(M' \cup H)) \setminus \mathsf{LM}(H)$
while $\exists \mathfrak{m} \in Q : \mathsf{LM}(f) \mid \mathfrak{m}$ **do**
    $\mathfrak{m}' \leftarrow \max\{\mathfrak{m} \in Q : \mathsf{LM}(f) \mid \mathfrak{m}\}$
    $(M', h) \leftarrow \textsc{MulFullReduce}(L', M', \mathfrak{m}'/\mathsf{LT}(f), \mathsf{Tail}(f))$
    $H \leftarrow H \cup \{\mathfrak{m}' + h\}$
    $Q \leftarrow \mathsf{M}(\mathsf{Tail}(M' \cup H)) \setminus \mathsf{LM}(H)$
while $H \neq \{\}$ **do**
    Select $h \in H$ such that $\mathsf{LM}(h) = \min(\mathsf{LM}(H))$
    $H \leftarrow \{g - ch : g \in H, c$ is the coefficient of LM(h) in Tail(g)$\}$
    $M' \leftarrow \{g - ch : g \in M', c$ is the coefficient of LM(h) in Tail(g)$\}$
    $M' \leftarrow M' \cup \{h\}$
    $H \leftarrow H \setminus \{h\}$
$(L', P') \leftarrow \textsc{Update}(L', P, \mathsf{LM}(f))$
return (L', M', P')

Algorithm 5.6: UpdateReduce Algorithm.



(a) *F_4 matrix* **(b)** M4GB*'s M*

Columns represent monomials, with non-reducible monomials last. Rows represent polynomials, with S-polynomial halves at the bottom (CD) and reductor polynomials at the top (AB). We remark that A in M4GB *is a diagonal submatrix, represented as rowlabels of B.*

Figure 7: *Visualization of F_4's matrix and* M4GB*'s M*

5.4 Comparison with the F_4 Algorithm

In this section we shall discuss a comparison between M4GB algorithm and the F_4 algorithm.

The F_4 algorithm achieves the reduction of S-polynomials by translating the reduction of many S-polynomials to the computation of (reduced)-row echelon form of the coefficient matrix. The M4GB algorithm performs addition and scalar multiplication on tail of polynomials, which can be efficiently implemented as scalar multiplication and addition of coefficient vectors. Both algorithms keep track of prior reduced polynomials to speed up computations in the next iterations. The F_4 algorithm achieved this using the SIMPLIFY function, which requires the algorithm to store all previously constructed coefficient matrices and their corresponding (reduced)-row echelon forms. In the M4GB algorithm, this is clearly achieved by the set M that not only contains the tail-reduced intermediate basis but also tail-reduced multiples thereof that have been used as reductors.

It should be noted that the M4GB algorithm has a more native description instead of translating desired computations into an external linear algebra computation. However, there are more important differences between the M4GB and the F_4 algorithm that would indicate that the former is more computational and memory efficient than the latter.

Firstly, the M4GB algorithm reduces one critical pair at a time in contrast to the F_4 algorithm that selects many critical pairs. In fact, the best known selection strategy for the F_4 algorithm is to choose all critical pairs of the smallest degree (normal selection strategy). This seems to be the most efficient even if more critical pairs are selected than necessary, because it reduces the overhead associated in translating to and from coefficient matrices. Also reducing the number of critical pairs may not significantly decrease the matrix size anyway. Unfortunately, when a relatively small number of these critical pairs would be sufficient, the F_4 algorithm wastes a significant amount of computation and memory. This is easily seen in the complexity graph for a growing set of problems where there are significant "jumps" whenever the degree of regularity increase (see Figure 8). By operating on a single critical pair (or small batches) at a time, the M4GB algorithm does not have this disadvantage as can be seen from our experiments.

Secondly, the M4GB algorithm operates on coefficient vectors over non-

reducible monomials, which in effect eliminates unnecessary computations and memory use involving reducible monomials. In Figure 7 we have visualized the M4GB's M against matrices used by the F_4 algorithm to showcase this benefit. One can directly see that the F_4 algorithm works with the upper-triangular matrix A as well as larger matrices C and D. Note that for the M4GB's M, instead of representing A as a diagonal matrix, we represent A as a single column of leading terms (instead of coefficients), or equivalently as the labels for the coefficient rows in B. It is well known that matrices generated by the F_4 algorithm have special structure and most row pivots are known in advance, namely those in A related to the reductor polynomials. Linear algebra software can take advantage of this special structure, but inherently must spent computation and memory in keeping track of coefficients related to reducible monomials. In contrast, whenever a reducible terms arises, the M4GB algorithm acts on this and reduces it without ever having to store the reducible term in the result.

5.5 Correctness

We shall now discuss the correctness of the M4GB algorithm to compute a Gröbner bases. We limit ourselves to proving the correctness of full-reduction in the initial basis of the ideal and the S-polynomials since the algorithm itself performs the same steps as Buchberger's algorithm for any selection strategy and Update function.

The MulFullReduce algorithm computes $\mathfrak{t} \cdot f$ by multiplying each term $\mathfrak{s} \in \mathsf{T}(f)$ by $\mathfrak{t}$ and if $\mathfrak{t}\cdot\mathfrak{s}$ is non-reducible by L, the result is added to h. Otherwise, when the term $\mathfrak{r} = \mathfrak{t} \cdot \mathfrak{s}$ is reducible by L it will immediately reduce it with a reductor $g \in M$ with $\mathsf{LM}(g) = \mathsf{LM}(\mathfrak{r})$ retrieved by calling GetReductor:

$$h \leftarrow h + \mathfrak{r} - (r/\mathsf{LT}(g)) \cdot g = h - (r/\mathsf{LT}(g)) \cdot \mathsf{Tail}(g).$$

Assuming correctness of GetReductor we have that (L, M) satisfies the M4GB invariant at every step and that any such g is tail-reduced by L itself*. Thus $\mathsf{Tail}(g)$ is full-reduced by L and the resulting h is always full-reduced by L.

For the correctness of GetReductor, if for the input term $\mathfrak{t}$ there exists a polynomial $g \in M$ with $\mathsf{LM}(g) = \mathsf{LM}(\mathfrak{t})$ then it immediately returns g which is tail-reduced by L due to M4GB invariant. Otherwise, it selects a monomial $\mathfrak{m} \in L$ such that $\mathfrak{m} \mid \mathsf{LM}(\mathfrak{t})$ and retrieves the unique $r \in M$ with $\mathsf{LM}(r) = \mathfrak{m}$ (which always exists since $L \subset \mathsf{LM}(M)$). It then computes

$$\mathfrak{t}/\mathsf{LT}(r) \cdot r = \mathfrak{t} + \mathfrak{t}/\mathsf{LT}(r) \cdot \mathsf{Tail}(r)$$

in tail-reduced form $\mathfrak{t} + \ell'$ by calling MulFullReduce to compute the full-reduced tail ℓ'. Note that the set $M \cup \{\mathfrak{t}+\ell'\}$ still satisfies the M4GB invariant.

We will now discuss the correctness of UpdateReduce. It first updates the set M so that all polynomials $g \in M$ are not tail-reducible by $L \cup \{\mathsf{LM}(f)\}$. Starting with $H = \{\mathsf{LC}(f)^{-1} \cdot f\}$, it will repeatedly insert $\mathfrak{m}'/\mathsf{LT}(f) \cdot f$ into H for the largest monomial $\mathfrak{m}' \in \mathsf{M}(\mathsf{Tail}(M \cup H)) \setminus \mathsf{LM}(H)$ that is reducible

* We also remark that L remains unchanged, i.e., a call to GetReductor does not cause h to not be tail-reduced w.r.t. L.

by f that still needs to be handled. Note that for any such $\mathfrak{m}'$, it will call MulFullReduce which only calls GetReductor$(L, M, \mathfrak{r})$ for $\mathfrak{r} < \mathfrak{m}'$. The sequence of chosen monomials $\mathfrak{m}'$ in line 4 of Algorithm 5.6 is monotonic decreasing and thus finite, as any polynomial h in line 5 has $\mathsf{LM}(h) < \mathfrak{m}'$. After this procedure, at the beginning of step 8, for any monomial $\mathfrak{m}' \in \mathsf{M}(\mathsf{Tail}(M \cup H))$ reducible by f, there exists $h \in H$ with $\mathsf{LM}(h) = \mathfrak{m}'$. At the same time, every $g \in M \cup H$ are tail-reduced by L due to the M4GB invariant and the correctness of MulFullReduce. Therefore, after processing all $h \in H$ in line 9-13 we have that no monomial $\mathfrak{m}' \in \mathsf{M}(\mathsf{Tail}(M))$ is reducible by $L \cup \mathsf{LM}(f)$. Since eventually $\mathsf{LC}(f)^{-1} \cdot f \in M$ due to the step in line 1 and 12, the M4GB invariant will be broken during the steps in line 8-13, however this is fixed when $\mathsf{LM}(f)$ is inserted to L by the Update function in line 14.

For the correct application of Gebauer-Möller installation, let $g_1, g_2, \ldots$ be the sequence of polynomials passed to the UpdateReduce function for M4GB. Note that Gebauer-Möller installation may remove an element g_i from the intermediate basis but keep the critical pair (g_i, h) for some polynomial h in the ideal. It suffices to use a proper references (integer indexing or leading monomial) in the set of critical pairs. Since the polynomials $g_1, g_2, \ldots$ are inserted to M, never removed, never replaced, and only further tail-reduced by UpdateReduce, in M4GB we represent critical pairs by leading monomials and hence Algorithm 5.5 extracts the correct polynomials in line 10.

Lastly, it remains to verify that the steps in line 11-14 of Algorithm 5.5 indeed compute the full-reduction of the S-polynomial of f and g. The full-reduction of $\mathsf{Spoly}(f, g)$ can be computed as the differences of the full-reduction of $(u/\mathsf{LT}(f)) \cdot \mathsf{Tail}(f)$ and $u/\mathsf{LT}(g) \cdot \mathsf{Tail}(g)$ where $u = \mathsf{LCM}(\mathsf{LM}(f), \mathsf{LM}(h))$, which is computed in line 12-13 of Algorithm 5.5 since full-reduction distributes over polynomial addition for fixed basis. Let $G = \{f \in M : \mathsf{LM}(f) \in L\}$ is fixed and for any monomial u reducible by G, a fixed tail-reduced polynomial h is given by GetReductor. Let f_1, f_2 be any two polynomials with coefficients $c_1, c_2 \in \mathbb{F}$ for monomial u respectively. In the full-reduction of f_1, f_2 and $f_1 + f_2$ the only tail-reduced polynomial that affects the terms with monomial u is the identical corresponding h with $\mathsf{LM}(h) = u$. Without loss of generality, assume that h is monic, then $f_1 - c_1 h, f_2 - c_2 h$ and $(f_1 + f_2) - ch$ are full-reduced if and only if $c_1 + c_2 = c$, which implies that full-reduction distributes over polynomial addition for fixed tail-reduced reductors for reducible monomials.

5.6 Efficient Implementation

We provide a proof-of-concept implementation of the M4GB algorithm. The implementation is tailored for dense overdefined systems over a finite field. Note that an overdefined system of equations over a finite field typically have a unique solution over the base field. In particular, we evaluate our implementation against systems of polynomials equations that represent public-keys in multivariate public-key and digital signature algorithm. The source code of the M4GB algorithm, written in `C++11`, is available under the GPLv3 license at

`https://github.com/cr-marcstevens/m4gb`.

Our implementation of the algorithm supports degree-reverse lexicographic ordering, which is considered as a favorable ordering for Gröbner bases computation. It consists mainly of finite field, monomials, parser, and threadpool

implementations. It supports prime fields and even-characteristic finite fields of small size. To facilitate timing and memory comparison, we also provide convenient wrappers for FGb and OpenF4. The implementation of the M4GB algorithm is also configurable at compile-time to optimize the run-time performance and to produce more efficient compiled code. It automatically selects the most efficient data types based on the order of the finite field and the number of variables in the polynomial ring.

The M4GB algorithm performs computations on leading terms and tails separately and frequently retrieves polynomials by their leading monomial. The most efficient data structure for M is a hash-map, mapping the leading monomial to the leading coefficient and the tail.

Operations on the tails of polynomials in M are scalar multiplication and polynomial addition which, in the case of dense polynomials, are more efficient to compute if these tails are stored as coefficient vectors relative to a global list of non-reducible monomials in increasing order and are truncated by removing trailing zeros. The global list stores monomials up to the largest least common multiple of leading monomials in the set of critical pairs. It enables a simpler implementation that does not have to deal with ad-hoc column insertions but at the same time it allows a faster way to determine all reducible and non-reducible monomials using an approach comparable to the sieve of Eratosthenes [Sho08, pg. 115]. The monomials are represented as machine-size integers rather than as exponent vectors. We implemented a fast order-preserving encoder/decoder for monomials that utilizes small look-up tables that fit CPU cache memory.

For system of equations that represent the public-key of multivariate cryptosystem, the order of the underlying finite field $\mathbb{F}_q$ is typically small with $q \leq 256$. We have chosen to implement finite field operations in a relatively simple way by using log and reverse-log lookup-tables, which remains performant at least up to $q \leq 65535$. For $q \leq 256$ we additionally use a two-dimensional array as a multiplication look-up table that easily fits into CPU cache memory. Furthermore, for prime fields with $q \leq 31$ we use a three-dimensional array as a multiply-and-add look-up table, i.e., the entry $A[i][j][k]$ stores the value $((i \cdot j) + k) \pmod q$ for all $i, j, k \in \mathbb{F}_q$.

5.7 Benchmark and Performance Comparison

We compared our implementation of the M4GB algorithm against existing state-of-the-art implementations of Gröbner bases algorithm: FGb [*], Magma version 2.20-6 [†], and OpenF4[‡]. While the first two are well-known implementations of Gröbner bases algorithm, the latter came to our attention when it was proposed as an optional package for SageMath [§]. Unlike Magma and FGb, OpenF4 is an open-source implementation of F_4 algorithm that uses SIMD CPU instructions which makes it an attractive competitor to existing closed-source implementation of Gröbner bases algorithm.

[*] Available at `http://www-polsys.lip6.fr/~jcf/FGb/C/index.html`.
[†] Available at `http://magma.maths.usyd.edu.au/magma/`.
[‡] Available at `https://github.com/nauotit/openf4`.
[§] See `https://trac.sagemath.org/ticket/18749`.

5.7.1 Machines

There are two different machines used for benchmarking purposes :

A) Dual Intel Xeon E5-2650 v3 @ 2.3 GHz with 128GB RAM and

B) Quad Intel Xeon E5-4640 @ 2.4 GHz with 132GB RAM.

The first machine has been used to run and compare the performance of M4GB, FGb, and OpenF4 while the comparison of M4GB with Magma was done in the second machine. Both machines are Non-Uniform Memory Access (NUMA) machine where memories are distributed accross different CPU chips. To ensure that the cost of process and memory transfers among CPU chips does not affect the experimental result, we enforce the processes to run in one CPU chip and its associated memory using `numactl` available in most UNIX system. We are also aware that the processors in both machines support Intel Hyper-Threading technology, and we ensure all processes run in the physical cores.

5.7.2 Testcases

The test cases used to benchmark and compare the chosen implementations of Gröbner bases algorithms consist of polynomial equation systems relating to multivariate public-key encryption and signature schemes. These polynomial systems are dense, quadratic, overdefined, and all coefficients are randomly chosen from the finite field. The public challenges to solve MQ problems* are based on three different finite fields: $\mathbb{F}_2, \mathbb{F}_{2^8}, \mathbb{F}_{31}$. Since $\mathbb{F}_{2^8}$ is not supported by FGb and $\mathbb{F}_2$ needs more specialized data structure for the polynomials, our benchmarks are limited to the field $\mathbb{F}_{31}$. The number of variables (n) and equations (m) are chosen such that the execution can be completed within a reasonable time.

Based on the number of variables and equations, there are two types of equation system used in the experiment. The first type is a strongly over-defined system of equations with $m = 2n$. The second type is a weakly overdefined system of equations with $m = n+1$. The first type of equation system represents equation system from multivariate public-key cryptography, while the second type of equation system is the result of hybrid-approach in solving underdefined system of equations that represent multivariate signature (see Chapter 6).

5.7.3 Results

Our benchmarking results for $m = 2n$ and $m = n + 1$ are listed in the Table 5 and Table 6 respectively. In order to allow comparisons in one single graph and table, the runtime and memory usage of Magma had to be projected. The actual results of comparison between Magma and M4GB is available in Table 8 and Table 9. Note that in light of the large ratios between the performance of Magma and the performance of M4GB, the margin of error caused by the projection for Magma should be insignificant. For the same reason and by taking into account the time to complete the executions, it was not worth repeating the experiments many times for the same parameters to obtain more accurate expected time complexities.

* `https://www.mqchallenge.org`

		Total CPU time (sec)			
n	m	OpenF4	FGb	Magma (projected)	M4GB
20	40	206	470	232.17	57
21	42	472	1002	500.26	170
22	44	1145	3118	1616.73	424
23	46	2274	6849	3184.82	1060
24	48	10293	64700	31167.61	2556
25	50	-	151653	77678.58	5575
26	52	-	360055	183628.74	15517
27	54	-	767543	409451.87	46548
		Memory (MB)			
20	40	4240	112	361.84	73
21	42	6640	165	577.34	121
22	44	14368	525	853.84	226
23	46	26135	918	1324.16	395
24	48	161945	1561	8872.94	663
25	50	-	2765	19718.78	1471
26	52	-	4607	25197	3328
27	54	-	8180	39844.84	6799

Table 5: *Performance comparison for systems with n variables and $m = 2n$ equations over $\mathbb{F}_{31}$. The corresponding graph is given in Figure 8.*

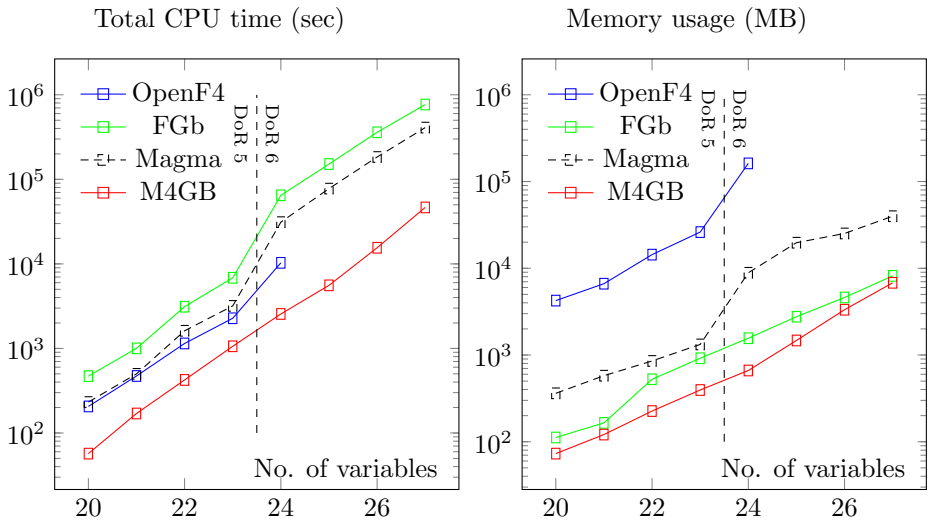


Figure 8: *The graph of CPU time and memory usage (log-y scale) with n variables and $m = 2n$ equations over $\mathbb{F}_{31}$. The vertical dashed line separates the parameter with different degree of regularity (DoR). The corresponding data is described in Table 5.*

		Total CPU time (sec)			
n	m	OpenF4	FGb	Magma (projected)	M4GB
10	11	2.99	5	3.29	0.98
11	12	8.73	21	11.172	2.6
12	13	36.76	134	59.08	13.92
13	14	172.49	642	286.4	58.18
14	15	1258	5850	2810.75	393.19
15	16	7225	36361	17265.5	2424
		Memory (MB)			
10	11	101	33	32.09	17
11	12	341	50	64.12	16
12	13	1463	112	113.59	31
13	14	7622	323	281.53	74
14	15	33460	1098	1104	250
15	16	117396	4118	3320	837

Table 6: *Performance comparison for systems with n variables and $m = n + 1$ equations over $\mathbb{F}_{31}$. The corresponding graph is given in Figure 9.*

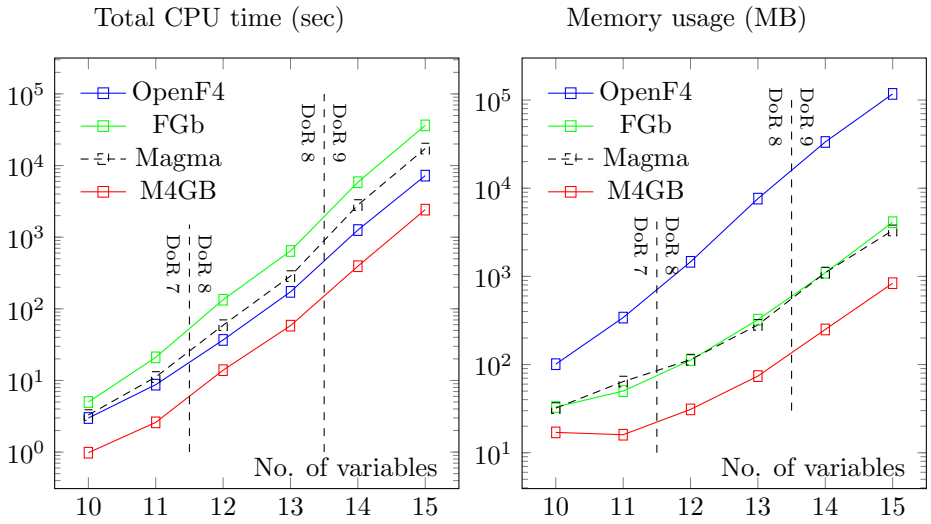


Figure 9: *The graph of CPU time and memory usage (log-y scale) with n variables and $m = n+1$ equations over $\mathbb{F}_{31}$. The vertical dashed line separates the parameter with different degree of regularity (DoR). The corresponding data is described in Table 6.*

5.7.4 Observations and Conclusions

In Table 5 one can see a clear trade-off between the CPU time and the memory usage among OpenF4, FGb, and Magma. OpenF4 performs best in terms of speed followed by Magma and FGb. However, FGb has the least memory usage followed by Magma and OpenF4. The efficiency of time execution of OpenF4 is likely due to its usage of SIMD instructions to perform Gaussian elimination. In terms of CPU time, OpenF4 performs 1.05 up to 3 times faster than Magma. However, its memory usage is at least 11 times more than Magma, which does not allow us to run OpenF4 for $n \geq 25$.

In comparison to OpenF4 (which is the fastest implementation among OpenF4, FGb, and Magma), our implementation of M4GB runs 2.15 and up to 4 times faster and this puts M4GB as the fastest one. In comparison with FGb (which uses the least memory among OpenF4, FGb, and Magma), M4GB uses less memory by a factor between 1.2 and 2.35 and this puts M4GB as an implementation with the least memory usage.

The benchmarking results for $m = n+1$ in Table 6 shows the same ordering in terms of CPU time performance, that is M4GB is the fastest followed by OpenF4, Magma, and FGb. In terms of memory usage, M4GB is also the most efficient one followed by both FGb and Magma which are comparably similar, and OpenF4. M4GB performs 2.6 up to 3.3 faster than OpenF4 and uses less memory by a factor of 1.9 up to 4.4 compared to Magma and FGb.

In order to understand the overall efficiency, we also computed the ratio of time-memory product of other implementations relative to the time-memory product of M4GB in Table 7. For the systems with n variables and $2n$ equations, OpenF4 has the highest ratio, ranging from 141.94 ($n = 23$) to 983.64 ($n = 24$), mainly due to its high memory usage. This is followed by Magma with ratio ranging from 10.07 ($n = 23$) to 186.77 ($n = 25$). Lastly, FGb has overall the smallest ratio ranging from 8.04 ($n = 21$) to 59.6 ($n = 24$). For systems with $n+1$ equations and n variables, OpenF4 also has the highest ratio from 18.13 ($n = 10$) up to 428.22 ($n = 14$). This is followed by FGb, with ratio ranging from 9.9 ($n = 10$) to 73.8 ($n = 15$). Lastly, Magma has the closest ratio to M4GB ranging from 6.34 ($n = 10$) to 31.57 ($n = 14$).

n	m	FGb	Magma	OpenF4
20	40	12.65	20.19	209.91
21	42	8.04	14.04	152.36
22	44	17.08	14.40	171.68
23	46	15.02	10.07	141.94
24	48	59.6	163.19	983.64
25	50	51.13	186.77	-
26	52	32.12	89.60	-
27	54	19.84	51.55	-

n	m	Magma	FGb	OpenF4
10	11	6.34	9.9	18.13
11	12	17.22	25.24	71.56
12	13	15.55	34.78	124.63
13	14	18.73	48.16	305.37
14	15	31.57	65.34	428.22
15	16	28.25	73.8	418.05

Table 7: *The ratio of time-memory product of Magma, FGb, and OpenF4 relative to the time-memory product of* M4GB *for systems with $m = 2n$ equations (left) and $m = n + 1$ equations (right) where n is the number of variables. The corresponding graph is given in Figure 10.*

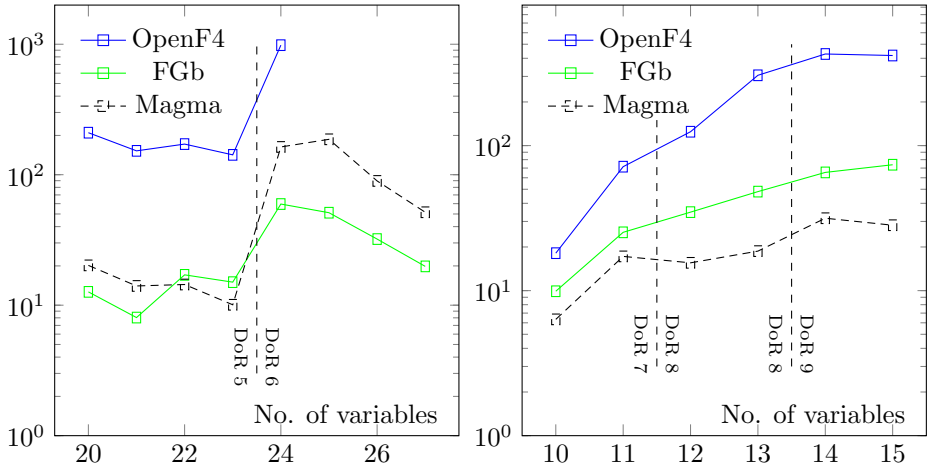


Figure 10: *The ratio of time-memory product of Magma, FGb, and OpenF4 (log-y axis) relative to the time-memory product of* M4GB *for system with* $2n$ *equations (left) and* $n+1$ *equations (right) where* n *is the number of variables. The vertical dashed line separates the parameter with different degree of regularity (DoR). The corresponding data is described in Table 7.*

		CPU time (sec)		Memory(MB)	
n	m	M4GB	Magma	M4GB	Magma
20	40	73.28	178.41	78	361.48
21	42	222.27	384.43	117	577.34
22	44	560.11	1242.39	230	853.84
23	46	1425.68	2447.40	354	1324.16
24	48	3474.01	23951.03	696	8872.94
25	50	7647.64	59692.81	1416	19718.78
26	52	21432.34	141111.18	3049	25197.97
27	54	64944.98	314647.02	6181	39844.84

Table 8: *Performance comparison of* M4GB *and Magma for systems with* n *variables and* $m = 2n$ *equations over* $\mathbb{F}_{31}$.

		CPU time (sec)		Memory(MB)	
n	m	M4GB	Magma	M4GB	Magma
10	11	1.74	2.35	23	32
11	12	4.36	7.98	24	64
12	13	15	42.20	51	114
13	14	51	204.57	104	281
14	15	326.96	2007.68	363	1104
15	16	1951.04	12332.51	1046	3320

Table 9: *Performance comparison of* M4GB *and Magma for systems with* n *variables and* $m = n + 1$ *equations over* $\mathbb{F}_{31}$.