PMNS for Cryptography : A Guided Tour

Nicolas Méloni, François Palma, Pascal Véron

Manuscript submitted to AIMS Journal

doi:

PMNS FOR CRYPTOGRAPHY : A GUIDED TOUR

Nicolas Méloni✉, François Palma✉, Pascal Véron✉

Laboratoire IMath, Université de Toulon
La Garde, France

Abstract. The Polynomial Modular Number System (PMNS) offers an alternative to the conventional binary multi-precision representation system for large integers. Its effectiveness has been demonstrated for various cryptosystems using prime field arithmetic [2, 4, 6], with prime sizes ranging from 256 to 736 bits. However, as the size of p increases, the relative performance of PMNS compared to standard arithmetic diminishes. Furthermore, the generation process of a PMNS has a worst-case complexity of $\mathcal{O}(2^n)$, where n denotes the number of symbols used to represent an integer modulo p in this representation system. In this paper, we present several alternatives and improvements to the construction and implementation processes of PMNS, which are tailored to the size of p.

1. **Introduction.** Modern cryptography relies heavily on finite field arithmetic and especially on modular arithmetic. The standard way to represent a field element is through a 2^k-ary positional number system. In this system, an element a of $\mathbb{Z}/p\mathbb{Z}$, where p is a prime, is represented as an $(n = \lfloor \log_2(p)/k \rfloor + 1)$-element array $(a_0, \ldots, a_{n-1})$ corresponding to the integer

$$a = \sum_{i=0}^{n-1} a_i 2^{ki}$$

with $0 \leqslant a_i < 2^k$ or $-2^{k-1} \leqslant a_i < 2^{k-1}$. All arithmetic operations are performed modulo p. Modular multiplication is the most investigated operation, as many cryptosystems rely heavily on it. It is typically performed by a standard integer multiplication followed by a modular reduction. Various approaches have been proposed to accelerate modular reduction modulo a prime number. Two main cases are usually considered, depending on whether the prime can be freely chosen or not. Mersenne primes, which are primes of the form $2^t - 1$, are a typical example of the former case. In that case, a modular reduction can be computed easily as a shift-and-add operation. Generalizations of Mersenne primes have been proposed to extend the range of possible candidates, but they are limited to a small number of primes [3]. In the latter case, when the cryptosystem does not allow for enough freedom to choose a generalized Mersenne number, more generic algorithms must be used.

The Polynomial Modular Number System (PMNS) was introduced by Bajard, Imbert, and Plantard as an alternative to the standard radix 2^k multi-precision representation, with the goal of speeding up modular arithmetic when special primes

Key words and phrases. Polynomial Modular Number System, Finite field, Modular arithmetic.

are not available [1]. In this system, a field element is represented as a polynomial of bounded degree, and field arithmetic is performed using polynomial arithmetic. Specifically, an element a of $\mathbb{Z}/p\mathbb{Z}$ is represented as a polynomial $A(X) = \sum_{i=0}^{n-1} a_i X^i \mod E(X)$, where E is a degree n polynomial and $|a_i| < \rho$ for some ρ.

A key advantage of PMNS is that, even if p has no special form, the polynomial E can usually be chosen such that polynomial reduction modulo $E(X)$ is fast (for example, by setting $E(X) = X^n - 1$). In this case, the modular reduction that occurs in the positional number system is replaced by a fast polynomial reduction (known as external reduction) followed by a coefficient reduction (known as internal reduction). However, the internal reduction operation becomes the critical point.

Previous studies have shown that the PMNS is highly competitive with state-of-the-art algorithms for integers ranging from 256 to 736 bits [2, 4, 6]. However, as the size of integers increases beyond 1024 bits, it becomes increasingly difficult to generate good parameters to construct a PMNS for larger integers. On top of that relative performance of PMNS compared to standard arithmetic decreases.

In this work we propose to address these issues with a two-fold approach. First, we generalize the Montgomery-like internal reduction method to allow for the generation of good parameters for larger primes. Second, we address the decrease in PMNS relative efficiency due to two phenomena.

First, as the size of p increases, the relative size of the PMNS data structure (i.e., the degree of the polynomials) increases faster than that of the radix 2^k standard representation leading to additional computations. To address this, we propose a method to improve the relative size of the PMNS data structure.

Second, most libraries switch to subquadratic multiplication algorithms (such as Karatsuba or Toom-Cook) beyond 2000 bits when PMNS uses quadratic matrix vector products. To address this, we show that PMNS can be performed faster by parallelizing the computation and taking advantage, under some conditions, of the Toeplitz matrix form.

The rest of the paper is organized as follows. Section 2 provides the fundamental knowledge required to understand PMNS. In Section 3, we present a generalized Montgomery algorithm to generate PMNS for large primes. In Section 4 we examine the reasons for the decreasing efficiency of PMNS as p grows and propose a practical solution for software implementation on 64- bit architecture, based on 128-bit arithmetic. In Section 5 two methods, based on parallelism and Toeplitz matrices, are proposed to improve the PMNS efficiency, which can be combined. In Section 6, we detail our software implementation and results.

2. **Mathematical Background.**

In this section, we will provide a brief overview of the necessary mathematical background related to PMNS.

2.1. **PMNS.**

Definition 2.1. Let $p \geqslant 3$, $n \geqslant 2$, $\gamma \in [1, p-1]$ and $\rho \in [1, p-1]$. Let $E \in \mathbb{Z}[X]$ a monic polynomial of degree n, such that $E(\gamma) \equiv 0 \pmod{p}$. A PMNS is a set $\mathcal{B} \subset \mathbb{Z}[X]$ such that :

1. $\forall A \in \mathcal{B}$, $\deg(A) < n$,
2. $\forall A(X) = \sum_{i=0}^{n-1} a_i X^i \in \mathcal{B}$, $-\rho < a_i < \rho$ for all i,

3. $\forall a \in \mathbb{Z}/p\mathbb{Z}$, $\exists A \in \mathcal{B}$ such that $A(\gamma) \equiv a \pmod{p}$.

With the PMNS, elements of a finite field are represented by polynomials with degree at most $n-1$ and bounded coefficients. Finite field arithmetic operations are then performed on these polynomials subject to additional constraints:

1. Given two field elements a and b, compute a polynomial $A(X)$ representing a and a polynomial $B(X)$ representing b in $\mathcal{B}$.
2. Compute $C(X) = A(X) \odot B(X) \bmod E(X)$, where $\odot$ denotes polynomial multiplication or polynomial addition, $E(X)$ is the external reduction polynomial, and mod denotes the polynomial remainder operation. This step is called *external reduction*.
3. Find a polynomial $\tilde{C}(X)$ such that $\deg C(X) = \deg \tilde{C}(X)$, $\tilde{C}(\gamma) \equiv C(\gamma) \bmod p$ and $|\tilde{c}_i| < \rho$ for all coefficients $\tilde{c}_i$ of $\tilde{C}(X)$. This step is called *internal reduction*.

These additional constraints ensure that the arithmetic operations produce results in the appropriate residue class modulo the prime p and with coefficients bounded by a given threshold ρ.

Step 3 is the most critical point from an efficiency standpoint. This is because step 1 is carried out only once during the entire cryptographic computation and is often precomputed offline. Step 2 involves polynomial arithmetic, which is a fast operation. Therefore, only step 3 requires special attention and is the main bottleneck in terms of performance.

2.2. **Montgomery Algorithm.** Let $\mathbb{Z}_{n-1}[X]$ be the set of polynomials of degree at most $n-1$ in $\mathbb{Z}[X]$. Let us consider the set $\mathfrak{L}$ given by:

$$\mathfrak{L} = \left\{ (x_0, \ldots, x_{n-1}) \in \mathbb{Z}^n : \sum_{i=0}^{n-1} x_i \gamma^i \equiv 0 \mod p \right\}.$$

It is an n-dimensional lattice that can be viewed as a subset of $\mathbb{Z}_{n-1}[X]$ which consists of polynomials that vanish at γ modulo p. In order to solve step 3, we can look for a polynomial $T(X) \in \mathfrak{L}$ such that $|C(X) - T(X)|_\infty < \rho$. We can then set $\tilde{C}(X) = C(X) - T(X)$. Finding such a polynomial is closely related to solving the closest vector problem in a lattice, which is known to be an NP-hard problem [12]. Therefore, the efficiency of constructing a PMNS for a given integer p mainly depends on the efficiency of the algorithm used to solve step 3.

To summarize, in order to generate a PMNS with efficient internal reduction for a given integer p, we need an efficient process to generate all the parameters of the PMNS that guarantee the existence of an efficient internal reduction process. The choice of parameters and the way they are defined depend mainly on the internal reduction algorithm used.

The classical algorithm used to perform step 3 is called the Montgomery-like reduction algorithm and is described in [11] (see Alg. 1). This algorithm depends on a parameter ϕ which for practical reasons is set to 2^k where k is the size of the hardware register being used, typically 2^{64}. The algorithm also mainly depends on the existence of a polynomial $M(X) \in \mathfrak{L}$ invertible modulo (E, ϕ). The existence and the effective construction of this polynomial has been widely studied in [5, 6]. It is proven in [6] that the output of Algorithm 1 is in $\mathcal{B}$ as soon as ρ and ϕ satisfy

$$\rho \geqslant 2\|\mathcal{M}\|_1 \quad \text{and} \quad \phi \geqslant 2w\rho\,,$$

Algorithm 1 Coefficients reduction [11]

Require: $\mathcal{B} = (p, n, \gamma, \rho, E)$ a PMNS, $V \in \mathbb{Z}_{n-1}[X]$, $M \in \mathbb{Z}_{n-1}[X]$ such that $M(\gamma) \equiv 0 \pmod{p}$, $\phi \in \mathbb{N} \setminus \{0\}$ and $M' = -M^{-1} \bmod (E, \phi)$.
Ensure: $S(\gamma) = V(\gamma)\phi^{-1} \pmod{p}$, with $S \in \mathbb{Z}_{n-1}[X]$
1: $Q \leftarrow V \times M' \bmod (E,\ \phi)$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow (V + T)/\phi$
4: return S

where

$$\mathcal{M} = \begin{pmatrix} m_0 & m_1 & \dots & m_{n-1} \\ \dots & \dots & \dots & \dots \\ \vdots & \vdots & & \vdots \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{matrix} \leftarrow M \\ \leftarrow X.M \bmod E \\ \\ \leftarrow X^{n-1}.M \bmod E \end{matrix} \tag{1}$$

and w is a constant depending on $E(X)$.

With that in mind, for a given prime p, in the context of a software implementation, the PMNS parameter generation process requires the following steps :

1. Choose $\phi = 2^k$ and $n > \frac{\log_2 p}{k}$.
2. Find an irreducible degree n polynomial $E(X)$ and γ a root of $E(X) \bmod p$.
3. Build the lattice $\mathfrak{L}$.
4. Apply LLL on $\mathfrak{L}$ to obtain a reduced basis $\mathcal{L}$.
5. Find a polynomial $M(X) \in \mathfrak{L}$ with small coefficients and invertible modulo (E, ϕ).
6. Set $\rho = 2\|\mathcal{M}\|_1$.
7. Check that $\phi \geqslant 2w\rho$.

Note that, once the parameters have been generated, they induced a minimum value for the parameter ρ. If it is larger than some expected value, or if step 7 fails, one must go through the whole process all over again from either step 1 or 2. As p grows in size, step 4 can become time-consuming but the LLL algorithm remains a polynomial time algorithm. However, step 5, which involves finding a polynomial $M(X)$ with small coefficients, is the most critical step in terms of efficiency whose worst case scenario complexity is exponential in the degree of $E(X)$ [6, Algorithm 8]. The following section is devoted to addressing that critical issue.

3. Generating PMNS for large primes.

The main challenge in the PMNS parameter generation process is that, in the worst-case scenario, step 5 requires an exhaustive search among a set of 2^n polynomials in the lattice $\mathfrak{L}$. On a 64-bit architecture, n is at least as large as $n_{\text{opt}} = \lfloor (\log_2 p)/64 \rfloor + 1$. For instance, for 256-bit primes p, this exhaustive search involves only 2^5 iterations and is thus feasible. However, for a 4096-bit integer, if we want to construct a PMNS with $n \geqslant 65$, an exhaustive search among a set of at least 2^{65} polynomials is required, which is extremely challenging and computationally expensive.

To overcome this issue, we propose a generalized version of the Montgomery-like algorithm that eliminates the need for any search.

3.1. Montgomery algorithm over lattices.

The Montgomery algorithm for integers can be described as follows [10]. Let $c \in \mathbb{Z}$, $m \in \mathbb{N}$ (m odd), such that $2^{k-1} \leqslant m < 2^k (= \phi)$. To compute a representative of c in $\mathbb{Z}/m\mathbb{Z}$, the algorithm

adds a multiple qm of m to c so that c and $c+qm$ are in the same equivalence class in $\mathbb{Z}/m\mathbb{Z}$. Now, q is chosen such that $c+qm$ is a multiple of ϕ. In other words, one computes q such that $c+qm \equiv 0 \bmod \phi$. From this, one can deduce that $q \equiv -m^{-1}c \bmod \phi$ and the algorithm outputs the value $(c+qm)/\phi$ which is a representative of $c\phi^{-1}$ in $\mathbb{Z}/m\mathbb{Z}$.

C. Negre and T. Plantard generalized this idea in [11] to design an internal reduction process on $\mathbb{Z}[X]$. The operation $c+qm$ becomes $C(X)+Q(X)M(X)$ and this implies that $M(X)$ must satisfy $M(\gamma) \equiv 0 \bmod p$ so that the output result be equal to $C(\gamma)/\phi$ in $\mathbb{Z}/p\mathbb{Z}$. Conditions on ϕ and ρ are given in [11] so that $\|(C(X)+Q(X)M(X) \bmod E(X))/\phi\|_\infty < \rho$.

In the Montgomery multiplication algorithm, adding qm to c means adding a representative of 0 in $\mathbb{Z}/m\mathbb{Z}$. In the Montgomery internal reduction algorithm, adding $Q(X)M(X) \bmod E(X)$ is equivalent to adding a polynomial which vanishes in $\gamma \bmod p$. We have seen in Subsection 2.2 that the set $\mathfrak{L}$ is an n-dimensional lattice. Any polynomial in $\mathfrak{L}$ can be computed using a basis of this lattice. The set of polynomials of the form $Q(X)M(X) \bmod E(X)$ is only a sublattice of this lattice. Thus, we propose to modify the computation of $C(X)+Q(X)M(X)$ by $C(X)+Z(X)$ where $Z(X) \in \mathfrak{L}$.

Let $L_0(X), \ldots, L_{n-1}(X)$ be a reduced basis of $\mathfrak{L}$, and let $\mathcal{L}$ be the matrix where the i^{th} row contains the coefficients of $L_{i-1}(X)$. Let $C=(C_0,\ldots,C_{n-1})$ be the coefficients of $C(X)$. We aim to find a vector $q \in \mathbb{Z}^n$ such that $C+q\mathcal{L} \equiv 0 \bmod \phi$. This equality gives $q = C(-\mathcal{L}^{-1}) \bmod \phi$. From [11, Definition 3], we have that $\det(\mathcal{L})=p$ which is coprime with ϕ for $\phi = 2^k$, hence $-\mathcal{L}^{-1} \bmod \phi$ always exists. Algorithm 2 sums up this new internal reduction. This algorithm does not need to precompute a specific polynomial $M(X)$ that might require an exhaustive search.

Algorithm 2 Coefficients reduction, new version

Require: $\mathcal{B} = (p, n, \gamma, \rho, E)$ a PMNS, $C \in \mathbb{Z}_{n-1}[X]$, $\mathcal{L}$ a reduced basis of $\mathfrak{L}$, $\phi \in \mathbb{N} \setminus \{0\}$ and $-\mathcal{L}^{-1} \bmod \phi$.
Ensure: $S(\gamma) \equiv C(\gamma)\phi^{-1} \bmod p$, with $S \in \mathbb{Z}_{n-1}[X]$
1: $q \leftarrow C(-\mathcal{L}^{-1}) \bmod \phi$
2: $S \leftarrow (C+q\mathcal{L})/\phi$
3: return S

Proposition 3.1. *Let $C(X)$ be a polynomial of degree at most $n-1$, if*

$$\|C\|_\infty \leqslant \phi(\rho - \|\mathcal{L}\|_1)$$

then the output S of Algorithm 2 (with C as input) is such that $\|S\|_\infty < \rho$ (i.e., $S \in \mathcal{B}$).

Proof. Step 1 of algorithm 2 ensures that $\|q\|_\infty < \phi$. Now,

$$\begin{array}{rcl} \|q\mathfrak{L}\|_\infty & = & \max_i |\sum_j q_j \mathcal{L}_{ji}| \\ & < & \phi \max_i \sum_j |\mathcal{L}_{ji}| \end{array}$$

Hence

$$\|q\mathfrak{L}\|_\infty < \phi \|\mathcal{L}\|_1 .$$

Now,

$$\|C + q\mathfrak{L}\|_\infty \leqslant \|C\|_\infty + \|q\mathfrak{L}\|_\infty < \|C\|_\infty + \phi\|\mathfrak{L}\|_1 \,.$$

Thus, as soon as

$$\|C\|_\infty + \phi\|\mathfrak{L}\|_1 \ < \rho\phi \,,$$

then

$$\|S\|_\infty < \rho \,.$$

□

Now, we give sufficient conditions on ϕ and ρ so that the polynomial $C(X)$ meets the requirements of proposition 3.1.

Proposition 3.2. *Let $A(X)$ and $B(X)$ be two elements of $\mathcal{B}$ and let $C(X) = A(X)B(X) \bmod E(X)$. Let $w > 0$ such that $\|C(X)\|_\infty \leqslant w\|A(X)\|_\infty\|B(X)\|_\infty$, if $\phi \geqslant 2w\rho$ and $\rho \geqslant 2\|\mathcal{L}\|_1$ then $\|C\|_\infty \leqslant \phi(\rho - \|\mathcal{L}\|_1)$.*

Proof. Since $A(X)$ and $B(X)$ belong to $\mathcal{B}$, then

$$\|C\|_\infty \leqslant w\rho^2 \,,$$

hence if

$$w\rho^2 < \phi\rho - \phi\|\mathcal{L}\|_1 \,,$$

then $C(X)$ will satisfy the constraint on its infinity norm. Now,

$$\begin{aligned} w\rho^2 < \phi\rho - \phi\|\mathcal{L}\|_1 \quad &\Leftrightarrow \quad \phi\|\mathcal{L}\|_1 + w\rho^2 < \phi\rho \\ &\Leftrightarrow \quad \tfrac{w\rho^2}{\phi} + \|\mathcal{L}\|_1 < \rho \,. \end{aligned}$$

Hence, if

$$\frac{w\rho^2}{\phi} \leqslant \frac{\rho}{2} \quad \text{and} \quad \|\mathcal{L}\|_1 \leqslant \frac{\rho}{2} \,,$$

we obtain the required result. □

Remark 1. A tight bound on the value w is given in [6, subsection 4.3]. It only depends on the shape of the polynomial $E(X)$.

Remark 2. For a software implementation, ϕ is often chosen equal to 2^{64}. Hence to satisfy the bound $\phi \geqslant 2w\rho$, it is better to choose w and ρ "small". The value w depends on the coefficients of $E(X)$, which explains why sparse polynomials with small coefficients are used to build a PMNS.

With our generalized Montgomery algorithm, the PMNS parameter generation process requires the following steps:

1. Let $n \geqslant n_{\text{opt}} (= \lfloor (\log_2 p)/64 \rfloor + 1)$.
2. Find an irreducible degree n polynomial $E(X)$ and γ a root of $E(X) \bmod p$.
3. Build the lattice $\mathfrak{L}$.
4. Apply LLL on $\mathfrak{L}$ to obtain a reduced basis $\mathcal{L}$.
5. Set $\rho = 2\|\mathcal{L}\|_1$ and $\phi = 2w\rho$ (proposition 3.2).

Let us notice once again that we do not need to perform the exhaustive search of a polynomial $M(X)$, which ensures that a PMNS can always be generated, whatever the prime p and the selected polynomial $E(X)$. Practical implementations usually set additional constraints on ρ and ϕ, typically $\rho < 2^{64}$ and $\phi = 2^{64}$. If those are not met, repeat the generation process either from step 1 or 2 with a different n or polynomial E.

Alg./size of p	256	512	1024
Montgomery	121	432	1709
This work	122	443	1728

TABLE 1. Number of cycles to compute a PMNS modular multiplication, gcc 12.1.0, i9-11900KF.

Remark 3. As p grows in size, the PMNS becomes less efficient compared to more standard approaches. Indeed, most software libraries switch to subquadratic multiplication algorithms when p is too large and, to generate good PMNS parameters, it is often necessary to chose n further away from the optimal value $n_{\text{opt}} = \lfloor (\log_2 p)/k \rfloor + 1$ (where k is the target architecture register size) inducing additional computational costs compared to standard representations. Throughout the remainder of this paper, we explore these challenges and present two approaches to address them.

Table 1 shows that, in practice, the performance of our internal reduction process is similar to that of the Montgomery-like reduction. The test protocol is detailed in Section 6.

4. PMNS 128-bit.

In the previous section, we have demonstrated that it is feasible to generate PMNS parameters without requiring an exhaustive search of exponential complexity. However, in this section, we show that even though the PMNS parameter generation process can be performed efficiently, the data structure of PMNS still grows at a faster rate than the standard binary representation. This results in an automatic loss in competitiveness of the PMNS arithmetic. We propose a new implementation of the PMNS algorithm that addresses the issue of the relative size of the PMNS data structure. This new implementation is based on 128-bit arithmetic and enables us to circumvent this issue.

4.1. Data structure size problem.

For a software implementation on a 64-bit architecture, the parameter ϕ is set to 2^{64}. At the end of the parameter generation process, if $2w\rho \geqslant 2^{64}$ then we increment n by 1 and repeat. This explains why, in practice, n moves away from n_{opt}, degrading performances. Hence, for any p, it is of importance that n be not "too far" from n_{opt}. Unfortunately, this condition is hard to meet. Indeed, from proposition 3.2, the parameters ρ and ϕ must satisfy the following bounds to guarantee the existence of the PMNS and the consistency of the internal reduction process :

$$\phi \geqslant 2w\rho \quad \text{and} \quad \rho \geqslant 2\|\mathcal{L}\|_1 \,. \tag{2}$$

A classical result in lattice theory states that the infinity-norm of a "short" vector in $\mathfrak{L}$ is about $p^{1/n}$. Hence, among the lines of the matrix $\mathcal{L}$, which is a reduced basis of $\mathfrak{L}$, there is a line L_i whose infinity norm is greater than $p^{1/n}$. Hence,

$$p^{1/n} \leqslant \|L_i\|_\infty \leqslant \sum_j |L_{ji}| \leqslant \|\mathcal{L}\|_1 \,.$$

From equation 2, we deduce that n must be chosen such that

$$p^{1/n} \leqslant \frac{\phi}{4w} \,.$$

In practice, as ϕ is equal to 2^{64}, the more p grows , the more it will be difficult to find an integer n near from n_{opt} which satisfies this inequality.

Proposition 4.1. *For any $\varepsilon > 0$, there exists an integer p such that*

$$p^{\frac{1}{n_{\text{opt}}+\varepsilon}} > \frac{\phi}{4w}\,.$$

Proof. Notice that since $w > 1$, $\frac{\phi}{4w} < 2^{62}$. Now,

$$n_{\text{opt}} + \varepsilon < \frac{\log_2 p}{64} + \varepsilon + 1 \Leftrightarrow \frac{\log_2 p}{n_{\text{opt}} + \varepsilon} > \frac{64 \log_2 p}{\log_2 p + 64\varepsilon + 64}\,.$$

Hence

$$p^{\frac{1}{n_{\text{opt}}+\varepsilon}} > (2^{64})^{\frac{\log_2 p}{\log_2 p + 64\varepsilon + 64}}\,.$$

As

$$\lim_{p\to\infty} \frac{\log_2 p}{\log_2 p + 64\varepsilon + 64} = 1\,,$$

there still exists an integer p such that

$$p^{\frac{1}{n_{\text{opt}}+\varepsilon}} > 2^{62}\,.$$

□

This proves that as p grows, we must choose a n far away from n_{opt}.
As an example, let us suppose that we aim to build a PMNS for a 2048-bit integer, then $n_{\text{opt}} = 33$. On one hand, we have

$$p^{1/n_{\text{opt}}} \simeq 2^{62.06}\,,$$

and on the other hand, since $w \geqslant n$ (from the definition given in [6, subection 4.3]),

$$\frac{2^{64}}{4w} \simeq 2^{56.96}\,.$$

To satisfy the required constraint, we have to choose $n \geqslant 37$. The global performances of the PMNS suffer in consequence.

4.2. **128-bit PMNS vs 64-bit.** We have shown that the parameters must satisfy $p^{1/n} \leqslant \frac{\phi}{4w}$. Practically speaking, for a given p, there are thus two way to meet this condition: either increase n or ϕ. The usual approach is to set $\phi = 2^{64}$ so that one must increase n. The reason why ϕ is set that way is that the reduction algorithm involves divisions by ϕ that can be costly for a random value and virtually free when $\phi = 2^k$ where k matches the register size of the target device. Hence, rather than increasing n, an alternative is thus to choose $\phi = 2^{128}$ instead of 2^{64}. Doing so still guarantees fast divisions and allows to set much lower value for n. As a consequence, the software implementation of the PMNS has to be upgraded in order to deal with 128 and 256-bit integer operations on polynomial coefficients.

Let n_{64} (resp. n_{128}), the actual value of n for $\phi = 2^{64}$ (resp. for $\phi = 2^{128}$), Table 2 shows how n moves away from n_{opt} depending on the size of the prime p. When $\phi = 2^{64}$, for integers whose size exceeds 1024 bits, the value of n_{64} will certainly give rise to a PMNS multiplication process whose complexity will be higher than the classical multiplication algorithms when applied to the same operands split into n_{opt} 64-bit blocks (as it is done in various multi-precision libraries). For $\phi = 2^{128}$, the gap between n_{128} and n_{opt} remains negligible and justifies the use of this value for ϕ. From a practical point of view, we can observe that $n_{128} <$

size of p	256	512	1024	2048	4096	8192
n_{opt64}	5	9	17	33	65	129
n_{64}	5	9	19	40	83	187
n_{opt128}	3	5	9	17	33	65
n_{128}	3	5	9	18	36	72

TABLE 2. Optimal polynomial degrees vs practical ones used in implementation per size of prime considered.

size of p	256	512	1024	2048	4096	8192
Red-64	122	443	1709	7690	33471	169070
Red-128	245	720	2367	10920	39446	**156310**

TABLE 3. Comparative table of performances for Red-64 and Red-128 in number of processor cycles for one modular multiplication on Intel processor i9-11900KF with gcc 12.1.0.

$2n_{64}$. Hence performances should be better using $\phi = 2^{128}$, but this has to be tempered considering the cost of a software implementation of 128-bit arithmetic. The reference implementation with $\phi = 2^{64}$ makes use of internal compiler `__int128` registers which are 128-bit integer words handled directly by the compiler. This allows the algorithm to completely forgo any multi-precision considerations such as dealing with carries and leave it all to the compiler to be done internally in assembly. No such equivalent 256-bit register exists at the current time so, for $\phi = 2^{128}$, multi-precision algorithms have to be used which lead to slower calculations mainly due to carry propagation. Each 128-bit coefficient will be stored on two 64-bit registers and arithmetic on this coefficient has to consider this representation.

Both Karatsuba and the Schoolbook version of the multiplication of two elements were evaluated and the Schoolbook version has been found to be slightly faster so far. This means that, for $\phi = 2^{128}$, any coefficient multiplication involves 4 distinct multiplication operations on 64-bit integers. Our internal reduction process involves two vector-matrix products. For $\phi = 2^{64}$, each operation involves n_{64}^2 multiplications between 64-bit variables, whereas for $\phi = 2^{128}$ it involves $4n_{128}^2$ between 64-bit variables. For 1024-bit primes and beyond, since $n_{64} > 2n_{128}$, it means that for a given prime p, the 64-bit version of PMNS should be slower than its 128-bit counterpart. Unfortunately, the 128-bit version has additional carry operations dragging down the complexity function and giving us worse results until higher values of p widen the gap substantially between n_{64} and n_{128}. Table 3 shows that for primes larger than 8000 bits, reducing the size of the data structure leads to faster implementation as long as quadratic algorithms are used. For the sequel of the paper, we will note :

- **Red-64** : the 64-bit version of our internal reduction algorithm,
- **Red-128** : the 128-bit version of our internal reduction algorithm.

5. **Faster PMNS for large primes.** As the size of the prime p exceeds 2000 bits, modular multiplication implementations often switch from classical schoolbook algorithms to subquadratic alternatives like Karatsuba or Toom-Cook. A first approach

to lower our execution time is to optimize the computation of $A(X)B(X) \bmod E(X)$ using Toeplitz matrices.

5.1. **Toeplitz matrix version.** When $E(X) = X^n - \lambda$, we can optimize the computation of $A(X)B(X) \bmod E(X)$ using Toeplitz matrices. Let $A(X)$ and $B(X)$ be two elements of a PMNS $\mathfrak{B}$, to compute a representative of $A(X)B(X)$ in $\mathfrak{B}$, we first have to compute $A(X)B(X) \bmod E(X)$. We could use Karatsuba algorithm to compute the product of two polynomials and then reduce the result. It appears that this operation can be more efficient if seen as a vector matrix product involving Toeplitz matrices.

5.2. **Toeplitz vs Karatsuba.**

Definition 5.1. An $n \times n$ matrix is a matrix defined by $2n-1$ values $a_0, a_1, \ldots, a_{2n-2}$ such that

$$\begin{pmatrix} a_0 & a_1 & a_2 & \cdots & \cdots & a_{n-1} \\ a_n & a_0 & a_1 & \ddots & & \vdots \\ a_{n+1} & a_n & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_1 & a_2 \\ \vdots & & \ddots & a_n & a_0 & a_1 \\ a_{2n-2} & \cdots & \cdots & a_{n+1} & a_n & a_0 \end{pmatrix}$$

There is a natural link between Toeplitz matrices and modular polynomial multiplication when the reduction polynomial $E(X)$ is equal to $X^n - \lambda$. As an example let us consider $A(X) = a_0 + a_1X + a_2X^2 + a_3X^3$ and $B(X) = b_0 + b_1X + b_2X^2 + b_3X^3$, and let $C(X) = A(X)B(X) \bmod X^4 - \lambda$. A straight forward computation gives

$$\begin{aligned} C(X) \quad = \quad & a_3b_0X^3 + a_2b_1X^3 + a_1b_2X^3 + a_0b_3X^3 + \\ & \lambda a_3b_3X^2 + a_2b_0X^2 + a_1b_1X^2 + a_0b_2X^2 + \\ & \lambda a_3b_2X + \lambda a_2b_3X + a_1b_0X + a_0b_1X + \\ & \lambda a_3b_1 + \lambda a_2b_2 + \lambda a_1b_3 + a_0b_0\,. \end{aligned}$$

This last result can be computed as

$$\begin{pmatrix} b_0 & b_1 & b_2 & b_3 \end{pmatrix} \times \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ \lambda a_3 & a_0 & a_1 & a_2 \\ \lambda a_2 & \lambda a_3 & a_0 & a_1 \\ \lambda a_1 & \lambda a_2 & \lambda a_3 & a_0 \end{pmatrix}$$

The matrix used in this formula is a Toeplitz matrix defined by the values a_0, a_1, a_2, a_3, λa_3, λa_2 and λa_1. Now, a classical result states that the vector matrix product in the Toeplitz context can be efficiently computed using the following proposition [7].

Proposition 5.1. *Let*

$$A = \begin{pmatrix} A_0 & A_1 \\ A_2 & A_0 \end{pmatrix}$$

an $n \times n$ Toeplitz matrix where each submatrix is a $n/2$ square matrix. Let $B = \begin{pmatrix} B_0 & B_1 \end{pmatrix}$ a vector of size n where each B_i is of size $n/2$. The product $B \times A$ can be computed as $(P_0 - P_2, P_0 - P_1)$, where

$$\begin{aligned} P_0 \quad &= \quad (B_0 + B_1)A_0 \\ P_1 \quad &= \quad B_1(A_0 - A_2) \\ P_2 \quad &= \quad B_0(A_0 - A_1)\,. \end{aligned}$$

In [7] the authors show that this approach can be used recursively to obtain a complexity of $\mathcal{O}(n^{\log_2 3})$ similar to that of Karatsuba. Although the two complexities are asymptotically the same, in the next subsection we show that the Toeplitz approach is less costly in practice.

Proposition 5.2. *Let $A(X)$ and $B(X)$ be two polynomials of degree at most $n-1$ and let $E(X) = X^n - \lambda$. Let $T(n)$ be the execution time needed to compute $A(X)B(X) \bmod E(X)$, then, on a 64-bit architecture :*

- $T(n) = 3T(n/2) + nA_{64} + 4nA_{128} - 4A_{128}$, *using Karatsuba method,*
- $T(n) = 3T(n/2) + (\frac{5n}{2} - 2)A_{64} + nA_{128}$, *using Toeplitz method.*

where A_{128} (resp. A_{64}) is the execution time of a 128-bit (resp. 64-bit) addition.

Proof. Let $A(X) = A_H(X)X^{n/2} + A_L(X)$ and $B(X) = B_H(X)X^{n/2} + B_L(X)$ where the degree of $A_H(X), A_L(X), B_H(X)$ and $B_L(X)$ is at most $\frac{n}{2} - 1$, then

$$A(X)B(X) = A_H(X)B_H(X)X^n + N(X)X^{n/2} + A_L(X)B_L(X)$$

with

$$N(X) = (A_H(X) + A_L(X))(B_H(X) + B_L(X)) - A_H(X)B_H(X) - A_L(X)B_L(X)\,.$$

To reduce modulo $E(X)$ the product $A(X)B(X)$, let

- $A_HB_H(X) = A_HB_H^{(H)}(X)X^{n/2} + A_HB_H^{(L)}(X)$, with $\deg A_HB_H^{(H)}(X) \leqslant \frac{n}{2} - 2$ and $\deg A_HB_H^{(L)}(X) \leqslant \frac{n}{2} - 1$,
- $A_LB_L(X) = A_LB_L^{(H)}(X)X^{n/2} + A_LB_L^{(L)}(X)$, with $\deg A_LB_L^{(H)}(X) \leqslant \frac{n}{2} - 2$ and $\deg A_LB_L^{(L)}(X) \leqslant \frac{n}{2} - 1$,
- $N(X) = N_H(X)X^{n/2} + N_L(X)$, with $\deg N_H(X) \leqslant \frac{n}{2} - 2$ and $\deg N_L(X) \leqslant \frac{n}{2} - 1$.

We thus obtain

$$\begin{aligned} A(X)B(X) \bmod E(X) \quad = \quad & A_LB_L^{(L)}(X) + \lambda N_H(X) + \lambda A_HB_H^{(L)}(X) + \\ & (N_L(X) + A_HB_H^{(L)}(X) + \lambda A_HB_H^{(H)}(X))X^{n/2}\,. \end{aligned}$$

For the sequel of this analysis, let us consider the case where $A(X)$ and $B(X)$ are two elements of a PMNS, meaning that their coefficients are strictly less than ρ with $\rho < 2^{63}$ on a 64-bit architecture (see equation 2). Then one step of the Karatsuba algorithm needs to compute :

- $A_H(X) + A_L(X)$ and $B_H(X) + B_L(X)$, which involves at most $\frac{n}{2}$ additions between 64-bit words,
- a recursive call to compute the three products in $N(X)$,
- $2(n-1)$ additions on 128-bit words to compute $N(X)$,
- $\frac{n}{2} - 1 + \frac{n}{2}$ additions on 128-bit words to compute $A_LB_L^{(L)}(X) + \lambda N_H(X) + \lambda A_HB_H^{(L)}(X)$,
- $\frac{n}{2} + \frac{n}{2} - 1$ additions on 128-bit words to compute $N_L(X) + A_HB_H^{(L)}(X) + \lambda A_HB_H^{(H)}(X)$.

which gives a total cost of n 64-bit additions, $4n-4$ 128-bit additions and 3 recursive calls.

Now, concerning the Toeplitz approach, one step needs to compute :

- $\frac{n}{2}$ additions on 64-bit words for $B_0 + B_1$,
- $2\frac{n}{2} - 1$ additions on 64-bit words for $A_0 - A_2$ (resp. $A_0 - A_1$) since each one is a $\frac{n}{2} \times \frac{n}{2}$ Toeplitz matrix depending only on $2\frac{n}{2} - 1$ coefficients,

Size	1024	2048	4096	8192
GnuMP				
n	16	32	64	128
Low level	1715	5501	16897	53042
Classical Mont.	1534	4653	14522	44587
Mont. CIOS	1187	4661	16793	61478
This work				
n	19	40	84	187
Red-64	1709	7690	33471	169070
n	-	-	-	72
Red-128	-	-	-	156310

TABLE 4. Number of cycles to compute a modular multiplication using GnuMP and PMNS on Intel processor i9-11900KF with gcc 12.1.0.

- a recursive call to compute the three vectors P_0, P_1 and P_2,
- $2\frac{n}{2}$ additions on 128-bit words to compute $P_0 - P_2$ and $P_0 - P_1$,

which gives a total cost of $\frac{n}{2} + 2n - 2$ 64-bit additions, n 128-bit additions and 3 recursive calls.
Now we have to evaluate the complexity of the last step of each method. For Karatsuba , the last step is a product of two 64-bit integers which has to be done using the `__int128` `gcc` type.
The same observation can be done for Toeplitz method. Hence, the difference between the two methods relies on the splitting and reconstruction process. The Toeplitz method has a lower complexity for these two steps. □

Remark 4. There exists other ways to split the initial matrix [8]. For example one can use a 3-way split or 5-way split depending on the initial size of the matrix, and one can mix them during the recursive process. These multi-way splittings all lead to subquadratic algorithms.

Even if, using Toeplitz matrices, computing $A(X)B(X) \bmod E(X)$ becomes subquadratic, the two other steps of our internal reduction algorithm (Algorithm 2) remain quadratic. We thus propose to take into account the parallel nature of PMNS to optimize these last two steps.

5.3. **Parallel PMNS (PPMNS).** Proposition 3.2 guarantees that for any prime p, we can build a PMNS with a suitable internal reduction process. Unfortunately, for prime p whose size exceeds 1024, PMNS performances are not competitive with GnuMP library if no parallelism is used. For 1024-bit primes our performances are similar to low level GnuMP functions, but for higher sizes, our performances are worse (see Table 4, where n stands for the number of 64-bits block – or 128-bit blocks for Red-128 – used to represent an integer). It is all the more obvious when taking into account the two undocumented implementations of the classical Montgomery multiplication on integers (respectively named Classic Montgomery and Montgomery CIOS [9] in Table 4). The PMNS multiplication process we propose, involves three vector-matrix products whose complexity is quadratic in n, while GnuMP (or OpenSSL) switches to algorithms whose complexity is subquadratic in

n_{opt}, when n_{opt} is larger than some threshold computed at compile time. More precisely, to compute $ab \mod p$, first GnuMP low level functions computes ab using subquadratic algorithm depending on the size of a and b, then the result is reduced modulo p using a division algorithm depending on the size of the operands. The more n moves away from n_{opt}, the worse the PMNS performances get, compared to a classical multi-precision library. Now, as the operations in PMNS can be parallelized, one can hope that on a multi-core processor, the PMNS will give better performances despite the high difference between n and n_{opt}. Let us denote by $M(n)$ the computational cost of ab, $R(n)$ that of the modular reduction and $VM(n)$ that of the vector-matrix product. On a δ-core processor, if $VM(n)/\delta < M(n_{\text{opt}}) + R(n_{\text{opt}})$, then the PPMNS will perform better than any classical multi-precision library. In section 6, we provide a proof of concept targeting 8192-bit integers.

6. **Implementation and results.** All the results given in this paper have been computed using the following procedure :

- the *Turbo-Boost*® is deactivated during the tests;
- 1000 runs are executed in order to "heat" the cache memory, i.e. we ensure that the cache memory (data and instruction) is in an enough stabilized state in order to avoid untimely cache faults;
- one generates 500 random data sets, and for each data set the execution clock cycle numbers over a batch of 1000 runs is recorded;
- the performance is the median value.

All the source code is available on Github[1].

6.1. **A proof of concept for 8192-bit integers.** From the preceding results, it appears that a suitable implementation of a PMNS depends on the size of the parameter p. For integers used in elliptic curve cryptography (from 256 bits to 1024 bits), the internal reduction we proposed in section 3.1 guarantees that for any prime p, one can always build a 64-bit version of a PMNS with an efficient reduction process which outperforms low level GnuMP functions and is competitive with the undocumented Montgomery multiplication functions (until 512 bits). Moreover any polynomial $E(X)$ can be used to build the PMNS which enlarges the set of PMNS linked to a prime p.
For larger sizes, our internal reduction process is not competitive with subquadratic algorithms used by GnuMP, hence we have to consider the parallel version of PMNS (PPMNS). A major advantage is that, despite the increasing value of n, we can keep $\phi = 2^{64}$ and the corresponding source code only relies on processor 64-bit arithmetic instead of a costly software 128-bit arithmetic. For any polynomial $E(X)$, the multiplication process in the PMNS will involve three vector-matrix product which are well suited for parallel implementation. As a proof of concept, we consider in this section 8192-bit integers. Depending on the size of the operands a and b , the GnuMP library switches to different algorithms in order to compute ab. The thresholds to select the multiplication algorithm is chosen when compiling the source code of the library. On a Skylake processor, for GnuMP 6.2.1, the thresholds for sizes 1664 up to 13248 are (from `mpn/x86_64/skylake/gmp-mparam.h`):

```
#define MUL_TOOM22_THRESHOLD              26
#define MUL_TOOM33_THRESHOLD              73
#define MUL_TOOM44_THRESHOLD             208
```

[1] https://github.com/francoispalma/PMNS/

```
...
#define DC_DIV_QR_THRESHOLD                    55
```

This means that up to $25 \times 64 = 1600$-bit integers, GnuMP uses the classical schoolbook algorithm to compute the product ab. For size in the range 1664 to 4608, GnuMP uses Toom22 which is similar to Karatsuba algorithm. Finally it switches to the classical 3-way split Toom-Cook algorithm up to 13248-bit integers. For 8192-bit integers, we can obtain a rough estimation of the number of multiplication done by GnuMP. Such integers are split into 128 64-bit blocks. Then the product of two such sequences is done using 3-way split Toom-Cook and involves 5 multiplications between 43 64-bit blocks. Each multiplication is done using a Karatsuba-like method which in turn requests 3 multiplications on 22 64-bit blocks, each one being computed using the classical schoolbook algorithm. Hence, the total process involves $22^2 \times 3 \times 5$ multiplications on 64-bit integers, leading to 7260 multiplications. Notice that this is a very optimistic estimation as we do not take into account all the pre and post processing of the Toom-Cook and Karatsuba algorithms. Next to reduce modulo p, a divide and conquer algorithm is used for integers whose size is greater than $55 \times 64 = 3520$ bits[2]. As mentioned in GnuMP documentation, for 8192-bit integers, the complexity is about $2.63M(8192)$ where $M(8192)$ is the cost of the product of two 8192-bit integers. Such a product will be computed using a 3-way split Toom-Cook step followed by a Karatsuba step which will boil down to a total of 15 schoolbook multiplications on 1350-bit integers, that is to say about 19000 64-bit multiplications. To sum up to compute $ab \bmod p$ at least 26260 64-bit multiplications are performed.

Now, for 8192-bit integers, one can build a PMNS with $n = 187$. For such a value, if $E(X)$ has no particular shape, the multiplication process will exactly involve $3 \times 187 \times 187 = 104907$ 64-bit multiplications. On a δ-core processor, the parallelized version of the PMNS will perform better if $104907/\delta \leqslant 26260$, which gives $\delta \geqslant 4$. In practice, we observe that on a 5-core processor the PPMNS gives better performances than low-level GnuMP functions and , on a 8-core processor, we beat the undocumented Montgomery modular multiplication functions (see Table 5).

5-core	6-core	8-core	Low level	Classical Mont.	Mont. CIOS
n=187			n=128		
51979	**46224**	**40778**	53042	44567	61478

TABLE 5. Number of cycles to compute a modular multiplication between two 8192-bit integers using GnuMP and PPMNS (for any $E(X)$) on Intel processor i9-11900KF with gcc 12.1.0.

When $E(X) = X^n - \lambda$, the first vector-matrix product can be done using a Toeplitz decomposition if n is a multiple of 2 or 3. Even if $n = 187$ is the smallest value that can be used to build a PMNS for a 8192-bit integer, we consider here $n = 189 = 3 \times 63$. This way, we can use the 3-way split decomposition which involves 6 vector-matrix products between vectors of size 63 and 63×63 matrices and perform them using a classical vector-matrix product since experimental results show that there is no point in splitting further. The next two steps are done using classical vector-matrix product. Hence, on a 6-core processor, the total number of 64-bit multiplications done by one core will be $63^2 + \frac{2 \times 189^2}{6} = 15876$ which is less

[2] https://gmplib.org/manual/Divide-and-Conquer-Division

than the number of multiplications done by GnuMP. Table 6 gives the results we obtained for the polynomial $E(X) = X^{189} - 2$.

6-core	Low level	Classical Mont.	Mont. CIOS
n=189		n=128	
42510	53042	44567	61478

TABLE 6. Number of cycles to compute a modular multiplication between two 8192-bit integers using GnuMP and PPMNS (for $E(X) = X^{189}-2$) on an Intel processor i9-11900KF with gcc 12.1.0.

The results we obtained show that the parallelized version of PMNS is the best way to manage large integers. When the polynomial $E(X)$ is arbitrary, the vector-matrix product is well suited for parallelization since the use of δ-core leads to approximately a gain factor of δ. When $E(X) = X^n - \lambda$ and $n = 0 \bmod 3$, one can take advantage of the 3-way split Toeplitz decomposition on a 6-core processor.
For integer ranging from 2048-bit up to 8192-bit, our experimental results show that the PPMNS does not perform faster than GnuMP functions (see Table 7). For 2048-bit and 4096-bit integers we ran several benchmarks using from 1 core to 8 cores. The best performances are summarized in Table 7. For 2048-bit integers, the size of the matrices are too small to benefit from multi-threading. For 4096-bit integers, the performances of our 64-bit version using 6 cores are close to those of low level GnuMP functions. In any case, it appears that within this size range, PMNS are not well suited.

6.2. **Toeplitz and internal Montgomery reduction.** Even if there is no guarantee that a suitable polynomial $M(X)$ can be computed in reasonable time when $E(X) = X^n - \lambda$ with λ odd, there is a huge advantage to consider again the Montgomery-like internal reduction. Indeed, steps 1 and 2 of Algorithm 1 involve polynomial modular reductions. As mentioned in the previous section, such an operation leads to a vector-matrix product with a Toeplitz matrix. Hence, all the internal reduction process can be performed using Toeplitz matrices. There are two alternatives to perform the vector-matrix product :

- either using the recursive splitting detailed in Subsection 5.2,

Method/Size	2048	4096
n	32	64
Low level	5501	16897
Classical Mont.	4653	14524
Mont. CIOS	4607	16793
n	40	84
Red-64	7491 (1-core) 10165 (2-core)	36195 (1-core) 17756 (6-core)
Red-128	10920 (1-core) 15314 (2-core)	39446 (1-core) 25797 (4-core)

TABLE 7. Number of cycles to compute a modular multiplication for 2048-bit and 4096-bit integers using GnuMP, PMNS and PPMNS on an Intel processor i9-11900KF with gcc 12.1.0.

Size	2048	4096	8192
8 cores			
Toeplitz Montgomery-like	14421	18193	37026
This work	14529	22782	40778

TABLE 8. Number of cycles to compute a modular multiplication for 2048,4096 and 8192-bit integers using Toeplitz form vs random matrix form on Intel processor i9-11900KF with gcc 12.1.0.

Size	1024	1664	2048
Low level	1715	4039	5501
Classical Mont.	1539	3466	4653
Montgomery CIOS	1187	3153	4607
Toeplitz (this work)	1449	3454	5489

TABLE 9. Number of cycles to compute a modular multiplication for 1024, 1664 and 2048-bit integers using Toeplitz form vs GnuMP on Intel processor i9-11900KF with gcc 12.1.0.

- or using a classical vector-matrix product algorithm taking into account that a Toeplitz matrix is defined by $2n-1$ coefficients instead of n^2 and is thus stored in a 1-dimensional array.

Our experimental results show that the parallel version of the Montgomery-like reduction algorithm using the Toeplitz recursive splitting performs worse than the parallel version of the Montgomery-like reduction algorithm using the classical vector-matrix product. For 8192-bit integers, if the polynomial $M(X)$ exists and can be computed in reasonable time, we obtain better performances than our internal reduction process (see Table 8).

For integers ranging from 1024 to 2048 bits, we have shown that there is no point in using the PPMNS. Within this range, the Toeplitz recursive splitting on a single core outperforms some GnuMP functions as summarized in Table 9.

Concerning the polynomial $M(X)$, if $E(X) = X^n - \lambda$ with λ odd, we exhibit in the next proposition, particular cases where it can be computed in linear time.

Proposition 6.1. *Let n an odd integer such that $1+X+\cdots+X^{n-1}$ is irreducible in $\mathbb{F}_2[X]$, then one can compute in linear time a polynomial $M(X)$ invertible modulo (E, ϕ).*

Proof. From [6, corollary 2 and proposition 5], the polynomial $M(X)$ exists iff the determinant of the matrix $\mathcal{M}$ is odd, where

$$\mathcal{M} = \begin{pmatrix} m_0 & m_1 & \dots & m_{n-1} \\ \cdots & \cdots & \cdots & \cdots \\ \vdots & \vdots & & \vdots \\ \cdots & \cdots & \cdots & \cdots \end{pmatrix} \begin{matrix} \leftarrow M \\ \leftarrow X.M \bmod E \\ \\ \leftarrow X^{n-1}.M \bmod E \end{matrix} \tag{3}$$

Equivalently, this means that this matrix is invertible over $\mathbb{F}_2$. Now, as $E(X) = X^n+1$ over $\mathbb{F}_2$, the matrix $\mathcal{M}$ is a circulant one and there is an isomorphism between

the set of circulant matrices and the algebra of polynomials modulo X^n+1. Hence $\mathcal{M}$ is invertible iff $M(X)$ is coprime with X^n+1 in $\mathbb{F}_2[X]$.

Let L the matrix whose rows are a reduced basis of the set of polynomials that vanishes in $\gamma \bmod p$. We know that $\det(L) = p$, so L is invertible in $\mathbb{F}_2$. Hence, there exists a line in L which has an odd number of odd coefficients. Otherwise, in $\mathbb{F}_2$, all the lines of the matrix will contain an even number of ones, which implies that the all one vector will be in the kernel of the linear application defined by L, meaning that L is not invertible.

Let n such that $O(X) = 1 + X + \cdots + X^{n-1}$ is irreducible over $\mathbb{F}_2$ (which implies n odd), then x^n+1 is the product of two irreducible polynomials, $X+1$ and $O(X)$. Let $M(X)$ the polynomial which contains an odd number of odd coefficients, if $M(X)$ is coprime with $O(X)$ in $\mathbb{F}_2[X]$, then $M(X)$ is coprime with $E(X)$ since it does not vanish in 1. Otherwise, it means that $M(X) = O(X)$. In this case, there are only two alternatives:

- either there exists in the matrix L another line with an odd number of odd coefficients, then the corresponding polynomial $\bar{M}(X)$ cannot be equal to $O(X)$, and thus $\bar{M}(X)$ is the polynomial we are searching for,
- or all the other lines of L have an even number δ of odd coefficients. Notice that since $\det(L) = 1 \bmod 2$, then $\delta > 0$, otherwise L will contain a line with only even coefficients which would give a null vector in $\mathbb{F}_2$. Let L_i any line with δ odd coefficients, add the corresponding polynomial to $O(X)$ to obtain a polynomial $M(X)$ with an odd number of odd coefficients coprime with $O(X)$.

□

If we consider that until $n = 40$ an exhaustive search is reasonable to find $M(X)$, here are some $n > 40$ which meets our requirements : 53, 59, 61, 67, 83, 101, 107, 131, 139, 149, 163, 173, 179, 181, 197, 211, 227, 269, 293.

7. **A roadmap for using PMNS.** In this work, we have developed several new ways to use PMNS:

- the original Montgomery-like internal reduction adapted to the Toeplitz recursive splitting (the drawback being that it relies on finding a suitable polynomial $M(X)$),
- a novel internal reduction process that generalizes the Montgomery-like version (the advantage being that there is no need to search for a suitable polynomial $M(X)$),
- a 128-bit version of the two previous approaches,
- a parallel version of all of the above.

Performance wise, the 128-bit version outperforms its 64-bit counterpart only for integers larger than 8192 bits. Unfortunately within this range GnuMP uses faster algorithms so that there is no real point in considering the 128-bit version of PMNS. As a consequence, it is important to carefully consider the specific requirements of each application to determine the most appropriate method for using PMNS. Considering this fact, the roadmap to use PMNS is as follows :

- up to 1024 bits, the 64-bit generalized version is competitive with GnuMP,
- from 1024 to 2048 bits, the Toeplitz splitting method is well suited,
- in the 2048 to 4096 bits range, it appears that PMNS are not well suited to perform modular multiplications,

- above 4096 bits, the parallel version PPMNS is a way to be competitive with GnuMP again.

REFERENCES

1. J.-C. Bajard, Laurent Imbert, and Thomas Plantard, *Modular number systems: Beyond the mersenne family*, Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, 2004, pp. 159–169.
2. Cyril Bouvier and Laurent Imbert, *An alternative approach for sidh arithmetic*, Public-Key Cryptography – PKC 2021 (Cham) (Juan A. Garay, ed.), Springer International Publishing, 2021, pp. 27–44.
3. Richard P. Brent and Paul Zimmermann, *Modern Computer Arithmetic*, Cambridge Monographs on Applied and Computational Mathematics, vol. 18, Cambridge University Press, 2010.
4. Titouan Coladon, Philippe Elbaz-Vincent, and Cyril Hugounenq, *MPHELL: A fast and robust library with unified and versatile arithmetics for elliptic curves cryptography*, ARITH 2021 (Torino, Italy), Transactions on Emerging Topics in Computing, June 2021.
5. Laurent-Stéphane Didier, Fangan Yssouf Dosso, and Pascal Véron, *Efficient modular operations using the Adapted Modular Number System*, Journal of Cryptographic Engineering (2020), 1–23.
6. Fangan Yssouf Dosso, Jean-Marc Robert, and Pascal Véron, *PMNS for Efficient Arithmetic and Small Memory Cost*, IEEE Transactions on Emerging Topics in Computing **10** (2022), no. 3, 1263 – 1277.
7. H. Fan and M.A. Hasan, *Alternative to the karatsuba algorithm for software implementation of GF(2^n) multiplication*, IET Information Security **3** (2009), 60–65(5).
8. M. Anwar Hasan and Christophe Nègre, *Multiway splitting method for toeplitz matrix vector product*, IEEE Transactions on Computers **62** (2013), 1467–1471.
9. Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski, *Analyzing and comparing montgomery multiplication algorithms*, IEEE Micro **16** (1996), no. 3, 26–33.
10. Peter L. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44** (1985), no. 170, 519–521.
11. Christophe Negre and Thomas Plantard, *Efficient modular arithmetic in adapted modular number system using lagrange representation*, Information Security and Privacy, 13th Australasian Conference, ACISP 2008, Wollongong, Australia, 2008, pp. 463–477.
12. Peter van Emde Boas, *Another np-complete problem and the complexity of computing short vectors in a lattice*, Tech. report, University of Amsterdam, Department of Mathematics,Netherlands, 1981.

Université de Toulon, Institut de Mathématiques, Toulon, France
E-mail address: francois.palma@univ-tln.fr
E-mail address: nicolas.meloni@univ-tln.fr
E-mail address: pascal.veron@univ-tln.fr