Modular Polynomial Multiplication Using RSA/ECC coprocessor

Aurélien Greuet[1], Simon Montoya[1,2], and Clémence Vermeersch[1]

[1] IDEMIA, Cryptography & Security Labs, Courbevoie, France.
firstname.lastname@idemia.com
[2] LIX, INRIA, CNRS, École Polytechnique, Institut Polytechnique de Paris, France.
firstname.lastname@lix.polytechnique.fr

Keywords: Post-Quantum Lattice-based Cryptography · Modular Polynomial Multiplication · Embedded devices

Abstract. Modular polynomial multiplication is a core and costly operation of ideal lattice-based schemes. In the context of embedded devices, previous works transform the polynomial multiplication to an integer one using Kronecker substitution. Then thanks to this transformation, existing coprocessors which handle large-integer operations can be repurposed to speed-up lattice-based cryptography. In a nutshell, the Kronecker substitution transforms by evaluation the polynomials to integers, multiplies it with an integer multiplication and gets back to a polynomial result using a radix conversion. The previous work focused on optimization of the integer multiplication using coprocessor instructions. In this work, we pursue the seminal research by optimizing the evaluation, radix conversion and the modular reductions modulo q with today's RSA/ECC coprocessor. In particular we show that with a RSA/ECC coprocessor that can compute addition/subtraction, (modular) multiplication, shift and logical AND on integers, we can compute the whole modular polynomial multiplication using coprocessor instructions. The efficiency of our modular polynomial multiplication depends on the component specification and on the cryptosystem parameters set. Hence, we assess our algorithm on a chip for several lattice-based schemes, which are finalists of the NIST standardization. Moreover, we compare our modular polynomial multiplication with other polynomial multiplication techniques.

1 Introduction

In the next few years, a quantum computer powerful enough to run Shor's algorithm [17] could emerge. Such a computer can break the entire cryptography based on the hardness of integer factorization and discrete logarithm like RSA or Elliptic Curve Cryptography (ECC). Due to this potential threat, national agencies started to study new proposals (e.g. [6]) and initiated standardization of quantum safe algorithms [14,8]. The most followed standardization by the community is the one of the National Institute of Standards and Technology (NIST), which was launched in 2016 [14]. This standardization aims to bring together an

important part of the community to determine future Key Encapsulation Mechanisms (KEMs) and signatures standards. In July 2020, the third round of this standardization started with seven finalists remaining, including four KEMs and three signatures. Among these seven finalists, five are based on lattice or assimilated problems [15]. Hence, the international community around post-quantum cryptography is very likely to include lattice-based standards. Therefore, optimizing and ensuring practical security of these cryptosystems is an important area of research.

Post-quantum cryptography will be deployed on embedded devices. On such devices, the amount of RAM or the CPU frequency are very limited: less than 60 kB of RAM and less than 100 MHz. Therefore, implementing efficient cryptosystems in these constrained environments is a real challenge. In order to speed-up the cryptographic algorithms, these devices may embed additional hardware coprocessors for symmetric and asymmetric cryptographic computations. Moreover, these coprocessors can provide additional security features as hardware and software security against faults and side-channel leakage. Most of the asymmetric coprocessors currently deployed are designed for the ECC or RSA schemes and not for lattice-based cryptosystems. However, the underlying arithmetic of these cryptosystems can be tweaked with the purpose of using an arithmetic close to the one used on RSA/ECC schemes. Therefore, re-purposing such asymmetric coprocessors is interesting to optimize lattice-based schemes and to facilitate the transition in the post-quantum world. Indeed, the easier the transition, the more it will be used and deployed.

Motivations & previous works. Lattice-based cryptography is believed to be a promising direction to provide efficient and secure post-quantum algorithms. One of the main operation in these schemes is *modular polynomial multiplication*. Researches have been conducted in the way of optimizing the polynomial multiplication operation using specific software instructions or by designing a specific hardware. However, most of the polynomial multiplication optimization are intended to ARM-Cortex M4, or less frequently to ARM-Cortex M3 CPU architecture. The ARM CPU is powerful and has a larger panel of interesting assembly instruction. However, all the embedded systems do not have such a powerful CPU and base their cryptographic efficiency on the additional coprocessor.

Moreover, the transition period should rely on *hybrid cryptography* which is the combination of a post-quantum algorithm and a classical one. Hence, such cryptography is both secure against quantum attacks, thanks to the post-quantum part, and secure against classical attacks, with at least the same security level as a pure classical crypto algorithm. Several governmental agencies (NIST, ANSSI, BSI) recommend and will impose in a few years the use of hybrid cryptography for long term security certification [2,6]. In this context, re-purposing the current asymmetric coprocessors to optimize the modular polynomial multiplication is of interest in terms of costs, ease of deployment and to propose optimization for a wide range of components.

The seminal work of Albrecht *et al.* in [1] re-purposes a RSA/ECC coprocessor to optimize polynomial multiplication on Kyber algorithm. To do so, they use techniques introduced in [11] which transform polynomial multiplication to an integer one using the Kronecker substitution [12]. Afterwards, another work in [18] adapts the previous technique on Saber algorithm.

The work of *Bos et al.* in [5] introduced `Kronecker+`, a generalization of the Kronecker substitution used by Albrecht *et al.* in [1]. This generalization allows trade-off between number of integer multiplications, size of the integers and the number of polynomial evaluations. Depending on the component and coprocessor specifications, `Kronecker+` allows a faster polynomial multiplication than Kronecker substitution.

In [10], the authors provide a variant of the Kronecker substitution and an adaptation of the schoolbook multiplication to perform hardware polynomial multiplication. Depending on the RSA/ECC coprocessor specifications, one of these algorithms can outperform the classical Kronecker substitution.

Our contribution. This work aims to perform modular polynomial multiplication in $R_{q,\delta} = \mathbb{Z}_q[X]/(X^N + \delta)$ using a RSA/ECC coprocessor, where $\delta \in \{-1, 1\}$. These rings are the most used by the lattice-based finalists of the NIST standardization.

The contemporary asymmetric coprocessor can perform integer operations and not polynomial ones. As we have seen previously, most techniques to repurpose current coprocessor to optimize polynomial multiplication on embedded devices are based on the Kronecker substitution. In $R_{q,\delta}$ this substitution can be summarized in four steps:

1. Convert polynomials in $R_{q,\delta}$ to integers in $\mathbb{N}$ of bit size `bitsize`. When polynomials have coefficients with a negative representation, this conversion requires additional operations.
2. Modular integer multiplication mod $2^{\texttt{bitsize}} + \delta$ of the obtained integers.
3. Convert back integer multiplication result to a polynomial in $\mathbb{Z}[X]/(X^N + \delta)$. Like Step 1, if the initial polynomials have coefficients with a negative representation this conversion requires additional operations.
4. Reduce the coefficients modulo q to have result over $R_{q,\delta}$.

All the previous works re-purpose the coprocessor only to optimize Step 2. All the other steps are implemented in software without the use of coprocessor.

In this work for most of the previous steps, we describe algorithms which allow to re-purpose existing coprocessor. Our work focuses on two mains contributions:

- Handle negative evaluation and radix conversion using RSA/ECC coprocessor (Steps 1 and 3).
- Perform modular reduction of the coefficients modulo q with a RSA/ECC coprocessor (Step 4).

These improvements are possible only if the coprocessor can handle the following integer operations: addition/subtraction, bitwise `AND`, logical shift, multiplication

and modular multiplication. Except the logical `AND` operation, most of current asymmetric coprocessors handle these operations. The logical `AND` is less common on the current RSA/ECC coprocessor. However adding this operation to an existing architecture is easier and cheaper than designing a new one for polynomial multiplication.

Organization. In Section 2 we introduce notations which we use in the rest of the paper. In Section 3 we present how to perform a polynomial multiplication in $\mathbb{N}[X]$ using the *Kronecker substitution*. Afterwards, in Section 4 we explain how to use the coprocessor instructions to perform the *Kronecker substitution* evaluation and radix conversation, since the polynomials are in $R_{q,\delta} = \mathbb{Z}_q[X]/(X^N+\delta)$, where $\delta \in \{-1, 1\}$. In Section 5 we describe modular reductions modulo q using coprocessor instructions. Finally, in Section 6 we present the results of our practical implementations of our algorithms on several lattice-based finalists.

2 Background

RSA/ECC coprocessor. The RSA/ECC coprocessor are designed to speed-up RSA or elliptic curves cryptosystems. To do so, these components provide a range of integer operations. In this work, we assume that we have access to a component which can perform, at least, addition/subtraction, bitwise `AND`, logical shift, multiplication and modular multiplication operations.

2.1 Element representation

Integers representation. Let $a \in \mathbb{N}$ such that $0 \leq a < 2^\ell$. In the following, we say that a is represented over ℓ bits to mean that a is stored in a machine buffer of ℓ bits.

Let $b \in \mathbb{Z}$ such that $-2^{\ell'-1} < b < 2^{\ell'-1}$. Let $\tilde{b}$ be the two's complement representation of b over ℓ' bits, defined by:

$$\tilde{b} = 2^{\ell'} + b \mod 2^{\ell'} \in \mathbb{N}$$

In the following, we say that b is represented over ℓ' bits to mean that the two's complement representation of b is stored in a machine buffer of ℓ' bits.

Let r be a $N\ell$-bit natural number. We denote by r_i the i-th digit of r in base 2^ℓ. In other words, $r = \sum_{i=0}^{N-1} r_i 2^{i\ell}$ with $0 \leq r_i < 2^\ell$. We use the following notation $r = (r_0, r_1, \ldots, r_{N-1})_\ell$.

Polynomial representation. Let $F(X) = f_0 + f_1 X + \ldots + f_{N-1}X^{N-1} \in \mathbb{Z}[X]$ of degree at most $N-1$. Let $\tilde{f}_i$ be a two's complement representation of a coefficient f_i.

Array representation. The usual machine representation of $F(X)$ is an array where the i-th item is $\tilde{f}_i$. To ease the reading, we denote in the following, f_i or $f[i]$ the coefficient associated to the i-th item. Moreover, unless otherwise specified, a polynomial is represented as an array.

Packed integer representation. A packed integer representation of $F(X)$ is the concatenation of all the $\tilde{f}_i$ into a buffer.

$$f = \tilde{f}_{N-1}|\ldots|\tilde{f}_1|\tilde{f}_0 \in \mathbb{N}$$

In this work, this representation is used to represent polynomials into a natural number. Afterwards, the polynomial arithmetic is carried out with operations on this natural number.

2.2 Notations

Rings. Let q be an integer. Denote by $R_{q,\delta}$ the polynomial ring $\frac{\mathbb{Z}_q[X]}{(X^N+\delta)}$, where $\delta \in \{-1, 1\}$. We represent an element $F(X) \in R_{q,\delta}$ as a polynomial of degree at most $N-1$ with coefficients in $\{0, \ldots, q-1\}$. $R^-_{q,\delta}$ denotes the elements of $R_{q,\delta}$ represented by a polynomial of degree at most $N-1$ with coefficients in $\{-\frac{q}{2}-1, \ldots, \frac{q}{2}\}$.

Integer operations. In the sequel, the algorithms are described using the following notations. Their purpose is to clarify the size of the manipulated operands.

- Let **add**(a,b,`bitlen`) (resp. **sub**(a,b,`bitlen`)) be the addition (resp. subtraction) between a and b. The values a and b are represented over `bitlen` bits.
- Let **lshift**(a,k,`bitlen`) (resp. **rshift**(a,k,`bitlen`)) be the left (resp. right) shift $a << k$ (resp. $a >> k$) over `bitlen` bits.
- Let **and**(a,b,`bitlen`) be the AND operation $a\&b$ over `bitlen` bits.
- Let **mult**(a,b,$\texttt{bitlen}_a$,$\texttt{bitlen}_b$) be the integer multiplication $a \times b$ where a (resp. b) is represented on $\texttt{bitlen}_a$ (resp. $\texttt{bitlen}_b$) bits.
- Let **modMult**(a,b,$\texttt{bitlen}_a$,$\texttt{bitlen}_b$,p) be the integer modular multiplication $a \times b \mod p$ where a (resp. b) is represented on $\texttt{bitlen}_a$ (resp. $\texttt{bitlen}_b$) bits.

Concatenation. Let $(\ell, k, N) \in \mathbb{N}^3$ with $\ell \leq k$ and $m \in \mathbb{N}$ represented over ℓ bits. In the following we denote by **concat**(m, k, N) the function that represents m on k bits and concatenates this new representation N times. Formally:

$$\textbf{concat}(m, k, N) = \sum_{j=0}^{N-1} m2^{jk} \in \mathbb{N}$$

Example 1. Let $m = 1$ then **concat**$(m, 8, 3) = 0\text{x}10101$.

Integer to polynomial. Let $(\ell, k, N) \in \mathbb{N}^3$, $\ell > k$ and $F(X) = f_0 + \ldots + f_{N-1}X^{N-1} \in \mathbb{Z}[X]$. For all i, let $\tilde{f}_i$ be the two's complement representation of f_i over k bits. We denote by:

$$f = \textbf{polyToN}(F(X), k, \ell) = \sum_{i=0}^{N-1} \tilde{f}_i 2^{i\ell}, f \in \mathbb{N}$$

Let $g = (g_0, g_1, \ldots, g_{N-1})_\ell \in \mathbb{N}$ a $N\ell$-bit number.

$$G(X) = \mathbf{NtoPoly}(g, \ell) = \sum_{i=0}^{N-1} g_i X^i$$

The obtained polynomial $G(X)$ belongs to $\mathbb{N}[X]$ and its degree is at most $N-1$.

Example 2. Let $F(X) = f_2X^2 + f_1X + f_0 = 2X^2 + 4X - 2$. Let $\tilde{f}_0 = 0\text{x}\text{E}$, $\tilde{f}_1 = 0\text{x}4$, $\tilde{f}_2 = 0\text{x}2$, be representations of all f_i over 4 bits. Then, $f = \mathbf{polyToN}(F(X), 4, 8) = 0\text{x}02040\text{E}$ and $\mathbf{NtoPoly}(f, 8) = 2X^2 + 4X + 14$

3 Multiplication in $\mathbb{N}[X]$ using Kronecker substitution

The Kronecker substitution was first introduced in [12]. We give here the main steps of this substitution. The idea of this substitution is to transform a polynomial multiplication to an integer one by evaluating the polynomials and get back to the result using a radix conversion. In the context of embedded devices, this transformation is of interest to perform polynomial multiplication by using the RSA/ECC coprocessor. Indeed, such coprocessor handles multiplication on integers. In this section we assume that our polynomials are defined over $\mathbb{N}[X]$.

3.1 Kronecker substitution

The Kronecker substitution multiplies two polynomials $F(X)$ and $G(X)$ using an integer multiplication. This substitution can be summarized in three steps:

1. Evaluation of $F(X)$ and $G(X)$ at 2^ℓ. The value ℓ is chosen such that all the coefficients after the polynomial multiplication are lower than 2^ℓ.
2. Integer multiplication $r = F\left(2^\ell\right) G\left(2^\ell\right), r \in \mathbb{N}$.
3. Get back to polynomial $R(X) \in \mathbb{N}[X]$ using radix conversion on r.

Evaluation. The first step of the Kronecker substitution is the polynomial evaluation at 2^ℓ. Since $F(X)$ has coefficients in $\mathbb{N}$ represented over k bits:

$$\textsc{Evaluation}_{\geq 0}(F(X), k, \ell) \coloneqq F\left(2^\ell\right) = \mathbf{polyToN}(F(X), k, \ell) \tag{1}$$

Example 3. Let $F(X) = 2X^2 + X + 3$ then, $F(2^8) = 0\text{x}020103 = \textsc{Evaluation}_{\geq 0}(F(X), 2, 8)$

Evaluation point. Let $R(X) = F(X)G(X)$ where $F(X), G(X) \in \mathbb{N}[X]$ of degree at most $N-1$. The evaluation point 2^ℓ is chosen such that for all $i \leq 2(N-1)$:

$$r_i \leq \max_{j \in \{0,\ldots,N-1\}}(f_j) \max_{j \in \{0,\ldots,N-1\}}(g_j) N < 2^\ell$$

By the fact that all the coefficients are non-negative, this evaluation is only a representation of all the f_i over ℓ bits. Then in an implementation, the evaluation does not require arithmetic operations.

Radix Conversion. Radix conversion aims to transform an integer into a polynomial. Let $f = (f_0, \ldots, f_{N-1})_\ell \in \mathbb{N}$, then:

$$F(X) = f_0 + \ldots + f_{N-1}X^{N-1} \coloneqq \textsc{Radix Conversion}_{\geq 0}(f) = \mathbf{NtoPoly}(f, \ell) \tag{2}$$

Example 4. Let f = 0x020103 then $F(X) = 2X^2 + X + 3$ = $\textsc{Radix Conversion}_{\geq 0}(f)$

The radix conversion converts a packed integer representation to an array one. Like the evaluation algorithm, in an implementation, the radix conversion does not require arithmetic operation.

Example of Kronecker substitution.

Example 5. Let $F(X) = 2X^2 + X + 3$ and $G(X) = X^2 + 1$. Then,

$$F(2^8) = \text{0x020103} = \textsc{Evaluation}_{\geq 0}(F(X), 2, 8)$$
$$G(2^8) = \text{0x010001} = \textsc{Evaluation}_{\geq 0}(G(X), 2, 8)$$

Afterwards we multiply the evaluated polynomials $r = F(2^8)G(2^8)$ = 0x201050103. Finally we obtain $R(X) = \textsc{Radix Conversion}_{\geq 0}(r) = 2X^4 + X^3 + 5X^2 + X + 3$.

4 Multiplication in $R_{q,\delta}$ using Kronecker substitution

In the previous section we perform polynomial multiplication as an integer one with polynomials in $\mathbb{N}[X]$. However, in the lattice-based schemes some polynomials, mainly the secret ones, have coefficients with a negative representation close to 0. Moreover, the reduction modulo $X^N + 1$ can also bring negative coefficients. Then in this section we focus on polynomial multiplication in $R_{q,\delta} = \mathbb{Z}_q[X]/(X^N + \delta)$. In $R_{q,\delta}$, the polynomial multiplication using Kronecker substitution is achieved as follows:

- Evaluation of polynomials considering negative coefficients.
- Integer multiplication modulo $2^{N\ell} + \delta$. The modular reduction ensures that after radix conversion the polynomial result is reduced modulo $X^N + \delta$.
- Radix conversion to obtain a polynomial in $\mathbb{Z}[X]/(X^N + \delta)$.
- Reduction modulo q of the polynomial coefficients.

Previous works [1,10,5] already achieve the evaluation and the radix conversion with negative coefficients. However, these algorithms are done using array representations. In this section we describe a way to realize these algorithms when the coefficients are on a packed integer representation. The main advantage of this representation is that it allows the use of existing coprocessor.

Negative representation. Our goal is to perform polynomial multiplication over $R_{q,\delta}$. Then, a way to avoid the negative coefficients is to represent them with a non-negative representation over $R_{q,\delta}$. However, the negative coefficients are close to 0, then the closest non-negative representation is nearby q. This involves that the evaluation point must be higher and then the integer operation are done on much larger integer; see [10] for more details on the impact on the evaluation point. Thus, for the sake of efficiency we use, when possible, our algorithms with the negative representation.

4.1 Evaluation with negative coefficients.

Let $F(X) = f_0 + f_1 X + \ldots + f_{N-1} X^{N-1} \in R^-_{q,\delta}$ and $\tilde{f}_i$ be the two's complement representation over k bits of f_i. Our goal is to evaluate $F(X)$ at 2^ℓ where $\ell > k$, then for $i = 0$ to $N-1$:

- If $f_i \geq 0$, then we only have to represent it on ℓ bits (as in Section 2).
- If $f_i < 0$, then we have to represent it with a two's complement over ℓ bits and propagate a borrow to the next coefficient. To obtain a two's complement representation from k bits to ℓ bits, we compute:

$$\tilde{f}_i + (2^\ell - 2^k) = 2^k + f_i + (2^\ell - 2^k) = 2^\ell + f_i$$

The Algorithm 1 computes the two's complement representation of the polynomial evaluation when the coefficients are in $\mathbb{Z}$. More precisely, this evaluation is done using arithmetic operations on a packed integers representation. To do so, we first represent the polynomial coefficients into a packed integers form, as defined in Equation 1. Afterwards, we use arithmetic operations in order to convert the two complement's representation from k to ℓ bits and to propagate the required borrows.

Algorithm 1 Evaluation

Input: $F(X) \in R^-_{q,\delta}$, $k, \ell \in \mathbb{N}$ where $\ell > k$.
Output: $\tilde{f} \in \mathbb{N}$ the two's complement representation of $F\left(2^\ell\right) \mod 2^{N\ell}$

1: $\texttt{mask} \leftarrow \textbf{concat}(1, \ell, N)$ //Precomputed
2: $\tilde{f} \leftarrow \textbf{polyToN}(F(X), k, \ell)$
3: $\texttt{neg} \leftarrow \textbf{rshift}(\tilde{f}, k-1, N\ell)$
4: $\texttt{neg} \leftarrow \textbf{and}(\texttt{neg}, \texttt{mask}, N\ell)$ // Detect negative coefficients
5: $\texttt{tmp} \leftarrow \textbf{mult}(\texttt{neg}, 2^\ell - 2^k, N\ell, 32)$
6: $\tilde{f} \leftarrow \textbf{add}(\tilde{f}, \texttt{tmp}, N\ell)$ // Two's complement representation of each coeff over ℓ bits
7: $\texttt{neg} \leftarrow \textbf{lshift}(\texttt{neg}, \ell, N\ell)$
8: $\tilde{f} \leftarrow \textbf{sub}(\tilde{f}, \texttt{neg}, N\ell)$ // Borrow propagation
9: **return** $\tilde{f}$

Remark 1. The value `mask` is always the same for a fixed scheme. Then, this integer can be precomputed and stored in Non-Volatile Memory (NVM).

Remark 2. The EVALUATION (Algorithm 1) returns the two's complement representation of $F\left(2^{\ell}\right) \mod 2^{N\ell}$. This implies:

- If $F\left(2^{\ell}\right) \geq 0$, then the returned value is equal to $F\left(2^{\ell}\right)$.
- Otherwise, the returned value is not equal to $F\left(2^{\ell}\right)$. This case occurs when the latest non-zero coefficient of $F(X)$ is negative.

To obtain the expected result after the Kronecker Substitution, the last case requires additional operations before the radix conversion. These additional operations are described in Section 4.2 paragraph **Two's complement representation of the evaluated polynomial**.

Example 6. Let $F(X) = 3X^2 - 2X + 2$, where all the coefficients are encoded with a two's complement representation over $k = 4$ bits. Let $N = 3$ and $\ell = 8$. The expected result is $F(2^8) = 0\text{x}02\text{FE}02$. This is obtained with EVALUATION$(F(X), k, \ell)$:

1. $\texttt{mask} \leftarrow \textbf{concat}(1, 8, 3) = 0\text{x}010101$
2. $\tilde{f} \leftarrow \textbf{polyToN}(F(X), 4, 8) = 0\text{x}030\text{E}02$
3. $\texttt{neg} \leftarrow \textbf{rshift}(\tilde{f}, 4-1, 3 \times 8) = 0\text{x}0061\text{C}0$
4. $\texttt{neg} \leftarrow \textbf{and}(\texttt{neg}, \texttt{mask}, 3 \times 8) = 0\text{x}000100$
5. $\texttt{tmp} \leftarrow \textbf{mult}(\texttt{neg}, 2^8 - 2^4, 3 \times 8, 32) = 0\text{x}00\text{F}000$
6. $\tilde{f} \leftarrow \textbf{add}(\tilde{f}, \texttt{tmp}, 3 \times 8) = 0\text{x}03\text{FE}02$
7. $\texttt{neg} \leftarrow \textbf{lshift}(\texttt{neg}, 8, 3 \times 8) = 0\text{x}010000$
8. $F(2^8) \leftarrow \textbf{sub}(\tilde{f}, \texttt{neg}, 3 \times 8) = 0\text{x}02\text{FE}02$

Evaluation point. Let $R(X) = F(X)G(X)$ where $F(X), G(X) \in R_{q,\delta}$. The evaluation point 2^{ℓ} is chosen such that for all $i \leq 2(N-1)$:

$$r_i \leq \max_{j \in \{0,\ldots,N-1\}}(|f_j|) \max_{j \in \{0,\ldots,N-1\}}(|g_j|)N < 2^{\ell-1}$$

4.2 Radix Conversion with negative coefficient representation.

As mentioned in [1,10], the radix conversion has to be adapted since some coefficients have negative representations. Two issues arise with the negative coefficients:

1. The evaluation and the integer multiplication propagate borrow between the polynomial coefficients.
2. The negative evaluation algorithm returns two's complement representation over $N\ell$ bits.

Borrow between the coefficients. The evaluation converts a polynomial to a packed integers representation. In the following of the Kronecker substitution, the obtained natural numbers are manipulated regardless the original polynomial structure. Therefore, borrows can be propagated between the coefficients.

However in order to retrieve the expected polynomial result, the radix conversion must compensate the propagated borrows by propagating back carries.

Let $\tilde{r} = (\tilde{r}_0, \tilde{r}_1, \ldots, \tilde{r}_{N-1})_\ell \in \mathbb{N}$ be the integer that we want to convert to a polynomial, where for all i, $\tilde{r}_i$ is a two's complement representation over ℓ bits of an integer $-2^{\ell-1} < r_i < 2^{\ell-1}$. In order to propagate back the carries, we transform the negative coefficients to non-negative ones by adding a multiple of our modulus q: `maxValue`. More precisely, `maxValue` is the smallest multiple of q such that for all i, $-\texttt{maxValue} \leq r_i < \texttt{maxValue}$. Moreover with the parameters that we use in Section 6, we have $\texttt{maxValue} < 2^{\ell-1}$. Then, by adding `maxValue` we got:

- If $r_i < 0$, then $2^\ell \leq \tilde{r}_i + \texttt{maxValue} = 2^\ell + r_i + \texttt{maxValue}$. Therefore a carry is propagated to $\tilde{r}_{i+1}$.
- If $r_i \geq 0$, then $\tilde{r}_i + \texttt{maxValue} = r_i + \texttt{maxValue} < 2^\ell$.

After adding `maxValue`, the values r_i are considered as natural numbers represented over ℓ bits. Then, the expected polynomial is obtained by using the radix conversion algorithm defined in Equation 2 on $\tilde{r}$.

This negative to non-negative conversion is possible because the polynomial multiplication is done over $R_{q,\delta}$. Indeed after reduction modulo q, the added value `maxValue` is equal to 0.

Two's complement representation of the evaluated polynomial. The second issue is due to the two's complement representation of the evaluated polynomial.

Let $F(X) = f_0 + \ldots + f_{N-1}X^{N-1} \in R^-_{q,\delta}$ of degree $N-1$ and $\ell \in \mathbb{N}$. Then Algorithm 1 returns the integer $f \leftarrow \textsc{Evaluation}(F(X), k, \ell)$, that is the two's complement representation of $F\left(2^\ell\right) \mod 2^{N\ell}$. Two cases are to be distinguished:

- $f_{N-1} > 0$, then $f = F\left(2^\ell\right) \in \mathbb{N}$.
- $f_{N-1} < 0$, then $f = 2^{N\ell} + F\left(2^\ell\right)$ is the two's complement of $F\left(2^\ell\right)$ modulo $2^{N\ell}$.

Only the second case will lead to a wrong result after the modular multiplication. Indeed, let $g \in \mathbb{N}$ and $f = 2^{N\ell} + F\left(2^\ell\right)$ we got:

$$\begin{aligned} r \mod \left(2^{N\ell}+\delta\right) = fg \mod \left(2^{N\ell}+\delta\right) &= 2^{N\ell}g + F\left(2^\ell\right)g \mod \left(2^{N\ell}+\delta\right) \\ &\neq F\left(2^\ell\right)g \mod \left(2^{N\ell}+\delta\right) \end{aligned}$$

Then in this case, before the radix conversion we must add or subtract g to r, depending on δ:

- $\delta = 1 : 2^{N\ell}g \mod \left(2^{N\ell}+1\right) = -g \mod \left(2^{N\ell}+1\right)$, then
$$r + g \mod \left(2^{N\ell}+1\right) = F\left(2^\ell\right)g \mod \left(2^{N\ell}+1\right)$$
- $\delta = -1 : 2^{N\ell}g \mod \left(2^{N\ell}-1\right) = g \mod \left(2^{N\ell}-1\right)$, then
$$r - g \mod \left(2^{N\ell}-1\right) = F\left(2^\ell\right)g \mod \left(2^{N\ell}-1\right)$$

Previously, we supposed that at most one polynomial can have negative coefficients. In case of lattice-based schemes, this is always the case.

Algorithm 2 RADIX CONVERSION

Input: $r, g, \texttt{maxValue} \in \mathbb{N}$, and $\texttt{sign} \in \{0, 1\}$
Output: $R(X) \in \mathbb{N}[X]/(X^N + \delta)$
1: $\texttt{max} \leftarrow \textbf{concat}(\texttt{maxValue}, \ell, N)$ //Can be precomputed
2: **if** sign eq 1 **then**
3: **if** δ eq 1 **then** $r \leftarrow \textbf{add}(r, g, N\ell)$ // To handle negative last coeff
4: **else** $r \leftarrow \textbf{sub}(r, g, N\ell)$
5: **else**
6: **if** δ eq 1 **then** $\texttt{dummy} \leftarrow \textbf{add}(r, g, N\ell)$ // For isochrony
7: **else** $\texttt{dummy} \leftarrow \textbf{sub}(r, g, N\ell)$
8: **end if**
9: $r \leftarrow \textbf{add}(r, \texttt{max}, N\ell)$ // Add maxValue to each coefficient
10: $R(X) \leftarrow$ RADIX CONVERSION$_{\geq 0}(r)$

4.3 Multiplication in $R_{q,\delta}$ using coprocessor

The Sections 4.1 and 4.2 are used to obtain a polynomial multiplication algorithm in $R_{q,\delta}$ using, mainly, a packed integer representation. More precisely, except for the modular reductions modulo q, the operations are done using this representation. All operations performed on the packed integers representation can be achieved with coprocessor as defined in Section 2.

The Polynomial Multiplication in $R_{q,\delta}$ algorithm is described in Algorithm 3.

Algorithm 3 POLYNOMIAL MULTIPLICATION IN $R_{q,\delta}$

Input: $(F(X), G(X)) \in (R^-_{q,\delta}, R_{q,\delta})$ of degree $N-1$. Let $k, \ell, q \in \mathbb{N}$ where $\ell > k$, and maxValue defined as above.
Output: $R(X) = F(X)G(X) \in R_{q,\delta}$
1: $f \leftarrow$ EVALUATION$(F(X), k, \ell)$
2: $G\left(2^\ell\right) \leftarrow$ EVALUATION$_{\geq 0}(G(X), k, \ell)$
3: $r \leftarrow \textbf{modMult}(f, G\left(2^\ell\right), N\ell, N\ell, 2^{N\ell} + \delta)$
4: $b \leftarrow \texttt{sign}(F[N-1])$ // if $F_{N-1} < 0$ then $b = 1$, otherwise $b = 0$.
5: $R(X) \leftarrow$ RADIX CONVERSION$(r, G\left(2^\ell\right), \texttt{maxValue}, b)$
6: $R(X) \leftarrow R(X) \mod q$ // Any modular reduction
7: **return** $R(X)$

In the following section we determine how to perform modular reductions modulo q using packed integers representation.

5 Reducing coefficients modulo q

In Section 4.3, we perform polynomial multiplication in $R_{q,\delta}$. However, the reduction modulo q is done after the radix conversion on a polynomial representa-

tion. In this section we show how to perform reduction modulo q using packed integers representation. As mentioned previously, such representation allows to repurpose existing RSA/ECC coprocessor.

Let $r = (r_0, \ldots, r_{N-1})_\ell \in \mathbb{N}$. In our context, r is obtained after the two first steps of the Kronecker substitution: polynomial evaluation and modular integer multiplication. Moreover, we have added `maxValue` like in Section 4.2. Then, each r_i is such that for all i: $0 \leq r_i < 2^{\texttt{maxValue}}$.

In the following we denote by *simultaneous reduction*, the fact of reducing all the $r_i \mod q$ by performing operations on r.

5.1 Power-of-two modulus

Some of lattice-based schemes, like Saber [9] and NTRU [7], use a power-of-two modulus. In this context, the simultaneous reduction is easy and fast. Indeed, the simultaneous reduction is achieved by the computation:

$$r \& \mathbf{concat}(q-1, \ell, N)$$

5.2 Prime modulus

Kyber [4] is a lattice-based KEM which perform polynomial multiplication over $R_{q,\delta}$, where q is a prime number. In this section we adapt Barrett [3] reduction to perform simultaneous reduction.

Barrett The Barrett reduction is introduced in [3]. The main idea is to precompute an approximation of a division and use it to perform modular reduction. Let $\alpha, \beta \in \mathbb{Z}$ and $a \in \mathbb{N}$ be an integer to reduce modulo $q \in \mathbb{N}$ of bit-length $k \in \mathbb{N}$. Barrett reduction precomputes $m = \left\lfloor \frac{2^{k+\alpha}}{q} \right\rfloor$ and computes:

$$a' = a - [((a >> (k+\beta)) \cdot m) >> (\alpha - \beta)]\, q$$

A special case is when $\alpha = \beta$, therefore the computation is

$$a' = a - [a >> (k+\beta)] \cdot m \cdot q$$

In this case, only one shift and one multiplication is performed ($m \cdot q$ is precomputed).

Depending on the parameters (α, β), $a' = a \mod q + tq$ where $0 \leq t < \left\lfloor \frac{a}{q} \right\rfloor$. Further details on the Barrett algorithms are given in [13].

Simultaneous modular reduction. To adapt this reduction to simultaneous reduction we need to perform logical `AND` after the shift operations. Indeed, because of the shift operations, noise coming from coefficient $i+1$ can overflow on

the coefficient i. The Algorithm 4 describes the simultaneous Barrett reduction.

Algorithm 4 SIMULT. BARRETT$_{\alpha,\beta}$

Input: $r = (r_0, \ldots, r_{N-1})_\ell \in \mathbb{N}$. Let $q \in \mathbb{N}$ of bit-length k and $m = \left\lfloor \frac{2^{k+\alpha}}{q} \right\rfloor$.
Output: $r' = (r'_0, \ldots, r'_{N-1})_\ell \in \mathbb{N}$, all r_i are reduced with BARRETT reduction
1: $\texttt{mask} \leftarrow \textbf{concat}(2^{\ell-\alpha+\beta} - 1, \ell, N)$ // Can be precomputed
2: $\texttt{mask}' \leftarrow \textbf{concat}(2^{\ell-k-\beta} - 1, \ell, N)$ // Can be precomputed
3: $\texttt{tmp} \leftarrow \textbf{rshift}(r, k+\beta, N\ell)$
4: $\texttt{tmp} \leftarrow \textbf{and}(\texttt{tmp}, \texttt{mask}', N\ell)$
5: $\texttt{tmp} \leftarrow \textbf{mult}(\texttt{tmp}, m, N\ell, 32)$ // Mult between a word and a large integer
6: $\texttt{tmp} \leftarrow \textbf{rshift}(\texttt{tmp}, \alpha-\beta, N\ell)$
7: $\texttt{tmp} \leftarrow \textbf{and}(\texttt{tmp}, \texttt{mask}, N\ell)$
8: $\texttt{tmp} \leftarrow \textbf{mult}(\texttt{tmp}, q, N\ell, 32)$ // Mult between a word and a large integer
9: $r' \leftarrow \textbf{sub}(r, \texttt{tmp}, N\ell)$
10: **return** r'

Final reduction Using the simultaneous Barrett Algorithm 4, the returned result $r' = (r'_0, \ldots, r'_{N-1})_\ell \in \mathbb{N}$ is such that, for all i, $r'_i = r'_i \mod q + t_i q$. With the parameters sets that we use in Section 6, for all i, $t_i \in \{0, 1, 2\}$.

Let k and c such that $q = 2^k - c$. Then $r'_i \geq 2q$ if and only if $r'_i + 2c$ has its $(k+1)$-th bit equal to one. This fact is used in Algorithm 5 to detect and subtract q to coefficients $\geq 2q$ in a packed integers representation.

After using the Algorithm 5, the r''_i are bounded by $2q$. In that case, this algorithm can be adapted replacing $2c$ by c (line 1) and $k+1$ by k (line 3). It follows that q is subtracted from each $r''_i \geq q$. Afterwards, each r''_i is necessary lower than q.

Algorithm 5 SIMULT. CONDITIONAL SUBTRACTION

Input: $r' = (r'_0, \ldots, r'_{N-1})_\ell$ with all $0 \leq r'_i < 3q$, where $q = 2^k - c$, $\ell, N \in \mathbb{N}$.
Output: $r'' = (r''_0, \ldots, r''_{N-1})_\ell$ with all $0 \leq r''_i < 2q$
1: $(\texttt{C}, \texttt{mask}) \leftarrow (\textbf{concat}(2c, \ell, N), \textbf{concat}(1, \ell, N))$ //Can be precomputed
2: $\texttt{tmp} \leftarrow \textbf{add}(r', \texttt{C}, N\ell)$ //Raised the $k+1$-th bit in each coeff
3: $\texttt{tmp} \leftarrow \textbf{rshift}(\texttt{tmp}, k+1, N\ell)$ // Move the $k+1$-th bit to position 0 in each coeff
4: $\texttt{tmp} \leftarrow \textbf{and}(\texttt{tmp}, \texttt{mask}, N\ell)$ // Detect the coeff $\geq 2q$
5: $\texttt{tmp} \leftarrow \textbf{mult}(\texttt{tmp}, q, N\ell, 32)$ // Mult between a word and a large integer
6: $r'' \leftarrow \textbf{sub}(r', \texttt{tmp}, N\ell)$ // Subtract q to each coeff $\geq 2q$
7: **return** r''

5.3 Modular polynomial multiplication using coprocessor

The Algorithm 6 performs polynomial multiplication in $R_{q,\delta}$ using operations on packed integers representation. All operations performed on this representation can be achieved with coprocessor as defined in Section 2.

Algorithm 6 Modular Polynomial Multiplication

Input: $(F(X), G(X)) \in (R^-_{q,\delta}, R_{q,\delta})$ of degree $N-1$. Let $k, \ell, q \in \mathbb{N}$ where $\ell > k$, and `maxValue` defined as above.
Output: $R(X) = F(X)G(X) \in R_{q,\delta}$

1: $\texttt{max} \leftarrow \textbf{concat}(\texttt{maxValue}, \ell, N)$ // Precomputed
2: $\left(f, G\left(2^\ell\right)\right) \leftarrow (\text{Evaluation}(F(X), \ell), \text{Evaluation}_{\geq 0}(G(X), \ell))$
3: $r \leftarrow \textbf{modMult}(f, G\left(2^\ell\right), N\ell, N\ell, 2^{N\ell} \pm \delta)$
4: $b \leftarrow \texttt{sign}(f[N-1])$
5: **if** b eq 1 **then**
6: **if** δ eq 1 **then** $r \leftarrow \textbf{sub}(r, G\left(2^\ell\right), N\ell)$ // To handle negative last coeff
7: **else** $r \leftarrow \textbf{add}(r, G\left(2^\ell\right), N\ell)$
8: **else**
9: **if** δ eq 1 **then** $\texttt{dummy} \leftarrow \textbf{sub}(r, G\left(2^\ell\right), N\ell)$ // For isochrony
10: **else** $\texttt{dummy} \leftarrow \textbf{add}(r, G\left(2^\ell\right), N\ell)$
11: **end if**
12: $r \leftarrow \textbf{add}(r, \texttt{max}, N\ell)$ //Negative to non negative representation for all r'_i
13: **if** q eq 2^k **then**
14: $\texttt{mask}' \leftarrow \textbf{concat}(2^k - 1, \ell, N)$
15: $r \leftarrow \textbf{and}(r, \texttt{mask}', N\ell)$
16: **else**
17: $r \leftarrow$Simult. Barrett(r)
18: $r \leftarrow$ Simult. Cond. Sub.(r, ℓ, N) // Can be applied twice if some $r_i \geq 2q$
19: **end if**
20: $R(X) \leftarrow$ Radix Conversion(r)
21: **return** $R(X)$

The Modular Polynomial Multiplication Algorithm 6 works as follows:

1. Line 2: Polynomial evaluations defined in Equation 1 and Algorithm 1.
2. Line 3: Modular integer multiplication modulo $2^{N\ell} + \delta$ of the evaluated polynomials.
3. Line 4 to 11: Handle the two's complement representation of the evaluated polynomial; see Section 4.2.
4. Line 12: Convert the negative representation to non negative one; see Section 4.2. This operation allows to perform simultaneous reduction mod q and radix conversion.
5. Line 13 to 19: Perform simultaneous reduction mod q. This ensures that the polynomial result has coefficients reduced mod q.
6. Line 20: Radix conversion defined in Equation 2 to obtain a polynomial result.

6 Applications and Results

In this section, after some preliminaries, the component on which we performed our experiments and the results obtained by implementing the `Modular Poly-`

`nomial Multiplication` (`MPM`), cf. Algorithm 6, and another polynomial multiplication depending of the evaluated scheme. The evaluated lattice-based algorithms are: `Kyber`, `Dilithium`, `NTRU`, and `Saber`.

6.1 Background

NTT NTT is an algorithm allowing to perform fast polynomial multiplication in $R_{q,1}$ [16]. Given a and $b \in R_{q,1}$, $a \times b$ is computed as $\mathrm{NTT}^{-1}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b))$, where $\circ$ is the coefficient-wise multiplication.

Theoretically, NTT has the best asymptotic complexity for multiplication in $R_{q,1}$. However, in constrained environments (e.g. smart cards), devices may have dedicated hardware to perform fast large-integer arithmetic. In this context, NTT can be outperformed by an algorithm relying on integer arithmetic, even if its theoretical complexity is worse than NTT.

Subdivision RSA/ECC coprocessors perform integer arithmetic with data in buffer size has a fixed limit. In our context after polynomial evaluation, the resulting integer is generally too large to fit in these buffers. In that case we use multiple-precision arithmetic. This arithmetic consists of dividing the manipulated integers into several smaller ones and then perform operations on these smaller integers.

In the case of integer multiplication we use two techniques to divide the integer multiplication into smaller ones: Karatsuba and Schoolbook. Let $f = f_I + f_S 2^{N\ell/2}$ and $g = g_I + g_S 2^{N\ell/2}$, where f_I, f_S, g_I, g_S are lower than $2^{N\ell/2}$.

Schoolbook: $fg = f_I g_I + (f_I g_S + f_S g_I)2^{N\ell/2} + 2^{N\ell} f_S g_S$

Karatsuba: $fg = f_I g_I + ((f_I + f_S)(g_I + g_S) - f_I g_I - f_S g_S)2^{N\ell/2} + 2^{N\ell} f_S g_S$

These techniques can be applied recursively in order to obtain a targeted integer size. Later on when presenting the results, we specify in a column named *subdivision* the multi-precision method that we use for the integer multiplication.

Evaluation point. In our context the Karatsuba subdivision requires to increase the size of the evaluation point by 1 bit at each subdivision. It is due to the computation $(f_I + f_S)(g_I + g_S)$. Indeed, this computation is performed on integers of length twice as small but with values twice as large.

In the following results, the evaluation point is chosen to take into account the negative coefficients and the Karatsuba subdivisions.

Polynomial distribution The following polynomial multiplications are performed between a polynomial $G(X) \in R_{q,\delta}$ and $F(X) \in R^{-}_{q,\delta}$. More precisely, the coefficients of $G(X)$ are sampled uniformly in $\{0, \ldots, q-1\}$ and the coefficients of $F(X)$ are sampled in a distribution $\mathcal{D}_\sigma$. Using a distribution $\mathcal{D}_\sigma$, the coefficients are represented in $\{-\sigma, \ldots, 0, \ldots, \sigma\}$.

Masked secret polynomial. Most of the time the polynomial using the distribution $\mathcal{D}_\sigma$ is the secret polynomial. In some use cases, an embedded implementation must be strongly secured against side-channel attacks. One way to do this is to mask the secret data. To do so, we split the sensitive data into shares $x = x_1 + x_2 \mod q$, where x_1, x_2 belongs to $\{0, \ldots, q-1\}$, and then we process the operations on each share separately. In our context the value q is much larger than the secret distribution. Therefore, that implies we will manipulate larger secret data and then it increases the evaluation point. For some assessments, in order to consider this security requirement, we suppose that the polynomial $F(X)$ is defined over $R_{q,\delta}$ and its coefficients are sampled uniformly in $\{0, \ldots, q-1\}$. In the following results, we denote this case by $\mathcal{U}_q$ distribution.

In the following results, we only specify the distribution of $F(X)$.

Target Assessments are done on a smart card component using a 32-bit architecture. In the following we refer to this device as Component A. Due to intellectual properties reasons, the component name or a detailed description cannot be given. Then, we only give the main characteristics of the component A:

- Standards 32-bit instructions (add, sub, shifts, bitwise and, xor, or, etc.).
- No CPU multiplication and division.
- A coprocessor which handles: logical `AND`, addition, subtraction, shifts, modular integer multiplication and the non-modular one.

The following results take into account a complete modular reduction. Moreover like the previous works [1,18,5,10], we assume that the inputs are already in the appropriate machine representation. This implies that the inputs are in:

- Polynomial representation for NTT, Karatsuba and schoolbook polynomial multiplication.
- Packed integers representation for the `MPM` algorithm.

6.2 Results

Kyber Kyber [4] is a lattice-based KEM finalist of the NIST standardization. The polynomial ring defined in Kyber is $R_{q,1} = \mathbb{Z}_q[X]/(X^N+1)$, where $q = 3329$ and $N = 256$. The polynomial multiplication used in the specification is the NTT algorithm. In this context, we have implemented two polynomial multiplications:

- A NTT multiplication. It is adapted from the reference implementation, in order to use the hardware Montgomery multiplication. Tables of roots of unity have been recomputed to handle the Montgomery arithmetic with $R = 2^{32}$, the smallest handled by the coprocessor, instead of $R = 2^{16}$. In addition, the multiplication followed by a Montgomery reduction is replaced by a call to the coprocessor Montgomery multiplication. In Table 1 we present timings from the NTT's implementation.

	NTT	Pointwise	NTT^{-1}
Cycles	98k	40k	106k

Table 1. Kyber NTT cycles on Component A

- The modular polynomial multiplication (MPM) described in Algorithm 6. For this algorithm we consider two distributions for the polynomial $F(X)$:
  - $\mathcal{D}_3$. In this case the modular reduction modulo q is done using $\texttt{Simult.Barrett}_{11,0}$. In order to completely reduce the coefficients we perform 2 final subtractions using the technique described in Section 5.2.
  - $\mathcal{U}_q$. In this case the modular reduction modulo q is done using $\texttt{Simult. Barrett}_{10,10}$ and then an application of $\texttt{Simult. Barrett}_{13,-2}$. Afterwards, a final subtraction is performed using the technique described in Section 5.2.

In Table 2, we describe the parameters used for MPM algorithms. More precisely, we describe ℓ such that the evaluation point is 2^ℓ, the maximum value to convert negative coefficients to non-negative ones, the subdivision used and the obtained cycles.

Distribution	ℓ	maxValue	Subdivision	Cycles MPM
$\mathcal{D}_3$	23	$3qn$	None	50k
$\mathcal{U}_q$	34	q^2n	2 calls to Karatsuba	67k

Table 2. Parameters and cost of one multiplication in $R_{q,1}$ for Kyber parameters

Comparison. The previous results take into account one execution of MPM algorithm and each NTT routine. In order to compare NTT and MPM algorithms, we must not only compare pointwise routine with MPM algorithm. Indeed, we must also take into account calls to the NTT and NTT^{-1} routines. Then, in order to compare the two polynomial multiplication methods we must determine how many times each algorithm is called.

The Table 3 describes the number of calls to NTT, pointwise multiplication and NTT^{-1} during the Key Generation, Encrypt and Decrypt routines. The number of calls depends on the Kyber's security parameters which are $k = 2/3/4$. Note that the number of pointwise matches the number of MPM calls.

	NTT	Pointwise/MPM	NTT^{-1}
Key Gen.	$2k$	k^2	0
Encrypt	k	$k^2 + k$	$k + 1$
Decrypt	k	k	1

Table 3. Number of call to NTT routines in Kyber

In order to fairly compare NTT and MPM algorithms we use:

- The official specification of Kyber for the NTT algorithm. The private and public keys are stored in the NTT domain.
- A tweaked version of Kyber for the MPM algorithm. The private and public keys are not stored in the NTT domain. Therefore, we do not need to apply NTT^{-1} to perform MPM algorithm.

The MPM algorithm is called with the $\mathcal{U}_q$ distribution parameters.

	Total cycles NTT	Total cycles MPM	Ratio (NTT/MPM)
$k = 2$			
Key Gen.	552k	**268k**	2
Encrypt	754k	**402k**	1.9
Decrypt	382k	**134k**	2.9
$k = 3$			
Key Gen.	948k	**603k**	1.6
Encrypt	1198k	**804k**	1.5
Decrypt	520k	**201k**	2.6
$k = 4$			
Key Gen.	1424k	**1072k**	1.3
Encaps	1722k	**1340k**	1.3
Decrypt	658k	**268k**	2.5

Table 4. Cycle count for all multiplications in Kyber for the $\mathcal{U}_q$ distribution parameters

Saber & NTRU

Saber. Saber [9] is a lattice-based KEM finalist of the NIST standardization. The polynomial ring used in Saber is $R_{q,1} = \mathbb{Z}_q[X]/(X^N + 1)$, where $N = 256$ and $q = 8192 = 2^{13}$. In this work we consider two distributions for the polynomial $F(X)$:

- $\mathcal{D}_5$. Other distributions are used in Saber. However we only describe the worst one for the MPM algorithm.
- $\mathcal{U}_q$.

Since the modulus is a power of two, the reductions are achieved using a logical AND with the appropriate mask.

NTRU. NTRU [7] is also a KEM finalist of the NIST competition. The polynomial ring used in NTRU is $R_{q,-1} = \frac{\mathbb{Z}_q[X]}{(X^N-1)}$. The modulus q and the value N depends on the security parameters. In this work we only consider NTRU HPS 1 parameters, where $N = 509$ and $q = 2048 = 2^{11}$.

The value of N does not allow to easily make subdivisions. To overcome this issue, we work on polynomials with $\tilde{N} = 512$ coefficients where the latest coefficients are equal to 0.

In this work, we consider only a $\mathcal{U}_q$ distribution. Since q is a power of two, the modular reductions are performed with a logical AND.

Comparison. The Saber and NTRU MPM algorithms are compared with the polynomial multiplication used in their reference implementations.

- Saber: A combination of a 4-way Toom-Cook and Karatsuba algorithms.
- NTRU: A schoolbook multiplication.

The polynomial multiplication of the reference implementations are achieved with the 32 bits coprocessor multiplication. The Table 5 describes the obtained results on Component A.

Distribution	ℓ	`maxValue`	`Subdivision`	Cycl.`MPM`	Cycl. ref.
Saber					
$\mathcal{D}_5$	25	$5qn$	None	47k	1405k
$\mathcal{U}_q$	36	q^2n	2 calls to Karatsuba	61k	1405k
NTRU					
$\mathcal{U}_q$	34	q^2n	3 calls to Karatsuba	173k	17256k

Table 5. Parameters and cost of one multiplication in $R_{q,\delta}$ for Saber and NTRU parameters

7 Conclusion

In this paper we pursue the previous works that optimize lattice-based schemes, by re-purposing today's RSA/ECC coprocessor. Indeed, we propose an algorithm, called `MPM`, which performs modular polynomial multiplication using coprocessor instructions. More precisely, our work allow to reprupose existing coprocessor to handle modular reductions and the negative coefficients during the polynomial multiplication.

Afterwards, we assess in practice the `MPM` algorithm for almost all NIST lattice-based finalists. This assessment is done on a component which has few CPU instruction and that bases the asymmetric cryptographic efficiency on its RSA/ECC coprocessor. The `MPM` algorithm is compared to software polynomial multiplications, as `NTT` or Karatsuba. The few CPU instruction minimizes the possible assembly optimization for the software algorithms. Therefore in this component, our algorithm multiplication brings a significant speed-up.

This attest that re-purposing standard asymmetric coprocessor to speed-up lattice-based cryptography is of interest especially in a context of hybrid cryptography deployment.

References

1. Albrecht, M.R., Hanser, C., Hoeller, A., Pöppelmann, T., Virdia, F., Wallner, A.: Implementing RLWE-based Schemes Using an RSA Co-Processor. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 169–208 (2019)
2. ANSSI: Technical position paper - ANSSI views on the Post-Quantum Cryptography transition, available at `https://www.ssi.gouv.fr/publication/anssi-views-on-the-post-quantum\-cryptography-transition/`
3. Barrett, P.: Implementing The Rivest Shamir And Adleman Public Key Encryption On A Standard Digital Signal Processor. CRYPTO' 86. CRYPTO 1986. Lecture Notes in Computer Science, vol 263. Springer, Berlin, Heidelberg. pp. 1156–1158 (1986)

4. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: Crystals – kyber: a cca-secure module-lattice-based kem. Cryptology ePrint Archive, Report 2017/634 (2017)
5. Bos, J.W., Renes, J., van Vredendaal, C.: Post-quantum cryptography with contemporary co-processors. USENIX (2021)
6. BSI: Migration zu Post-Quanten-Kryptografie - Handlungsempfehlungen des BSI
7. Chen, C., Danba, O., Hoffstein, J., Hülsing, A., Rijneveld, J., M.Schank, J., Schwabe, P., Whyte, W., Zhang, Z.: NTRU (2020)
8. for Cryptography Research, C.A.: National cryptographic algorithm design competition (2018)
9. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F.: Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. Cryptology ePrint Archive, Report 2018/230 (2018)
10. Greuet, A., Montoya, S., Renault, G.: Speeding-up ideal lattice-based key exchange using a RSA/ECC coprocessor. IACR Cryptol. ePrint Arch. p. 1602 (2020)
11. Harvey, D.: Faster polynomial multiplication via multipoint kronecker substitution (2007)
12. Kronecker, L.: Grundzüge einer arithmetischen theorie der algebraischen grössen. (abdruck einer festschrift zu herrn e. e. kummers doctor-jubiläum, 10. september 1881.). Journal für die reine und angewandte Mathematik **92**, 1–122 (1882)
13. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC press (2018)
14. Moody, D.: Post-Quantum Cryptography NIST's Plan for the Future (2016)
15. Moody, D., Alagic, G., Apon, D.C., Cooper, D.A., Dang, Q.H., Kelsey, J.M., Liu, Y.K., Miller, C.A., Peralta, R.C., Perlner, R.A., Robinson, A.Y., Smith Tone, D.C., Alperin Sheriff, J.: Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. Tech. rep., National Institute of Standards and Technology (Jul 2020)
16. Nussbaumer, H.J.: Number Theoretic Transforms, pp. 211–240. Springer Berlin Heidelberg, Berlin, Heidelberg (1982)
17. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM J. Comput. **26**(5), 1484–1509 (Oct 1997)
18. Wang, B., Gu, X., Yang, Y.: Saber on ESP32. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) Applied Cryptography and Network Security. pp. 421–440. Springer International Publishing, Cham (2020)