Multi-precision PMNS with CIOS reduction

Nicolas Méloni, François Palma, Pascal Véron

Université de Toulon, Institut de Mathématiques, Toulon, France.
`nicolas.meloni@univ-tln.fr, francois.palma@univ-tln.fr, pascal.veron@univ-tln.fr`

Abstract. The Polynomial Modular Number System (PMNS) is a representation system that has shown its effectiveness for finite-field arithmetic with ECC-size integers [4, 7, 9]. Efforts have also been made to adapt it to larger integers [8, 19, 24] with mixed results. In particular in [19], a gap in the 2048-bit to 8192-bit integer sizes was highlighted, for which PMNS seemed incapable of overcoming the competition. In this work, we propose a new way of using PMNS for larger sizes, which involves multi-precision polynomial coefficients and utilizes the Montgomery CIOS reduction, thus closing said gap.

Keywords: Modular Arithmetic, PMNS, Montgomery CIOS, Toeplitz Matrices

1 Introduction

Finite-field arithmetic is key to many cryptographic protocols, with Elliptic Curve Cryptography, for example, being used in the "vast majority" of TLS handshakes [18, footnote 2 page 8] with TLS 1.3 reported by Cloudflare to be "93% of the connections" [27] in 2024. Meanwhile, Post Quantum Cryptography (PQC) also uses finite-field arithmetic, with the preeminent example being Kyber [3], the new NIST standard for PQC [21] (although Modular Lattices do not need multi-precision due to their characteristic fields generally being much smaller). Another example in PQC is CSIDH whose prime sizes range from 1024 bits [6] up to 8192 bits [5].

PMNS are usable in any finite-field arithmetic context but mostly as a replacement for traditional multi-precision arithmetic. Instead of the usual 2^w-ary multi-precision where an integer a can be written as $a = \sum_{i=0}^{m-1} a_i 2^{wi}$, a corresponding polynomial $A(X) = \sum_{i=0}^{n-1} A_i X^i$ is chosen to represent a with the main advantages being the lack of carry propagation and the independence of each coefficient during multiplication. In practice, the efficiency of PMNS has been shown in [7], which described a specific library for ECC, named MPHELL, and compared it with other dedicated cryptographic libraries. The results show that on a 64-bit architecture, the PMNS representation gives the best results inside MPHELL for ECDSA/EdDSA signatures (generation and verification). Moreover, it also offers competitive timings on an ARM v8 architecture or an STM32F4 board. More recently, PMNS have been implemented as a faster alternative to the standard finite-field arithmetic provided in the Zcash library, which is a cryptocurrency based on bitcoin [11]. Furthermore, PMNS are well-suited to both SIMD [13] and multithreading approaches [19]. However, the number of coefficients required to represent integers in memory seems to balloon out in size as we consider larger prime fields, as noted in [19]. In that paper, the authors increased the size of the coefficients by using a 128-bit type made available by GCC on x86_64 CPUs, with mixed results.

In this work, we explain the observed degradation in size efficiency with concrete formulas. We also propose an adaptation of the well-established Montgomery CIOS method [16], originally designed for integer modular arithmetic, to a polynomial context using the PMNS representation. Furthermore, we incorporate the use of Toeplitz matrices into our methodology, which are particularly well-suited for polynomial modular reduction. Finally, we establish an alternative way to handle multi-precision sized coefficients in PMNS through the use of "reduced coefficients" [2]. By multi-precision, we mean that each coefficient of a polynomial is stored across multiple machine words. Together, these methods achieve very convincing results and finally bridge the gap for PMNS on large integers highlighted in [19]. Furthermore, we also improve the results with regard to the SIMD implementation presented in [8].
In Section 2 we provide background information related to PMNS. In Section 3 we provide a new bound for the minimal number of symbols required to represent an integer in PMNS. In Section 4 we adapt the Montgomery CIOS algorithm to PMNS in order to manage polynomial coefficients stored on multiple machine words. Finally, in Sections 5 and 6 we present a complexity analysis of our new algorithm followed by the implementation results.

2 Background

In this section, we give a brief overview of PMNS and the way operations are performed.

2.1 PMNS

Definition 1. *Let $p \geqslant 3$, $n \geqslant 2$, $\gamma \in [1, p-1]$ and $\rho \in [1, p-1]$. Let $E \in \mathbb{Z}[X]$ be 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* $A(\gamma) \equiv a \pmod p$.

Since we want $\|A\|_\infty < \rho$, bounds must be put on ρ to guarantee that each element of $\mathbb{Z}/p\mathbb{Z}$ will have at least one polynomial in $\mathcal{B}$ to represent it. This bound is given in [1, Theorem 4.2].
The condition is as follows. Let $\mathfrak{L} = \left\{(x_0, \dots, x_{n-1}) \in \mathbb{Z}^n : \sum_{i=0}^{n-1} x_i \gamma^i \equiv 0 \pmod p\right\}$, which corresponds to the lattice of all vectors of size n for which the associated polynomials vanish in γ modulo p. Let $\mathbf{B}$ be any basis of $\mathfrak{L}$. Then, as long as $\rho > \frac{1}{2}\|\mathbf{B}\|_1$, every element of $\mathbb{Z}/p\mathbb{Z}$ can be represented in $\mathcal{B}$. The way to construct a basis of $\mathfrak{L}$ is also explained by [1], which gives the following canonical basis:

$$\mathbf{B} = \begin{pmatrix} p & 0\,0 \dots\, 0\,0 \\ -\gamma & 1\,0 \dots\, 0\,0 \\ -\gamma^2 & 0\,1 \dots\, 0\,0 \\ \vdots & \ddots \quad \vdots \\ -\gamma^{n-2} & 0\,0 \dots\, 1\,0 \\ -\gamma^{n-1} & 0\,0 \dots\, 0\,1 \end{pmatrix} \qquad (1)$$

The first row of $\mathbf{B}$ corresponds to $(p, 0, \dots, 0)$ and we have $p \equiv 0 \pmod p$. The next $n-1$ rows are $(-\gamma^i, 0, \dots, 0, 1, 0, \dots, 0)$ which correspond to the polynomials $X^i - \gamma^i$ and we indeed have that $\gamma^i - \gamma^i \equiv 0 \pmod p$.

In practice, we can choose a short basis, since we generally want ρ to be as small as possible, originally to fit inside a machine word. Such a short basis can be obtained, for example, through the application of LLL [17]. For parameter consistency, it has been shown in [12, Section 3.2] that taking $\rho = \|\mathbf{B}_{LLL}\| - 1$ (with $\mathbf{B}_{LLL}$ being the LLL-reduced form of $\mathbf{B}$) is optimal as long as negative polynomial coefficients are allowed.

Remark 1. Note that the determinant of $\mathfrak{L}$ is $det(\mathbf{B}) = p$.

Remark 2. Note that any lattice reduction algorithm can be used, such as BKZ [26]. For small sizes, LLL often finds the shortest basis. Meanwhile, for larger sizes, the gain that can be obtained with regard to ρ using another algorithm is often negligible at the cost of a much longer execution time.

2.2 Operations

Let $\mathcal{B} = (p, n, \gamma, \rho, E)$, a PMNS. Let $a \in \mathbb{Z}/p\mathbb{Z}$ and $b \in \mathbb{Z}/p\mathbb{Z}$ with $A \in \mathcal{B}$ such that $A(\gamma) \equiv a \pmod p$ and $B \in \mathcal{B}$ such that $B(\gamma) \equiv b \pmod p$.
Performing $c = a \odot b \pmod p$ through our PMNS $\mathcal{B}$ is done by computing $C(X) = A(X) \odot B(X) \pmod{E(X)}$ with $\odot$ being either addition, subtraction or multiplication. $E(X)$ is called the *external reduction* polynomial and, since it is a monic polynomial of degree n, it allows the polynomial degree to remain bounded by n after each step. Indeed, since $E(\gamma) \equiv 0 \pmod p$, modular reduction by E does not change the value of the result and hence we will properly have $C(\gamma) \equiv c \pmod p$.
However, there is no guarantee that we will have $\|C\|_\infty < \rho$, so another step is

performed called the *internal reduction* which computes from C another polynomial $\tilde{C} \in \mathcal{B}$ such that $\tilde{C}(\gamma) \equiv c \pmod{p}$ and $\|\tilde{C}\|_\infty < \rho$.

External Reduction External reduction in PMNS is generally performed by choosing $E(X) = X^n - \lambda$ with small λ. The existence of such a polynomial in $\mathbb{Z}/p\mathbb{Z}$ only presupposes the existence of a sufficiently small nth residue. Since a polynomial modular reduction is performed, the shape of E has an impact on performance. Other shapes of E have been highlighted in [13] but, for our purposes, we will always consider E to be of shape $X^n - \lambda$. This is because it allows the use of sub-quadratic algorithms for the computation of $A(X) \times B(X) \pmod{E(X)}$, as noted in [19].

Indeed, let $A(X) = a_0 + a_1X + \cdots + a_{n-1}X^{n-1}$ and $B(X) = b_0 + b_1X + \cdots + b_{n-1}X^{n-1}$, let $C(X) = A(X) \times B(X) \pmod{E(X)}$ with $E(X) = X^n - \lambda$. We can write each c_i as $c_i = \sum\limits_{j=0}^{i} a_j b_{i-j} + \lambda \sum\limits_{j=1}^{n-i-1} a_{i+j} b_{n-j}$.
Hence, computing $C(X)$ can be viewed as the following vector-matrix computation:

$$(c_0, c_1, \ldots, c_{n-1}) = (a_0, a_1, \ldots, a_{n-1}) \times \begin{pmatrix} b_0 & b_1 & \ldots & b_{n-2} & b_{n-1} \\ \lambda b_{n-1} & b_0 & \ldots & b_{n-3} & b_{n-2} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \lambda b_2 & \lambda b_3 & \ldots & b_0 & b_1 \\ \lambda b_1 & \lambda b_2 & \ldots & \lambda b_{n-1} & b_0 \end{pmatrix} \tag{2}$$

The matrix being a Toeplitz matrix, we can use the recursive Toeplitz vector-matrix algorithm [15], which is a sub-quadratic algorithm, much like the Karatsuba or Toom-Cook algorithms. However, this algorithm has been shown to be slightly faster than a Karatsuba multiplication followed by a reduction in [19, Proposition 5.2].

Internal Reduction The Internal Reduction in a PMNS is closely related to the Closest Vector Problem (CVP). Indeed, considering the lattice $\mathfrak{L}$ defined in Section 2.1, if we find the element closest to a vector C we want to reduce, we can subtract that element without changing the value of the evaluation in γ, since the elements of $\mathfrak{L}$ vanish in γ modulo p. In doing so, we can reduce the norm of C. CVP is known to be NP-hard [14], but in our context, solving a ρ-approximation suffices. As such, we can use polynomial-time approximation algorithms instead. Several reduction methods have been proposed, the state-of-the-art being the Montgomery Internal Reduction from [22] (see Algorithm 1), which is an adaptation of the classical Montgomery modular reduction to polynomial coefficients. Note that it does not solve the CVP or an approximation, but it does allow us to obtain vectors with reduced norms. Let $M \in \mathbb{Z}[X]$ be such that $M(\gamma) \equiv 0 \pmod{p}$. To reduce $\|C\|_\infty$, the main idea is to compute a polynomial Q such that every coefficient of $C(X)+Q(X)\times M(X) \pmod{E(X)}$ is divisible by a parameter ϕ. Since Q must satisfy $C(X)+Q(X)\times M(X)\equiv 0$ $(\text{mod } E(X),\phi)$, we conclude that $Q(X)\equiv C(X)(-M^{-1}(X))$ $(\text{mod } E(X),\phi)$. We note $M'(X)\equiv -M^{-1}(X)$ $(\text{mod } E(X),\phi)$ in the sequel. Hence, $M(X)$ necessarily needs to be invertible modulo $E(X)$ modulo ϕ. Note that the parameter ϕ is often chosen as a power of 2 so that divisions can be performed with a simple binary shift.

Algorithm 1 Coefficients reduction [22]

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

Note that similarly to the External Reduction step, computing a polynomial multiplication by $M(X)$ and $M'(X)$ mod $E(X)$ can be done through the Toeplitz vector-matrix algorithm. We call $\mathcal{M}$ and $\mathcal{M}'$ the Internal Reduction Matrices associated with M and M' which correspond to the Toeplitz matrices needed for the computation of the previous algorithm. They are as follows:

$$\mathcal{M}=\begin{pmatrix} m_0 & m_1 & \dots & m_{n-1}\\ \lambda m_{n-1} & m_0 & \dots & m_{n-2}\\ \vdots & \ddots & \ddots & \vdots\\ \lambda m_1 & \lambda m_2 & \dots & m_0\end{pmatrix}\begin{array}{l}\leftarrow M\\ \leftarrow X.M \text{ mod } E\\ \\ \leftarrow X^{n-1}.M \text{ mod } E\end{array} \tag{3}$$

and

$$\mathcal{M}'=\begin{pmatrix} m'_0 & m'_1 & \dots & m'_{n-1}\\ \lambda m'_{n-1} & m'_0 & \dots & m'_{n-2}\\ \vdots & \ddots & \ddots & \vdots\\ \lambda m'_1 & \lambda m'_2 & \dots & m'_0\end{pmatrix}\begin{array}{l}\leftarrow M'\\ \leftarrow X.M' \text{ mod } E\\ \\ \leftarrow X^{n-1}.M' \text{ mod } E\end{array} \tag{4}$$

Consequently, internal reduction has a non-negligible cost, since it is at best sub-quadratic in n. As such a parameter δ has been introduced in [9] that refers to the number of "free" additions one can perform without needing to execute an internal reduction step before a multiplication.

Remark 3. $\mathcal{M}$ is a basis of a sub-lattice $\mathfrak{L}'\subset\mathfrak{L}$ as shown in [1, Proposition 4.1].

Remark 4. The use of the Montgomery Internal Reduction algorithm implies the need to use the Montgomery domain for operational consistency. This means that we will have $A(X) \equiv a\phi \pmod{p}$ to represent a and $B(X) \equiv b\phi \pmod{p}$ to represent b so that $C(X) = A(X) \times B(X) \times \phi^{-1} \pmod{E(X)}$ will be such that $C(\gamma) \equiv (a \times b)\phi \pmod{p}$.

Notice that since the multiplication in PMNS is a polynomial multiplication, we can improve this operation when the two operands are the same. We do not consider this case because we prioritize constant-time implementations for modular arithmetic.

Due to the redundancy inherent to PMNS, equality tests cannot be performed in a straightforward fashion like in a classical binary representation. Equality tests, if needed, should be performed either through Horner polynomial evaluation in γ or by testing the membership in a modular lattice. Indeed, let $\mathcal{B}$ be a PMNS. Let $A \in \mathcal{B}$ and $B \in \mathcal{B}$. Then $A(\gamma) \equiv B(\gamma) \pmod{p} \iff (A - B) \in \mathfrak{L} = \left\{(x_0, \ldots, x_{n-1}) \in \mathbb{Z}^n : \sum_{i=0}^{n-1} x_i\gamma^i \equiv 0 \pmod{p}\right\}$. The authors in [10] present PMNS with increased size parameters for faster equality tests, but the performance for standard operations is reduced as a result. In this paper, we only consider PMNS with optimal parameters with regard to the performance of modular multiplication. Hence, we only consider PMNS without equality tests.

3 A new definition of optimal n for PMNS

The parameter n_{opt} has been introduced in previous papers as the optimal value of n for a given prime size for generation purposes. More explicitly, no PMNS can exist with $n < n_{opt}$ without coefficients potentially overflowing from machine words in the middle of operations.

The complexity of the external and internal reductions depends mainly on the parameter n, which is the number of coefficients used to represent an element of $\mathbb{Z}/p\mathbb{Z}$ by a polynomial. Therefore, it is critical to minimize the parameter as much as possible.

Let σ be the number of values in a machine word (for example, 2^{64} for a 64-bit processor). In previous works, authors often set the optimal value of n as $n_{opt} = \left\lfloor \frac{\log_2(p)}{\log_2(\sigma)} \right\rfloor + 1$.

Prime size	256	512	1024	2048	4096	6144	8192
n_{opt}	5	9	17	33	65	97	129
n_{actual}	5	9	18	36	72	108	144

Table 1. size in number of 64-bit machine words.

Table 1 shows empirical results for the actual values of n for various integer sizes compared to the naive evaluation of n_{opt}, assuming 64-bit machine words.

As can be seen, while for smaller integer sizes, this bound can be reached, the gap widens as we increase in integer size. This has been highlighted in [19, Proposition 4.1]. In this section, we give an explanation for this gap and give a more accurate expression for the theoretical minimum value of n.

3.1 Minkowski's bound on the 1-norm of lattices

Let us first recall Minkowski's theorem from [20].
Minkowski's Theorem. *Let $\mathfrak{L} \subseteq \mathbb{R}^n$, a full-rank lattice. Let $S \subseteq \mathbb{R}^n$ be a convex centrally symmetric set. If $vol(S) > 2^n \det(\mathfrak{L})$, then S contains at least one non-zero lattice vector.*

Proposition 1. *Let $\mathfrak{L} \subseteq \mathbb{R}^n$, a full-rank lattice.*
Let $v \in \mathfrak{L}$ such that $\forall x \in \mathfrak{L} \setminus \{0\}, \|x\|_1 \geqslant \|v\|_1$. Then $\|v\|_1 \leqslant (n! \det(\mathfrak{L}))^{\frac{1}{n}}$.

Proof. Let $v \in \mathfrak{L}$ such that $\forall x \in \mathfrak{L} \setminus \{0\}, \|x\|_1 \geqslant \|v\|_1$. In other words, v is the shortest non-zero vector of $\mathfrak{L}$ in terms of 1-norm. Let $S = \{x \in \mathbb{R}^n : \|x\|_1 < \|v\|_1\}$. S is both convex and centrally symmetric, verifying the requirements for Minkowski's Theorem. S is a n-ball in L^1 norm of radius $\|v\|_1$ by construction, thus its volume is $vol(S) = \frac{2^n}{n!}(\|v\|_1)^n$. If we assume $\|v\|_1 > (n! \det(\mathfrak{L}))^{\frac{1}{n}}$, then we have $vol(S) > 2^n \det(\mathfrak{L})$. In that case, S contains at least one non-zero lattice vector according to Minkowski's theorem. This contradicts the assumption that v is the shortest non-zero vector of $\mathfrak{L}$ in terms of 1-norm since all elements of S have a smaller 1-norm than v by construction. Therefore, it follows that $\|v\|_1 \leqslant (n! \det(\mathfrak{L}))^{\frac{1}{n}}$. $\square$

This is a classical result on lattices that helps us in establishing a realistic bound on the 1-norm of $\mathcal{M}$. This is important because ρ, which provides a bound for our polynomial coefficients, directly depends on it. Since $\mathcal{M}$ is a sub-lattice of $\mathfrak{L}$ (see Remark 3), then, from [13, Proposition 3], a PMNS exists as soon as $\rho > \frac{1}{2}\|\mathcal{M}\|_1$. The matrix $\mathcal{M}$ is constructed from a short vector of $\mathfrak{L}$ which means the result from Proposition 1 helps us bound its 1-norm. Indeed, assuming that we choose the shortest vector, v_1, it follows that $\|\mathcal{M}\|_1 \leqslant |\lambda|\|v_1\|_1$. This gives us

$$\|\mathcal{M}\|_1 \leqslant |\lambda|(n!p)^{\frac{1}{n}} \tag{5}$$

by remarking that $\det(\mathfrak{L}) = p$ (see Remark 1).
Being able to construct $\mathcal{M}$ with v_1 is not guaranteed and, in the first place, finding the shortest vector in a lattice is an NP-hard problem, but this gives us a bound in the optimal case, which is the premise behind the choice of n_{opt}. Similarly, the optimal value of $|\lambda|$ is 1 in the case of $E(X) = X^n + 1$ with n a power of 2. This gives us $\|\mathcal{M}\|_1 \leqslant (n!p)^{\frac{1}{n}}$ in the optimal case.

3.2 New expression for n_{opt}

We want $\sigma \geqslant 2\rho$ so as to be able to fit our polynomial coefficients in a single machine word. Indeed, having $\sigma \geqslant \rho$ does not suffice, seeing as coefficients can be both positive and negative, so an extra bit is needed. As explained in the previous section, we cannot construct a PMNS such that $\rho < \frac{1}{2}\|\mathcal{M}\|_1$. Which means, since we want $\sigma \geqslant 2\rho$, we will at least need to have $\sigma \geqslant \|\mathcal{M}\|_1$. Hence, we can set a requirement on n to be $\sigma \geqslant (n!p)^{\frac{1}{n}}$. Which means:

$$p \leqslant 2^{n\log_2(\sigma)-\log_2(n!)},$$

thus $\log_2(p) \leqslant n\log_2(\sigma) - \log_2(n!)$. We can then set n_{opt} as the smallest n which verifies the above inequality. This gives us the following table:

Prime size	256	512	1024	2048	4096	6144	8192
old n_{opt}	5	9	17	33	65	97	129
new n_{opt}	5	9	17	34	70	105	141
n_{actual}	5	9	18	36	72	108	144

Table 2. size in number of 64-bit machine words.

The new approximation fits better with reality. The remaining difference can be attributed to the approximation error from LLL [23] (since we generally obtain our short bases with LLL in practice).

Note that we can write $n_{opt} = \left\lceil \frac{\log_2(p)+\log_2(n_{opt}!)}{\log_2(\sigma)} \right\rceil$ or, alternatively,

$$n_{opt} = \left\lfloor \frac{\log_2(p)+\log_2(n_{opt}!)}{\log_2(\sigma)} \right\rfloor + 1$$

Proposition 2. *Let p be such that $\log_2\left(\left(\left\lfloor \frac{\log_2(p)}{\log_2(\sigma)} \right\rfloor + 1\right)!\right) \geqslant \log_2(\sigma)$, then*

$$n_{opt} > \left\lfloor \frac{\log_2(p)}{\log_2(\sigma)} \right\rfloor + 1$$

Proof. First, notice that $\left\lfloor \frac{\log_2(p)+\log_2(n_{opt}!)}{\log_2(\sigma)} \right\rfloor + 1 \geqslant \left\lfloor \frac{\log_2(p)}{\log_2(\sigma)} \right\rfloor + 1$. Hence, since $n_{opt} = \left\lfloor \frac{\log_2(p)+\log_2(n_{opt}!)}{\log_2(\sigma)} \right\rfloor + 1$, then

$$n_{opt} \geqslant \left\lfloor \frac{\log_2(p)}{\log_2(\sigma)} \right\rfloor + 1$$

Then, as soon as $\log_2\left(\left(\left\lfloor \frac{\log_2(p)}{\log_2(\sigma)} \right\rfloor + 1\right)!\right) \geqslant \log_2(\sigma)$, then $\frac{\log_2(n_{opt}!)}{\log_2(\sigma)} \geqslant 1$.

Therefore

$$n_{opt} = \left\lfloor \frac{\log_2(p)+\log_2(n_{opt}!)}{\log_2(\sigma)} \right\rfloor + 1 \geqslant \left\lfloor \frac{\log_2(p)}{\log_2(\sigma)} + 1 \right\rfloor + 1 > \left\lfloor \frac{\log_2(p)}{\log_2(\sigma)} \right\rfloor + 1$$

$\square$

For example, for $\sigma = 2^{64}$, this bound is achieved for $p \geqslant 2^{1280}$. This is simply because 1280 is 64 times 20 and $21! > 2^{64}$ while $20! < 2^{64}$. Hence, past this bound, our new expression of n_{opt} will be strictly superior to the old expression, making the old bound impossible to reach.

4 Adaptations to make multi-precision PMNS more efficient

This section details the adaptations necessary to switch to multi-precision polynomial coefficients for our PMNS.

4.1 Reduced Coefficients

To address the expansion of n as the size of p increases, it is suggested in [19] to switch to $\phi = \sigma^2$ (using the 128-bit type made available by GCC on X86_64 CPUs). In so doing, an integer from $\mathbb{Z}/p\mathbb{Z}$ is now represented by a polynomial with coefficients stored over two machine words (128 bits) rather than a single one. However, the multi-precision used was a naive implementation with carry propagation.
A better method is to instead split the coefficients in equal parts among several machine words. This is reminiscent of non-saturated integer arithmetic with a radix chosen to minimize carry chains used in some cryptographic schemes. For example, for $\rho = 2^{108}$ with $\sigma = 2^{64}$, instead of storing the first 64 bits of a coefficient on the first machine word and the next 44 bits on another, we split the coefficients into two 54-bit values on the two machine words. All coefficients are thus bounded, which means we can anticipate any overflows and remove the need for carry propagation until the end step of operations to ensure that the result is back to the initial bounds on our coefficients. This technique is similar to the one used in [2] where it is called "reduced coefficients" polynomials, which is how we shall refer to it in the sequel of this paper.

For the sequel of this section, we assume that PMNS coefficients are larger than one machine word, and thus multi-precision is required. Let s be the number of machine words needed to fit a single coefficient. Since we want to perform the division by ϕ in s steps, we split each coefficient such that only values from $-\lceil \sqrt[s]{\phi} \rceil$ to $\lceil \sqrt[s]{\phi} \rceil - 1$ are contained in each machine word at the start. That is to say, let $a \in \mathbb{Z}/p\mathbb{Z}$, a polynomial A representing a in a PMNS will thus be written as $A(X) = \sum_{k=0}^{n-1} a_k X^k$ with $a_k = \sum_{i=0}^{s-1} a_{k_i} \lceil \sqrt[s]{\phi} \rceil^i$. Thus, $A(X) = \sum_{i=0}^{s-1} (\sum_{k=0}^{n-1} a_{k_i} X^k) \lceil \sqrt[s]{\phi} \rceil^i = \sum_{i=0}^{s-1} A_i \lceil \sqrt[s]{\phi} \rceil^i$ where

$$A_i = \sum_{k=0}^{n-1} a_{k_i} X^k \tag{6}$$

.

Traditionally, authors set ϕ as the size of a machine word and a given PMNS is valid if $2w\rho(\delta+1)^2 < \phi$ with δ a parameter introduced in [9] that refers to the number of "free" additions one can perform without needing to execute an internal reduction step before a multiplication and $w = |\lambda|(n-1)+1$ for $E(X) = X^n - \lambda$ [13]. As such, we can, for now, set the value of ϕ to $2^{\lceil \log_2(2w\rho(\delta+1)^2) \rceil}$.

Carry-less operations The main purpose of reduced coefficients is to delay all carry propagation to the last step. As such, for efficiency reasons, we must ensure that $|\lambda|\lceil \sqrt[s]{\phi} \rceil < \sigma$ otherwise we will not be able to construct the Toeplitz matrix from Equation (2) in such a way that all coefficients fit in a machine word. Furthermore, since we are using the Toeplitz vector-matrix algorithm, some additions are performed before the multiplication step. The maximum number of addition performed for one coefficient is n, with coefficients of size bounded by $|\lambda|\lceil \sqrt[s]{\phi} \rceil$, hence we need to enlarge the bound to be:

$$n|\lambda|\lceil \sqrt[s]{\phi} \rceil < \sigma \tag{7}$$

Note that some fine-tuning can be done depending on the number of recursion steps and the way the splitting is performed, but this bound is sufficient for a general approach.

Division by ϕ As noted in Section 4.1, we can set $\phi = 2^{\lceil \log_2(2w\rho(\delta+1)^2) \rceil}$. However, by necessity from Equation (7), we cannot set $\sqrt[s]{\phi} = \sigma$. Traditionally in PMNS, ϕ is set to σ. This is done for efficiency reasons, since a division by the size of a machine word in practice translates to a simple MOV instruction. Similarly, a modular reduction can be performed "for free" since anything above σ will overflow. In our case, since this isn't possible, we will need to perform actual divisions and modular reductions.

One of the most efficient ways to perform a modular reduction consists of simply applying a binary mask in memory. However, this is only possible for powers of 2. Similarly, powers of 2 allow the use of bit-shifting to perform divisions. Hence, it seems logical to choose ϕ to be a power of 2.

In our case, the divisions and reductions performed will be done with $\lceil \sqrt[s]{\phi} \rceil$. Thus, we also want $\lceil \sqrt[s]{\phi} \rceil$ to be a power of 2. This gives us, for efficiency reasons:

$$\phi = 2^{s\left\lceil \frac{\log_2(2w\rho(\delta+1)^2)}{s} \right\rceil} \tag{8}$$

4.2 Generation

In this section, we detail the generation process for multi-precision PMNS. A sample generation code is available at `https://github.com/francoispalma/PMNS/tree/master/multiprecision_pmns`.

Adaptation to multi-precision In previous works, authors generate a PMNS for a given p in the following manner:

1. ϕ is chosen (often the size of a machine word)
2. n_{opt} is computed with regards to ϕ and we set $n \leftarrow n_{opt}$ at the start
3. we find λ, a small nth root in $\mathbb{Z}/p\mathbb{Z}$ such that $E(X) = X^n - \lambda$ is irreducible in $\mathbb{Z}$
4. we choose γ among the roots of $E(X)$ in $\mathbb{Z}/p\mathbb{Z}$
5. we construct $\mathbf{B}$ as detailed in Equation (1) and apply LLL. We note $\mathbf{B}_{LLL}$ the LLL reduced form of $\mathbf{B}$
6. we set $\rho = \|\mathbf{B}_{LLL}\|_1 - 1$ (as explained in Section 2.1). If ρ is small enough with respect to ϕ (we need $\phi > 2w\rho(\delta+1)^2$ as explained in Section 4.1), the generation process ends. Else, we set $n \leftarrow n+1$ and go to step 3.

Note that this is a generalization of the process and does not necessarily fit all previous works. This process works and is good enough for us to expand on.

Remark 5. Note that in step 3 we choose the smallest possible λ. Notice that since the value of w is proportional to λ, and ϕ must satisfy $\phi > 2w(\delta+1)^2$, it is natural to choose λ as small as possible.

When switching to multi-precision, the value of ϕ is necessarily set differently. One can choose s, the number of machine words necessary for a single polynomial coefficient, at the start of the generation process and compute n_{opt} as detailed in Section 3.2. The process thus ends up being very similar to the one we have just described. However, another approach is possible:

1. we choose n
2. we find λ, a small nth root in $\mathbb{Z}/p\mathbb{Z}$ such that $E(X) = X^n - \lambda$ is irreducible in $\mathbb{Z}$
3. we choose γ among the roots of $E(X)$ in $\mathbb{Z}/p\mathbb{Z}$
4. we construct $\mathbf{B}$ as detailed in Equation (1) and apply LLL. We note $\mathbf{B}_{LLL}$ the LLL reduced form of $\mathbf{B}$. From it, we set $\rho = \|\mathbf{B}_{LLL}\|_1 - 1$ as explained in Section 2.1
5. We set s so that $\sigma^s \geqslant \rho$ (with σ the number of values in a machine word, for example, 2^{64} for a 64-bit processor) from which we compute ϕ as detailed in Equation (8).

Choosing n however we want is beneficial because it allows us to choose it to be optimal in terms of complexity (see Section 5 for more details) but also with many small factors for the Toeplitz recursive splitting. Note also that this process always yields a PMNS.

Choice of M Since Algorithm 1 requires $-M^{-1} \pmod{E(X), \phi}$, then necessarily M needs to have an inverse mod $E(X), \phi$. Previous works [9, 13] provide more details as to the exact conditions for M to be invertible. In this section, we explain a process for choosing M among the valid possibilities such that the 1-norm is minimal.

After constructing $\mathbf{B}$ as detailed in Equation (1) and applying LLL, M is often chosen as the row such that the associated Internal Reduction Matrix $\mathcal{M}$ has the smallest 1-norm and is invertible mod ϕ. However, some small optimizations can be performed.

Proposition 3. *Let $v = (v_0, v_1, \ldots, v_{n-1}) \in \mathfrak{L}$ and $\mathcal{V}$ be the internal reduction matrix associated with v (see Equation (3)). Let $w = (v_1, \ldots, v_{n-1}, \frac{v_0}{\lambda})$ and $\mathcal{W}$ be the internal reduction matrix associated with w. If $v_0 \equiv 0 \pmod{\lambda}$, then $w \in \mathfrak{L}$ and $\|\mathcal{W}\|_1 \leqslant \|\mathcal{V}\|_1$.*

Proof. Let $v = (v_0, v_1, \ldots, v_{n-1}) \in \mathfrak{L}$. We note $V(X)$ the polynomial associated with v. Let $w = (v_1, \ldots, v_{n-1}, \frac{v_0}{\lambda})$. We note $W(X)$ the polynomial associated with w. Then, if $v_0 \equiv 0 \pmod{\lambda}$, $W(X) \in \mathbb{Z}[X]$. Furthermore, notice that $W(X) \times X \equiv V(X) \pmod{E(X)}$ since $\frac{v_0}{\lambda}X^n \equiv v_0 \pmod{E(X)}$. Since $v \in \mathfrak{L}$, we have $V(\gamma) \equiv 0 \pmod{p}$ and therefore $W(\gamma) \equiv \frac{V(\gamma)}{\gamma} \equiv 0 \pmod{p}$ and thus $w \in \mathfrak{L}$.

For $M(X) = m_0 + m_1 X + \cdots + m_{n-1}X^{n-1}$ then $\|\mathcal{M}\|_1 = |m_0| + |\lambda| \sum_{i=1}^{n-1} |m_i|$ since the first column of $\mathcal{M}$ is always the one with the largest coefficients. Hence $\|\mathcal{W}\|_1 \leqslant \|\mathcal{V}\|_1$.

□

Remark 6. It is easy to see that the process can be repeated as long as the coefficient in the first position is divisible by λ. Note in particular that for $\lambda = 2$, this process will give a vector whose first coefficient is odd. This is notable because it has been shown in [9, Proposition 8] that it is a sufficient condition to guarantee that M is invertible by ϕ when ϕ is a power of 2, as is our case here.

Thus, selecting M can be done with the following heuristic:

1. We compute $\mathbf{B}_{LLL}$, the LLL-reduced form of $\mathbf{B}$.
2. For the first row of $\mathbf{B}_{LLL}$ which we can note as r_1, compute R_1 the corresponding polynomial.
3. Compute the successive $R_1/X^i \bmod E(X)$ as long as $R_1/X^i \bmod E(X)$ has integer coefficients.
4. Choose $M_1 = R_1/X^j \pmod{E(X)}$ with j the largest value such that M_1 is invertible mod $E(X)$ mod ϕ.
5. Repeat for each row of $\mathbf{B}_{LLL}$.
6. Choose M as the smallest 1-norm vector among the M_k.

Note that a given row is not guaranteed to yield a polynomial that is invertible mod $E(X)$ mod ϕ through this heuristic unless $\lambda = 2$. It has been shown in [9, Propositions 8 and 11] that for $E(X) = X^n - \lambda$ with even λ, at least one row of $\mathbf{B}_{LLL}$ is invertible, but for odd λ the question remains open for nonspecific n. In [9, Proposition 12] and [13, Proposition 7], the authors show that a valid M can be found by performing binary linear combinations of the rows of

$\mathbf{B}_{LLL}$ for odd λ although this may need a 2^n step exhaustive search.

Another consideration is sign. PMNS are well-suited for the use of vectorized instructions, and in particular they make great use of the AVX512IFMA set of instructions, as shown in [8]. However, those instructions only exist in unsigned form. It is possible to perform all operations in the PMNS using only unsigned coefficients; however, this requires finding a M with only positive coefficients. This also means that the value of ρ must be doubled to account for the necessity of every element of $\mathbb{Z}/p\mathbb{Z}$ having a polynomial representation with only positive coefficients, which halves the number of potential candidates.

The authors of [8] use a generation process that involves iterating over various random vectors whose norms are close to the upper bound on the shortest vector until one is found that is all positive and invertible. The considerations for our new multi-precision PMNS variant are opposite in that we do not necessarily require a very short vector since we can increase the size of our coefficients as needed. Instead, having a process that is guaranteed to find a positive invertible vector in $\mathfrak{L}$ is more useful.

We propose the following heuristic that systematically finds a positive linear combination of the rows of the reduced basis that is also guaranteed to be invertible for even λ.

1. We initialize $C(X)$ as the polynomial corresponding to the vector $(\|\mathbf{B_{LLL}}\|_1 + 1, \|\mathbf{B_{LLL}}\|_1 + 1, \dots, \|\mathbf{B_{LLL}}\|_1 + 1)$ and set $\rho' = \|\mathbf{B_{LLL}}\|_1$, a temporary ρ for generation purposes.
2. We compute $\phi' = 2^{s\left\lceil \frac{\log_2(2w\rho'(\delta+1)^2)}{s} \right\rceil}$, a temporary ϕ for generation purposes. We have a temporary valid PMNS for use in conversion operations.
3. We compute $c \in \mathbb{Z}/p\mathbb{Z}$ with $C(\gamma) \equiv c \pmod{p}$.
4. Using algorithm 5 from [12], we compute $\overline{C}(X)$, a polynomial representing c in our temporary PMNS. This guarantees $\|\overline{C}\|_\infty \leqslant \frac{1}{2}\|\mathbf{B}_{LLL}\|_1 + 1$ (as shown in [12, Proposition 1]).
5. Since $C(\gamma) \equiv \overline{C}(\gamma) \equiv c \pmod{p}$, we set $M(X) = C(X) - \overline{C}(X)$ and we properly have $M(\gamma) \equiv 0 \pmod{p}$. Furthermore, we know by construction that $\forall i, \frac{1}{2}\rho' \leqslant m_i \leqslant \frac{3}{2}\rho' + 2$. Hence, all coefficients of M are positive.

This M is not necessarily invertible. Assuming even λ, a sufficient condition can be found to construct an invertible M, using [9, Proposition 11] which shows that at least one row of $\mathbf{B}_{LLL}$ is such that its first coefficient is odd. As mentioned above, [9, Proposition 8] states that it is sufficient for even λ that the first coefficient of M is odd for M to be invertible mod E, ϕ with ϕ a power of 2, which is our case here.

Thus, the construction is as follows. Let us note $R_k(X)$ the polynomial equivalent of one of the rows of $\mathbf{B}_{LLL}$ whose first coefficient is odd. Since no coefficient of $\mathbf{B}_{LLL}$ is bigger than $\|\mathbf{B}_{LLL}\|_1$ (which is ρ') by definition of the 1-norm of a matrix, then $2 \times M(X) - R_k(X)$ will have all coefficients positive and the first coefficient odd. Indeed, as noted earlier, $\forall i, \frac{1}{2}\rho' \leqslant m_i \leqslant \frac{3}{2}\rho' + 2$. Hence $\forall i, \rho' \leqslant 2 \times m_i \leqslant 3\rho' + 4$. Hence subtracting $R_k(X)$ from $2 \times M(X)$ will not have any coefficient go negative. Also since the first coefficient of $R_k(X)$ is odd, then the first coefficient of the result of $2 \times M(X) - R_k(X)$ will always be odd. Hence this new polynomial will be invertible and all positive. Since we are subtracting an element of $\mathfrak{L}$ from another element of $\mathfrak{L}$, the result will be in $\mathfrak{L}$.

We can then continue as normal in the generation and set the real value of ρ as $\rho > 2 \times (\frac{1}{2}\|\mathcal{M}\|_1 + 1)$ with $\mathcal{M}$ the internal reduction matrix from Equation (3) and $\phi = 2^{s\left\lceil \frac{\log_2(2w\rho(\delta+1)^2)}{s} \right\rceil}$. As noted above, the value of ρ needs to be doubled, compared to a system where negative coefficients are allowed in order to represent all elements of $\mathbb{Z}/p\mathbb{Z}$.

One demerit of this heuristic is that we have $\|2 \times M(X) - R_k(X)\|_\infty \leqslant 4\rho' + 4$, which is much larger in terms of ∞-norm than a row of $\mathbf{B}_{LLL}$. Thus, this process is useful from a theoretical standpoint, but in practice we can improve on it. Indeed, one can continuously add and subtract rows of the short basis matrix, as long as the result is positive. The end result is not guaranteed to be invertible, however for $\lambda = 2$, we can always obtain an invertible polynomial from Remark 6.

4.3 Montgomery CIOS

As explained in Section 2.2, internal reduction in PMNS is done through the use of an adaptation of the Montgomery algorithm to polynomials. However, the classic Montgomery algorithm has several variants. In [16], Koç et al. proposed various ways to schedule the operations of the Montgomery algorithm, notably the Montgomery CIOS (Coarsely Integrated Operand Scanning) and FIOS (Finely Integrated Operand Scanning) methods. In [24], Noyez et al. show how to adapt the FIOS variant to PMNS in a hardware context, since this variant is well adapted. Meanwhile, in a software context, the CIOS variant is the state-of-the-art for many cryptographic protocols that rely on modular multiplication. The widely used OpenSSL and GMP libraries both propose software implementations, for example. Another example would be the reference implementation of [6] (`https://csidh.isogeny.org/software.html`). From our testing (see Section 6), Montgomery CIOS seems to have the edge over regular Montgomery until the 2048-bit range. For our purposes, this means that as long as $\sigma^s < 2^{2048}$, using the CIOS version seems logical. Algorithm 2 paraphrases the Montgomery CIOS algorithm from [16].

The parameter w from Algorithm 2 refers to the wordsize and the parameter W is set to $W = 2^w$. The principle is to first compute a times ($b \pmod W$)) before zeroing out the last w bits using the Montgomery reduction mod W and divide by W (which is seamless here due to the division being the size of a

Algorithm 2 Montgomery CIOS modular multiplication [16]

Require: $n \in \mathbb{N}$ the modulus, $a \in \mathbb{N}$ and $b \in \mathbb{N}$ the operands, w the wordsize of the computer with $W = 2^w$, r the smallest power of 2 such that $r > n$, $z = \lceil \frac{r}{W} \rceil$ and $n' \equiv -n^{-1} \pmod r$

Ensure: $t \equiv a \times b \times r^{-1} \pmod n$

```
1: for $i = 0 \ldots z-1$ do
2:     $C \leftarrow 0$
3:     for $j = 0 \ldots z-1$ do
4:         $C, S \leftarrow a_j \times b_i + c$
5:         $t_j \leftarrow S$
6:     end for
7:     $(C, S) \leftarrow t_z + C$
8:     $t_z \leftarrow S$
9:     $t_{z+1} \leftarrow C$
10:    $C \leftarrow 0$
11:    $m \leftarrow t_0 \times n'_0 \pmod W$
12:    for $j = 0 \ldots z-1$ do
13:        $(C, S) \leftarrow t_j + m \times n_j + C$
14:        $t_j \leftarrow S$
15:    end for
16:    $(C, S) \leftarrow t_z + C$
17:    $t_z \leftarrow S$
18:    $t_{z+1} \leftarrow t_{z+1} + C$
19:    for $j = 0 \ldots z-1$ do
20:        $t_j \leftarrow t_{j+1}$
21:    end for
22: end for
23: return $t$
```

machine word). We then compute a times $(\lceil \frac{b}{W} \rceil \pmod W)$ and add it to the running total before repeating the process. The end result is the same as the classical Montgomery algorithm, although we only need to compute the least significant word of the modular inverse of n (the modulus) by r (a power of 2 such that $\frac{r}{2} < n < r$) and the operations are scheduled differently.

We adapt it to PMNS in Algorithm 3. In it, $\mathbb{Z}_{n-1}[X]$ refers to the polynomials of degree of at most $n-1$ with coefficients in $\mathbb{Z}$. Each polynomial $P(X)$ is such that $P(X) = \sum_{i=0}^{s-1} P_i(\sqrt[s]{\phi})^i$ with $\forall i$, $\|P_i\|_\infty < \sqrt[s]{\phi}$ (see Equation (6)). In a similar fashion to the integer version, instead of computing $A(X) \times B(X) \pmod{E(X)}$ in one step, this CIOS variant first computes $A(X) \times B_0 \pmod{E(X)}$. Then, we compute a polynomial Q to zero out the lower bits of the result as normal and divide by $\sqrt[s]{\phi}$. We then compute $A(X) \times B_1 \pmod{E(X)}$ and add it to the running total before repeating the process of finding Q and so on. This is equivalent to the normal Montgomery Internal Reduction algorithm, just scheduled differently.

Algorithm 3 Multi-precision PMNS Montgomery CIOS modular multiplication

```
Require: $(p, n, \gamma, \rho, E)$ a PMNS with $E(X) = X^n - \lambda$, $A, B \in \mathbb{Z}_{n-1}[X]$ with $\|A\|_\infty < \rho$
    and $\|B\|_\infty < \rho$, $s$ the number of chunks coefficients are split into, $\phi = 2^h$ with
    $h \in \mathbb{N}^*$ a multiple of $s$, $M(X) \in \mathbb{Z}_{n-1}[X]$ such that $M(\gamma) \equiv 0 \pmod p$ and
    $M' \equiv -M^{-1} \pmod{E(X), \phi}$.
Ensure: $C(\gamma) \equiv A(\gamma)B(\gamma)\phi^{-1} \pmod p$, with $C \in \mathbb{Z}_{n-1}[X]$ such that $\forall i$, $\|C_i\|_\infty <$
    $\sqrt[s]{\phi}$ and $\|C\|_\infty < \rho$
 1: $R \leftarrow (0, \dots, 0)$
 2: for $i = 0 \dots s-1$ do
 3:     for $j = 0 \dots s-1$ do
 4:         $R_{i+j} \leftarrow R_{i+j} + A_j \times B_i \pmod{E(X)}$
 5:     end for
 6:     $T \leftarrow R_i \times M'_0 \pmod{E(X), \sqrt[s]{\phi}}$
 7:     for $j = 0 \dots s-1$ do
 8:         $Q_j \leftarrow T \times M_j \pmod{E(X)}$
 9:     end for
10:     for $j = 0 \dots s-1$ do
11:         $R_{i+j} \leftarrow R_{i+j} + Q_j$
12:     end for
13:     $R_{i+1} \leftarrow R_i / \sqrt[s]{\phi}$ # Exact Division
14: end for
15: for $i = 0 \dots s-1$ do
16:     $C_i \leftarrow R_{s+i} \pmod{\sqrt[s]{\phi}}$
17:     $R_{s+i+1} \leftarrow R_{s+i} / \sqrt[s]{\phi}$ # Euclidean Division
18: end for
19: return $C$
```

Each multiplication step in the algorithm makes use of the Toeplitz vector-matrix multiplication algorithm to ensure sub-quadratic complexity.

5 Complexity analysis

In this section we analyze the Multi-precision CIOS algorithm for PMNS and compare it to the single-precision classical Montgomery version from [22].

Step \ Method	Single-precision Montgomery	Multi-precision Montgomery CIOS
A times B	$n_{sp}^{2-\log_{n_{sp}}(2)}$	$s^2 \times n_m^{2-\log_{n_m}(2)}$
Multiplication by $\mathcal{M}'$	$n_{sp}^{2-\log_{n_{sp}}(2)}$	$s \times n_m^{2-\log_{n_m}(2)}$
Multiplication by $\mathcal{M}$	$n_{sp}^{2-\log_{n_{sp}}(2)}$	$s^2 \times n_m^{2-\log_{n_m}(2)}$
Total	$3n_{sp}^{2-\log_{n_{sp}}(2)}$	$s \times (2s+1) \times n_m^{2-\log_{n_m}(2)}$

Table 3. Complexity of the various step of each algorithm in the number of multiplications.

Table 3 summarizes the differences in terms of the number of multiplications, with n_{sp} being the single-precision version of the parameter n, s being the number of machine words needed to represent a single coefficient in our new multi-precision version of the PMNS and n_m being the multi-precision version of the parameter n. To clarify, in all cases we will have $s \times n_m \leqslant n_{sp}$. Obviously, for $s = 1$, the complexity of our new algorithm is identical to the single-precision version.

Not shown in the table is the fact that the division steps are performed through binary shifts in the multi-precision version, whereas they are performed by using MOV instructions in the single-precision version. In the same vein, modular reductions are considered to be "free" in the single-precision version because they are the result of natural overflow behavior of machine word sized multiplications, whereas they are performed with binary masks in the multi-precision version.

The exponent on the respective n of each variant comes from the usage of the Toeplitz vector-matrix algorithm, whose complexity was detailed in [15, Equation 22]. In this case, we assume the worst case which is prime n. The more n can be decomposed into small factors, the lower the exponent, which means that in the general case it is more favorable to choose $s \leqslant n_m$ since the overall complexity is quadratic in s but sub-quadratic in n_m.

The choice of s and n_m in the multi-precision case should be made according to the value of n_{opt} detailed in Section 3.2. That is to say:

$$n_m \times s \geqslant n_{opt} \text{ with } n_m \geqslant s$$

However, while choosing $s = 1$ may look the best on paper, reducing s too much means increasing n_m to the point where the value of w, with $w = |\lambda|(n_m - 1) + 1$, becomes much too large and we no longer have $2w\rho(\delta + 1)^2 \leqslant \sigma$ with σ the number of values in a single machine word (for example, 2^{64} for a 64-bit processor). Table 4 shows some empirical results of the value of $s \times n_m$ for various PMNS found with corresponding parameters.

Prime size	1024	2048	4096	6144	8192
n_{opt} from 3.2	17	34	69	105	141
$s = 1$	19	39	83	130	183
$s = 2$	18	36	80	120	160
$s = 3$	18	36	72	108	144
$s = 4$	20	36	72	108	144

Table 4. Evaluation of $s \times n_m$ for various values of s with $\sigma = 2^{64}$ compared with n_{opt} for various prime sizes.

Hence, the method should be to select the smallest value of $s \times n_m$, among which the smallest value of s should give a better time complexity.

6 Implementation results

Clock cycle measurements are performed according to the following recommendation from the relevant Intel white paper [25] with slight adaptation (taking the median instead of the minimum measurement) :

- We deactivate the *Turbo-Boost*®
- We try to minimize potential cache misses by "heating" the cache memory with 501 runs that are not measured
- Then we generate 1001 appropriate data sets for which 501 runs are executed and the clock cycles are measured by interrogating the Time Stamp Counter with calls to the RDTSC instruction.
- The performance is the median value

The source code is available at `https://github.com/francoispalma/PMNS/tree/master/multiprecision_pmns` .

6.1 Sequential

Method \ Prime size	1024	2048	4096	6144	8192
GMP Montgomery	1539	4653	14524	28028	44587
GMP Montgomery CIOS	1187	4607	16793	35283	61478
PMNS from [19]	1709	7491	33471	87671	169070
PMNS128 from [19]	2363	10304	39446	80726	146895
This work $s = 2$	1173	3874	16254	30795	54146
This work $s = 3$	1195	3901	**13444**	**26497**	**43835**
This work $s = 4$	1511	4218	14766	30973	45392
OpenSSL Montgomery CIOS	**968**	**3684**	14889	33506	58723

Table 5. Cycle count for modular multiplication on large integers using GCC 12.3.0 on intel processor i9-11900KF.

Note that our code is compiled with the flag **-fno-tree-vectorize** to ensure that no vectorized instructions are used. This is so that we can compare ourselves fairly to GMP which does not use vectorized instructions.

We can see in Table 5 that this new multi-precision method is better than the naive one from [19]. This new method makes PMNS competitive with GMP in the 2048 to 8192-bit range, which was highlighted as not being very favorable in [19]. The comparison with OpenSSL shows that the 1024 to 2048-bit range is, while still competitive, not better than the alternative. However, our choices allow for much better performances on larger integer sizes.

6.2 AVX512IFMA

The instruction set AVX512IFMA available on many modern CPUs has been noted in [8] to be particularly well-suited to PMNS. Although AVX2 allows for full vectorized 32-bit multiplication, in our tests this does not lead to better performance than regular sequential instructions due to the large increase in n from taking $\sigma = 2^{32}$. The trade-off of using AVX512IFMA compared to normal sequential instructions is that we will have $\sigma = 2^{52}$ because this instruction set only allows for full vectorized 52-bit multiplication and only positive coefficients can be used. This naturally leads to an increase in the value of the parameter n for a given prime size, but the speedup still seems to be worth the cost.

Method \ Prime size	1024	2048	4096	6144	8192
This work $s = 2$	747	1574	5227	12725	25220
This work $s = 3$	457	1577	5132	10891	15896
This work $s = 4$	-	-	5563	12844	19012

Table 6. Cycle count for modular multiplication on large integers with AVX512IFMA instructions using GCC 12.3.0 on intel processor i9-11900KF.

Method \ Prime size	1024	2048	4096	6144	8192
This work $s = 2$	1.570	2.461	2.572	2.082	1.738
This work $s = 3$	2.566	2.457	2.619	2.433	2.758
This work $s = 4$	-	-	2.417	2.063	2.306

Table 7. Ratio of above table values with the best sequential version of the same integer size.

In Table 6 we show the results we get for the same integer sizes as the one used in Table 5. In Table 7 we perform a ratio of the results in Table 6 by the best result of any PMNS variant in each column of Table 5. It is clear from those results that the vectorized instructions are faster at all sizes, which shows that they are indeed worth both the increase in parameter size and the slower generation process. In theory, we would expect to see a factor 4 improvement. However, memory management of 512-bit registers and the reduced σ mean that the ratio is lower. Furthermore, the sub-quadratic algorithms we use require subtraction, which may lead to negative coefficients. This needs to be handled manually through the use of sign masks, which adds an additional cost.

In Table 8 we compare ourselves with the AVX512IFMA OpenSSL implementation of Montgomery CIOS. Note that comparatively to Tables 4-7 OpenSSL only provides AVX512IFMA versions for integers of size 1024, 1536 and 2048

Method \ Prime size	1024	1536	2048
OpenSSL Montgomery CIOS	505	846	1177
This work	453	1065	1492

Table 8. Comparison of cycle count for modular multiplication on large integers with available sizes in OpenSSL using AVX512IFMA instructions with GCC 12.3.0 on intel processor i9-11900KF.

bits. As can be seen, our version seems to be worse for prime sizes above 1024-bit. This can be attributed in part to our code being written in C and comparing ourselves with the optimized assembly code used by OpenSSL. The performance of this work remains relatively competitive and further optimizations may lead to better results in the future.

Method \ Prime size	807	1214	1621	2029	2436	2844	3251
Results in [8]	779	1004	1965	3764	6233	9093	19157
This work	457	747	1573	2443	3244	5124	5125

Table 9. Comparison of cycle count for modular multiplication on PMNS, $\delta = 5$, with [8] using AVX512IFMA instructions with GCC 12.3.0 on intel processor i9-11900KF.

In Table 9 we compare ourselves with the results in [8] at the same exact integer sizes. In that paper, they set the parameter δ to 5 so the parameters we generated for this table similarly set δ to 5 for a fair comparison. As can be seen, this new multi-precision method seems to lead to better performance overall. The comparison is made with the code made available by the authors "as is" in [8], only switching the compiler to the same version of GCC for a fair comparison.

7 Conclusion

In this work, we give a more accurate bound on the number of symbols needed to represent an integer in a PMNS. We also present an adaptation of the Montgomery CIOS algorithm to PMNS, utilizing multi-precision oriented optimizations. We show that these changes allow for better performance on large integer sizes and help alleviate a deficiency in the 2048-bit to 8192-bit range for PMNS. Furthermore, we provide an efficient heuristic that solves the problem of finding a suitable polynomial M in an AVX512IFMA context. Combined with the new multi-precision CIOS algorithm, we obtain better performances than the previous results.

References

1. Bajard, J.C., Marrez, J., Plantard, T., Véron, P.: On Polynomial Modular Number Systems over $\mathbb{Z}/p\mathbb{Z}$. Advances in Mathematics of Communications **18**(3), 674–695 (Jun 2024). https://doi.org/10.3934/amc.2022018
2. Bernstein, D.J.: Curve25519: New diffie-hellman speed records. In: Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography. Lecture Notes in Computer Science, vol. 3958, pp. 207–228. Springer (2006). https://doi.org/10.1007/11745853_14
3. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehle, D.: Crystals - kyber: A cca-secure module-lattice-based kem. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 353–367 (2018). https://doi.org/10.1109/EuroSP.2018.00032
4. Bouvier, C., Imbert, L.: An alternative approach for sidh arithmetic. In: Garay, J.A. (ed.) Public-Key Cryptography – PKC 2021. pp. 27–44. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-75245-3_2
5. Campos, F., Chavez, J., Chi-Domínguez, J.J., Meyer, M., Reijnders, K., Rodríguez-Henríquez, F., Schwabe, P., Wiggers, T.: Optimizations and practicality of high-security csidh. IACR Communications in Cryptology (04 2024). https://doi.org/10.62056/anjbksdja
6. Castryck, W., Lange, T., Martindale, C., Panny, L., Renes, J.: Csidh: An efficient post-quantum commutative group action. In: Peyrin, T., Galbraith, S. (eds.) Advances in Cryptology – ASIACRYPT 2018. pp. 395–427. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_15
7. Coladon, T., Elbaz-Vincent, P., Hugounenq, C.: MPHELL: A fast and robust library with unified and versatile arithmetics for elliptic curves cryptography. In: ARITH 2021. Transactions on Emerging Topics in Computing, Torino, Italy (Jun 2021). https://doi.org/10.1109/ARITH51176.2021.00026
8. Didier, L.S., Robert, J.M., Dosso, F.Y., El Mrabet, N.: A software comparison of RNS and PMNS . In: 2022 IEEE 29th Symposium on Computer Arithmetic (ARITH). pp. 86–93. IEEE Computer Society, Los Alamitos, CA, USA (Sep 2022). https://doi.org/10.1109/ARITH54963.2022.00025
9. Didier, L.S., Dosso, F.Y., Véron, P.: Efficient modular operations using the Adapted Modular Number System. Journal of Cryptographic Engineering pp. 1–23 (2020). https://doi.org/10.1007/s13389-019-00221-7
10. Dosso, F.Y., Berzati, A., Mrabet, N.E., Proy, J.: Pmns revisited for consistent redundancy and equality test. Cryptology ePrint Archive, Paper 2023/1231 (2023), `https://eprint.iacr.org/2023/1231`
11. Dosso, F.Y., Duquesne, S., Mrabet, N.E., Gautier, E.: PMNS arithmetic for elliptic curve cryptography. Cryptology ePrint Archive, Paper 2025/467 (2025), `https://eprint.iacr.org/2025/467`
12. Dosso, F.Y., Mrabet, N.E., Méloni, N., Palma, F., Véron, P.: Friendly primes for efficient modular arithmetic using the polynomial modular number system. Cryptology ePrint Archive, Paper 2025/090 (2025), `https://eprint.iacr.org/2025/090`
13. Dosso, F.Y., Robert, J.M., Véron, P.: PMNS for efficient arithmetic and small memory cost. IEEE Transactions on Emerging Topics in Computing **10**(3), 1263–1277 (2022). https://doi.org/10.1109/tetc.2022.3187786
14. van Emde Boas, P.: Another np-complete problem and the complexity of computing short vectors in a lattice. Tech. Rep. 81-04, University of Amsterdam, Department of Mathematics,Netherlands (1981)

15. Hasan, M.A., Nègre, C.: Multiway splitting method for toeplitz matrix vector product. IEEE Transactions on Computers **62**, 1467–1471 (2013). https://doi.org/10.1109/TC.2012.95
16. Kaya Koc, C., Acar, T., Kaliski, B.: Analyzing and comparing montgomery multiplication algorithms. IEEE Micro **16**(3), 26–33 (1996). https://doi.org/10.1109/40.502403
17. Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische annalen **261**, 515–534 (1982). https://doi.org/10.1007/BF01457454
18. Mankowski, D., Wiggers, T., Moonsamy, V.: TLS $\rightarrow$ Post-Quantum TLS: Inspecting the TLS Landscape for PQC Adoption on Android . In: 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 526–538. IEEE Computer Society, Los Alamitos, CA, USA (Jul 2023). https://doi.org/10.1109/EuroSPW59978.2023.00065, `https://doi.ieeecomputersociety.org/10.1109/EuroSPW59978.2023.00065`
19. Méloni, N., Palma, F., Véron, P.: PMNS for Cryptography : A Guided Tour. Advances in Mathematics of Communications pp. 342–359 (2023). https://doi.org/10.3934/amc.2023033
20. Minkowski, H.: Zur Geometrie der Zahlen. Teubner, Leipzig (1905)
21. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard. Tech. Rep. Federal Information Processing Standards Publications (FIPS) 203, U.S. Department of Commerce, Washington, D.C. (2024). https://doi.org/10.6028/NIST.FIPS.203
22. Negre, C., Plantard, T.: Efficient modular arithmetic in adapted modular number system using lagrange representation. In: Information Security and Privacy, 13th Australasian Conference, ACISP 2008, Wollongong, Australia. pp. 463–477 (2008). https://doi.org/10.1007/978-3-540-70500-0_34
23. Nguyen, P.Q., Stehlé, D.: Lll on the average. In: Hess, F., Pauli, S., Pohst, M. (eds.) Algorithmic Number Theory. pp. 238–256. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11792086_18
24. Noyez, L., El Mrabet, N., Potin, O., Véron, P.: Modular multiplication in the AMNS representation : Hardware Implementation. In: Selected Areas in Cryptography. Montréal (Québec), France (Aug 2024), `https://hal.science/hal-04691484`, to appear in LNCS vol 15516
25. Paoloni, G.: White paper: How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures. Tech. rep., Intel Corporation (2010)
26. Schnorr, C.P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In: International Symposium on Fundamentals of Computation Theory. pp. 68–85. Springer (1991)
27. Westerbaan, B.: The state of the post-quantum internet (2024), https://blog.cloudflare.com/pq-2024/ last accessed 23 Oct 2024