Polynomial modular product verification and its implications

Pascal Giorgi, Bruno Grenet, Armelle Perret Du Cray

Polynomial modular product verification and its implications

Pascal Giorgi Bruno Grenet Armelle Perret du Cray
LIRMM, Univ. Montpellier, CNRS
Montpellier, France
{pascal.giorgi,bruno.grenet,armelle.perret-du-cray}@lirmm.fr

January 6, 2021

Abstract

Polynomial multiplication is known to have quasi-linear complexity in both the dense and the sparse cases. Yet no truly linear algorithm has been given in any case for the problem, and it is not clear whether it is even possible. This leaves room for a better algorithm for the simpler problem of verifying a polynomial product. While finding deterministic methods seems out of reach, there exist probabilistic algorithms for the problem that are optimal in number of algebraic operations.

We study the generalization of the problem to the verification of a polynomial product modulo a sparse divisor. We investigate its bit complexity for both dense and sparse multiplicands. In particular, we are able to show the primacy of the verification over modular multiplication when the divisor has a constant sparsity and a second highest-degree monomial that is not too large. We use these results to obtain new bounds on the bit complexity of the standard polynomial multiplication verification. In particular, we provide optimal algorithms in the bit complexity model in the dense case by improving a result of Kaminski and develop the first quasi-optimal algorithm for verifying sparse polynomial product.

1 Introduction

Polynomials are one of the most basic objects in computer algebra and the study of fast polynomial operations remains a very challenging task. Polynomials can be represented using either the dense representation, that stores all the coefficients in a vector, or the more compact sparse representation, that only stores nonzero monomials. Depending on which representation is chosen the problems might have a very different flavor leading to two very separate lines of research.

Polynomial multiplication is the most noticeable problem that attracted a lot of attention since many decades, culminating nowadays with quasi-optimal algorithms [2, 15]. Although such algorithms are really efficient in theory and in practice, there are not yet optimal and they often rely on complex approaches that can be error prone. Therefore, looking for rather simple procedure to verify the correctness of polynomial products is of great interest. From a theoretical perspective, the goal is then to provide asymptotically faster algorithms than those for multiplying polynomials, ultimately seeking for an optimal algorithm. In practice the objective is barely to find simpler and faster procedures that reveal easier to trust.

In this work, we intend to present the most recent advances in verifying polynomial products in both the dense and sparse case, to extend such results to either optimal algorithms or to more reliable solutions in practice. Finally, we extend the problem to some specific modular multiplication of polynomials which seems to not having been explored yet.

Dense polynomial multiplication We know from the early 60's that dense polynomial arithmetic is sub-quadratic, and that it can even be quasi-linear when the so-called FFT applies [4]. It has been more than two decades later that Cantor and Kaltofen [2] provide a quasi-linear algorithm without any assumption on the polynomial algebra. They show that two dense polynomials of degree less than n over an algebra $\mathcal{A}$ can be multiplied with $\mathcal{O}(n \log n \log\log n)$ operations in $\mathcal{A}$. In regards to the bit complexity model, the operations in the base ring $\mathcal{A}$ cannot count $\mathcal{O}(1)$ anymore, and the previous algorithms may not lead to the best complexity estimates for specific domains such as $\mathcal{A} = \mathbb{F}_q$ or $\mathcal{A} = \mathbb{Z}$. There, the use of Kronecker substitution together with fast integer multiplication turns out to be the best alternative [8, Section 8.4]. It has been showed by Harvey and van der Hoeven in [13] that one can reach a bit complexity of $\mathcal{O}(n \log q \log(n \log q) 4^{\log^*(n)})$ for polynomial multiplication over $\mathbb{F}_q[X]$ for any prime field $\mathbb{F}_q$. We shall mention that very recently, such complexity have been further improved to $\mathcal{O}(n \log q \log(n \log q))$ bit operations [15] under some mild hypothesis. For

polynomials with integer coefficients bounded by an integer C, the complexity falls down to multiplying two integers of bit length $\mathcal{O}(n(\log n + \log C))$ which gives $\mathcal{O}(n(\log^2 n + \log C \log n + \log C \log\log C) = \tilde{\mathcal{O}}(n \log C)$[1] when we assume that n-bits integer multiplication complexity is $\mathsf{I}(n) = \mathcal{O}(n \log n)$ [14]. For clarity in the presentation, we will often use $\mathsf{M}(n)$ as the number of operations in $\mathscr{R}$ required to multiply two dense polynomials of size n, while $\mathsf{M}_q(n)$ will denote the bit complexity for such multiplication over a prime field $\mathbb{F}_q$.

Sparse polynomial multiplication In the sparse representation, a polynomial $F = \sum_{i=0}^{n} f_i X^i \in \mathscr{R}[X]$ is expressed as a list of pairs (e_i, f_{e_i}) such that all the f_{e_i} are nonzero. We denote by $\#F$ the *sparsity* of the polynomial F which corresponds to its number of nonzero coefficients. Let F be a polynomial of degree n, and $\log C$ be a bound on bit length of its coefficients. Then, the size of the sparse representation of F is $\mathcal{O}(\#F(\log n + \log C))$ bits. Contrary to the dense case, note that *fast* algorithms for sparse polynomials must have a (poly-)logarithmic dependency on the degree, and that the size of the output does not exclusively depend on the size of the inputs. Indeed, the product of two polynomials F and G has at most $\#F\#G$ nonzero coefficients. But it may have as few as 2 nonzero coefficients, as shown by the following example.

Example 1. Let $F = X^{14} + 2X^7 + 2$, $G = 3X^{13} + 5X^8 + 3$ and $H = X^{14} - 2X^7 + 2$. Then $FG = 3X^{27} + 5X^{22} + 6X^{20} + 10X^{15} + 3X^{14} + 6X^{13} + 10X^8 + 6X^7 + 6$ has nine terms, while $FH = X^{28} + 4$ has only two.

Another difference with the dense case is that studying the complexity in a pure algebraic model remains meaningless, unless you assume a transdichotomous model on the degree, meaning that the integer computation on the exponents is always $\mathcal{O}(1)$ [3, 25]. The classical approach for computing the product of two polynomials of sparsity T is to generate all the T^2 possible monomials, and to sort them and merge those of equal degree to collect the monomials of the result. Using radix sort, this algorithm takes for instance $\mathcal{O}(T^2(\mathsf{I}(\log C) + \log n))$ bit operations over $\mathbb{Z}$ and it exhibits a T^2 factor in the space complexity, whatever the number of terms in the result. Many improvements have been proposed to reduce this space complexity, to extend the approach to multivariate polynomials, and to provide fast implementations in practice [19, 22, 23]. Yet, none of these results reduces the T^2 factor in the time complexity. In general, no complexity improvement is expected as the output polynomial may have as many as T^2 nonzero coefficients. However, this number of nonzero coefficients can be overestimated, giving the opportunity for output-sensitive algorithms. Such algorithms have first been proposed for special cases. Notably, when the output size is known to be small due to sufficiently structured inputs [26], especially in the multivariate case [17, 16], or when the support of the output is known in advance [18].

Output-sensitive multiplication algorithms try to take into account the two reasons that can decrease the sparsity of the product. The first one is exponent collisions, while the second one occurs when these collisions imply some coefficient cancellations. The exponent collision is captured by the *sumset* of the exponents of $F = \sum_{i=1}^{T} f_i X^{\alpha_i}$ and $G = \sum_{j=1}^{T} g_j X^{\beta_j}$, that is $\{\alpha_i + \beta_j : 1 \le i, j \le T\}$. Arnold and Roche call this set the *structural support* of the product FG and its size the *structural sparsity* [1]. If $H = FG$, then the structural sparsity S of the product FG satisfies $2 \le \#H \le S \le T^2$. Observe that although $\#H$ and S can be close, their difference can reach $\mathcal{O}(T^2)$ as shown by the next example.

Example 2. Let $F = \sum_{i=0}^{T-1} X^i$, $G = \sum_{i=0}^{T-1} (X^{iT+1} - X^{iT})$ and $H = FG$. We have $\#F = T$, $\#G = 2T$ and the structural sparsity of FG is $T^2 + 1$ while $H = X^{T^2} - 1$ has sparsity 2.

For polynomials with non-negative integer coefficients, no coefficient cancellation can occur and Cole and Hariharan describe a multiplication algorithm requiring $\tilde{\mathcal{O}}(S \log^2 n)$ operations in the RAM model with $\mathcal{O}(\log(Cn))$ word size [3], where $\log(C)$ bounds the bitsize of the coefficients. Arnold and Roche improve this complexity to $\tilde{\mathcal{O}}(S \log n + \#H \log C)$ bit operations for polynomials with both positive and negative integer coefficients [1]. A recent algorithm of Nakos avoids the dependency on the structural sparsity for the case of integer polynomials [25], using the same word RAM model as Cole and Hariharan. Unfortunately, the bit complexity of this algorithm, $\tilde{\mathcal{O}}((T \log + \#H \log^2 n) \log(Cn) + \log^3 n)$, is not quasi-linear. More recently, we propose in [9] the first quasi-optimal algorithm for sparse polynomial multiplication yielding a bit complexity of $\tilde{\mathcal{O}}(T'(\log n + \log C))$ where $T' = \max(T, \#H)$. More precisely, taking $k = T'(\log n + \log c)$ which is the bit length of the input and output, we are able to reach a bit complexity of $\mathcal{O}(k \log^2 k \log^2 T(\log T + \log\log k))$.

Verification of polynomial products Considering the non-optimality of polynomial multiplication in both representations, it is quite natural to ask whether it is rather a simple task or not to verify an instance of the problem. More formally, given three polynomials F, G and H, can we assert that H is equal to the product of F and G in less operations than computing the product itself? Furthermore, we want such procedure to

[1] Here, and throughout the article, $\tilde{\mathcal{O}}(f(n))$ denotes $\mathcal{O}(f(n)\log^k(f(n)))$ for some constant $k > 0$.

be as simple as possible and to not rely on polynomial multiplication if possible. Unfortunately, doing this with a deterministic procedure is not yet known, but using probabilistic algorithms lead to positive answers as shown by several papers [6, 29, 32, 10]. Here and henceforth, polynomials are assumed to have coefficients in an integral domain $\mathscr{R}$, rather than in a more general algebra.

For dense polynomials this verification amounts to choosing a random element α in a finite subset of $\mathscr{R}$ and to assert that $H(\alpha) - F(\alpha)G(\alpha)$ is zero. In that case, the complexity for the verification becomes $\mathcal{O}(n)$ operations in $\mathscr{R}$, which is optimal. Of course, the probability of error is less than one as soon as $\mathscr{R}$ has more than n elements. If this not the case, for instance when $\mathscr{R} = \mathbb{F}_2$, it is not desirable to choose α in a sufficient large extension of $\mathscr{R}$ to have $\mathcal{O}(n)$ elements. The latter would require an extension of degree $\mathcal{O}(\log n)$ and it would raise the complexity to $\mathcal{O}(n\mathsf{M}(\log n))$. This is actually larger than the complexity $\mathsf{M}(n)$ of computing the product. In [20], Gamin's solved the latter problem by replacing the evaluation, that corresponds to computing within $\mathscr{R}[X]/(X-\alpha)$, by doing a polynomial multiplication within $\mathscr{R}[X]/(X^i-1)$ for a random integer $i < n$. More precisely, by choosing $i = \mathcal{O}(n^{1-e})$ for some $0 < e < 1/2$, his verification algorithm runs in $\mathcal{O}(n)$ operations in $\mathscr{R}$, whatever its size, with a probability of error bounded away from one. While the result sounds optimal from a theoretical perspective, it might be mitigated for practical applications as it verifies polynomial multiplication of degree n by doing multiplication of polynomials of degree $\mathcal{O}(n^{1-e})$.

All these results remain valid under the bit complexity model, but the obtained complexity might not be optimal. For polynomials over $\mathbb{F}_q[X]$, both approaches using products in $\mathbb{F}_q[X]/(X-\alpha)$ or in $\mathbb{F}_q[X]/(X^i-1)$ lead to a bit complexity of $\mathcal{O}(n\mathsf{I}(\log q)) = \mathcal{O}(n\log q\log\log q)$. While being non optimal, they remain however asymptotically faster than the computation of the initial products by a factor $\log n/\log\log q$. Actually, Kaminski's approach has a better bit complexity than the standard method and can even yield a linear bit complexity in favorable cases. For polynomials over $\mathbb{Z}[X]$, the result is more surprising as it is possible to reach an optimal bit complexity of $\mathcal{O}(n\log C)$ for any input. This result should be attributed to Kaminski, as he provided in [20] all the necessary materials while not noticing the result explicitly. It seems surprising, but we haven't found any references advertising such result. Thus, we propose to provide the description of those optimal verifications of polynomial products.

For sparse polynomials, the verification of products remains less studied. It is misleading to think that using polynomial evaluation is satisfactory. Assuming that only T coefficients are nonzero, sparse polynomial evaluation is not quasi-linear in the input size $\mathcal{O}(T(\log n + \log C))$. Indeed, computing α^n requires $\mathcal{O}(\log n)$ operations in $\mathscr{R}$ which implies a complexity of $\mathcal{O}(T\log n)$ operations in $\mathscr{R}$ when applied to the T nonzero monomials. Since one needs to use a subset of $\mathscr{R}$ of size at least n to ensure a nonzero probability of success, this implies that the bit complexity is at least $\mathcal{O}(T\log^2 n)$. Using similar ideas as Kaminski's [20], we proposed recently in [9] to verify sparse polynomial identities by doing the computation in $\mathscr{R}[X]/(X^p-1)$ for a random prime p. In particular, we prove that choosing $p = \mathcal{O}(T^2\log n)$ ensures that $(H - FG) \bmod (X^p - 1) = 0$ implies that $H - FG = 0$ with good probability and that the computation can be done in quasi-linear time with $\tilde{\mathcal{O}}(T(\log n + \log C))$ bit operations.

Another important measure for randomized verification algorithms is the probability of failure. All the known verification algorithms are True-biased one-sided Monte Carlo algorithms. This means that they always return True if $H = FG$ and return False with probability at least $1-\epsilon$ otherwise. Given an algorithm with error probability at most ϵ, we can attain any smaller probability of error τ by repeating $\mathcal{O}(\frac{\log\tau}{\log\epsilon})$ rounds of the algorithm. This shows that the complexity of the algorithm is actually dependent on the target error probability. In our results, we always explicitly indicate this dependency. We can distinguish several *regimes* of values for the error probability: the constant regime $\epsilon = \mathcal{O}(1)$, the inverse polynomial regime $\epsilon = 1/n^{\mathcal{O}(1)}$ and the inverse exponential regime $\epsilon = 1/2^{\mathcal{O}(n)}$, where n is the input degree. Given an algorithm with constant error probability, one can attain any smaller constant probability using a constant number of rounds. This keeps the same asymptotic complexity. The same is true for two probabilities inside the inverse polynomial regime. To get to the inverse polynomial regime from the constant regime, the number of rounds must be $\mathcal{O}(\log n)$, slightly increasing the asymptotic complexity. The inverse exponential regime can then be attained using a polynomial number of rounds. In our context of linear and quasi-linear algorithms, the inverse exponential regime is not attainable in general. The best known verification algorithms have linear bit complexity in the inverse polynomial regime.

Contributions As an extension of our prior work [9], we propose to study more generally the verification of polynomial multiplication in $\mathscr{R}[X]/P$ where P is a monic sparse polynomial. In the dense case, this generalizes a work from one of the authors on the probabilistic verification on polynomial middle product [10]. By reusing our modular product's verification, we show that we can address the difficulty of Kaminski's approach that verifies polynomial products using products of roughly the same degree, more than $\sqrt{n}$. In particular, we

show that we can avoid the dependency on polynomial multiplication in every cases. When dealing with finite field arithmetic it is quite common to rely on irreducible polynomials that are sparse [24]. Therefore, having the possibility to verify multiplication over finite fields in less operations than computing the product seems of great interest. In particular, we show that the verification of products in $\mathbb{F}_{q^s}$ can be done in $\mathcal{O}(s\#P)$ operations in $\mathbb{F}_q$ where $P \in \mathbb{F}_q[X]$ is the monic irreducible polynomial of degree s used to define $\mathbb{F}_{q^s}$. Clearly for irreducible polynomial with constant sparsity, as it is often the case over $\mathbb{F}_2$ [11] and more generally $\mathbb{F}_q$ [24], this offers an optimal verification procedure. Finally, for sparse polynomials, this work extends our prior result for $\mathscr{R}[X]/(X^p-1)$ in [9] that was of great importance to achieve the first quasi-linear time algorithm for sparse polynomial multiplication. We hope our new insight on this problem will leverage other fast algorithms for sparse polynomial operations, especially for the division problem [27].

All our techniques and results extend to the more general problem of verifying a polynomial identity of the form $\sum_i F_i G_i \bmod P = 0$, where the sum may have an arbitrary number of terms. It would be interesting to be able to extend these results to more general polynomial identities. As a very simple example, given F_1, F_2, F_3, H and $P \in \mathscr{R}[X]$ for some integral domain $\mathscr{R}$, what is the complexity of the verification of $H = F_1F_2F_3 \bmod P$? Obviously if the inputs are dense polynomials, the computation of $F_1F_2F_3 \bmod P$ can be done in quasi-linear time. But the question is to design an algorithm than runs faster than performing the computation. In the sparse case, the computation may increase the input size quite a lot and even a quasi-linear time algorithm is lacking. More generally, the problem is to verify identities of the form $\sum_i \prod_j \sum_k \cdots \prod_\ell F_{i,j,k,\ldots,\ell} \bmod P = 0$. This problem can be phrased as a *Modular Polynomial Identity Testing* (Modular PIT) problem. The standard Polynomial Identity Testing (PIT) problem takes as input an arithmetic circuit, or equivalently a straight-line program, and consists in deciding whether the polynomial it represents is zero. In this extension, a polynomial P is also given as input and the question is whether the polynomial represented by the circuit is divisible by P. The standard PIT problem admits polynomial-time, and even quasi-linear-time, randomized algorithm. A very important open question is whether it also admits a polynomial-time *deterministic* algorithm. For the Modular PIT problem, the question is already to design efficient *randomized* algorithms. If the dense case, the challenge is to obtain faster algorithms than performing the product, ideally linear-time algorithms. In the sparse case, it is not even known how to solve the problem in randomized polynomial-time. Our results may be seen as a first step towards this goal.

Outline We start our work in Section 2 by introducing all the technical materials that serve to demonstrate our main results. Then, Section 3 is devoted to the study of the evaluation of modular multiplication. In particular, we provide algorithms and their thorough analysis for evaluating $(FG) \bmod P$ on α without computing $FG \bmod P$. The results of that section serve to derive efficient algorithms in Section 4 for the verification of modular multiplication of polynomials. Finally, we present in Section 5 the more general results on the verification of classical polynomial multiplication. In particular, we extend the work of Kaminski [20] for the dense case with a thorough analysis of its bit complexity that enables to reach optimal verification. We also give a more detailed presentation of our first quasi-optimal algorithm for the sparse case that appears in [9].

2 Preliminaries

2.1 Notations and complexities

Let $Q \in \mathscr{R}[X]$ be a degree-n polynomial. We denote its coefficient of degree i by q_i. The *sparsity* of Q is its number of nonzero monomials and is denoted by $\#Q$. The *support* of Q is the set $\mathrm{supp}(Q) = \{i : q_i \neq 0\}$. If Q is a polynomial over $\mathbb{Z}$, we denote by $\|Q\|_\infty$ its norm, defined as $\max_{0\leq i\leq n} |q_i|$. We denote by $\log(\cdot)$ the base-2 logarithm and by $\ln(\cdot)$ the natural logarithm. We also use $\log_b(\cdot)$ to denote the base-b logarithm defined by $\log_b(x) = \frac{\log x}{\log b}$.

We work in this paper with dense and sparse polynomials. A dense polynomial is represented as the vector of its coefficients, which has size $n+1$ for a degree-n polynomial. A sparse polynomial is represented by the list of its nonzero monomials. We consider that we work, for sparse polynomials, with an abstract structure of *sparse vector*. In practice, this can be implemented by several data structures, depending on the operations that need to be performed. A standard choice in sparse polynomial arithmetic is the use of heaps [19, 22, 23]. To get better complexities, we might resorting to van Emde Boas Trees [5, Chapter 20] as in Section 3.3.We also use sparse vectors in some algorithms to represent data which are not directly polynomials. The underlying data structure is the same as for sparse polynomials.

Complexity of dense polynomial multiplications We denote by $\mathsf{M}(n)$ the number of ring operations needed to compute a product of degree-n dense polynomials over an integral domain. We can take $\mathsf{M}(n) = \mathcal{O}(n \log n \log\log n)$ [2]. We denote by $\mathsf{M}_q(n)$ the bit complexity of the multiplication of two degree-n dense polynomials over a finite field $\mathbb{F}_q$. The best known bounds on $\mathsf{M}_q(n)$ are $\mathcal{O}(n \log q \log(n \log q) 4^{\log^*(n \log q)})$ [13] unconditionally and $\mathcal{O}(n \log q \log(n \log q))$ assuming the existence of some Linnik constant [15]. To simplify the notation, we assume the existence of this Linnik constant. The cost of multiplying two elements in an extension field $\mathbb{F}_{q^d}$ is the cost of degree-d polynomial multiplications, that is $\mathcal{O}(\mathsf{M}_q(d))$. Let $F, G \in \mathbb{Z}[X]$ of degree n and norm C. Their product has norm at most nC^2. To compute FG, we can evaluate both F and G on some power of 2 larger than nC^2, multiply the resulting integers (that have size $n \log(nC^2)$), and read the coefficients of FG directly on the output integer. Let $\mathsf{I}(m) = \mathcal{O}(m \log m)$ denote the bit complexity of multiplying two m-bit integers [14]. Then the bit complexity of multiplying F and G is $\mathsf{I}(n \log(nC^2)) = \mathcal{O}(n \log^2 n + n \log n \log C + n \log C \log\log C)$.

Complexity of polynomial evaluation The evaluation of a dense degree-n polynomial $F \in \mathscr{R}[X]$ on a point $\alpha \in \mathscr{R}$ requires $\mathcal{O}(n)$ operations in $\mathscr{R}$ using for instance Horner scheme. If α lies in an extension ring $\mathscr{R}_{\text{ext}}$ of $\mathscr{R}$, the evaluation requires $\mathcal{O}(n)$ operations in $\mathscr{R}_{\text{ext}}$. If F has coefficients in a finite field $\mathbb{F}_q$, this translates directly to a linear number of operations in $\mathbb{F}_q$. Now if F has coefficients in $\mathbb{Z}$, one must take into account the growth of the integers during the computation. Using a divide-and-conquer approach to use balanced integer multiplications, the cost of the evaluation is $\mathcal{O}(\mathsf{I}(n \log C) \log(n \log C))$ bit operations where $C = \max(|\alpha|, \|F\|_\infty)$. We note that this cost is quasi-linear in the worst case output size while using Horner scheme would have been quadratic.

To evaluate a sparse polynomial $F \in \mathscr{R}[X]$ on $\alpha \in \mathscr{R}$, we compute the relevant powers of α and then perform $\#F$ multiplications and additions in $\mathscr{R}$. Computing each power independently yields $\mathcal{O}(\#F \log n)$ operations in $\mathscr{R}$. Using simultaneous exponentiation [31], the cost is reduced to $\mathcal{O}(\log n + \#F \log n / \log\log n)$ operations in $\mathscr{R}$. Again, this directly translates to operations in $\mathbb{F}_q$ if $\mathscr{R} = \mathbb{F}_q$. For polynomials with integer coefficients, the growth is much more severe than in the dense case. Indeed, α^n has $\mathcal{O}(n \log |\alpha|)$ bits. This implies that the bit complexity is at least linear in n which is exponentially larger than the input size. The cost is actually not better than with dense polynomials.

2.2 Bounds on polynomial products and modular reductions

Reducing a polynomial modulo P changes its norm and sparsity. We provide bounds on these growths. They rely on the *gap* between the degree of P and its *second degree*, that is the degree of its second highest-degree monomial.

Definition 1. Let $P = X^n + \sum_{i=0}^{k} p_i X^i$ for $k < n$ and $p_k \neq 0$. The *second degree* of P is the integer k. The *gap parameter* γ of P is $\gamma = \frac{1}{n}(n-k)$.

In particular, the second degree of P is $(1-\gamma)n$. The parameter γ is between 0 and 1. If γ is close to 0, the polynomial actually has no gap, while $\gamma = 1$ corresponds to a binomial $aX^n + b$. We note that given this definition, $\frac{1}{\gamma}$ is always upper bounded by n. Polynomials with a large gap are also known as *sedimentary* polynomials [24]. A polynomial is said t-sedimentary if it is of the form $X^n + H$ where $\deg(H) = t$. A t-sedimentary polynomial is a polynomial with gap parameter $(n-t)/n$ and conversely a monic polynomial with a gap parameter γ is $(1-\gamma)n$-sedimentary.

The norm of the product of two polynomials is classically related to their norms and degrees. This can be slightly refined using the sparsities instead of the degrees.

Lemma 2.1. *Let F and G be two polynomials over $\mathbb{Z}$. Then $\|FG\|_\infty \leq \min(\#F, \#G)\|F\|_\infty \|G\|_\infty$.*

Proof. Let $H = \sum_k h_k X^k$ be the product of F and G. Then $h_k = \sum_{i+j=k} f_i g_j$. Let $T = \min(\#F, \#G)$. Then the sum to define h_k has size of most T. Since $|f_i| \leq \|F\|_\infty$ for all i and $|g_j| \leq \|G\|_\infty$ for all j by definition, $|h_k| \leq T\|F\|_\infty \|G\|_\infty$, whence the result. □

The modular reduction of polynomials has a bigger impact on the norm. It is actually related to several parameters such as the gap parameter of the divisor and the difference of the degrees. The following example shows a large increase in the norm, as well as a densification of the result.

Example 3. Let $P = X^{80} + 7X^{65} + 2X^{61} - 8X^{59} + X^{56} + 3$ and $Q = X^{131} + 4X^{120} + 8X^{118} - 3X^{108} - 3X^{80} + X^{71} + 5X^{32}$. Here the gap parameter of P is $\gamma = \frac{3}{16}$, $\#P = 6$, $\#Q = 7$ and $\|P\|_\infty = \|Q\|_\infty = 8$. The polynomial $Q \bmod P$ has degree 79, sparsity 53 and norm 11912.

The following proposition bounds the growth on the different parameters of the polynomial after a modular reduction.

Proposition 2.2. *Let Q be a sparse polynomial of degree at most $n-1+k$ and P a monic polynomial of degree n with $\#P \geq 2$. The polynomial $Q \bmod P$ has at most $\#Q(\#P-1)^{\lceil \frac{k}{\gamma n} \rceil}$ monomials. If Q and P are defined over $\mathbb{Z}$, $\|Q \bmod P\|_\infty \leq \|Q\|_\infty (\#P\|P\|_\infty)^{\lceil \frac{k}{\gamma n} \rceil}$.*

Proof. We analyse the growth of the norm and the sparsity while performing the euclidean division.

Instead of following the classical quadratic algorithm, we first reduce once all the monomials of Q with degree at least n to obtain a new dividend. We repeat this process until the dividend has degree less than n. Let us define the sequence $(Q^{[i]})_i$ by $Q^{[0]} = Q$ and $Q^{[i+1]} = (Q^{[i]} \bmod X^n) + (Q^{[i]} \text{ quo } X^n)(X^n - P)$. Then $Q^{[i]} \bmod P = Q \bmod P$ for all i. Since $\deg(Q^{[i]} \text{ quo } X^n) = \deg(Q^{[i]}) - n$ and $\deg(X^n - P) \leq (1-\gamma)n$, $\deg(Q^{[i+1]}) \leq \max(n-1, \deg(Q^{[i]}) - \gamma n)$, whence $\deg(Q^{[i]}) \leq \max(n-1, \deg(Q) - i\gamma n)$.

Also, $\#Q^{[i+1]}$ is at most $\#Q^{[i]}(\#P-1)$, thus $\#Q^{[i]} \leq \#Q(\#P-1)^i$. Finally,

$$\|Q^{[i+1]}\|_\infty \leq \|Q^{[i]}\|_\infty (1 + \min(\#Q^{[i]}, \#P-1)\|P\|_\infty).$$

Therefore, $\|Q^{[i]}\|_\infty \leq (\#P\|P\|_\infty)^i \|Q\|_\infty$.

Since $\deg(Q^{[i]}) \leq n + k - 1 - i\gamma n$, $\deg(Q^{[i]}) < n$ if $i = \lceil \frac{k}{\gamma n} \rceil$. This implies that $Q^{[i]} = Q \bmod P$. □

2.3 Random primes and random irreducible polynomials

We collect in this section some useful results to produce random prime numbers and random irreducible polynomials over finite fields.

Proposition 2.3 ([28])**.** *If $\lambda \geq 21$, there are at least $\frac{3}{5}\lambda/\ln\lambda$ prime numbers in $[\lambda, 2\lambda]$.*

Using this proposition together with Miller-Rabin probability test, we can produce integers that are prime with good probability [30].

Proposition 2.4. *There exists an algorithm* RandomPrime(λ, ϵ) *that returns an integer q in $[\lambda, 2\lambda]$, such that q is prime with probability at least $1-\epsilon$. It requires $\mathcal{O}(\log(\frac{1}{\epsilon})\log^2(\lambda)\mathsf{I}(\log\lambda)\log\log\lambda)$ bit operations.*

Proposition 2.5. *Let $H \in \mathcal{R}[X]$ be a nonzero polynomial of degree at most n and sparsity at most T, $0 < \epsilon < 1$ and $\lambda = \max(21, \frac{10}{3\epsilon} T \ln n)$. With probability at least $1-\epsilon$,* RandomPrime$(\lambda, \frac{\epsilon}{2})$ *returns a prime number p such that $H \bmod X^p - 1 \neq 0$.*

Proof. It is sufficient, for $H \bmod X^p - 1$ to be nonzero, that there exist one exponent e of H that is not congruent to any other exponents e_j modulo p. In other words, it is sufficient that p does not divide any of the $T-1$ differences $\delta_j = e_j - e$.

Noting that $\delta_j \leq n$, the number of primes in $[\lambda, 2\lambda]$ that divide at least one δ_j is at most $\frac{(T-1)\ln n}{\ln\lambda}$. Since there exists $\frac{3}{5}\lambda/\ln\lambda$ primes in this interval, the probability that a prime randomly chosen from it divides at least one δ_j is at most $\epsilon/2$. RandomPrime$(\lambda, \epsilon/2)$ returns a prime in $[\lambda, 2\lambda]$ with probability at least $1-\epsilon/2$, whence the result. □

The following two propositions will be useful to either reduce integer coefficients modulo some prime number or to construct irreducible polynomials over finite fields.

Proposition 2.6. *Let $H \in \mathbb{Z}[X]$ be a nonzero polynomial, $0 < \epsilon < 1$ and $\lambda \geq \max(21, \frac{10}{3\epsilon}\ln\|H\|_\infty)$. Then with probability at least $1-\epsilon$,* RandomPrime$(\lambda, \frac{\epsilon}{2})$ *returns a prime q such that $H \bmod q \neq 0$.*

Proof. Let h_i be a nonzero coefficient of H, a random prime from $[\lambda, 2\lambda]$ divides h_i with probability at most $\frac{5}{3}\ln\|H\|_\infty/\lambda \leq \epsilon/2$. Since RandomPrime$(\lambda, \epsilon/2)$ returns a prime in $[\lambda, 2\lambda]$ with probability at least $1-\epsilon/2$ the result follows. □

Proposition 2.7 ([30, Chapter 19])**.** *The number of irreducible monic polynomial of degree d over a field $\mathbb{F}_q$ is between $\frac{q^d}{2d}$ and $\frac{q^d}{d}$.*

Proposition 2.8 ([30, Chapter 20])**.** *There exists an algorithm that, given a finite field $\mathbb{F}_q$, an integer d and $0 < \epsilon < 1$, computes a degree-d polynomial in $\mathbb{F}_q[X]$ that is irreducible with probability at least $1-\epsilon$. It requires $\mathcal{O}(\log(\frac{1}{\epsilon})d^2\mathsf{M}(d)(\log q + \log\log d))$ operations in $\mathbb{F}_q$ or $\mathcal{O}(\log(\frac{1}{\epsilon})d^4\log q)$ operations in $\mathbb{F}_q$ if using only naive polynomial multiplications.*

Remark 2.9. *Shoup [30] presents Las Vegas algorithms for Propositions 2.4 and 2.8. We consider Monte Carlo versions of his algorithms. Also, he analyses the complexities with naive algorithms. Our complexity estimates use fast integer and polynomial arithmetic.*

3 Evaluation for polynomial multiplication in a quotient ring

As seen earlier, the verification of polynomial multiplication mainly relies on the evaluation of the polynomial identity at a random point. In this section we present algorithms to efficiently compute the evaluation of a modular product $(FG) \bmod P$ on a point α, without computing $(FG) \bmod P$. There, the modulus P is always considered as a sparse polynomial, while F and G can be either dense or sparse.

Section 3.1 describes our method in the simpler case where P is a binomial. We obtain linear-time evaluations, whether F and G are dense or sparse. Section 3.2 generalizes the method to the product of two dense polynomials *modulo* a sparse modulus, and Section 3.3 presents the case of a sparse modular product.

3.1 Evaluation of a product modulo a binomial

Let us first present our method to evaluate a modular product $FG \bmod P$ where $P = X^n - 1$. This special case illustrates our more general method. It also has its own interest since it is used as the main tool for the verification of a product of two polynomials in Section 5, either for dense or sparse representation.

We first describe the algorithm for dense polynomials F and G.

Theorem 3.1. *Let F and G be two polynomials in $\mathscr{R}[X]$ of degrees less than n and $\alpha \in \mathscr{R}$. The polynomial $(FG) \bmod X^n - 1$ can be evaluates on α using $\mathscr{O}(n)$ operations in $\mathscr{R}$.*

Proof. Let $H = FG$ and $M = H \bmod X^n - 1$. We denote by f_i (resp. g_i, h_i, m_i) the coefficient of degree i of the polynomial F (resp. G, H, M). Let also $\vec{g} = (g_0, \ldots, g_{n-1})^T$, $\vec{h} = (h_0, \ldots, h_{2n-2})^T$ and $\vec{m} = (m_0, \ldots, m_{n-1})^T$.

It is a well-known fact that considering F as fixed, the multiplication by F is a linear map described by a Toeplitz matrix. More precisely, we have $\vec{h} = T_F \vec{g}$ where

$$T_F = \begin{pmatrix} f_0 & & & \\ f_1 & f_0 & & \\ \vdots & & \ddots & \\ f_{n-1} & \cdots & \cdots & f_0 \\ & f_{n-1} & & f_1 \\ & & \ddots & \vdots \\ & & & f_{n-1} \end{pmatrix}.$$

Since $M = H \bmod X^n - 1$, $m_i = h_i + h_{i+n-1}$ for $0 \le i < n-1$ and $m_{n-1} = h_{n-1}$. Therefore, $\vec{m} = C_F \vec{g}$ where C_F is the circulant matrix

$$C_F = \begin{pmatrix} f_0 & f_{n-1} & \cdots & f_1 \\ f_1 & f_0 & \cdots & f_2 \\ \vdots & \vdots & & \vdots \\ f_{n-1} & f_{n-2} & \cdots & f_0 \end{pmatrix}.$$

On the other hand, evaluating M on α corresponds to the inner product $\vec{\alpha}_n \vec{m}$ where $\vec{\alpha}_n = (1, \alpha, \ldots, \alpha^{n-1})$. Therefore, our aim is to compute $\vec{\alpha}_n C_F \vec{g}$. The standard way to perform this evaluation corresponds to first computing $\vec{m} = C_F \vec{g}$ and then $\vec{\alpha}_n \vec{m}$. As noticed by Giorgi [10], the bracketing $(\vec{\alpha}_n C_F)\vec{g}$ yields a faster algorithm due to the structure of the matrix C_F.

Let $\vec{c} = \vec{\alpha}_n C_F$. Then $c_{j+1} = \sum_{\ell=0}^{n-1} \alpha^\ell f_{(\ell-j-1) \bmod n} = f_{n-j-1} + \alpha \sum_{\ell=0}^{n-2} \alpha^\ell f_{(\ell-j) \bmod n}$. Since for $j > 0$ we have $\sum_{\ell=0}^{n-2} \alpha^\ell f_{(\ell-j) \bmod n} = c_j - \alpha^{n-1} f_{n-j-1}$, we obtain the recurrence relation

$$\begin{cases} c_{j+1} = \alpha c_j - P(\alpha) f_{n-j-1} & \text{for } j \ge 0 \\ c_0 = F(\alpha) \end{cases} \tag{1}$$

where $P = X^n - 1$ and $c_0 = F(\alpha)$.

It is immediate that exploiting such recurrence relation for computing the evaluation of $(FG) \bmod P$ leads to a complexity of $\mathscr{O}(n)$ operations in $\mathscr{R}$. Indeed, once c_0 and $P(\alpha) = \alpha^n - 1$ are computed each other c_j can be computed sequentially at cost $\mathscr{O}(1)$. □

For completeness, we provide the full description of this method in Algorithm 3.1.

We can actually be more precise on the number of operations required by Algorithm 3.1. In particular when α does not lie into $\mathscr{R}$ but in an extension $\mathscr{R}_{\text{ext}}$ of $\mathscr{R}$, we can distinguish between operations in $\mathscr{R}$ and $\mathscr{R}_{\text{ext}}$. In the next corollary, we call *scalar multiplications* those that are multiplications of an element of $\mathscr{R}_{\text{ext}}$ by an element of $\mathscr{R}$. The following analysis minimizes the number of non-scalar multiplications.

Algorithm 1 EVALUATIONMODULOBINOMIAL

Input: $F, G \in \mathscr{R}[X]$ with $\deg(F), \deg(G) < n$, and $\alpha \in \mathscr{R}$.
Output: $(FG \bmod X^n - 1)(\alpha)$

1: $c \leftarrow F(\alpha)$
2: $P_\alpha \leftarrow \alpha^n - 1$
3: $\beta \leftarrow c g_0$
4: **for** $j = 1$ to $n-1$ **do**
5: $\quad c \leftarrow \alpha c - P_\alpha f_{n-j}$
6: $\quad \beta \leftarrow \beta + c g_j$
7: **return** β

Corollary 3.2. *Let F and G be two polynomials in $\mathscr{R}[X]$ of degree less than n and $\alpha \in \mathscr{R}_{ext}$. The polynomial $(FG) \bmod X^n - 1$ can be evaluated on α using $2n-2$ multiplications and $3n-2$ additions in $\mathscr{R}_{ext}$, and $3n-2$ scalar multiplications.*

Proof. We can first compute $\alpha^2, \alpha^3, \ldots, \alpha^n$ using $(n-1)$ multiplications. Then, $F(\alpha)$ can be computed using $(n-1)$ scalar multiplications and additions, and $P(\alpha) = \alpha^n - 1$ require one more addition. The initial value $c g_0$ of β requires one scalar multiplication. Then each iteration of the loop require one multiplication, two scalar multiplications and two additions. Therefore, the complete evaluation require $3n-2$ additions, $2n-2$ multiplications and $3n-2$ scalar multiplications. □

To minimize the total number of multiplications instead, we remark that one can also evaluate F using Horner's scheme with $(n-2)$ multiplications, one scalar multiplication and $(n-1)$ additions. Then α^n has to be computed using at most $2\log n$ multiplications. This results in $3n-2$ additions, $2n-3+2\log n$ multiplications and $2n$ scalar multiplications. The total number of multiplications (scalar or not) is a bit less.

We now turn to the analysis of the algorithm for F and G given in sparse representation.

Theorem 3.3. *Let F and G be two sparsely represented polynomials in $\mathscr{R}[X]$ of degrees less than n and $\alpha \in \mathscr{R}$. The polynomial $(FG) \bmod X^n - 1$ can be evaluated on α using $\mathcal{O}((\#F + \#G)\log n)$ operations in $\mathscr{R}$.*

Proof. We use the same notations as in the previous proof. If the support of G is $\mathrm{supp}(G) = \{j_0, \ldots, j_{\#G-1}\}$ with $j_0 < \cdots < j_{\#G-1}$, the inner product $\vec{c}\vec{g}$ is equal to $\sum_{k=0}^{\#G-1} c_{j_k} g_{j_k}$. This means that only the $\#G$ entries $c_{j_0}, \ldots, c_{j_{\#G-1}}$ of $\vec{c}$ need to be computed. Applying the recurrence relation (1) as many times as necessary, we obtain the new recurrence relation

$$\begin{cases} c_{j_{k+1}} = \alpha^{j_{k+1}-j_k} c_{j_k} - P(\alpha) \sum_{\ell=j_k+1}^{j_{k+1}} \alpha^\ell f_{n-\ell} & \text{for } k \geq 0 \\ c_{j_0} = ((X^{j_0} F) \bmod X^n - 1)(\alpha). \end{cases} \tag{2}$$

The initial value c_{j_0} can be computed using $\mathcal{O}(\#F \log n)$ operations in $\mathscr{R}$ since it needs $\#F$ exponentiations of α with exponent bounded by n. Most values of $f_{n-\ell}$ are actually equal to zero since F is sparse.

A nonzero coefficient f_t of F appears in the definition of $c_{j_{k+1}}$ if and only if $n - j_{k+1} \leq t < n - j_k$. Thus, each f_t is used exactly once to compute all the c_{j_k}'s. Since for each summand, one needs to compute α^ℓ for some $\ell < n$, the total cost for computing all the sums is $\mathcal{O}(\#F \log n)$ operations in $\mathscr{R}$. Similarly, the computation of $\alpha^{j_{k+1}-j_k} c_{j_k}$ for all $k \in [0, \#G-2]$ costs $\mathcal{O}(\#G \log n)$ operations in $\mathscr{R}$ plus $\#G-1$ additions of $\mathcal{O}(\log n)$-bit integers to get the exponents. As one operation in $\mathscr{R}$ requires at least one bit operation, the integer additions that costs $\mathcal{O}(\#G \log n)$ bit operations are negligible. The last remaining step is the final inner product which costs $\mathcal{O}(\#G)$ operations in $\mathscr{R}$, whence the result. □

As in the dense case, one can be more precise on the complexity if α liesin an extension $\mathscr{R}_{\mathrm{ext}}$. In contrary to the dense case where there is more operations in $\mathscr{R}$ than in $\mathscr{R}_{\mathrm{ext}}$, one can note that the number of operations in $\mathscr{R}$ is negligible in the sparse case.

Corollary 3.4. *Let F and G be two sparsely represented polynomials in $\mathscr{R}[X]$ of degrees less than n and $\alpha \in \mathscr{R}_{ext}$. The polynomial $(FG) \bmod X^n - 1$ can be evaluated on α using $\log n + \mathcal{O}((\#F + \#G)\log n / \log\log n)$ operations in $\mathscr{R}_{ext}$ plus $\#G - 1$ additions of $\mathcal{O}(\log n)$-bit integers.*

Proof. We first notice that in the sparse case the operations on α dominate the complexity. These operations are operations in $\mathscr{R}_{\mathrm{ext}}$. To improve the complexity estimates, we remark that in the sparse settings we need to compute α^t for several values of t. The computation of c_{j_0} requires to know $\#F$ values of α^t, more precisely

those with $t = \ell - j_0 \bmod n$ for each nonzero coefficient f_ℓ of F. To apply Equation (2), one needs to compute α^t for $t = j_{k+1} - j_k$, $1 \le k < \#G$, and for $t = \ell$ where $f_{n-\ell} \neq 0$. The value α^n is also needed to compute $P(\alpha)$. Finally, the inner product requires to compute α^t for each nonzero g_t. Altogether, one needs α^t for at most $2(\#F+\#G)$ values of t, each at most n. They can be computed independently using fast exponentiation, using at most $\mathcal{O}((\#F+\#G)\log n)$ multiplications, as it is done in Theorem 3.3. Actually, Yao [31] shows that these values of α^t can be computed simultaneously using only $\log n + \mathcal{O}((\#F+\#G)\log n/\log\log n)$ multiplications. Once these α^t have been computed, computing c_{j_0} and the c_{j_k}'s by means of Equation (2), as well as the inner product $\vec{c}\vec{g}$, only require $\mathcal{O}(\#F+\#G)$ operations.

□

3.2 Evaluation of a dense modular product

In this section, we extend the previous algorithm to the evaluation of a polynomial $FG \bmod P$ where P is any monic sparse polynomial. We first consider the case where F and G are given in dense representation. The case where they are given is sparse representation is postponed to the next section.

The algorithm goes along the same lines as the evaluation modulo X^n-1. Let $F^{[i]} = (X^i F) \bmod P$. We can rewrite $FG \bmod P = \sum_{i=0}^{n-1} g_i F^{[i]}$ where g_i is the coefficient of degree i in G. The evaluation of this equality on a point α yields the formula

$$(FG \bmod P)(\alpha) = \sum_{i=0}^{n-1} g_i F^{[i]}(\alpha). \tag{3}$$

To make use of this formula, we need to be able to efficiently evaluate each $F^{[i]}$ on α. Note that consecutive $F^{[i]}$'s are bound by the recurrence relation $F^{[i+1]} = (XF^{[i]}) \bmod P$. Since $\deg(F^{[i]}) = n-1$, $(XF^{[i]}) \bmod P = XF^{[i]} - f^{[i]}_{n-1}P$ where $f^{[i]}_{n-1}$ is the coefficient of degree $n-1$ of $F^{[i]}$. Consequently we have the following recurrence relation

$$\begin{cases} F^{[i+1]}(\alpha) = \alpha F^{[i]}(\alpha) - f^{[i]}_{n-1}P(\alpha) & \text{for } i \ge 0 \\ F^{[0]}(\alpha) = F(\alpha) \end{cases} \tag{4}$$

The evaluations of each $F^{[i]}$ on α can thus be computed iteratively from $F(\alpha)$, only knowing the coefficient $f^{[i]}_{n-1}$ of $F^{[i]}$ for $0 < i < n-1$.

We first present an algorithm to compute these coefficients. Note that we did not need such an algorithm when $P = X^n - 1$ since these coefficients were given for free as we had $f^{[i]}_{n-1} = f_{n-1-i}$. In the general case, the computation is based on the recurrence relation $F^{[i+1]} = XF^{[i]} - f^{[i]}_{n-1}P$, which implies

$$f^{[i+1]}_k = f^{[i]}_{k-1} - f^{[i]}_{n-1}p_k \tag{5}$$

for $0 < k \le n-1$. This allows to compute each $f^{[i]}_{n-1}$, starting from the values of $f^{[0]}_k$ for all k. These values are given as input since $F^{[0]} = F$ by definition. Note that since P is a sparse polynomial, Equation (5) actually reduces to an equality $f^{[i+1]}_k = f^{[i]}_{k-1}$ in many cases. Algorithm 2 takes this into account and only performs the required updates.

Algorithm 2 LEADINGCOEFFICIENTS

Input: Two polynomials P and F in $\mathcal{R}[X]$, with $\deg(F) < \deg(P) = n$ and P monic.
Output: The vector $[f^{[0]}_{n-1}, f^{[1]}_{n-1}, \ldots, f^{[n-2]}_{n-1}]$, where $f^{[i]}_{n-1}$ is the coefficient of degree $n-1$ of $F^{[i]} = (X^i F) \bmod P$.

1: $V \leftarrow [f_{n-1}, f_{n-2}, \ldots, f_1]$
2: **for** $i = 0$ to $n-2$ **do**
3: **for** $k \in \mathrm{supp}(P)$ such that $i < k < n$ **do**
4: $V[i+n-k] \leftarrow V[i+n-k] - p_k V[i]$
5: **return** V

Lemma 3.5. *Algorithm 2 is correct. It uses $\mathcal{O}(n\#P)$ operations in $\mathcal{R}$.*

Proof. The number of operations is clear: all operations are performed at Step 4 and it is called $\mathcal{O}(n\#P)$ times. Note that the external for loop can be stopped as soon as there exists no $k \in \mathrm{supp}(P)$ such that $i < k < n$. In other words, i never goes beyond $\deg(X^n - P) - 1$.

To show the correctness of Algorithm 2, we prove by induction that after iteration i of the external loop, the following property $\mathcal{P}(i)$ holds:

$$V[j] = f^{[j]}_{n-1} \text{ for any } j \le i+1 \text{ and } V[j] = f^{[i+1]}_{n-(j-i)} \text{ for } j > i+1.$$

Before the first iteration, $\mathscr{P}(-1)$ holds since it reads $V[j] = f_{n-j-1}^{[0]}$ for all j, and $F = F^{[0]}$ by definition.

Suppose that $\mathscr{P}(i-1)$ holds. In particular, $V[j] = f_{n-1}^{[j]}$ for $j \leq i$. During iteration i, only $V[i+1]$ to $V[n-2]$ can be modified so these equalities remain after that iteration. For $j > i$, $V[j] = f_{n-(j-i+1)}^{[i]}$ before the iteration by hypothesis. After the iteration, it becomes

$$V[j] = f_{n-(j-i+1)}^{[i]} - p_{n-j+i}V[i] = f_{n-j+i-1}^{[i]} - p_{n-j+i}f_{n-1}^{[i]}.$$

Equation (5) shows that $V[j] = f_{n-j+i}^{[i+1]}$ after Step 4, and $\mathscr{P}(i)$ holds.

To conclude, after the last iteration, $V[j] = f_{n-1}^{[j]}$ for all $j \leq n-2$ and the algorithm is correct. □

We can now make use of Algorithm 2 to evaluate $FG \bmod P$ on a point α. In the following algorithm, we assume that α belongs to some extension ring $\mathscr{R}_{\text{ext}}$ of $\mathscr{R}$. Our analysis distinguishes between operations in $\mathscr{R}$ and in $\mathscr{R}_{\text{ext}}$.

Algorithm 3 MODULAREVALUATION

Input: $P, F, G \in \mathscr{R}[X]$ with $\deg(F), \deg(G) < \deg(P) = n$, P monic, and $\alpha \in \mathscr{R}_{\text{ext}}$.
Output: $(FG \bmod P)(\alpha)$

1: $V \leftarrow [f_{n-1}^{[0]}, \ldots, f_{n-1}^{[n-2]}]$ using a call to LEADINGCOEFFICIENTS(P, F)
2: $P_\alpha \leftarrow P(\alpha)$
3: $F_\alpha \leftarrow F(\alpha)$
4: $\beta \leftarrow F_\alpha g_0$
5: **for** $i = 1$ to $n-1$ **do**
6: $\quad F_\alpha \leftarrow \alpha F_\alpha - V[i-1]P_\alpha$
7: $\quad \beta \leftarrow \beta + F_\alpha g_i$
8: **return** β

Theorem 3.6. *Algorithm 3 is correct. It uses $\mathscr{O}(n\#P)$ operations in $\mathscr{R}$ and $\mathscr{O}(n)$ operations in $\mathscr{R}_{ext}$.*

Proof. Step 6 relies on Equation (4) to compute $F_\alpha = F^{[i]}(\alpha)$. Step 7 uses this evaluation together with Equation (3) to correctly compute $(FG \bmod P)(\alpha)$. The first step requires $\mathscr{O}(n\#P)$ operations in $\mathscr{R}$ by Lemma 3.5. (It does not depend on α.) The other steps require $\mathscr{O}(n)$ operations in $\mathscr{R}_{\text{ext}}$. □

As before, we notice that the operations in $\mathscr{R}_{\text{ext}}$ are sometimes *scalar* multiplications, that is multiplications of an element of $\mathscr{R}_{\text{ext}}$ by an element of $\mathscr{R}$. We provide an analysis that minimizes the number of non-scalar multiplications.

Corollary 3.7. *Let P, F, G and α as in Algorithm 3. Then $(FG) \bmod P$ can be evaluated on α using $2n-2$ multiplications and $(3n-5+\#P)$ additions in $\mathscr{R}_{ext}$, $(3n-3+\#P)$ scalar multiplications in $\mathscr{R}_{ext}$, and $(n-1)(\#P-1)$ multiplications and additions in $\mathscr{R}$.*

Proof. We first note that the number of operations performed by Algorithm 2 is at most $(n-1)(\#P-1)$ multiplications and additions in $\mathscr{R}$. In Algorithm 3, we need to evaluate both P and F on α. To minimize the number of non-scalar multiplications, we first compute $\alpha^2, \ldots, \alpha^n$ using $n-1$ multiplications in $\mathscr{R}_{\text{ext}}$. We can then compute P_α using $\#P-1$ scalar multiplications and $\#P-1$ additions, and F_α using $n-1$ scalar multiplications and $n-2$ additions. Then, the initialization of β and the for loop require $n-1$ multiplications, $2n-1$ scalar multiplications and $2n-2$ additions. This results in $2n-2$ multiplications in $\mathscr{R}_{\text{ext}}$, $(3n-3+\#P)$ scalar multiplications in $\mathscr{R}_{\text{ext}}$, and $(3n-5+\#P)$ additions in $\mathscr{R}_{\text{ext}}$. □

In a different context where the aim is specifically to compute the evaluation with no restriction to the use of polynomial arithmetic one can first compute the polynomial $FG \bmod P$ and then evaluate it on $\alpha \in \mathscr{R}_{\text{ext}}$. Such method requires $\mathscr{O}(n \log n \log\log n)$ operations in $\mathscr{R}$ for the polynomial multiplication $F \times G$ and division by P and $\mathscr{O}(n)$ operations in $\mathscr{R}_{\text{ext}}$ for the evaluation. Thus we see that if P verifies $\#P < \log n \log\log n$ our technique is more efficient.

3.3 Evaluation of a sparse modular product

In this section, we adapt and analyse the previous algorithms for polynomials F and G given in sparse representation. Our results depend on the difference between the highest and the second highest exponents in P. Recall that the *gap parameter* γ is a measure of this difference, defined by $1-\gamma = \frac{1}{n}\max\{k < n : p_k \neq 0\}$.

In particular, the second highest exponent with nonzero coefficient in P is $(1-\gamma)n$. Proposition 2.2 gives a relation between the gap parameter and the sparsity of $(FG) \bmod P$. The potential growth of the sparsity induced by the reduction modulo P explains the dependency of our results on the gap parameter.

As G is sparse, Equation (3) becomes

$$(FG \bmod P)(\alpha) = \sum_{i \in \mathrm{supp}(G)} g_i F^{[i]}(\alpha), \tag{6}$$

with the same notations as in the previous section. The recurrence relation $F^{[i+1]} = XF^{[i]} - f_{n-1}^{[i]} P$ still holds, hence $F^{[i+1]}(\alpha) = \alpha F^{[i]}(\alpha) - f_{n-1}^{[i]} P(\alpha)$ too. The goal now is to efficiently compute $F^{[i]}(\alpha)$ for all $i \in \mathrm{supp}(G)$ only, not for all indices i. When γ is not close to zero, there are actually few indices i such that $f_{n-1}^{[i]} \neq 0$. In fact, the number of such indices depends on $\#F, \#P$ and γ. Let $I = \{i_1, \ldots, i_t\}$ denote this set of indices. We will prove in Lemma 3.8 that this set is of size $\mathcal{O}(\#F \#P^{\lceil 1/\gamma - 1 \rceil})$. We decide to first provide some arguments and an explicit algorithm to prove this claim.

An important remark is that for any $0 \leq j < n-1$, in particular those verifying $j \in \mathrm{supp}(G)$, if we assume i to be the largest index in I not larger than j, then Equation (5) implies

$$F^{[j]}(\alpha) = \alpha^{j-i} F^{[i]}(\alpha). \tag{7}$$

Therefore, the recurrence relation given in (4) becomes

$$\begin{cases} F^{[i_{k+1}]}(\alpha) = \alpha^{i_{k+1}-i_k-1}(\alpha F^{[i_k]}(\alpha) - f_{n-1}^{[i_k]} P(\alpha)) & \text{for } k \geq 0 \\ F^{[i_1]}(\alpha) = \alpha^{i_1} F^{[0]}(\alpha) \end{cases} \tag{8}$$

To efficiently use Equations (7) and (8) to perform the evaluation, we need to provide a sparse variant of Algorithm 2. It computes a sparse representation of the vector $V = [f_{n-1}^{[0]}, \ldots, f_{n-1}^{[n-2]}]$, that is the sparse vector $\{(i, f_{n-1}^{[i]}) : f_{n-1}^{[i]} \neq 0\}$.

The idea of Algorithm 4 is to mimic Algorithm 2 in the sparse settings. For simplicity of the presentation, we first consider to store V as a sparse vector as it is sufficient to our needs for proving our claims on the size of the set I. We will show in Corollary 3.9 that we must require another structure to minimize the complexity attached to data management.

The initial nonzero values in V are the nonzero coefficients of $F = F^{[0]}$, with $V[i] = f_{n-1-i}$ if $n-i-1 \in \mathrm{supp}(F)$. Let now consider the external loop in Algorithm 2. Iteration i does not require any operation if $f_{n-1}^{[i]} = 0$ since Equation (5) reduces to $f_k^{[i+1]} = f_{k-1}^{[i]}$ in that case. Therefore, we must loop over indices i such that $f_{n-1}^{[i]}$ is nonzero. For such an index i, the same updates as in Step 4 of Algorithm 2 are required. For $k \in \mathrm{supp}(P)$, $i < k < n$, we must perform the update $V[i+n-k] \leftarrow V[i+n-k] - p_k f_{n-1}^{[i]}$. If $V[i+n-k]$ is already nonzero, its value is already stored in V and must be updated. Otherwise, the new value $-p_k f_{n-1}^{[i]}$ must be inserted in V with index $i+n-k$.

It remains to be able to only loop over the indices i such that $f_{n-1}^{[i]} \neq 0$. Let us assume that iteration i has been performed since $f_{n-1}^{[i]} \neq 0$. The proof of Lemma 3.5 shows that $V[i+1]$ then contains $f_{n-1}^{[i+1]}$. Therefore in the sparse setting, we know that iteration $i+1$ has to be performed if, and only if, $V[i+1] \neq 0$. More generally, the next index to be considered is the index of the next nonzero entry of V after $V[i]$. Algorithm 4 below uses such method for computing all the indices i such that $f_{n-1}^{[i]}$ is nonzero.

Algorithm 4 SPARSELEADINGCOEFFICIENTS

Input: P, $F \in \mathscr{R}[X]$ with $\deg(F) < \deg(P) = n$, and P monic
Output: The list $\{(i, f_{n-1}^{[i]}) : 0 \leq i < n-1, f_{n-1}^{[i]} \neq 0\}$, sorted by increasing values of i.

```
1: L ← empty list
2: V ← {(i, f_{n-1-i}) : n-1-i ∈ supp(F)} (sparse vector)
3: while V is not empty do
4:     (i, v) ← extract the element of minimal index from V
5:     if v ≠ 0 then
6:         Add (i, v) to the list L
7:         for k ∈ supp(P) such that i < k < n do
8:             V[i+n-k] ← V[i+n-k] - p_k v
9: return L
```

Lemma 3.8. *Algorithm 4 is correct. If the polynomial P has a gap parameter γ, the algorithm uses $\mathcal{O}(\#F \#P^{\lceil 1/\gamma - 1 \rceil})$ operations in $\mathscr{R}$ and additions of $\mathcal{O}(\log n)$-bits integers. In particular, there are at most $\#F \#P^{\lceil 1/\gamma - 1 \rceil}$ indices i such that $f_{n-1}^{[i]} \neq 0$.*

Proof. As explained above, Algorithm 4 is an adaptation of Algorithm 2 to the sparse settings that only computes those $f_{n-1}^{[i]}$ that are nonzero, using Equations (7) and (8) in place of Equation (4). Instead of considering all the $F_i[n-1]$ one after the other it only considers those which are not zero. Let us call "iteration i" the iteration in the while loop that extract a pair (i,v) from V. To prove the correctness, we prove by induction that at the end of iteration i,

$$V = \{(j, f_{n-j+i}^{[i+1]}) : j > i, f_{n-j+i}^{[i+1]} \neq 0\} \text{ and } L = \{(j, f_{n-1}^{[j]}) : j \leq i, f_{n-1}^{[j]} \neq 0\}.$$

Before the loop ("iteration -1") the property holds, that is L is empty and V contains exactly the pairs $(j, f_{n-j-1}^{[0]})$ such that $f_{n-j-1}^{[0]} \neq 0$.

Let us assume that the property holds at then end of iteration ℓ, and let (i,v) be the pair extracted at the next iteration. We first prove that $f_{n-1}^{[i]} = v$ and $f_{n-1}^{[j]} = 0$ for $\ell < j < i$. By minimality of i and induction hypothesis, $f_{n-j+\ell}^{[\ell+1]} = 0$ for $\ell < j < i$ at the end of iteration ℓ. In particular, $f_{n-1}^{[\ell+1]} = 0$. By Equation (5), $f_{n-j+\ell+1}^{[\ell+2]} = f_{n-j+\ell}^{[\ell+1]} - f_{n-1}^{[\ell+1]} p_{n-j+\ell+1} = 0$. And an easy recurrence shows that $f_{n-1}^{[j]} = 0$ for $\ell < j < i$. Now this implies that $f_{n-1}^{[i]} = f_{n-2}^{[i-1]} = \cdots = f_{n-i+\ell}^{[\ell+1]}$. Yet by induction hypothesis, at the end of iteration ℓ, V contains $(j, f_{n-j+i}^{[\ell+1]})$ if $f_{n-j+i}^{[\ell+1]} \neq 0$. Therefore, if $f_{n-1}^{[i]}$ is nonzero, $f_{n-i+\ell}^{[\ell+1]}$ is nonzero too and V contains the pair $(i, f_{n-i+\ell}^{[\ell+1]}) = (i, f_{n-1}^{[i]})$. In other words, the value v extracted from V is indeed equal to $f_{n-1}^{[i]}$ and the property holds for L after iteration i.

Now with the same argument, $f_{n-1-j+i}^{[i]} = f_{n-j+\ell}^{[\ell+1]}$ for $\ell < j < i$. Right before iteration i, V contains then the pairs $(j, f_{n-1-j+i}^{[i]})$ for $f_{n-1-j+i}^{[i]} \neq 0$. After iteration i, such pairs are replaced by $(j, f_{n-1-j+i}^{[i]} - p_{n-j+i} f_{n-1}^{[i]})$, that is $(j, f_{n-j+i}^{[i+1]})$ by Equation (5). And if $f_{n-1-j+i}^{[i]} = 0$ but $p_{n-j+i} \neq 0$, a new pair $(j, -p_{n-j+i} f_{n-1}^{[i]}) = (j, f_{n-j+i}^{[i]})$ is inserted into V. Therefore, the property holds for V too after iteration i.

The second point is to count the number of operations. Since the while loop stops when V is empty, the number of operations in $\mathscr{R}$ is at most twice the number of pairs that are inserted into V during the algorithm and the same number of additions in $\mathbb{Z}$ are performed as the index of each pair is computed by two additions of number at most n. We will classify the pairs by *generations*. Initially, V contains $\#F$ pairs which form generation 0. New pairs can be inserted into V when a pair (i,v) is extracted. If (i,v) is a pair of generation t, the new pairs inserted at iteration i belong to generation $t+1$. At any iteration, at most $\#P-1$ pairs are inserted into V. Therefore, there are at most $\#F(\#P-1)$ pairs of generation 1, $\#F(\#P-1)^2$ pairs of generation 2, and in general $\#F(\#P-1)^t$ pairs of generation t. Now we need to bound the number of generations. Note the pairs of generation 0 have an index i between 0 and $n-1$. But at generation 1, the new pairs have index $(i+n-k)$ for some $k \in \mathrm{supp}(P)$, $k < n$. There comes the gap into account: If P has gap parameter γ, the largest exponent less than n in $\mathrm{supp}(P)$ is $(1-\gamma)n$ by definition. Therefore, at generation 1, all pairs have an index at least $i+n-(1-\gamma)n \geq \gamma n$. At generation 2, all pairs have then an index at least $2\gamma n$. At generation t, all pairs have an index at least $t\gamma n$. Since indices are bounded by $n-1$, there cannot be any pair of generation t if $t\gamma n \geq n$. In other words, the largest possible generation is $t = \lceil 1/\gamma - 1 \rceil$. Altogether, the total number of pairs inserted into V is at most

$$\sum_{t=0}^{\lceil 1/\gamma - 1 \rceil} \#F(\#P-1)^t = \#F \, \frac{(\#P-1)^{\lceil 1/\gamma \rceil} - 1}{\#P-2}$$

if $\#P > 2$, and is at most $\lceil 1/\gamma \rceil \#F$ if $\#P = 2$. To simplify the exposition, we bound both of them by $\#F\#P^{\lceil 1/\gamma - 1 \rceil}$ in the following. Note that of course, this number is also a bound on the number of pairs that are extracted from V during the algorithm.

This has two consequences. First, the number of extracted pairs is a bound on the size of the list at then end of the algorithm. Therefore there are at most $\mathscr{O}(\#F\#P^{\lceil 1/\gamma - 1 \rceil})$ nonzero values $f_{n-1}^{[i]}$. Second, this number also bounds the total number of executions of Step 8, that is the total number of operations. □

Corollary 3.9. *All operations of* INSERTION, REMOVAL, MINIMUM *and* SEARCH *of pairs (i,v) in the data structure V within Algorithm 4 can be done with $\mathscr{O}(\#F\#P^{\lceil 1/\gamma - 1 \rceil} \log\log n)$ bit operations.*

Proof. By definition the size of the sparse vector V is at most n. Therefore, using a data structure for V of type *van Emde Boas* tree with a universe of size n, ensures that any requested operations can be done with $\mathscr{O}(\log\log n)$ bit operations, see [5, Chapter 20]. □

Remark 3.10. *As the bit-complexity of all the operations in $\mathbb{Z}$ of Algorithm 4 is $\mathscr{O}(\#F\#P^{\lceil 1/\gamma - 1 \rceil} \log n)$ the cost driven by the data structure of V is negligible.*

Our algorithm to compute the evaluation of the polynomial $FG \bmod P$ on some point α, when F and G are given in sparse representation, relies on Equations (6), (8) and (7). More precisely, we first compute each $F^{[i]}(\alpha)$ for indices i such that $f_{n-1}^{[i]} \neq 0$ by means of Equation (8). From these values, we get each $F^{[j]}(\alpha)$ for $j \in \mathrm{supp}(G)$ by means of Equation (7). Finally, we deduce $(FG \bmod P)(\alpha)$ using Equation (6).

In Algorithm 5, all these computations are intertwined. The idea is to loop over all indices j such that either $f_{n-1}^{[j]} \neq 0$ or $j \in \mathrm{supp}(G)$. If $f_{n-1}^{[j]} \neq 0$, we update the value $F^{[j]}(\alpha)$ using Equation (8). If $j \in \mathrm{supp}(G)$, we accumulate partial evaluations of $(FG \bmod P)(\alpha)$ using Equations (6) and (7).

Algorithm 5 SPARSEMODULAREVALUATION

Input: P, F and $G \in \mathscr{R}[X]$, with $\deg(F), \deg(G) < \deg(P) = n$, P monic and $\alpha \in \mathscr{R}_{\mathrm{ext}}$.
Output: $(FG \bmod P)(\alpha)$

1: $V \leftarrow \{(i, f_{n-1}^{[i]}) : 1 \leq i < n-1, f_{n-1}^{[i]} \neq 0\}$, using a call to SPARSELEADINGCOEFFICIENTS(P, F)
2: **if** $f_{n-1}^{[0]} = 0$ **then** insert $(0,0)$ in V
3: $P_\alpha \leftarrow P(\alpha)$
4: $F_\alpha \leftarrow F(\alpha)$
5: $\beta \leftarrow F_\alpha g_0$ ▷ $\beta \leftarrow 0$ if $0 \notin \mathrm{supp}(G)$
6: $i \leftarrow 0$
7: **for** $j \in \mathrm{supp}(V) \cup \mathrm{supp}(G) \setminus \{0\}$, by increasing order **do**
8: **if** $j \in \mathrm{supp}(V)$ **then**
9: $F_\alpha \leftarrow \alpha^{j-i-1}(\alpha F_\alpha - V[i]P_\alpha)$ ▷ Equation (8)
10: $i \leftarrow j$
11: **if** $j \in \mathrm{supp}(G)$ **then**
12: $\beta \leftarrow \beta + \alpha^{j-i} F_\alpha g_j$ ▷ Equations (6) and (7)
13: **return** β

Theorem 3.11. *Algorithm 5 is correct. It uses $\mathscr{O}(\#F\#P^{\lceil \frac{1}{\gamma}-1 \rceil})$ operations in $\mathscr{R}$, $\mathscr{O}((\#F\#P^{\lceil \frac{1}{\gamma}-1 \rceil} + \#G)\log n)$ operations in $\mathscr{R}_{ext}$.*

Proof. We prove that at the end of iteration j, $F_\alpha = F^{[j]}(\alpha)$ if $j \in \mathrm{supp}(F)$ and $\beta = \sum_i g_i F^{[i]}(\alpha)$ where the sum ranges over indices $i \in \mathrm{supp}(G) \cap \{0, \ldots, j\}$. The property is satisfied after iteration 0 (before entering the loop) since $F_\alpha = F^{[0]}(\alpha)$ and $\beta = F_\alpha g_0 = g_0 F^{[0]}(\alpha)$. Let us assume that the property holds before entering iteration j. Index i denotes the previous index that belongs to $\mathrm{supp}(F)$. Therefore, if $j \in \mathrm{supp}(F)$, Equation (8) ensures that F_α has the right value after iteration j since $V[i] = f_{n-1}^{[i]}$. And Equations (6) and (7) justify that β also has the right value if $j \in \mathrm{supp}(G)$.

The evaluations $P(\alpha)$ and $F(\alpha)$ require $\mathscr{O}(\#P \log n)$ and $\mathscr{O}(\#F \log n)$ operations in $\mathscr{R}_{\mathrm{ext}}$ respectively. Steps 9 and 12 each require $\mathscr{O}(\log n)$ operations in $\mathscr{R}_{\mathrm{ext}}$ to compute powers of α and $\mathscr{O}(1)$ additions with integers of size $\mathscr{O}(\log n)$ to compute the appropriate exponent. These steps are executed $\mathscr{O}(\#V + \#G)$ times. Since $\#V = \mathscr{O}(\#F\#P^{\lceil 1/\gamma - 1 \rceil})$ this gives a total of $\mathscr{O}((\#F\#P^{\lceil \frac{1}{\gamma}-1 \rceil} + \#G)\log n)$ operations in $\mathscr{R}_{\mathrm{ext}}$ plus $\mathscr{O}((\#F\#P^{\lceil \frac{1}{\gamma}-1 \rceil} + \#G)\log n)$ bit operations for the integer additions. Since we can easily assume that one operation in $\mathscr{R}_{\mathrm{ext}}$ will cost more than one bit operation, the latter complexity is dominated by the computation part in $\mathscr{R}_{\mathrm{ext}}$. The cost of Step 1 is given by Lemma 3.8

. We can still use *van Edme Boas* tree to iterate over the union of the supports of V and G at Step 7 with a total of $\mathscr{O}((\#F\#P^{\lceil \frac{1}{\gamma}-1 \rceil} + \#G)\log\log n)$ bit operations which is less than the number bit operations required by the additions in $\mathbb{Z}$ and thus negligible. □

Obviously, since polynomial multiplication over integral domains is commutative, the roles of F and G can be exchanged in Algorithm 5. In particular if $\#G < \#F$, this exchange decreases the complexity in Theorem 3.11. In other words, the statement remains valid if $\#F$ is replaced by $\min(\#F, \#G)$ and $\#G$ by $\max(\#F, \#G)$. The same remark applies to subsequent results.

Remark 3.12. *As in Corollary 3.4, we can decrease the number of operations over $\mathscr{R}_{ext}$ by using simultaneous exponentiation on α. This results in $\log n + \mathscr{O}((\#F\#P^{\lceil 1/\gamma - 1 \rceil} + \#G)\log n / \log\log n)$ operations in $\mathscr{R}_{ext}$.*

If the gap parameter γ is close to $\frac{1}{n}$, the polynomial $FG \bmod P$ is in general a dense polynomial even if FG is sparse and the dense modular evaluation will be more appropriate. On the contrary, $FG \bmod P$ remains sparse if γ is close to 1, in particular if $\gamma \geq \frac{1}{2}$.

Remark 3.13. *If $\gamma \geq \frac{1}{2}$, the evaluation requires $\mathscr{O}((\#F\#P + \#G)\log n)$ operations in $\mathscr{R}_{ext}$.*

Remark 3.14. *The factor $\#F\#P^{\lceil \frac{1}{\gamma}-1\rceil}+\#G$ in the complexity may be larger than the actual sparsity of $FG \bmod P$. For instance, the sparsity can be 0 if P divides FG. Yet, it is smaller than the general bound $\#F\#G(\#P-1)^{\lceil \frac{1}{\gamma}\rceil}$ given by Proposition 2.2. Thus in general it is more efficient to use our method than to directly evaluate $FG \bmod P$ if the polynomial is known.*

4 Verification of polynomial modular product

This section is devoted to the verification of polynomial modular product. That is, given F, G, H and P, such that $\deg(F), \deg(G), \deg(H) < \deg(P) = n$, we want to test whether $H = FG \bmod P$. The idea is classical, that is to evaluate the identity at a random point. Contrary to the more straightforward verification of polynomial multiplication, we cannot do such evaluation directly since we do not know the polynomial $Q = (FG - (FG \bmod P))/P$. As seen in the previous section, we provide new algorithms to do such evaluation efficiently without reverting to the computation of Q. We remind that P is always taken monic. Note that this is a mild assumption since $(FG) \bmod P = (FG) \bmod (\lambda P)$ for any invertible constant λ.

In the following, all algorithms are analysed both when the polynomials F, G and H are dense and when they are sparse. On the other hand, P is always considered as a sparse polynomial. We shall recall that γ denotes the gap parameter of P, defined by $1-\gamma = \deg(P - X^n)$, and it serves to control the densification of the modular reduction.

We first begin in Section 4.1 with an abstract case where the polynomials are defined over an integral domain. There, we analyse the algorithms by counting the number of ring operations. In Sections 4.2 and 4.3 we discuss some adaptations of the algorithm to the case of integers and small finite fields in order to provide finer analysis in the bit complexity model.

4.1 Modular product verification in $\mathscr{R}[X]$

Algorithm 6 depicted below is straightforward from Theorems 3.6 or 3.11. We mainly provide his description to serve as a starting point for its adaptations in the next sections. The algorithm covers both the dense and the sparse case. The only difference is at Step 1.

Algorithm 6 MODULARVERIFICATION

Input: F, G, H and $P \in \mathscr{R}[X]$, P monic of degree n and F, G and H of degrees $< n$; $0 < \epsilon < 1$.
Output: True if $H = FG \bmod P$, False with probability at least $1-\epsilon$ otherwise.

1: **if** F, G and H are given in sparse representation **then**
2: **if** $\#H > \#F\#G(\#P-1)^{\lceil 1/\gamma\rceil}$ **then return** False ▷ Proposition 2.2
3: $\alpha \leftarrow$ random element from a subset $\mathscr{E}$ of $\mathscr{R}$, of size $\geq \frac{1}{\epsilon}(n-1)$
4: $\beta \leftarrow (FG \bmod P)(\alpha)$ ▷ using Theorem 3.6 or 3.11
5: **return** True if $\beta = H(\alpha)$, False otherwise

Theorem 4.1. *If $\mathscr{R}$ has at least $\frac{1}{\epsilon}(n-1)$ elements, Algorithm 6 is correct.*
If F, G and H are dense, the algorithm uses $\mathcal{O}(\#Pn)$ operations in $\mathscr{R}$.
If F, G and H are sparse, the algorithm uses $\mathcal{O}((\#F\#P^{\lceil \frac{1}{\gamma}-1\rceil} + \#G + \#H)\log n)$ operations in $\mathscr{R}$.

Proof. Step 1 dismisses a trivial mistake if the polynomials are sparse. If $H = (FG) \bmod P$, $H(\alpha) = (FG \bmod P)(\alpha)$ for any α and the algorithm always returns True. Otherwise, let $\Delta = H - (FG) \bmod P$. Then Δ has degree $< n$, hence has at most $n-1$ roots since it is nonzero. Therefore, the probability that α, randomly chosen in $\mathscr{E}$, is a root of Δ is at most $(n-1)/\frac{1}{\epsilon}(n-1) = \epsilon$. The algorithm returns True in that case with probability at most ϵ.

The complexity is given by the cost of a single modular product evaluation that is stated in Theorem 3.6 for the dense case and in Theorem 3.11 for the sparse case. □

If $\mathscr{R}$ is not large enough, the algorithm fails and it is customary to revert to an extension ring $\mathscr{R}_{\text{ext}}$ to perform the evaluation in a larger set. Using Theorems 3.6 and 3.11, we get the following extension when the polynomials are evaluated on a random point of $\mathscr{R}_{\text{ext}}$ rather than $\mathscr{R}$.

Corollary 4.2. *Let $\mathscr{R}$ be an integral domain with less than $\frac{1}{\epsilon}(n-1)$ elements, and $\mathscr{R}_{ext}$ an extension ring of $\mathscr{R}$ with at least $\frac{1}{\epsilon}(n-1)$ elements. Then Algorithm 6 can be adapted by choosing a random element from $\mathscr{R}_{ext}$, with the same probability of success. It uses $\mathcal{O}(n\#P)$ operations in $\mathscr{R}$ and $\mathcal{O}(n)$ operations in $\mathscr{R}_{ext}$ if F, G and H are dense, and $\mathcal{O}((\#F\#P^{\lceil \frac{1}{\gamma}-1\rceil} + \#G + \#H)\log n)$ operations in $\mathscr{R}_{ext}$ if they are sparse.*

In the dense case, Algorithm 6 uses an optimal number of operations in $\mathscr{R}$ as soon as $\#P$ is constant. It is always faster than a modular product when $\#Pn < \mathsf{M}(n)$ that is when $\#P < \log(n)\log\log(n)$ for a general ring $\mathscr{R}$. In the sparse case, Theorem 4.1 is not linear in the input size. Indeed, $\#P$ is raised to a potentially large power $\lceil \frac{1}{\gamma} - 1 \rceil$ and more importantly there is a factor $\log n$ in the number of operations in $\mathscr{R}$ while the input has only $(\#F + \#G + \#H)$ elements of $\mathscr{R}$. Nevertheless, the efficiency of verification has to be compared with the cost of computing $FG \bmod P$. We assume the latter to be done with a sparse multiplication followed by a sparse division with P. This hypothesis seems reasonable as no work have been done to optimize such operation yet. Letting aside the division, the number of operations in $\mathscr{R}$ for sparse multiplication could be either $\mathcal{O}(\#F\#G)$ with naive approach or $\tilde{\mathcal{O}}(\#F + \#G + \#(FG))$ using [9]. Assuming $\#P$ to be constant, our verification has a complexity of $\mathcal{O}((\#F + \#G + \#H)\log n)$ which is always faster when $\#(FG) = o(\#F\#G/\log n)$. If we assume that $\#(FG) = \mathcal{O}(\#F\#G)$, our verification will be faster at least when $n = 2^{\mathcal{O}(\#F+\#G)}$ and $\#P$ constant. Depending on the cost of the division, our algorithm could be faster in more cases.

Of course, these conditions are not very restrictive. We use Algorithm 6 in Section 5 to verify classical polynomial multiplication, where P will be a binomial of degree either logarithmic in the sparsity or polynomial in the input degree.

Yet the efficiency of Algorithm 6 depends heavily on the integral domain $\mathscr{R}$. Indeed the complexity of polynomial multiplication in $\mathscr{R}[X]$ can be faster than $\mathcal{O}(n \log n \log\log n)$ operations in $\mathscr{R}$ [13, 15]. Furthermore, if $\mathscr{R}$ is small, one operation in $\mathscr{R}_{\text{ext}}$ corresponds to a non-constant number of operations in $\mathscr{R}$. In the following sections we consider polynomials over the integers or finite fields and we provide thorough analyses together with adapted versions when necessary.

4.2 Modular product verification in $\mathbb{Z}[X]$

If the polynomials are defined over $\mathbb{Z}$, there is no difficulty with the size of $\mathscr{E}$ in Algorithm 6. However we must prevent the integers growth during the evaluation. It is very classical to choose a random prime q and to map the whole computation into $\mathbb{F}_q$. To do so, we must ensure two properties on the prime q. First, q must be large enough to use the algorithm, that is at least $\frac{1}{\epsilon}(n-1)$. Second, if $H \neq (FG) \bmod P$, we need this inequality to hold modulo q as well. For this second property, we define $\Delta = H - (FG) \bmod P$. To ensure that Δ does not vanish modulo q, we need that at least one coefficient of Δ is nonzero modulo q. We then need to bound its coefficients to assess the latter fact.

Proposition 4.3. *The coefficients of Δ are bounded by $\|H\|_\infty + \min(\#F, \#G)\|F\|_\infty\|G\|_\infty(\#P\|P\|_\infty)^{\lceil \frac{1}{\gamma} \rceil}$.*

Proof. The coefficient of Δ are bounded by $\|H\|_\infty + \|FG \bmod P\|_\infty$. The bound follows from Proposition 2.2, with $Q = FG$ and $\|Q\|_\infty \leq \min(\#F, \#G)\|F\|_\infty\|G\|_\infty$ by Lemma 2.1. □

Using this bound and Proposition 2.6 we can determine an appropriate prime q to adapt the Algorithm 6 to the integer case. This is done in the Algorithm ModularVerificationOverZ below.

Algorithm 7 ModularVerificationOverZ

Input: F, G, H and $P \in \mathbb{Z}[X]$, P monic of degree n and F, G and H of degrees $< n$; $0 < \epsilon < 1$.
Output: True if $H = FG \bmod P$, False with probability at least $1 - \epsilon$ otherwise.

1: $\Delta_\infty \leftarrow \|H\|_\infty + \min(\#F, \#G)\|F\|_\infty\|G\|_\infty(\#P\|P\|_\infty)^{\lceil \frac{1}{\gamma} \rceil}$
2: $q \leftarrow$ RandomPrime$(\lambda, \frac{\epsilon}{2})$ where $\lambda = \max(21, \frac{2}{\epsilon}n, \frac{20}{3\epsilon}\ln\Delta_\infty)$
3: $(F_q, G_q, H_q, P_q) \leftarrow (F \bmod q, G \bmod q, H \bmod q, P \bmod q)$
4: **return** ModularVerification$(F_q, G_q, H_q, P_q, \frac{\epsilon}{2})$

Theorem 4.4. *Algorithm 7 is correct. If $C = \max(\|P\|_\infty, \|F\|_\infty, \|G\|_\infty, \|H\|_\infty)$ and $T = \max(\#F, \#G, \#H)$, the algorithm requires $\mathcal{O}(\log^2(\frac{n}{\epsilon}\log C)\log\log(\frac{n}{\epsilon}\log C)\log\frac{1}{\epsilon}\mathsf{I}(\log(\frac{n}{\epsilon}\log C)))$ bit operations to get a prime number, plus*

- $\mathcal{O}((\#Pn + n\frac{\log C}{\log(\frac{n}{\epsilon}\log C)})\mathsf{I}(\log(\frac{n}{\epsilon}\log C)))$ *bit operations if F, G and H are dense, or*
- $\mathcal{O}((T\#P^{\lceil \frac{1}{\gamma}-1 \rceil}\log n + (T + \#P)\frac{\log C}{\log(\frac{n}{\epsilon}\log C)})\mathsf{I}(\log(\frac{n}{\epsilon}\log C)))$ *bit operations if F, G and H are sparse.*

Proof. To ensure that ModularVerification works properly, we need q to be at least $\frac{2}{\epsilon}(n-1)$. This is the case since $\lambda \geq \frac{2}{\epsilon}n$. The algorithm always returns the correct answer when $H = (FG) \bmod P$. Otherwise,

it may incorrectly return True in two cases: Either $H_q = (F_q G_q) \bmod P_q$ while the equality does not hold over $\mathbb{Z}$, or ModularVerification incorrectly returns True. Both situations occur with probability at most $\frac{\epsilon}{2}$. Indeed, by Proposition 4.3, Δ_∞ is always a bound on $\|\Delta\|_\infty$ where $\Delta = H - (FG) \bmod P$. Therefore, Proposition 2.6 shows that with probability at least $\frac{\epsilon}{2}$, the number q chosen at Step 2 is a prime number such that $\Delta \bmod q \neq 0$. The error probability of Algorithm 7 is thus at most ϵ.

Let us now analyse the complexity of the algorithm. As a first step, we shall express q in terms of the input size. Since $\#P, \#F, \#G \leq n$, $\Delta_\infty = \mathcal{O}(n^{1+\lceil \frac{1}{\gamma} \rceil} C^{2+\lceil \frac{1}{\gamma} \rceil})$. Thus, $\ln \Delta_\infty = \mathcal{O}(\frac{1}{\gamma}(\log n + \log C)) = \mathcal{O}(n(\log n + \log C))$ since $\frac{1}{\gamma} \leq n$. This implies $q = \mathcal{O}(\frac{n}{\epsilon}(\log n + \log C))$ and $\log q = \mathcal{O}(\log(\frac{n}{\epsilon} \log C))$.

By Proposition 2.4, Step 2 costs $\mathcal{O}(\log \frac{1}{\epsilon} \log^2 q \log\log q \mathsf{I}(\log q))$ bit operations, that is

$$\mathcal{O}\left(\log \tfrac{1}{\epsilon} \log^2(\tfrac{n}{\epsilon} \log C) \log\log(\tfrac{n}{\epsilon} \log C) \mathsf{I}(\log(\tfrac{n}{\epsilon} \log C))\right). \tag{9}$$

Step 3 requires $\mathcal{O}(n \frac{\log C}{\log q} \mathsf{I}(\log q))$ bit operations in the dense case, and $\mathcal{O}((T + \#P) \frac{\log C}{\log q} \mathsf{I}(\log q))$ bit operations in the sparse case. By Theorem 4.4, Step 4 requires $\mathcal{O}(\#P n \mathsf{I}(\log q))$ bit operations in the dense case and $\mathcal{O}(T \#P^{\lceil \frac{1}{\gamma} - 1 \rceil} \log n \mathsf{I}(\log q))$ bit operations in the sparse case. Adding the complexities of all these steps leads to the claimed bit complexity. □

Remark 4.5. *In most cases, the cost of finding a prime number is negligible in comparison to the rest of the algorithm. In the dense case, it is negligible as long as $\epsilon = 1/n^{\mathcal{O}(1)}$. In the sparse case, it is negligible when the degree n is not too large compared to the other input parameters. More precisely, this is the case when $\frac{n}{\epsilon} = ((T + \#P) \log C)^{\mathcal{O}(1)}$.*

When computing a multiplication followed by a modular reduction with polynomials in $\mathbb{Z}[X]$, the size of the coefficients can grow significantly as shown by the bound on $\|FG \bmod P\|_\infty$ given in the proof of Proposition 4.3. At the opposite, our verification algorithm is done with bounded integers of bit length $\mathcal{O}(\log(\frac{n}{\epsilon} \log C))$. This is logarithmic in the input size in dense representation and linear in the sparse one. Our verification therefore avoids paying the coefficient growth, contrary to the direct computation. Taking this growth into account to compare our verification algorithm with the computation seems hard. We only detail the cases where our algorithm is already faster even without considering the coefficient growth. We shall mention that for sparse polynomial, $\#(FG \bmod P)$ might be smaller that $\#H$. In our analysis, we assume for simplicity that both have approximately the same size.

Remark 4.6. *For $\epsilon = 1/n^{\mathcal{O}(1)}$, Algorithm 7 is faster than the polynomial modular product*

(i) when the polynomials are dense and $\#P < \min(\frac{\log n}{\log\log n}, \frac{\log C}{\log\log\log C})$;

(ii) when the polynomials are sparse and $\frac{n}{\epsilon} = ((T + \#P) \log C)^{\mathcal{O}(1)}$.

Proof. We assume $\epsilon = 1/n^{\mathcal{O}(1)}$. In particular, $\log \frac{n}{\epsilon} = \mathcal{O}(\log n)$. To simplify the analysis, we place ourselves in the case of a negligible cost for finding the prime q, as described in Remark 4.5. In the sparse case, this implies that $\frac{n}{\epsilon}$ must be polynomial in $(T + \#P) \log C$. Therefore $\log n < \min(T, \#P)$ and $\mathsf{I}(\log(n \log C)) = \mathsf{I}(\log\log C)$. This makes our verification faster than computing the modular product since the latter requires at least $\#(FG \bmod P) = \mathcal{O}(T^2 \#P^{\lceil \frac{1}{\gamma} \rceil})$ operations on integers of bit length $\log C$.

For the dense case, we compare to the cost of multiplying two polynomials of degree n and coefficients bounded by C. As seen in the introduction, this reduces to integer multiplication with Kronecker substitution and it costs $\mathsf{I}(n(\log n + \log C)) = \mathcal{O}(n(\log^2 n + \log n \log C + \log C \log\log C)$ bit operations. By Theorem 4.4 our verification needs $\mathcal{O}(\#P n \mathsf{I}(\log n + \log\log C) + n \log C \log(\log n + \log\log C))$ bit operations. The second term is dominated by the complexity of multiplying the polynomials when $\epsilon = 1/n^{\mathcal{O}(1)}$. The first term is

$$\mathcal{O}(\#P n((\log n + \log\log C) \log\log n + (\log n + \log\log C) \log\log\log C)).$$

When $\#P < \min(\frac{\log n}{\log\log n}, \frac{\log C}{\log\log\log C})$, this term is bounded by $\mathcal{O}(n(\log^2 n + \log n \log C + \log C \log\log C))$, that is the complexity of computing the product. □

4.3 Modular product verification in $\mathbb{F}_q[X]$

The situation over a finite field $\mathbb{F}_q$ is different since there is no growth to prevent. When q is large enough, Theorem 4.1 applies directly. Otherwise, one can revert to computing in a sufficiently large extension field of $\mathbb{F}_q$, where Corollary 4.2 can be applied. We first give the precise complexity bounds for these two cases.

Corollary 4.7. *Let F, G, H and $P \in \mathbb{F}_q[X]$ as in Algorithm 6. One can test whether $H = (FG) \bmod P$ using Algorithm 6, with $\mathcal{O}(\log \frac{1}{\epsilon}(\log_q \frac{n}{\epsilon})^2 \mathsf{M}(\log_q \frac{n}{\epsilon})(\log q + \log\log_q \frac{n}{\epsilon}))$ operations in $\mathbb{F}_q$ to get an irreducible polynomial of degree $\mathcal{O}(\log_q \frac{n}{\epsilon})$, plus*

- *$\mathcal{O}(n\#P + n\mathsf{M}(\log_q \frac{n}{\epsilon}))$ operations in $\mathbb{F}_q$ if F, G and H are dense, or*
- *$\mathcal{O}(T\#P^{\lceil \frac{1}{\gamma}-1 \rceil} \log n \mathsf{M}(\log_q \frac{n}{\epsilon}))$ operations in $\mathbb{F}_q$ if F, G and H are sparse with at most T nonzero monomials.*

Proof. Let us assume that $q < \frac{1}{\epsilon}(n-1)$ otherwise Theorem 4.1 applies straightforwardly. In that case, Corollary 4.2 requires to choose a random point in an extension $\mathbb{F}_{q^d}$ of $\mathbb{F}_q$ with at least $\frac{1}{\epsilon}(n-1)$ elements. More precisely, we use Proposition 2.8 to produce with probability $1-\frac{\epsilon}{2}$ an irreducible polynomial of degree d over $\mathbb{F}_q$, where d is the smallest integer such that $q^d \geq \frac{2}{\epsilon}(n-1)$. The algorithm may be incorrect if either the polynomial used to defined $\mathbb{F}_{q^d}$ fails to be irreducible, or if the Algorithm 6 fails. If we choose α at random in $\mathbb{F}_{q^d}$, the error probability of Algorithm 6 is at most $\frac{\epsilon}{2}$. This gives a total probability of error of at most ϵ.

By definition, the degree of the extension is $d = \mathcal{O}(\log_q \frac{n}{\epsilon})$. The cost of generating the irreducible polynomial of degree d is $\mathcal{O}(\log \frac{1}{\epsilon} d^2 \mathsf{M}(d)(\log q + \log d))$ by Proposition 2.8. Using Corollary 4.2 and the fact that an operation in $\mathbb{F}_{q^d}$ costs $\mathsf{M}(d)$ operations in $\mathbb{F}_q$, we obtain the claimed costs. □

Remark 4.8. *If $\epsilon = 1/n^{\mathcal{O}(1)}$, the cost of getting an irreducible polynomial is negligible in the dense case. Then the algorithm requires $\mathcal{O}(\#Pn + n\mathsf{M}(\log_q n))$ operations in $\mathbb{F}_q$. If we add the degree constraint $\log n = o(T)$, the cost of getting an irreducible polynomials is also negligible in the sparse case and the algorithm requires $\mathcal{O}(T\#P^{\lceil \frac{1}{\gamma}-1 \rceil} \log n \mathsf{M}(\log_q n))$ operations in $\mathbb{F}_q$.*

As $\#(FG \bmod P)$ is bounded by $T^2\#P^{\lceil \frac{1}{\gamma} \rceil}$ and computing $FG \bmod P$ requires at least one operation in $\mathbb{F}_q$ for each monomial, this remark gives directly a case where the verification is faster than the modular product in general. Moreover, naive algorithms can be used to perform products in the extension of $\mathbb{F}_q$.

Remark 4.9. *When $\epsilon = 1/n^{\mathcal{O}(1)}$ and $n = T^{\mathcal{O}(1)}$, Algorithm 6 in the sparse case is in general faster than the modular multiplication and requires $\tilde{\mathcal{O}}(T\#P^{\lceil \frac{1}{\gamma}-1 \rceil})$ bit operations if $q < \frac{n}{\epsilon}$.*

In the dense case, we can see from Remark 4.8 that the verification complexity might in fact be larger than the cost of computing the modular product $FG \bmod P$. Indeed, assuming $\mathsf{M}_q(n) = \mathcal{O}(n \log q(\log n + \log\log q))$ bit operations [15], we have $\mathsf{M}_q(\log_q \frac{n}{\epsilon}) = \mathcal{O}(\log \frac{n}{\epsilon}(\log\log \frac{n}{\epsilon} + \log\log q))$. While the cost of computing $FG \bmod P$ uses $\mathsf{M}_q(n)$ bit operations, our verification requires $\mathcal{O}(\#Pn \log q \log\log q + n \log \frac{n}{\epsilon}(\log\log \frac{n}{\epsilon} + \log\log q))$ bit operations. When $\#P$ is not a constant, the latter is always larger. The following remark precise when we can expect a positive result.

Remark 4.10. *Assuming $\#P$ to be constant and $\epsilon = 1/n^{\mathcal{O}(1)}$, Algorithm 6 in the dense case with $\mathscr{R} = \mathbb{F}_q$ is asymptotically faster than the modular multiplication when*

(i) *$\log \frac{n}{\epsilon} < q < \frac{n}{\epsilon}$, since modular multiplication costs $\mathcal{O}(n \log q \log n)$ bit operations while verification, which does need an extension, is $\mathcal{O}(n \log \frac{n}{\epsilon} \log\log \frac{n}{\epsilon})$.*

(ii) *$\frac{n}{\epsilon} < q < 2^{\frac{n}{\epsilon}}$, since modular multiplication costs $\mathcal{O}(n \log q \log n)$ bit operations while verification, which does not use an extension, is $\mathcal{O}(n \log q \log\log q)$;*

If the field is very large $q > 2^{\frac{n}{\epsilon}}$, Algorithm 6 is asymptotically as fast as the modular multiplication. The dominant factor in both complexity is $\mathcal{O}(n \log q \log\log q)$.

We shall mention that the verification cost in (i) of Remark 4.10 assumes the use of fast multiplication of polynomials in order to check fast multiplication of polynomial of degree $\mathcal{O}(n)$. Even though this dependency is not a problem in theory it might not be satisfactory in practice. One solution would be to use a naive polynomial multiplication for the extension field arithmetic but this further tightens the superiority of the verification.

Remark 4.11. *Assuming that extension field arithmetic is done naively using quadratic polynomial multiplication, Algorithm 6 remains faster than modular multiplication only when $n^{1/3} < q$ in the dense case.*

We now propose an novel method that enables us to improve all the dense cases where an extension field is necessary, while not relying on any polynomial arithmetic. More precisely, we show that fast verification does exist when $q < \log n$, which is of great interest for the field $\mathbb{F}_2$. It is based on the evaluation of polynomials on matrices rather than scalars, combined with Freivalds algorithm for verifying matrix multiplication [7].

Indeed, choosing α from an extension field inherently leads to depend on polynomial multiplication. Instead of picking a random point that is probably not a root of $\Delta = H - (FG) \bmod P$ when $\Delta \neq 0$, we pick a polynomial $R \in \mathbb{F}_q[X]$ of degree $k < n$ that is probably not a divisor of $\Delta \neq 0$. To test whether R divides Δ, we evaluate Δ on the companion matrix C_R of R, defined by

$$C_R = \begin{pmatrix} 0 & 0 & \cdots & 0 & -r_0 \\ 1 & 0 & \cdots & 0 & -r_1 \\ 0 & 1 & \cdots & 0 & -r_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -r_k \end{pmatrix}$$

where $R = \sum_{i=0}^{k} r_i X^i$. This strategy relies on the fact that R is the minimal polynomial of its companion matrix. Therefore, $R(C_R) = 0$ and any polynomial Δ such that $\Delta(C_R) = 0$ must be a multiple of R. In other words, R divides Δ if and only if $\Delta(C_R) = 0$. We will show that taking R irreducible over $\mathbb{F}_q$ of degree $k = \mathcal{O}(\log n)$ makes this approach faster then the one using extension field when ϵ is constant. Furthermore, it will extend the possibility to have fast verification for any fields, whatever the size of the polynomials.

To check whether $\Delta(C_R) = 0$, we need to evaluate H and $(FG) \bmod P$ on C_R, and to verify that the evaluations match. Of course, one cannot directly evaluate those polynomials on C_R as it would cost $\mathcal{O}(nk^\omega)$ operations in $\mathbb{F}_q$ for the dense case, where $\omega < 2.3729$ is the best exponent for matrix multiplication [21]. Since $k = \mathcal{O}(\log n)$, this would not give any improvement to Remark 4.10.

Instead, we rely on the so-called Freivald's technique to verify matrix multiplication [7]. The idea is that the matrix product $C = A \times B \in \mathcal{R}^{k \times k}$ can be verified by asserting that $uC = (uA) \times B$ for a random vector $u \in \{0,1\}^n$ with a probability of error of $1/2$. To assert that two polynomials evaluations on the matrix C_R match, it is sufficient to verify that their projection by the vector u are equal. Given a degree-n polynomials $H \in \mathbb{F}_q[X]$, one can compute $uH(C_R)$ in $\mathcal{O}(nk)$ operation in $\mathbb{F}_q$ using Horner evaluation:

$$uH(C_R) = u \sum_{i=0}^{n} h_i {C_R}^i = \left(\sum_{i=1}^{n} h_i u {C_R}^{i-1} \right) C_R + uh_0. \tag{10}$$

Since matrix-vector product with C_R only costs $\mathcal{O}(k)$ operations in $\mathbb{F}_q$, and Horner procedure only uses n of those matrix-vector products, the cost is clear.

Remark 4.12. *It is sufficient to replace the evaluation of $F(\alpha)$ and $P(\alpha)$ by $uF(C_R)$ and $uP(C_R)$ in Algorithm 3 (*ModularEvaluation*) to reach a complexity of $\mathcal{O}(n(\#P + \deg(R)))$ operations in $\mathbb{F}_q$ for computing $u(FG \bmod P)(C_R)$ in the dense case. More informally, it is sufficient to say that any of the operations in $\mathcal{R}_{ext}$ have now the cost of one matrix-vector product with C_R.*

Theorem 4.13. *Let F, G, H and $P \in \mathbb{F}_q[X]$, as in Algorithm 6. We can check whether $H = FG \bmod P$ in $\mathcal{O}(\#Pn + n \log_q n \log \frac{1}{\epsilon})$ operations in $\mathbb{F}_q$ with a probability of error at most ϵ if $H \neq FG$.*

Proof. Let $0 < \epsilon_1 < \frac{1}{4}$ be a fixed probability. The algorithm needs two steps. First it computes with probability at least $1 - \frac{1}{\epsilon_1}$ an irreducible polynomial R of degree $d = \lceil \log_q \frac{2n}{\epsilon_1} \rceil$ using Proposition 2.8. Second, it computes $uH(C_R)$ and $u(FG \bmod P)(C_R)$ for some random vector $u \in \{0,1\}^d$. If both evaluations are distinct, the algorithm returns False. Otherwise, it repeats $\mathcal{O}(\log \frac{1}{\epsilon})$ these two steps until one of the repetition fails. If this is the case it return false, otherwise the algorithm return true.

If $H = (FG) \bmod P$, the algorithm always returns True. Let us assume that $H \neq (FG) \bmod P$, and let $\Delta = H - (FG) \bmod P$. For the algorithm to return True, each repetition must ensure that $uH(C_R) = u(FG \bmod P)(C_R)$. This may happen if either R divides Δ, whence $H(C_R) = (FG \bmod P)(C_R)$, or R does not divide Δ but $uH(C_R) = u(FG \bmod P)(C_R)$. Since there are at least $q^d/2d$ irreducible polynomials of degree d in $\mathbb{F}_q$ by Proposition 2.7 and at most n/d of them divide Δ, the probability that R divides Δ is at most $2n/q^d \leq \epsilon_1$ provided R is irreducible. Taking into account the probability that R is not irreducible, the probability that R divides Δ is at most $2\epsilon_1$. Then, using Freivalds standard argument, if R does not divide Δ, the probability $u\Delta(C_R) = 0$ is at most $\frac{1}{2}$. Altogether, the probability that one iteration returns True is at most $\frac{1}{2} + 2\epsilon_1 < 1$. Therefore, the probability that $\log \frac{1}{\epsilon} / \log(\frac{1}{2} + 2\epsilon_1)$ independent iterations all return True is at most ϵ.

Let us now analyse the complexity of the algorithm. Since ϵ_1 is a constant, the second step uses $\mathcal{O}(\#Pn + n \log_q n)$ operations in $\mathbb{F}_q$ using Remark 4.12. The first step is negligible, even if naive polynomial arithmetic is used. Note that in this complexity, $\#Pn$ is the cost of Algorithm 2 (LeadingCoefficients). Since it is deterministic and only depends on P and F, it can be called only once rather than at each iteration.

We then get the complexity $\mathcal{O}(\#Pn + n \log_q n \log \frac{1}{\epsilon})$. □

Remark 4.14. *Note that compared to using evaluation at α in an extension field, no operation depends on ϵ. If ϵ is fixed, the new method replaces a factor $\mathsf{M}(\log_q n)$ in the complexity by $\log_q n$. Moreover our new approach requires only simple computations: additions of vectors, multiplication of a vector by a scalar and matrix-vector product with a companion matrix. Furthermore, when $\#P$ and ϵ are constants, the verification is always faster than the modular multiplication, whatever the size of q.*

This new method still requires some polynomial arithmetic even if only naive polynomial multiplication is used. This is because the algorithm called to provide the degree-d polynomial R relies on polynomial products and GCDs to ensure that R is probably irreducible. In order to remove the dependency to polynomial arithmetic, we can just choose a random monic degree-d polynomial R and compute the evaluation on C_R even if R is not irreducible. This implies to take several random polynomials R to reach the target probability ϵ.

Corollary 4.15. *Let F, G, H and $P \in \mathbb{F}_q[X]$, as in Algorithm 6. Without using any polynomial multiplication we can check whether $H = FG \bmod P$ in $\mathcal{O}(\#Pn + n(\log_q n)^2 \log \frac{1}{\epsilon})$ operations in $\mathbb{F}_q$ with a probability of error at most ϵ if $H \neq FG$.*

Proof. In the proof of Theorem 4.13, we replace one evaluation on C_R with R irreducible with probability at least $1-\epsilon_1$ by few evaluations on several C_R with R random monic polynomial of degree d. As the probability of a random monic polynomial R to be irreducible is at least $\frac{1}{2d}$ by Proposition 2.7, we need to generate $\mathcal{O}(d \log \frac{1}{\epsilon_1})$ random polynomials R to reach a probability at least $1-\epsilon_1$ that at least one of them is irreducible. Thus the evaluation part of the algorithm is repeated $\mathcal{O}(d) = \mathcal{O}(\log_q n)$ times since ϵ_1 is constant. □

Even if this new approach does not improve the complexity from evaluation at α in an extension field using naive polynomial multiplication, it allows to remove completely the dependency to any polynomial multiplication algorithm While this result might not being seen useful as first sight, it will be used in Section 5.1 to provide efficient verification for polynomial multiplication. Indeed, in that case we will need to use verification with P being of degree smaller than the input degree.

This new method using companion matrix also works in the sparse case. Indeed, any power α^t in Algorithm 5 (SPARSEMODULAREVALUATION) are now replaced with C_R^t. However, only few powers C_R^t with $1 < t < n$ are relevant and we cannot compute all of them as in the dense case. This implies that computing $u(FG \bmod P)(C_R)$ instead of $(FG \bmod P)(C_R)$ is useless in that case. Indeed, using fast exponentiation together with the structure of the powers of companion matrices [12] already yield a complexity of $\mathcal{O}((\#F\#P^{\lceil \frac{1}{\gamma}-1 \rceil} + \#G) \log n \log_q^2 n)$ operations in $\mathbb{F}_q$, and we cannot hope to lower this down by some random vector projection. In that case, using Freivald technique is useless and we have a better probability of success. Choosing $\mathcal{O}(\log \frac{n}{\epsilon} \log \frac{1}{\epsilon})$ polynomials R at random, at least one of them is irreducible and does not divide $\Delta = H - FG \bmod P$ with probability $1-\epsilon$. This leads to an algorithm which does not use any polynomial product in the sparse case too. Even though it is asymptotically not as fast as the verification in an extension field where naive polynomial arithmetic is used, it is still quasi-linear.

Corollary 4.16. *Let $P \in \mathbb{F}_q[X]$ be monic of degree n and F, G, $H \in \mathbb{F}_q[X]$ of degree less than n and sparsity at most T, and $0 < \epsilon < 1$. Using a direct evaluation on a companion matrix, we can check whether $H = FG \bmod P$ with a probability of error at most ϵ if $H \neq FG$ in $\mathcal{O}(T\#P^{\lceil \frac{1}{\gamma}-1 \rceil} \log n (\log_q \frac{n}{\epsilon})^3 \log \frac{1}{\epsilon})$ operations in $\mathbb{F}_q$, without performing any polynomial product.*

5 Polynomial product verification

In this section we study the simpler problem of verifying a classical polynomial multiplication. Given three polynomials F, G and $H \in \mathcal{R}[X]$ of respective degrees n, n and $2n$, the classical idea to verify $H = FG$ simply falls down to testing $H(\alpha) = F(\alpha)G(\alpha)$ for some random α in a large enough set $\mathcal{S}$. As mentioned in the introduction, this strategy may or may not have an optimal bit complexity, depending on the context. Here we are concerned with two difficulties that arise in either the dense or the sparse cases.

If the polynomials are dense, the verification through evaluation requires a number of operations in $\mathcal{S}$ that is linear in the input polynomials degree n. When $\mathcal{R}$ has more than n elements, taking $\mathcal{S} \subset \mathcal{R}$ is sufficient to use evaluation. However, multiplication in $\mathcal{R}$ has not a linear bit complexity and best known results remain quasi-linear [2, 15, 14]. The evaluation therefore leads to a quasi-linear bit complexity of $\mathcal{O}(n\mathsf{I}(\log n)) = \mathcal{O}(n \log n \log\log n)$. When $\mathcal{R}$ is too small, for instance with a small finite field, $\mathcal{S}$ is classically taken as a field extension of $\mathcal{R}$, large enough to make it unlikely that α is a root of $H - FG$. Therefore, each

operation in $\mathscr{S}$ corresponds to an operation over $\mathscr{R}[X]$ with non-negligible degree, meaning that the number of operations in $\mathscr{R}$ is no more linear in the inputs degree n. As mentioned in the introduction, Kaminski's approach [20] circumvents the later problem by replacing the evaluation with a computation in $\mathscr{R}[X]/(X^i-1)$ for random integer i in a prescribed range. There, his algorithm is able to verify dense polynomial products with a linear number of operations in $\mathscr{R}$ whatever the ring size. However, the same difficulty as for large rings may arise. Since operations in $\mathscr{R}$ do not have an linear bit complexity, unless say $\mathscr{R} = \mathbb{F}_2$, this is not always sufficient to reach an optimal bit complexity for the verification. In Section 5.1, we present Kaminski's approach [20] and we provide a thorough analysis in the bit complexity model. In particular, we show that it is possible to get optimal verification in the bit complexity model for any polynomial in $\mathbb{Z}[X]$ and for some polynomials in $\mathbb{F}_q[X]$, depending on the relation between q and n.

If the polynomials F, G and H are sparse with at most T nonzero coefficients, the evaluation requires a number of operations in $\mathscr{S}$ that is $\mathscr{O}(T \log n)$. However the input bit size is given by the size of the exponents *plus* the size of the coefficients, that is $\mathscr{O}(T + \log n)$ bits. Since $\mathscr{S}$ has to be of size at least $\mathscr{O}(\log n)$, the bit complexity of evaluation would be $\mathscr{O}(T \log^2 n)$ which is not even quasi-linear. In Section 5.2 we develop a novel method, already appearing in [9], to verify sparse polynomial multiplication with a quasi-linear bit complexity of $\tilde{\mathscr{O}}(T + \log n)$.

5.1 Dense polynomial product verification

In [20], Kaminski describes an algorithm to verify a polynomial product $H = FG \in \mathscr{R}[X]$ using a linear number of operations in $\mathscr{R}$, regardless of its size. His method chooses at random a polynomial P that probably do not divides $\Delta = H - FG$ if $\Delta \neq 0$. Then he verifies $H = FG \in \mathscr{R}[X]/P$ using fast polynomial multiplication. Surprisingly, taking P of degree $o(n)$ in his algorithm enables to reach a linear number of operations in $\mathscr{R}$. In the following we will often use $\delta > 1.78107$ to be some constant value related to Euler's constant.

5.1.1 Kaminski's algorithm

The first step is to randomly select a polynomial from a fixed set, such that it most probably does not divide $\Delta = H - FG$ if $\Delta \neq 0$. A standard approach could be to consider irreducible polynomials. This would be the direct generalization of the evaluation method. However, Kaminski considers polynomials that are instead of the form $X^i - 1$, for some integers $i > 0$. These polynomials have two advantages: Reduction modulo $X^i - 1$ has a linear cost and all their divisors are cyclotomic polynomials so their least common multiple (lcm) has specific properties.

Proposition 5.1 ([20]). *For any integer set $I \subset \mathbb{N}$, $\prod_{i \in I} \Phi_i$ divides $\operatorname{lcm}\{X^i - 1 : i \in I\}$, where Φ_i is the i-th cyclotomic polynomial in $\mathscr{R}[X]$.*

Kaminski also gives a lower bound on the degree of $\operatorname{lcm}\{X^i - 1 : i \in I\}$, depending in I. In particular the proposition implies that a nonzero polynomial, divisible by k polynomials, of the form $X^i - 1$, cannot have a too small degree. In the converse direction, a nonzero polynomial Δ of degree at most $2n$ cannot have too many divisors of the form $X^i - 1$. This is the content of Kaminski's main theorem.

Theorem 5.2 ([20]). *Let Δ be a nonzero polynomial in $\mathscr{R}[X]$ of degree $\leq 2n$ and $0 < e < \frac{1}{2}$. Let $k = \lceil 2\delta n^e \ln\ln(n^{1-e}) \rceil$. At most $k-1$ polynomials in the set $\{X^i - 1 | n^{1-e} \leq i < 2n^{1-e}\}$ divide Δ.*

Kaminski's approach is then to choose a random integer $i \in [n^{1-e}, 2n^{1-e}[$, to reduce the input polynomials modulo $X^i - 1$ and to assert the equality in $\mathscr{R}[X]/(X^i - 1)$. We provide in Algorithm KaminskiVerification a more precise description of this approach.

Algorithm 8 KaminskiVerification

Input: $F, G, H \in \mathscr{R}[X]$ of degree n, n and $2n$; and $0 < e < \frac{1}{2}$.
Output: True if $H = FG$, False with probability at least $1 - (\lceil 2\delta n^e \ln\ln(n^{1-e}) \rceil - 1)/n^{1-e}$ otherwise.

1: $i \leftarrow$ random integer in $[n^{1-e}, 2n^{1-e}[$
2: $F_i, G_i, H_i \leftarrow F \bmod X^i - 1, G \bmod X^i - 1, H \bmod X^i - 1$
3: $M \leftarrow F_i G_i$ ▷ Using a fast multiplication algorithm
4: $M_i \leftarrow M \bmod X^i - 1$
5: **return** $M_i = H_i$

Theorem 5.3 ([20]). *Let F, G and $H \in \mathscr{R}[X]$ of degree at most n, n and $2n$, $0 < e < \frac{1}{2}$ and an integer k as in Theorem 5.2. Algorithm 8 uses $\mathcal{O}(n)$ operations in $\mathscr{R}$, and its failure probability is at most $(k-1)/n^{1-e}$ if $H \neq FG$.*

Remark 5.4. *To be more precise, Algorithm 8 requires $\mathcal{O}(n)$ additions in $\mathscr{R}$ at Step 2 to compute the first three reductions modulo $X^i - 1$, $\mathsf{M}(n^{1-e})$ operations in $\mathscr{R}$ to compute the product at Step 3, and $\mathcal{O}(n^{1-e})$ additions in $\mathscr{R}$ to compute the last reduction.*

One shall remark that the product in Step 3 must be computed with a subquadratic algorithm such that $\mathsf{M}(n^{1-e}) = \mathcal{O}(n)$ since $e < 1/2$. If the parameter e is taken close enough to $1/2$, Karatsuba's algorithm suffices to reach a linear number of operations. The failure probability is $\mathcal{O}(\log\log n/n^{1-2e})$, whence the need to have $e < 1/2$. We can bound this probability by $\mathcal{O}(\frac{1}{n^{e'}})$ for any positive integer $e' < 1-2e$. In order to reach a probability ϵ of error, the algorithm should be repeated $\mathcal{O}(\log_n \frac{1}{\epsilon})$ times. Note that this number of rounds is constant if ϵ is taken as $1/n^{\mathcal{O}(1)}$.

The drawback of such approach is to crucially rely on a somewhat fast multiplication algorithm, and to perform multiplications of polynomials of degrees more than $\sqrt{n}$. This means that optimal verification of the product of two degree-n polynomials uses a product of polynomials of degrees close to n. In some contexts, such as verifying an implementation, relying on the same problem is definitively problematic.

We note that all steps starting from Step 3 aim to verify $H_i = F_i G_i \bmod X^i - 1$ deterministically. It is easy to see that those steps can be replaced by our probabilistic modular product verification developed in Section 4. For polynomials over the integers or finite fields, this method does not require any polynomial multiplications at all.

Corollary 5.5. *If $\mathscr{R} = \mathbb{Z}$ or a finite field, and F, G and $H \in \mathscr{R}[X]$ of degrees n, n and $2n$. We can check whether $H = FG$ with a probability of failure at most ϵ if $H \neq FG$. This requires $\mathcal{O}(n \log_n \frac{1}{\epsilon})$ additions in $\mathscr{R}$ plus $o(n \log_n \frac{1}{\epsilon})$ operations in $\mathscr{R}$, without reverting to any polynomial multiplication. In particular, the algorithm uses an optimal number of operations in $\mathscr{R}$ when $\epsilon = 1/n^{\mathcal{O}(1)}$.*

Proof. We replace the last three steps of Algorithm 8 by a modular product verification, with a probability of failure at most $1/n$. Over $\mathbb{Z}$ or large finite fields, the complexity of this part is given by the dense version of Theorem 4.1 with $\#P = 2$ and degree $i = \mathcal{O}(n^{1-e})$. Over small finite fields, we rely instead on Corollary 4.15. In both cases, one can achieve a failure probability at most $1/n$ with at most $\mathcal{O}(\log n)$ repetitions of the algorithm, for a total number of operations in $\mathscr{R}$ that remains $o(n)$.

The total probability of failure of the modified algorithm is then $1/n + \mathcal{O}(1/n^{e'}) = \mathcal{O}(1/n^{e'})$ for some $e' > 0$. We can repeat this modified algorithm for $\mathcal{O}(\log_n \frac{1}{\epsilon})$ rounds to get the announced failure probability and complexity. □

5.1.2 Analysis in the bit complexity model

In [20], Kaminski only details the algebraic complexity of its polynomial product verification, and no further insights on the bit complexity are given. We now perform this analysis for polynomials over finite fields and over $\mathbb{Z}$. We surprisingly prove that his algorithm remains linear in number of bit operations in many cases. For polynomials over $\mathbb{F}_q$, the algorithm fails to be linear only when q is doubly exponentially larger than the degree. For polynomials over $\mathbb{Z}$, a similar condition applies. However, we are able to describe a variant of the algorithm that has linear bit complexity for polynomials with large coefficients. Hence we prove that polynomial product verification over $\mathbb{Z}$ has linear bit complexity in all cases. Our variant is based on integer product verification, for which Kaminski actually gives also in [20] a linear-time algorithm. Of course all those algorithms are therefore optimal.

The next theorem provides the bit complexity analysis of Kaminski's algorithm over finite fields.

Theorem 5.6. *Let F, G and $H \in \mathbb{F}_q[X]$ of degrees n, n and $2n$, and $0 < e < \frac{1}{2}$. Algorithm 8 requires $\mathcal{O}(n \log q + n^{1-e} \log q \log\log q)$ bit operations. When $\log\log q = \mathcal{O}(n^e)$, one can verify if $H = FG$ with failure probability at most ϵ if $H \neq FG$, using $\mathcal{O}(n \log q \log_n \frac{1}{\epsilon})$ bit operations which is optimal when $\epsilon = 1/n^{\mathcal{O}(1)}$.*

Proof. We apply the count of operations given in Remark 5.4. The additions give the term $\mathcal{O}(n \log q)$. The bit complexity of the product of degree-$\mathcal{O}(n^{1-e})$ polynomials over $\mathbb{F}_q$ is $\mathsf{M}_q(n^{1-e}) = \mathcal{O}(n^{1-e} \log q \log(n \log q))$, which is $\mathcal{O}(n \log q + n^{1-e} \log q \log\log q)$. We obtain the claimed complexity.

The second part directly follows from the observation that $\mathcal{O}(\log_n \frac{1}{\epsilon})$ rounds of the algorithm yield a failure probability at most ϵ. □

Note that the bound $\log\log q = \mathcal{O}(n^e)$ to get a linear number of bit operations in $n\log q$ is only valid when using the fastest known multiplication algorithm. If we replace by a slower algorithm, the bound becomes smaller. For instance, using Karatsuba's algorithm the product of degree-$\mathcal{O}(n^{1-e})$ polynomials uses $\mathcal{O}(n^{(1-e)\log 3}\log q\log\log q)$ ring operations. For the algorithm to still have an optimal complexity, we need that $n^{(1-e)\log 3}\log\log q = \mathcal{O}(n)$. This implies $e \geq 1 - 1/\log 3 \simeq 0.367$, and the bound becomes $\log\log q = \mathcal{O}(n^{1-(1-e)\log 3})$. If we take e close to $1/2$, say 0.45, the bound reads $\log\log q = \mathcal{O}(n^{0.13})$ while it is $\log\log q = \mathcal{O}(n^{0.45})$ using the fastest multiplication algorithm.

Further, as mentioned previously, using a fast multiplication algorithm for the verification of a polynomial product is problematic. We now analyse the bit complexity of our variant that does not use any polynomial product, that is of Corollary 5.5. We show that the same complexity and the same bound on q can be obtained without any polynomial product.

Remark 5.7. *Let F, G, $H \in \mathbb{F}_q[X]$ of degrees n, n and $2n$, and $0 < e < \frac{1}{2}$. Algorithm 8 can be implemented using a modular product verification and without any polynomial product. This variant has bit complexity $\mathcal{O}(n\log q + n^{1-e}\log q\log\log q)$. When $\log\log q = \mathcal{O}(n^e)$, one can verify if $H = FG$ with failure probability at most ϵ if $H \neq FG$, using $\mathcal{O}(n\log q\log_n\frac{1}{\epsilon})$ bit operations, which is optimal when $\epsilon = 1/n^{\mathcal{O}(1)}$, and without reverting to any polynomial product.*

Proof. The proof simply consists in using Corollary 4.15 in place of Remark 5.4 in the previous proof. □

Now we consider F, G and $H \in \mathbb{Z}[X]$ with $\|F\|_\infty, \|G\|_\infty, \|H\|_\infty \leq C$. We first analyse the bit complexity of Algorithm 8 and provide conditions for the algorithm to use a linear number of bit operations. Later we propose a variant to be able to verify $H = FG$ with a linear number of bit operations for any integer polynomials..

Theorem 5.8. *Let F, G and $H \in \mathbb{Z}[X]$ of degrees n, n and $2n$, and norms at most C, and $0 < e < \frac{1}{2}$. Algorithm 8 requires $\mathcal{O}(n\log C + n^{1-e}\log C\log\log C)$ bit operations. When $\log\log C = \mathcal{O}(n^e)$, one can verify if $H = FG$ with failure probability at most ϵ if $H \neq FG$, using $\mathcal{O}(n\log C\log_n\frac{1}{\epsilon})$ bit operations which is optimal when $\epsilon = 1/n^{\mathcal{O}(1)}$.*

Proof. The first three reductions require $\mathcal{O}(n)$ additions in $\mathbb{Z}$ to compute F_i, G_i and H_i, whose norms are at most n^eC. A careful computation of these additions using a binary tree uses $\mathcal{O}(\sum_{i=1}^{\log n}\frac{n}{2^i}\log(iC) = \mathcal{O}(n\log C)$ bit operations. Then the polynomial product is performed with inputs of degree n^{1-e} and norm n^eC. As discussed in the introduction, it requires $\mathsf{I}(n^{1-e}(\log(n^eC) + \log n^{1-e}))$ bit operations, that is $\mathcal{O}(n^{1-e}(\log n + \log C)(\log n + \log\log C)) = \mathcal{O}(n\log C + n^{1-e}\log C\log\log C)$. Finally the last reduction is performed with degree $2n^{1-e}$ and norm $n(n^eC)^2$ in $\mathcal{O}(n\log C)$ bit operations.

Repeating $\mathcal{O}(\log_n\frac{1}{\epsilon})$ times the algorithm provides the second part of the theorem. □

As for polynomials over finite fields, the final computations can be replaced by a modular product verification. Here this yields a slightly better complexity. This improvement translates into an exponentially smaller constraint on the norm C for the algorithm to be optimal.

Remark 5.9. *Let F, G and $H \in \mathbb{Z}[X]$, of degrees n, n and $2n$ and norms at most C, and $0 < e < \frac{1}{2}$. Algorithm 8 can be implemented using a modular product verification and without any polynomial product. This variant has bit complexity $\mathcal{O}(n\log C + n^{1-e}\log(C)\log\log\log(C))$. When $\log\log\log C = \mathcal{O}(n^e)$, one can verify if $H = FG$ with failure probability at most ϵ if $H = FG$, using $\mathcal{O}(n\log C\log_n\frac{1}{\epsilon})$ bit operations, which is optimal when $\epsilon = 1/n^{\mathcal{O}(1)}$, and without reverting to any polynomial product.*

Proof. The proof is once again similar, using the dense part of Theorem 4.4 for the modular product verification. This verification is performed on polynomials of degrees $\mathcal{O}(n^{1-e})$ and norm at most n^eC. Its bit complexity is then $\mathcal{O}(n^{1-e}(\mathsf{I}(\log(n\log C)+\log(C)\log\log(n\log C))))$ which is $\mathcal{O}(n\log C+n^{1-e}\log(C)\log\log\log(C))$. This proves the first part of the remark. The second part relies on repetition of Algorithm 8. □

As long as the coefficients are not insanely huge compared to the degree, the previous remark applies and the polynomial product verification is linear. More precisely, this corresponds to C ranging from $\mathcal{O}(1)$ to $2^{2^{\mathcal{O}(n)}}$. To deal with this extreme case of huge coefficients, we develop another approach that is valid as soon as $\log n = \mathcal{O}(\log C)$. This means that all cases are covered with an optimal bit complexity. We shall mention that both methods are applicable when C is ranging from $n^{\mathcal{O}(1)}$ to $2^{2^{\mathcal{O}(n)}}$, which could be interesting when designing the most efficient implementation.

To treat the huge coefficient case, we rely on a result of Kaminski about the verification of the product of two integers. His technique is similar to the polynomial case: He reduces s-bit integers modulo $2^i - 1$ for some i between s^{1-e} and $2s^{1-e}$, and then performed the product with reduced integers.

Theorem 5.10 ([20]). *Let a, b, c be integers of at most s, s and $2s$ bits, $0 < e < \frac{1}{2}$ and $k = \lceil 2\delta s^e \ln\ln(s^{1-e})\rceil$ where $\delta > 1.78107$. We can check whether $ab = c$ in $\mathcal{O}(s)$ bit operations with a probability of error at most $(k-1)/s^{1-e}$ if $ab \neq c$.*

To verify a polynomial product $H = FG$ over $\mathbb{Z}$, we use the same idea as for computing the product. We use Kronecker substitution. If we evaluate each polynomial on β that is some large power of two, the coefficients of FG can directly be read on the digits of the integer $F(\beta)G(\beta)$. These evaluations at β require no operation. The polynomial product verification is thus reduced to an integer product verification $H(\beta) = F(\beta)G(\beta)$.

Theorem 5.11. *Let F, G, $H \in \mathbb{Z}[X]$ of respective degrees n, n and $2n$, and norm at most C. If $\log n = \mathcal{O}(\log C)$, we can check whether $H = FG$ with failure probability at most ϵ if $H \neq FG$, using $\mathcal{O}(n \log C \log_{n\log C} \frac{1}{\epsilon})$ bit operations, which is optimal when $\epsilon = 1/n^{\mathcal{O}(1)}$.*

Proof. As F and G have norm C and degree n, FG has norm at most nC^2. Let β be the first power of 2 greater than nC^2. Then $H = FG$ if and only if $H(\beta) = F(\beta)G(\beta)$.

The integers $F(\beta)$, $G(\beta)$ and $H(\beta)$ have bit length $\mathcal{O}(n \log \beta) = \mathcal{O}(n \log(nC)) = \mathcal{O}(n \log C)$ since $\log n = \mathcal{O}(\log C)$. As β is a large enough power of 2, the evaluation on β does not require any operation. Therefore all the cost comes from the verification of $F(\beta)G(\beta) = H(\beta)$. This is linear in the size of $F(\beta)$, $G(\beta)$ and $H(\beta)$ by Theorem 5.10, hence linear in $n \log C$.

To get the appropriate probability bound, we use $\mathcal{O}(\log_{n\log C} \frac{1}{\epsilon})$ round of this algorithm. This is supported by the fact that the probability bound in Theorem 5.10 is $1/s^{\mathcal{O}(1)}$. □

5.2 Quasi-linear sparse product verification

Given three sparse polynomials F, G and H in $\mathcal{R}[X]$, we want to assert that $H = FG$. As already mentioned, evaluating the polynomials at a random point α cannot yield a quasi-linear algorithm. Our approach is to take a random prime p and to verify the equality modulo $X^p - 1$ through modular product verification. This method is explicitly described in Algorithm 9 that works over any large enough integral domain $\mathcal{R}$. We further extend the description and the analysis of this algorithm for the specific cases $\mathcal{R} = \mathbb{Z}$ and $\mathcal{R} = \mathbb{F}_q$.

Algorithm 9 SPARSEVERIFICATION

Input: $H, F, G \in \mathcal{R}[X]$; $0 < \epsilon < 1$.
Output: True if $H = FG$, False with probability at least $1 - \epsilon$ otherwise.

1: Define $0 < \epsilon_1 < \frac{3}{10}$ and $0 < \epsilon_2 < 1$ such that $\frac{10\epsilon_1}{3} + (1 - \frac{10\epsilon_1}{3})\epsilon_2 \leq \epsilon$
2: $n \leftarrow \deg(H)$
3: **if** $\#H > \#F\#G$ or $n \neq \deg(F) + \deg(G)$ **then return** False
4: $\lambda \leftarrow \max(21, \frac{1}{\epsilon_1}(\#F\#G + \#H)\ln n)$
5: $p \leftarrow$ RANDOMPRIME$(\lambda, \frac{5\epsilon_1}{3})$
6: $(F_p, G_p, H_p) \leftarrow (F \bmod X^p - 1,\ G \bmod X^p - 1,\ H \bmod X^p - 1)$
7: **return** True if $H_p = (F_p G_p) \bmod X^p - 1$, False otherwise ▷ using Theorem 4.1 with probability ϵ_2

Theorem 5.12. *If $\mathcal{R}$ is an integral domain of size $\geq \frac{2}{\epsilon_1\epsilon_2}(\#F\#G + \#H)\ln(n)$, Algorithm 9 works as specified. Assuming that $n = \deg(H)$ and $T = \max(\#F, \#G, \#H)$, it requires $\mathcal{O}(T \log(\frac{1}{\epsilon} T \log n))$ operations in $\mathcal{R}$, and $\mathcal{O}(T \log n \log\log(\frac{1}{\epsilon} T \log n))$ bit operations plus $\mathcal{O}(\log \frac{1}{\epsilon} \log^3(\frac{1}{\epsilon} T \log n) \log^2 \log(\frac{1}{\epsilon} T \log n))$ bit operations to obtain a prime p.*

Proof. Step 3 dismisses two trivial mistakes and ensures that n is a bound on the degree of each polynomial.

If $H = FG$, the algorithm always returns True. Otherwise, there are two sources of failure. Either $X^p - 1$ divides $H - FG$. Since this polynomial has at most $\#H + \#F\#G$ terms, this failure occurs with probability at most $\frac{10\epsilon_1}{3}$ by Proposition 2.5. Or $X^p - 1$ does not divide $H - FG$ but the modular product verification fails. This occurs with probability at most ϵ_2. Altogether, the failure probability is at most $\frac{10\epsilon_1}{3} + (1 - \frac{10\epsilon_1}{3})\epsilon_2 \leq \epsilon$.

To analyse the complexity, we consider $\epsilon_1, \epsilon_2 \sim \epsilon$ (for example $\epsilon_1 = \frac{3\epsilon}{20}$ and $\epsilon_2 = \frac{\epsilon}{2}$). Let us remark that $p = \mathcal{O}(\frac{1}{\epsilon} T^2 \log n)$. To get the prime p, Step 5 requires only $\mathcal{O}(\log \frac{1}{\epsilon} \log^3 p \log^2 \log p)$ bit operations by Proposition 2.4. This gives the announced complexity once $\log p$ is replaced by $\mathcal{O}(\log(\frac{1}{\epsilon} T \log n))$.

The operations in Step 6 are T divisions by p on integers bounded by n. Their cost is $\mathcal{O}(T \frac{\log n}{\log p} \mathsf{I}(\log p)) = \mathcal{O}(T \log n \log\log p)$ bit operations, that is $\mathcal{O}(T \log n \log\log(\frac{1}{\epsilon} T \log n)))$, plus T additions in $\mathcal{R}$.

In Step 7, F_p, G_p and H_p have degree $p = \mathcal{O}(\frac{1}{\epsilon}T^2 \log n)$ and at most T monomials. They are still sparse and we can use the sparse version of Theorem 4.1 with $P = X^p - 1$. The verification of $H_p = F_p G_p \bmod X^p - 1$ thus requires $\mathcal{O}(T \log p) = \mathcal{O}(T \log(\frac{1}{\epsilon} T \log n))$ operations in $\mathscr{R}$. Other steps have negligible cost. □

To clarify the complexity, we will use the notation $\mathcal{O}_\epsilon(f(n))$ as a shortcut for $\mathcal{O}(f(n)\log^k \frac{1}{\epsilon})$ for some k. Using this notation, the complexity of Algorithm 9 becomes $\mathcal{O}_\epsilon(T \log(T \log n))$ operations in $\mathscr{R}$ plus $\mathcal{O}_\epsilon(T \log n \log\log(T \log n)))$ bit operations as getting the prime p is logarithmic in T and $\log n$.

The rest of the section is dedicated to the bit complexity analysis of this algorithm over integers or finite fields. Our goal is to have bit complexities that are as close as possible to linear. To ease the comparison with truly linear complexity, we express these bit complexities in terms of the total bit size s of the input. A degree-n polynomial with T monomials has bit size $s = \mathcal{O}(T(\log n + \log q))$ if it has coefficients in $\mathbb{F}_q$, and $s = \mathcal{O}(T(\log n + \log C))$ if it has coefficients in $\mathbb{Z}$ of absolute value at most C.

We first note that reducing the input polynomials modulo $X^p - 1$ at Step 6 is already non-linear. Indeed, we proved that this step has bit complexity $\mathcal{O}_\epsilon(T \log n \log\log(T \log n))$, which is $\mathcal{O}_\epsilon(s \log\log s)$. We shall prove that in some cases, this step is actually the dominant term in the complexity.

We begin with the analysis over the integers.

Corollary 5.13. *Let F, G and $H \in \mathbb{Z}[X]$ of degree at most n, with norm at most C and sparsity at most T. Then Algorithm 9*

has bit complexity $\mathcal{O}_\epsilon(s \log s \log\log s)$, where $s = T(\log n + \log C)$ is the input size.

Proof. The modification only concerns Step 7, where we use Theorem 4.4 for the modular product verification with

$P = X^p - 1$ and F_p, G_p, H_p that have sparsity T and norm TC. So this step costs

$$\mathcal{O}_\epsilon(T \log p |(\log(p \log C)) + T \log(TC) \log\log(p \log TC)).$$

Since $T \leq n$, $T \log p = \mathcal{O}_\epsilon(T \log n) = \mathcal{O}_\epsilon(s)$. And $\log(p \log C) = \mathcal{O}_\epsilon(\log(T \log n \log C)) = \mathcal{O}_\epsilon(\log(T \log n) + \log\log C) = \mathcal{O}_\epsilon(\log s)$. Thus the first term is $\mathcal{O}_\epsilon(s \log s \log\log s)$. Also, $T \log(TC) = \mathcal{O}(T \log nC) = \mathcal{O}(s)$.

As $\log(p \log TC) = \mathcal{O}_\epsilon(\log(T \log n) + \log\log C) = \mathcal{O}_\epsilon(\log S)$, the second term is $\mathcal{O}_\epsilon(s \log\log s)$.

Since Step 6 is unchanged and has bit complexity $\mathcal{O}_\epsilon(s \log\log s)$, the result follows. □

The complexity is actually better for very sparse polynomials.

Remark 5.14. *If F, G, $H \in \mathbb{Z}[X]$ of bit size s have sparsity at most $T = \Theta(\log^k n)$ for some k, Algorithm 9 has bit complexity $\mathcal{O}_\epsilon(s \log\log s)$.*

Proof. The input size is $s = \Theta(\log^{k+1} n + \log^k n \log C)$. In this case, $\log p = \mathcal{O}_\epsilon(\log\log n)$. In the previous proof, there is one dominant term of order $\mathcal{O}_\epsilon(s \log s \log\log s)$, while the other terms are already of order $\mathcal{O}_\epsilon(s \log\log s)$. It is sufficient to prove that with the new assumption, the dominant term is also $\mathcal{O}_\epsilon(s \log\log s)$.

The dominant term $\mathcal{O}_\epsilon(s \log s \log\log s)$ in the complexity comes from the term $\mathcal{O}_\epsilon(T \log p |(\log(p \log C)))$. Since $\log(p \log C) = \mathcal{O}_\epsilon(\log\log n + \log\log C)$, this dominant term becomes

$$\mathcal{O}_\epsilon(\log^k n \log\log n(\log\log n + \log\log C) \log(\log\log n + \log\log C)).$$

Note that $\log\log n$ and $\log\log C$ are both $\mathcal{O}(\log s)$, therefore this can be rewritten $\mathcal{O}_\epsilon(\log^k n \log^2 s \log\log s)$. Since $\log^k n = \mathcal{O}(s^{k/(k+1)})$, this yields $\mathcal{O}_\epsilon(s \log\log s)$. □

We now switch to polynomials over finite fields. There are more cases to consider, depending on the size of the field with respect to the degree and sparsity of the inputs. The first easy case is the case of large finite fields: If there are enough points for the evaluation, the generic algorithm keeps its guarantee of success while offering a quasi-linear bit complexity.

Corollary 5.15. *Let F, G and $H \in \mathbb{F}_q[X]$ of degree at most n and sparsity at most T where $q > \frac{2}{\epsilon_1 \epsilon_2}(\#F \#G + \#H) \ln n$. Then Algorithm 9 has bit complexity $\mathcal{O}_\epsilon(s \log^2(s))$ where $s = T(\log n + \log q)$ is the input size.*

Proof. It is still enough to analyse Step 7. Each ring operation in $\mathbb{F}_q$ costs $\mathcal{O}(\log(q) \log\log(q))$ bit operations which implies that the bit complexity of Step 7 is $\mathcal{O}_\epsilon(T \log(T \log n) \log(q) \log\log(q))$. Since both $T \log q$ and $T \log n$ are $\mathcal{O}(s)$ and $\log\log q = \mathcal{O}(\log s)$, the result follows. □

If the field is not large enough, we need to use some extension field. This slightly modifies the algorithm but actually yields a better complexity bound than for large finite fields. This is due to the fact that in that case, we choose an extension of the exact appropriate size. Note that the probability of success remains unchanged.

Corollary 5.16. *Let F, G and $H \in \mathbb{F}_q[X]$ of degree at most n and sparsity at most T where $q < \frac{2}{\epsilon_1\epsilon_2}(\#F\#G + \#H)\ln n$. Algorithm 9 has bit complexity $\mathcal{O}_\epsilon(s\log s\log\log s)$, where $s = T(\log n + \log q)$ is the input size.*

Proof. By Corollary 4.7 Step 7 requires $\mathcal{O}_\epsilon(T\log p\mathsf{M}_q(\log_q p) + (\log_q p)^2\mathsf{M}_q(\log_q p)(\log q + \log\log_q p))$ operations in $\mathbb{F}_q$. Since $\log p = \mathcal{O}_\epsilon(\log(T\log n)) = \mathcal{O}_\epsilon(\log s)$ and $\log q = \mathcal{O}_\epsilon(\log s)$ too, the second term is polylogarithmic in s. As $\log p = \mathcal{O}_\epsilon(\log(T\log n))$ the first term is $\mathcal{O}_\epsilon(T\log(T\log n)\mathsf{M}_q(\log_q(T\log n)))$. Since $\log(T\log n) = \mathcal{O}(\log n)$, $T\log(T\log n) = \mathcal{O}(s)$. Furthermore, $\log(T\log n) = \mathcal{O}(\log s)$ and the first term simplifies to $\mathcal{O}_\epsilon(s\mathsf{M}_q(\log_q s))$. Now $\mathsf{M}_q(\log_q s) = \mathcal{O}(\log s\log\log s)$. Altogether $T\log p\mathsf{M}_q(\log_q p) = \mathcal{O}_\epsilon(s\log s\log\log s)$. The result follows. □

Again, we note that for very sparse polynomials over some fields, the complexity is even better.

Remark 5.17. *Let F, G and $H \in \mathbb{F}_q[X]$ of degree at most n and sparsity at most T, where $q < \frac{2}{\epsilon_1\epsilon_2}(\#F\#G + \#H)\ln n$. The bit complexity of Algorithm 9 is*

(i) $\mathcal{O}_\epsilon(s\log s)$ *if* $\log_q(T\log n) = \mathcal{O}(1)$,

(ii) $\mathcal{O}_\epsilon(s\log\log s)$ *if* $T = \Theta(\log^k n)$ *for some constant* k.

Proof. The most significant term in the complexity is $\mathcal{O}_\epsilon(T\log(T\log n)\mathsf{M}_q(\log_q(T\log n))$. In the first case, it becomes $\mathcal{O}_\epsilon(T\log(T\log n)\mathsf{M}_q(1)) = \mathcal{O}_\epsilon(T\log(T\log n)\log q\log\log q)$. As $\log q = \mathcal{O}_\epsilon(\log\log n)$, $T\log q\log\log q = \mathcal{O}_\epsilon(s)$ and the complexity becomes $\mathcal{O}_\epsilon(s\log s)$. In the second case, the most significant term can be bounded by $\mathcal{O}_\epsilon(T\log^3(T\log n))$. But $T = \mathcal{O}(s^{k/(k+1)})$, and this most significant term becomes $\mathcal{O}_\epsilon(s)$ only. The global bit complexity is then dominated by Step 6 and is $\mathcal{O}_\epsilon(s\log\log s)$. □

To conclude, the bit complexity of Algorithm 9 over integers or finite fields range from $\mathcal{O}_\epsilon(s\log\log s)$ in the most favorable cases, to $\mathcal{O}_\epsilon(s\log^2 s)$ in more complicated situations. We note that in the best cases, the complexity is actually dominated by the cost of the modular reduction of the exponents of the input polynomials.

Remark 5.18. *Verification of a sparse product is always faster than computing the sparse product over $\mathbb{Z}$ or $\mathbb{F}_q$.*

Proof. Assuming $s = T(\log n + \log\zeta)$ to be the input size of the sparse polynomial F,G and H. Over $\mathbb{Z}$ we have $\log\zeta = \log C$ where C is the norm of the coefficients, while $\log\zeta = \log q$ when in $\mathbb{F}_q$. The best know result for computing the product FG needs $\mathcal{O}_\epsilon(s\log^2(s)\log^2(T)(\log T + \log\log s))$ bit operations [9]. Taking the worst case complexity for our verification yields a cost of $\mathcal{O}_\epsilon(s\log^2 s)$. This means that we are always faster by a factor $\mathcal{O}(\log^2(T)(\log T + \log\log s))$. Of course, for some small finite fields we are even beyond this value. □

References

[1] A. Arnold and D. S. Roche. Output-sensitive algorithms for sumset and sparse polynomial multiplication. In *ISSAC '15*, pages 29–36. ACM, 2015.

[2] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991.

[3] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *STOC*, pages 592–601. ACM, 2002.

[4] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19:297–301, 1965.

[5] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[6] R. A. Demillo and R. J. Lipton. A probabilistic remark on algebraic program testing. *Information Processing Letters*, 7(4):193 – 195, 1978.

[7] R. Freivalds. Fast probabilistic algorithms. In *Mathematical Foundations of Computer Science*, volume 74, pages 57–69. Springer Berlin Heidelberg, 1979.

[8] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra (third edition)*. Cambridge University Press, 2013.

[9] P. Giorgi, B. Grenet, and A. Perret du Cray. Essentially optimal sparse polynomial multiplication. In *Proceedings of the 2020 international symposium on symbolic and algebraic computation*, ISSAC, pages 202–209. ACM, 2020.

[10] P. Giorgi. A probabilistic algorithm for verifying polynomial middle product in linear time. *Information Processing Letters*, 139:30 – 34, 2018.

[11] S. W. Golomb. *Shift register sequences*. Aegean Park Press, 1982.

[12] D. Gries and G. Levin. Computing fibonacci numbers (and similarly defined functions) in log time. *Inf. Process. Lett.*, 11:68–69, 1980.

[13] D. Harvey and J. van der Hoeven. Faster polynomial multiplication over finite fields using cyclotomic coefficient rings. *Journal of Complexity*, 54:101404, 2019.

[14] D. Harvey and J. van der Hoeven. Integer multiplication in time $O(n \log n)$. To appear in *Ann. of Math.*, March 2019.

[15] D. Harvey and J. van der Hoeven. Polynomial multiplication over finite fields in time O(n log n). working paper or preprint, March 2019.

[16] J. van der Hoeven, R. Lebreton, and É. Schost. Structured FFT and TFT: Symmetric and Lattice Polynomials. In *ISSAC'13*, pages 355–362. ACM, 2013.

[17] J. van der Hoeven and G. Lecerf. On the Complexity of Multivariate Blockwise Polynomial Multiplication. In *ISSAC'12*, pages 211–218. ACM, 2012.

[18] J. van der Hoeven and G. Lecerf. On the bit-complexity of sparse polynomial and series multiplication. *J. Symb. Comput.*, 50:227–254, 2013.

[19] S. C. Johnson. Sparse polynomial arithmetic. *ACM SIGSAM Bulletin*, 8(3):63–71, 1974.

[20] M. Kaminski. A note on probabilistically verifying integer and polynomial products. *J. ACM*, 36(1):142–149, January 1989.

[21] F. Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303, New York, NY, USA, 2014. ACM.

[22] M. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *ISSAC*, page 263. ACM, 2009.

[23] M. Monagan and R. Pearce. Sparse polynomial division using a heap. *J. Symb. Comput.*, 46(7), 2011.

[24] G. L. Mullen and D. Panario. *Handbook of Finite Fields*. Chapman & Hall/CRC, 1st edition, 2013.

[25] V. Nakos. Nearly optimal sparse polynomial multiplication. *IEEE Transactions on Information Theory*, 66(11):7231–7236, 2020.

[26] D. S. Roche. Chunky and equal-spaced polynomial multiplication. *Journal of Symbolic Computation*, 46(7):791 – 806, 2011. doi:10.1016/j.jsc.2010.08.013.

[27] D. S. Roche. What can (and can't) we do with sparse polynomials? In *ISSAC*, 2018.

[28] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois J. Math.*, 6(1):64–94, 03 1962. http://projecteuclid.org/euclid.ijm/1255631807.

[29] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, October 1980.

[30] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, second edition, 2008.

[31] A. C. Yao. On the Evaluation of Powers. *SIAM Journal on Computing*, 5(1):100–103, 1976.

[32] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Symbolic and Algebraic Computation. In: Lecture Notes in Comput. Sci., vol. 72*, pages 216–226. Springer-Verlag, 1979.