Speeding-up Ideal Lattice-Based Key Exchange Using a RSA/ECC Coprocessor

Aurélien Greuet[1], Simon Montoya[1,2] and Guénaël Renault[3,2]

[1] IDEMIA France, Paris La Défense, France, firstname.lastname@idemia.com
[2] LIX, INRIA, CNRS, École Polytechnique, Institut Polytechnique de Paris, France, firstname.lastname@lix.polytechnique.fr
[3] ANSSI, Paris, France, firstname.lastname@ssi.gouv.fr

Abstract. Polynomial multiplication is one of the most costly operations of ideal lattice-based cryptosystems. In this work, we study its optimization when one of the operand has coefficients close to 0. We focus on this structure since it is at the core of lattice-based Key Exchange Mechanisms submitted to the NIST call for post-quantum cryptography. In particular, we propose optimization of this operation for embedded devices by using a RSA/ECC coprocessor that provides efficient large-integer arithmetic. In this context, we compare Kronecker Substitution, already studied in [AHH+19], with two specific algorithms that we introduce: KSV, a variant of this substitution, and an adaptation of the schoolbook multiplication, denoted Shift&Add. All these algorithms rely on the transformation of polynomial multiplication to large-integer arithmetic. Then, thanks to these algorithms, existing coprocessors dedicated to large-integer can be re-purposed in order to speed-up post-quantum schemes. The efficiency of these algorithms depends on the component specifications and the cryptosystem parameters set. Thus, we establish a methodology to determine which algorithm to use, for a given component, by only implementing basic large-integer operations. Moreover, the three algorithms are assessed on a chip ensuring that the theoretical methodology matches with practical results. They are also compared to reference software implementations such as NTT or schoolbook multiplication.

Keywords: Post-Quantum Lattice-based Cryptography · Polynomial Multiplication · Smart cards

1 Introduction

The emergence of academic and industrial projects on the design of a potential quantum computer that can break most of the current public-key cryptosystems with Shor's Algorithm [Sho97], led national agencies to study new proposals (e.g. [BSI]) and start standardization of quantum safe algorithms [Moo16, fCR18]. In particular, the National Institute of Standards and Technology (NIST) launched, in 2016, a standardization call for post-quantum safe key exchange and signature [Moo16]. In July 2020, they announced seven finalists including four Key Exchange Mechanism (KEM) for a future standardization. Among these four KEMs, three are based on lattice problems [MAA+20]. For the Chinese competition, the lattice-based KEM LAC [XYD+19], won the first prize in January 2020 [fCR20]. Hence international post-quantum cryptography standards are very likely to include lattice-based cryptosystems. Thus, optimizing and ensuring the practical security of these algorithms is an important area of research.

Constrained environments like smart cards can be very limited in terms of CPU frequency or amount of RAM, especially when compared to regular computers. Working

on specific optimizations is often necessary to get efficient implementation. These devices may embed dedicated hardware coprocessors to accelerate symmetric and asymmetric cryptographic computations. However, most of today asymmetric coprocessors are designed for RSA or elliptic curve cryptography (ECC) and are not adapted for lattice-based cryptography. Performances and ease of deployment are parts of the NIST selection criteria. Then, re-purposing existing components to optimize lattice-based KEM can facilitate the transition to quantum safe algorithms.

Motivation & Previous works

Polynomial multiplication is the most costly operation for ideal lattice-based algorithms. A lot of research has been done on the design of efficient hardware to speed-up polynomial multiplication, see e.g. [YZMZBY+20, DFA+20, SRB20]. However, the transition period should rely on hybrid mechanisms, mixing both classical and post-quantum asymmetric cryptography. Thus, both large modular arithmetic and operations related with post-quantum cryptography, like polynomial multiplication, have to be handled.

Nowadays, hardware accelerating large modular arithmetic are designed and deployed. Then, re-purposing these coprocessors to optimize polynomial multiplication is relevant in terms of costs and ease of deployment for an hybrid cryptography world.

The previous work of Albrecht *et al.* in [AHH+19] optimizes Kyber round 1 algorithm with a RSA/ECC coprocessor, which handles large-integer arithmetic. They transform polynomial multiplication to an integer multiplication with Kronecker Substitution. The work of Wang *et al.* in [WGY20] re-use the Kronecker Substitution with such coprocessor to optimize Saber algorithm.

More recently, an independent work of Bos *et al.* in [JWBJRCvV20] introduced `Kronecker+`, an algorithm using a variant of Nussbaumer allowing to combine with Kronecker Substitution. Theoretically, `Kronecker+` allows a faster polynomial multiplication than Kronecker Substitution. However, the theoretical approach does not provide an estimation of the costs of the polynomial transformations required to apply `Kronecker+`, which does not ensure a faster algorithm in practice.

Contribution

In this paper we follow the approach initiated in [AHH+19] to improve polynomial multiplication for lattice-based KEMs using a RSA/ECC coprocessor. More precisely, the coprocessor performs few basic operations on large integers: multiplication, addition, subtraction, right/left shift. These operations are implemented in most of current RSA/ECC coprocessors.

Our work focus on polynomial multiplication when one the operand has small coefficients. To take advantage of this structure, we study a variant of Kronecker Substitution and an adaptation of the schoolbook multiplication, called Shift&Add. Both methods allow to handle polynomial multiplication with operations on large integers.

Compared to Kronecker Substitution, its variant replaces large integer multiplications with additions, shifts and multiplications between a large integer and a coefficient. For small coefficients, the latter is expected to be cheaper than a regular multiplication. Shift&Add handles polynomial multiplications with only integer additions and shifts. With Shift&Add, the smaller the coefficients are, the fewer operations are performed.

Thereafter, we propose a methodology to help the comparison between Kronecker Substitution, its variant and Shift&Add, for a given KEM and a given coprocessor. To this end, we give theoretical complexity estimates for the three algorithms, expressed in terms of basic operations like addition, multiplication, shift and evaluation. Then, by measuring the performance of these basic operations, a developer can determine the fastest algorithm without having to fully develop each algorithm.

Finally, we verify that practical results are in accordance with this methodology for seven parameter sets from the NIST PQC process. We also compare our implementations with NTT or, when NTT cannot be used, a naive software implementation. Like in [AHH+19], the use of a coprocessor always provides a consequent speed-up. In addition, we show that for some setups, Shift&Add is faster than Kronecker Substitution.

Organization

In Section 2 we introduce some notations and describe the three algorithms that we use to perform polynomial multiplication with a coprocessor. Section 3 is devoted to discuss the side-channel aspects of the proposed algorithms. In Section 4, we show how to determine the evaluation point and establish the complexity of the three algorithms in terms of basic coprocessor operations. Finally, in Section 5, we assess our algorithms with different set of parameters and show that the practical are consistent with our theoretical study.

2 Algorithms

In this section we present the algorithmic material used in this work. We first detail some notations and well know algorithmic techniques that will be developed in our contributions presented at the end of this section.

2.1 Notation and Preliminaries

The arithmetic that we study comes naturally from the definition of the ideal lattice-based cryptography and is given as follows.

Rings. For an integer $q \geq 1$, let $\mathbb{Z}_q$ be the residue class group modulo q such that $\mathbb{Z}_q$ can be represented as $\{0, \ldots, q-1\}$. We define R_q being the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$.

Modular reduction. Let $a, b \in \mathbb{N}$, we denote by $a \mod^{(+)} b$ the unique integer $a' \equiv a \mod b$ such that $0 \leq a' < b$ and $a \mod^{(-)} b$ the unique integer $a' \equiv a \mod b$ such that $-\frac{b}{2} \leq a' < \frac{b}{2}$. In the following, we denote by $a \mod b = a \mod^{(+)} b$

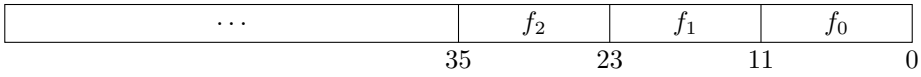
Polynomials. A polynomial in R_q is represented by a polynomial of degree at most $(n-1)$ with coefficients in $\mathbb{Z}_q$. Given $f \in R_q$, we denote by f_i the coefficient associated with the monomial x^i.

Polynomial representation. Polynomials are represented as byte strings. Let $f(x)$ be a polynomial of degree $n-1$ with all its coefficients $0 \leq f_i \leq \beta$. Then, each coefficient is encoded on $\lfloor \log_2(\beta) \rfloor + 1$ bits. The coefficients are packed as a string of size $n(\lfloor \log_2(\beta) \rfloor + 1)$ bits to represent $f(x)$.

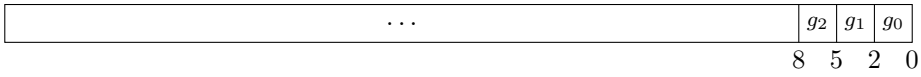
Let $g(x)$ be a polynomial of degree $n-1$ with all its coefficients $-\frac{\delta}{2} \leq g_i \leq \frac{\delta}{2}$. Then, each coefficient is encoded on $\lfloor \log_2(\delta) \rfloor + 1$ such that it is represented as $g_i \mod^{(+)} 2^{\lfloor \log_2(\delta) \rfloor + 1}$. As previously, all the coefficients are packed as a string of size $n(\lfloor \log_2(\delta) \rfloor + 1)$ bits to represent $g(x)$.

Example 1. Let $f(x)$ and $g(x)$ be two polynomials over R_q.

Let $f(x)$ be a polynomial with non-negative coefficients lower or equal to $q-1 = 3328$, then each coefficient is encoded on $\lfloor \log_2(3328) \rfloor + 1 = 12$ bits:



Let $g(x)$ be a polynomial with coefficients lying in $\{-\frac{\delta}{2}, \ldots, \frac{\delta}{2}\}$ with $\delta = 4$, then they are encoded on $\lfloor \log_2(4) \rfloor + 1 = 3$ bits:



Large integer operations In the sequel, we introduce three algorithms: Kronecker Substitution (Alg. 3), Kronecker Substitution Variant (Alg. 4) and Shift&Add (Alg. 5). For these algorithms the integer operations (multiplication, addition, subtraction, shift) are implicitly performed on large integers.

Number Theoretic Transform (NTT) NTT is an algorithm allowing to perform fast polynomial multiplication in R_q [Nus82, LN16]. Given a and $b \in R_q$, $a \times b$ 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. 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.

In the following paragraph, we describe a well know technique coming from computer algebra that allows to perform polynomial multiplication with integer arithmetic.

Kronecker Substitution The main idea of this substitution is to transform polynomial multiplication to an integer multiplication by evaluating the operands and to get back to the result using a radix conversion. More precisely if $f = \sum_{i=0}^{n-1} f_i X^i$ and $g = \sum_{i=0}^{n-1} g_i X^i$ are two polynomials with non-negative coefficients, by considering the product of their evaluations at an integer B we obtain the integer

$$\tilde{f}\tilde{g} = \left(\sum_{i=0}^{n-1} f_i B^i\right)\left(\sum_{i=0}^{n-1} g_i B^i\right)$$

that can be expressed in the base B by radix conversion. Then one obtains the evaluation of fg at B

$$\widetilde{fg} = \sum_{j=0}^{2n-2}\left(\sum_{i=0}^{j} f_i g_{j-i}\right) B^j$$

and thus deduce the value of each of the coefficients of the corresponding polynomial fg. This radix conversion is possible only if each of the resulting coefficients is smaller than B. The integer B is usually chosen as a power b^ℓ where b is equal to 2 or 10 depending on the context. In all the cases, ℓ as to be chosen sufficiently large in order to make the radix conversion effective. More precisely, ℓ could be chosen as the smallest integer such that b^ℓ is greater than the maximum of the coefficients of the resulting polynomial.

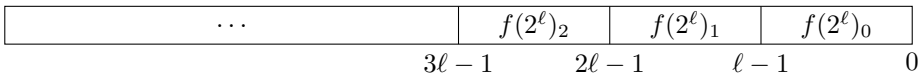
When some of the coefficients of the polynomials are negative, we have to adapt this reconstruction. In the following algorithms, we present the evaluation process and the radix conversion for a general polynomial of degree $n-1$. We take care of the sign of the coefficients by storing this information in a variable that is given as one of the outputs of the first algorithm. We explain more precisely this procedure in the following.

Algorithm 1 EVALUATION

Input: $f(x) \in \mathbb{Z}[x]$ of degree $n-1$, $\ell \in \mathbb{N}$
Output: $(-1)^{\mathtt{neg}} \times f(2^\ell), \mathtt{neg} \in \{0,1\}$

1: $\mathtt{carry} \leftarrow 0$
2: **if** $f_{n-1} < 0$ **then**
3: $\quad \mathtt{neg} \leftarrow 1$ // Ensure that the result integer is always positive
4: **else**
5: $\quad \mathtt{neg} \leftarrow 0$
6: **end if**
7: **for** $i = 0$ to $n-1$ **do**
8: $\quad \mathtt{tmp} \leftarrow (-1)^{\mathtt{neg}} f_i + \mathtt{carry} \mod {}^{(+)} 2^\ell$
9: $\quad$ **if** $\mathtt{tmp} > 2^{\ell-1}$ **then**
10: $\quad\quad \mathtt{carry} \leftarrow -1$
11: $\quad$ **else**
12: $\quad\quad \mathtt{carry} \leftarrow 0$
13: $\quad$ **end if**
14: $\quad f(2^\ell) \leftarrow f(2^\ell) + \mathtt{tmp} \times 2^{i\ell}$
15: **end for**

The integer obtained after evaluation at 2^ℓ of a polynomial of degree $n-1$, can be viewed as the concatenation of n integers of bitsize ℓ.



EVALUATION algorithm 1 always returns a positive integer. Indeed, if the highest degree coefficient of a polynomial $g(x)$ is negative then the algorithm returns the evaluation of $-g(x)$. The parameter $\mathtt{neg}$ indicates, for the latter radix conversion, if the evaluation algorithm returned the evaluation of $g(x)$ or $-g(x)$. In the following we denote $f(2^\ell)_i = (f(2^\ell) >> i(\ell-1)) \& (2^\ell - 1)$.

Algorithm 2 RADIX CONVERSION

Input: $r(2^\ell) = (r(2^\ell)_0, r(2^\ell)_1, \ldots, r(2^\ell)_{n-1})$ and $n, \ell, \mathtt{neg} \in \mathbb{N}$
Output: $r(x) = r_0 + r_1 x + \ldots + r_{n-1} x^{n-1} \in \mathbb{Z}[x]$

1: $\mathtt{carry} \leftarrow 0$
2: **for** $i = 0$ to $n-1$ **do**
3: $\quad r_i \leftarrow r(2^\ell)_i + \mathtt{carry} \mod {}^{(+)} 2^\ell$
4: $\quad$ **if** $r_i > 2^{\ell-1}$ **then**
5: $\quad\quad r_i \leftarrow (-1)^{\mathtt{neg}+1}(2^\ell - r_i)$
6: $\quad\quad \mathtt{carry} \leftarrow 1$
7: $\quad$ **else**
8: $\quad\quad r_i \leftarrow (-1)^{\mathtt{neg}} r_i$
9: $\quad\quad \mathtt{carry} \leftarrow 0$
10: $\quad$ **end if**
11: **end for**

The following example presents Kronecker Substitution by using a decimal radix.

Example 2. Let $f(x) = 8x^2 + 3x + 2$ and $g(x) = -5x^2 - 4x + 1$. Note that the coefficients r_i of the result are such that $-\frac{10^2}{2} \leq r_i \leq \frac{10^2}{2}$. Hence, we evaluate f and g at 10^2 in order to compute $f(x) \times g(x)$ using integer multiplication: $f(10^2) = 080302$, $\mathtt{neg}_f = 0$ and

$g(10^2) = 050399$, $\texttt{neg}_g = 1$. After this evaluation, these two integers can be multiplied:

$$f(10^2) \times g(10^2) = 080302 \times 050399 = 4047140498$$

Since the evaluation was done at 10^2, the resulting polynomial can be interpolated by reading the coefficients 2 digits by 2 digits. The two first digits are $98 \geq \frac{10^2}{2}$, that represents the negative number $-(10^2 - 98) = -2$ and propagate a carry for the next coefficient. After that, we got $4 < 50$ plus the previous carry to obtain 5. And so on for the other coefficients. However, $\texttt{neg}_g = 1$, then reading the coefficients like this gives $-(f(x)g(x))$.

$$\begin{aligned} -(f(x) \times g(x)) &= 40x^4 + 47x^3 + 14x^2 + (4+1)x - (10^2 - 98) \\ &= 40x^4 + 47x^3 + 14x^2 + 5x - 2 \\ f(x) \times g(x) &= -40x^4 - 47x^3 - 14x^2 - 5x + 2 \end{aligned}$$

For sake of clarity, examples are given in 10-radix but in our algorithms and implementation, a power of 2 is used as the integer B.

Algorithm 3 KRONECKER SUBSTITUTION

Input: $f(x) \in \mathbb{N}[x], g(x) \in \mathbb{Z}[x]$ and $\ell \in \mathbb{N}$
Output: $r(x) = f(x)g(x)$
1: $(f(2^\ell), \texttt{neg}_f), (g(2^\ell), \texttt{neg}_g) \leftarrow$ evaluation of $f(x), g(x)$ at 2^ℓ
2: $r(2^\ell) \leftarrow f(2^\ell)g(2^\ell)$
3: $r(x) \leftarrow$ radix conversion of $(r(2^\ell), \texttt{neg}_f \oplus \texttt{neg}_g)$

In the following Kronecker Substitution or KS refer to the Algorithm 3.

2.2 Polynomial Multiplication Using the Structure

The polynomials arising in ideal lattice based cryptosystems are structured, we show in the sequel how gain in efficiency by using them.

2.2.1 Notations

To perform an arithmetic operation using an hardware accelerator, the operands and an opcode for the operation to perform must be set into the coprocessor. When the choice of the operation depends on the value of a secret, we denote these sequences of instructions by $\texttt{c} \leftarrow \texttt{Op}\,(\texttt{a}, \texttt{b})$, where $\texttt{Op} \in \{\texttt{Add}, \texttt{Sub}\}$ represents the opcode. This notation allows to simplify the constant-time implementation of an algorithm (see Section 2.2.4 below).

In the following algorithm descriptions, `@var` denotes the address of the variable `var`. Given a pointer `ptr`, `*ptr` stands for the value stored at address `ptr`. The bitwise exclusive-or is denoted by $\oplus$, the bitwise logical `and` by `&` and $a >> b$ (resp. $a << b$) stands for the logical right shift (resp. logical left shift) by b bits of the value a.

2.2.2 Kronecker Substitution Variant

Classical Kronecker Substitution multiplies two integers of length $b^\ell \times n$, where b^ℓ is the evaluation point B and $n - 1$ the degree of the polynomials. As mentioned above, ℓ is determined by the maximum coefficient value of the result. In this variant, n multiplications are done on an integer of length $b^\ell \times n$ by an integer of length b^k with $k < \ell$.

Example 3. Let $f(x) = 8x^2 + 3x + 2$ and $g(x) = g_2x^2 + g_1x + g_0 = 5x^2 + 4x + 1$ as above. Then,

$$\begin{aligned} f(10^2)g(10^2) &= (f(10^2) \times g_2)(10^2)^2 + (f(10^2)g_1)10^2 + f(10^2)g_0 \\ &= (080302 \times 5)(10^2)^2 + (080302 \times 4)10^2 + 080302 \times 1 \\ &= 4015100000 + 32120800 + 080302 \\ &= 4047301102 \end{aligned}$$

The resulting polynomial is recovered with radix conversion like in classical Kronecker.

A multiplication of two large integers is replaced with 3 multiplications of a large integer by a small coefficient and some additions and shifts. This technique has an interest when considering the multiplication of a polynomial with small coefficients by a generic polynomial. Such multiplications are used in some lattice-based key exchanges. Algorithm 4 multiplies two polynomials of degree $n-1$ such that $(|fg(x)|)_i \leq 2^\ell$.

Algorithm 4 KRONECKER SUBSTITUTION VARIANT

Input: $f(x) \in \mathbb{N}[x], g(x) \in \mathbb{Z}[x]$ and $n, \ell \in \mathbb{N}$
Output: $r(x) = f(x)g(x)$

1: $(f(2^\ell), \mathtt{neg}_f) \leftarrow$ evaluation of $f(x)$ at 2^ℓ
2: **for** $i = n-1$ to 0 **do**
3: **if** $g_i \geq 0$ **then**
4: $\mathtt{Op} \leftarrow \mathtt{Add}$ // Addition will be performed line 9
5: **else**
6: $\mathtt{Op} \leftarrow \mathtt{Sub}$ // Subtraction will be performed line 9
7: **end if**
8: $c \leftarrow |g_i|$
9: $r(2^\ell) \leftarrow \mathtt{Op}\left(r(2^\ell), c \times f(2^\ell)\right)$
10: $r(2^\ell) \leftarrow r(2^\ell) << \ell$
11: **end for**
12: $r(x) \leftarrow$ radix conversion of $(r(2^\ell), \mathtt{neg}_f)$

2.2.3 Shift&Add

We now present an adaptation of the schoolbook polynomial multiplication, denoted SHIFT&ADD, where polynomials are represented as integers, after a Kronecker-like evaluation. It relies only on additions and left shifts. This technique is of interest when one of the operand has small coefficients.

The basic idea is explained in Example 4 while a full description is given in Algorithms 5.

Example 4. Let $f(x) = 9x^2 + 8x + 3$ and $g(x) = g_2x^2 + g_1x + g_0 = 2x^2 - 1$. Let $r = 0$. The computation of $f(x) \times g(x)$ is done as follow:

Step 1. Evaluate f: $f(10^3) = 009008003$

Step 2. Since $g_2 = 2$:

1. $r \leftarrow r + f(10^3) \times \left(10^3\right)^2$;
2. $r \leftarrow r + f(10^3) \times \left(10^3\right)^2$;

Step 3. Since $g_1 = 0$, do nothing;

Step 4. Since $g_0 = -1$, $r \leftarrow r - f(10^3) \times \left(10^3\right)^0$;

This leads to

$$\begin{aligned} f(10^3)g(10^3) &= 2f(10^3)(10^3)^2 - f(10^3) \\ &= 2(009008003 \times (10^3)^2) - 009008003 \\ &= 18015996991997 \end{aligned}$$

By radix conversion,

$$\begin{aligned} f(x)g(x) &= 18x^4 + (15+1)x^3 - (10^3 - (996+1))x^2 - (10^3 - (991+1))x - (10^3 - 997) \\ &= 18x^4 + 16x^3 - 3x^2 - 8x - 3. \end{aligned}$$

Algorithm 5 Shift&Add

Input: $f(x) \in \mathbb{N}[x], g(x) \in \mathbb{Z}[x]$ with all $g_i \in \{-\frac{\delta}{2}, \ldots, 0, \ldots, \frac{\delta}{2}\}$ and $n, \ell, q \in \mathbb{N}$
Output: $r(x) = f(x)g(x)$

1: $(f(2^\ell), \mathtt{neg}_f) \leftarrow$ evaluation of $f(x)$
2: $\mathtt{tmp} \leftarrow [\,]$ // dummy buffer for constant time implementation
3: **for** $i = n-1$ to 0 **do**
4: **if** $g_i \geq 0$ **then**
5: $\mathtt{Op} \leftarrow \mathtt{Add}$ // addition will be done line 15
6: **else**
7: $\mathtt{Op} \leftarrow \mathtt{Sub}$ // subtraction will be done line 15
8: **end if**
9: **for** $j = 0$ to $\frac{\delta}{2} - 1$ **do**
10: **if** $j < |g_i|$ **then**
11: $\mathtt{buff} \leftarrow \mathtt{@}r(2^\ell)$ // `Op` in line 15 will be kept
12: **else**
13: $\mathtt{buff} \leftarrow \mathtt{@tmp}$ // `Op` in line 15 will be discarded
14: **end if**
15: $\mathtt{*buff} \leftarrow \mathtt{Op}\left(r(2^\ell), f(2^\ell)\right)$
16: **end for**
17: $r(2^\ell) \leftarrow r(2^\ell) << \ell$
18: **end for**
19: $r(x) \leftarrow$ radix conversion of $(r(2^\ell), \mathtt{neg}_f)$

2.2.4 Isochronous implementations.

As it will be explained in Section 3, side-channel attacks must be taken into consideration. Hence, Algorithms 4 and 5 are intended to be isochronous: the execution time does not depend on a secret value. In the sequel we assume that for a given operands size, additions and subtractions have same execution time.

At each loop iteration, the same number of additions or subtractions and shifts is performed. However, the time taken to execute conditional assignments (lines 3 to 7 in Algo. 4, lines 4 to 8 and lines 10 to 14 in Algo. 5) can depend on the condition, that itself depends on a secret. Likewise, the computation of absolute value (line 8 in Algo. 4 and line 10 in Algo. 5) must be handled carefully to be isochronous.

We show in Algorithm 6 how to achieve the pointer selection for lines 10 to 14 in Algorithm 5. This pointer selection is done without branches and without table accesses. Thus, its execution time depends neither on any secret value nor on cache access.

Since Algorithm 6 computes an absolute value and performs a conditional assignment, the same techniques can be used to make Algorithms 4 and 5 isochronous.

Algorithm 6 Isochronous pointers selection

Input: Coefficient $g_i \in \left\{-\frac{\delta}{2}, \ldots, \frac{\delta}{2}\right\}$ encoded on $k = \lfloor \log_2(\delta) \rfloor + 1$ bits, $j \in \mathbb{N}$, $R =$ bitsize of CPU registers.

Output: $\texttt{buff} \leftarrow \begin{cases} \texttt{@}r(2^\ell) \text{ if } |g_i| < j \\ \texttt{@tmp} \text{ else.} \end{cases}$

1: $s \leftarrow g_i >> (k-1)$ // 0 if $g_i \geq 0$, 1 else
2: $t \leftarrow (s \oplus 1) - 1$ // 0 if $g_i \geq 0$, $2^R - 1 = \texttt{0xFF...FF}$ else
3: $t \leftarrow t \ \texttt{\&} \ \left(2^k - 1\right)$ // previous result on k bits
4: $\texttt{abs} = (t \oplus g_i) + s$ // $|g_i| = g_i$ if $g_i \geq 0$, $\overline{g_i} + 1 = \left(g_i \oplus \left(2^k - 1\right)\right) + 1$ else
5: $t \leftarrow \left((\texttt{abs} - j) >> (R-1)\right) - 1$ // 0 if $|g_i| < j$, $2^R - 1 = \texttt{0xFF...FF}$ else
6: $\texttt{switch} = \texttt{@}r(2^\ell) \oplus \texttt{@tmp}$
7: $\texttt{buff} = (\texttt{switch \& } t) \oplus \texttt{\&}r(2^\ell)$ // $\texttt{@}r(2^\ell)$ if $|g_i| < j$, @tmp else

3 Considerations on Side-Channel Attacks

This work focus on implementations on embedded devices, thus the side-channel aspect must be considered.

Simple Attacks. To avoid simple attacks like SPA, we make our multiplication algorithms isochronous: the execution time does not depend on any secret. For most of the hardware accelerators, large-integer arithmetic timings depend only on the size of operands. Hence, we assume that the execution time of shift, addition, subtraction and multiplication on large integers does not depend on the processed data. Moreover, exploiting coprocessors leaks by SPA during their computation is hard in practice. Then, we assume that an hardware addition cannot be distinguished from an hardware subtraction with a simple power consumption or EM attack. In addition, in our experiments, the CPU is set to perform the multiplication and division between registers in constant time. These instructions are used to compute modular reductions.

Under these assumptions, it is clear that a straightforward implementation of Kronecker Substitution does not have any operation depending on the manipulated data.

In addition, we explained in Section 2.2.4 and showed in Algorithm 4 and 5 how to compute isochronous KSV and Shift&Add, based on techniques described in Algorithm 6. With such techniques and since by assumption, addition cannot be distinguished from a subtraction, from an attacker point of view, the execution of the same instructions are performed at each loop iteration in Algorithm 5 regardless of the secret. Thus, SPA-like attack cannot reveal secret information.

Differential/Correlation Attacks. Several physical attacks against Post-Quantum cryptosystems have been studied [CCA+20]. Among them, some correlation attacks have been done on polynomial multiplication, see e.g. [EFGT17, OSPG18, RJH+18, RdCR+16, RRDC+16] and references therein.

These attacks are based on the fact that power consumption or electromagnetic emissions are correlated with the data being manipulated. Such an attack targets an intermediate variable of the form $s \times m$, where s is a small part of the secret and m a known input, like a message or a public key. Since s is small and m is known, the attacker can make a guess on s, compute $s \times m$ and predict, for a given leakage model, the expected consumptions or emissions for a series of different m's. Then, using physical measurements, like power consumption traces, and statistical tools, the correct key guess can be found: it is likely to be the one for which the correlation between predictions and real measurements is the strongest.

Masking is the classical countermeasure against such attacks [MOP08]. The sensitive data is split in two shares, each share being manipulated individually. For multiplication in lattice-based cryptography, the secret polynomial s has a structure, e.g. small coefficients, that can be exploited for performance optimizations. Hence, to keep this structure, one can split the known part: for a given public value m, consider a random m_1 and set $m_2 = m - m_1$. Then $s \times m$ can be computed as $(s \times m_1) + (s \times m_2)$, each $(s \times m_i)$ being processed independently. Since m_1 and m_2 are unknown and behave like randoms, the computation of $(s \times m_i)$ can not be correlated to $s \times m$, so that order 1 attacks will fail [MOP08]. With this countermeasure, the overhead is roughly an extra addition and an extra multiplication.

In [RdCR+16], the additively-homomorphic property of R-LWE schemes is used to mask, by adding the encryption of a random message before the decryption process. This random message encryption can be precomputed, so that the overhead is only an extra addition.

Recall that our goal is to determine, given a lattice-based scheme and a device, the fastest polynomial multiplication. For both kind of masking, the overhead does not depend on the choice of the multiplication algorithm: if unmasked multiplication A is faster than unmasked multiplication B, then the same result holds for their masked versions. Then in the sequel, we focus on comparing the basic unmasked versions of polynomial multiplication algorithms.

4 Complexity

This section is devoted to compare the performance of Kronecker substitution (KS), Kronecker substitution variant (KSV) and Shift&Add when using an existing RSA/ECC hardware accelerator. Hence, the complexity is given in terms of basic arithmetic operations performed by such accelerators: addition, multiplication and multiplication by a power of 2 (left shift) on large integers. Moreover, the number of evaluation differs between KS, KSV and Shift&Add. Then, the evaluation step is considered in the following complexities. Since the cost of these operations depends on the operand sizes, we first determine the minimal value for the parameter ℓ.

Our work is focused on lattice-based key exchange submitted to the NIST PQC standardization process. The polynomial multiplication of these key exchanges is over $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. To compute such a multiplication, we first multiply over $\mathbb{Z}[x]$. The result is then reduced modulo $x^n + 1$ and each coefficient is reduced modulo q. Since this reduction step is the same for the three methods, its cost is not relevant for a theoretical comparison between them. Hence, it is not included in the complexity computation. Likewise, radix conversion step is the same and is not included in the complexity. However, these costs are considered in the performance results of Section 5.

4.1 Choice of ℓ

As explained in Section 2, polynomial multiplication can be reduced to integer multiplication. To this end, the polynomials are evaluated at a point such that the result can be recovered by radix conversion. This evaluation point is determined by the maximum coefficient of the multiplication result.

In the following, we suppose that the polynomial evaluation is done by Algorithm 1. Then, each evaluated polynomial is represented as a positive integer.

Proposition 1. *Let f and g be polynomials of degree $n-1$ such that for all $i \in \{0, \ldots, n-1\}$, $0 \leq f_i \leq \beta$ and $-\frac{\delta}{2} \leq g_i \leq \frac{\delta}{2}$. Then*

- $\forall i \in \{0, \ldots, 2n-2\}, |(f(x)g(x))_i| < 2^{\ell-1}$, *where* $\ell = \lfloor \log_2(n\beta\delta) \rfloor + 1$.

- *Each coefficient of $f(x)g(x)$ is encoded on at most ℓ bits.*
- $\log_2(f(2^\ell)) < n\ell$ *and* $\log_2(\left|g(2^\ell)\right|) < n\ell$.

Proof. Let $r(x) = f(x)g(x)$. Then $r(x)$ is of degree $2n-2$ and its k-th coefficient is $r_k = \sum_{i=0}^{k} f_i g_{k-i}$. To prove the first assertion, we first consider the coefficients r_k for $k \leq n-1$. Since for all i, $0 \leq f_i \leq \beta$ and $|g_i| \leq \delta/2$, we get, for $k \leq n-1$:

$$|r_k| = \left|\sum_{i=0}^{k} f_i g_{k-i}\right| \leq \sum_{i=0}^{k} |f_i|\,|g_{k-i}| \leq \sum_{i=0}^{k} \beta\frac{\delta}{2} \leq \sum_{i=0}^{n-1} \beta\frac{\delta}{2} \leq n\beta\frac{\delta}{2}.$$

For $k \geq n$, note that since f (resp. g) has degree $n-1$, $f_i = 0$ (resp. $g_i = 0$) for $i \geq n$. Hence,

$$|r_k| = \left|\sum_{i=0}^{k} f_i g_{k-i}\right| = \left|\sum_{i=0}^{n-1} f_i g_{k-i} + \underbrace{\sum_{i=n}^{k} f_i g_{k-i}}_{=0}\right| = \left|\sum_{i=0}^{n-1} f_i g_{k-i}\right| \leq n\beta\frac{\delta}{2}.$$

Thus, for $k \in \{0,\ldots,2n-2\}$, $|r_k| \leq n\beta\frac{\delta}{2} < 2^{\lfloor \log_2(n\beta\frac{\delta}{2})\rfloor+1} = 2^{\ell-1}$. Each coefficient of the result is $-n\beta\frac{\delta}{2} \leq r_k \leq n\beta\frac{\delta}{2} < 2^{\ell-1}$, then to handle the negative case each coefficient of the result is at most encoded on ℓ bits.

We prove now the third assertion.

$$f(2^\ell) = \sum_{i=0}^{n-1} f_i 2^{i\ell} \leq (2^\ell-1)\sum_{i=0}^{n-1} 2^{i\ell} = (2^\ell-1)\frac{2^{n\ell}-1}{(2^\ell-1)} = 2^{n\ell}-1 < 2^{n\ell}$$

Thus, $\log_2\left(f(2^\ell)\right) \leq n\ell$. Likewise, $\log_2\left(\left|g(2^\ell)\right|\right) \leq n\ell$. □

Remark 1. The previous proposition applies with polynomial $f(x)$ with non-negative coefficients. In our context, the negative coefficients of a polynomial in R_q can be replaced with their non-negative equivalent in $\{0,\ldots,q-1\}$.

4.2 Complexity Estimates

In this section, we estimate the complexity of our multiplication algorithms. We express them in terms of the following basic operations. Let $E(n)$ be the evaluation complexity function for a polynomial of degree $n-1$. Let $M(x,y)$ and $A(x,y)$ be the multiplication and addition (or subtraction) of integers complexity functions depending on the bitsize of the inputs.

Example 5. Let a and b be two integers, let $x = \lfloor \log_2(a)\rfloor + 1$ and $y = \lfloor \log_2(b)\rfloor + 1$. Then the cost of computing $a \times b$ (resp. $a+b$) is $M(x,y)$ (resp. $A(x,y)$).

Likewise, $S(x,s)$ denotes the shift complexity function where x is the bitsize of the integer to shift on s bits.

4.2.1 Kronecker Substitution (KS)

Proposition 2. *Let $f(x)$ and $g(x)$ be two polynomials of degree $n-1$. Each coefficient of $f(x)$ is defined over $\mathbb{N}$ and each coefficient of $g(x)$ is defined over $\mathbb{Z}$. If for a given ℓ every coefficient $|(f(x)g(x))_i|$ is lower than $2^{\ell-1}$, then the multiplication complexity of $f(x)g(x)$ with Kronecker substitution is $2E(n) + M(n\ell, n\ell)$.*

Proof. Let $f(x)$ and $g(x)$ be polynomials of degree $n-1$. To compute $f(x) \times g(x)$ with Kronecker Substitution, the following steps are performed:

1. Evaluation of $f(x)$ and $g(x)$ at 2^ℓ. According to Proposition 1, $\log_2\left(f(2^\ell)\right) \leq n\ell$ and $\log_2\left(\left|g(2^\ell)\right|\right) \leq n\ell$.
2. Multiplication of two integers of bitsize $n\ell$.
3. Radix conversion of the coefficients.

Then, Kronecker Substitution complexity is $2E(n) + M(n\ell, n\ell)$. □

4.2.2 Kronecker Substitution Variant (KSV)

Proposition 3. *Let $f(x)$ and $g(x)$ be polynomials of degree $n-1$. Each coefficient of $f(x)$ are defined over $\mathbb{N}$ and each coefficient of $g(x)$ are defined over $\mathbb{Z}$. If for a given ℓ every coefficient $|(f(x)g(x))_i|$ is lower than $2^{\ell-1}$ and all coefficients of $g(x)$ fit on k bits, then the polynomial multiplication complexity of $f(x)g(x)$ with Kronecker substitution variant is $E(n) + n(M(n\ell, k) + S(n\ell, \ell) + A(n\ell, n\ell))$.*

Proof. Let $f(x)$ and $g(x)$ be polynomials of degree $n-1$, such that each bit representation for g_i fits on k bits. Computing $f(x) \times g(x)$ with Kronecker Substitution Variant is done by doing:

1. The evaluation of $f(x)$ at 2^ℓ. Then, $\log_2\left(f(2^\ell)\right) \leq n\ell$.
2. A "`for`" loop with n iterations, each step being:
   - A multiplication between an integer of bitsize $n\ell$ and a coefficients of bitsize k,
   - An addition or subtraction between two integers of size $n\ell$,
   - A ℓ-shift of an integer of bitsize $n\ell$.
3. A radix conversion of the coefficients.

Then the complexity of the multiplication using Kronecker Substitution Variant is $E(n) + n(M(n\ell, k) + S(n\ell, \ell) + A(n\ell, n\ell))$. □

4.2.3 Shift&Add

Proposition 4. *Let $f(x)$ and $g(x)$ be polynomials of degree $n-1$ with coefficients in $\mathbb{Z}$. If for a given ℓ every coefficient $|(f(x)g(x))_i|$ is lower than $2^{\ell-1}$ and all coefficients of $g(x)$ belong to $\{-\frac{\delta}{2}, \ldots, 0, \ldots, \frac{\delta}{2}\}$, then the polynomial multiplication in Shift&Add costs $E(n) + n(S(n\ell, \ell) + \frac{\delta}{2}A(n\ell, n\ell))$.*

Proof. Let $f(x)$, $g(x)$ be polynomials of degree $n-1$. Algorithm Shift&Add computes the multiplication as follow:

1. Evaluate $f(x)$ at 2^ℓ. Then, $\log_2\left(f(2^\ell)\right) \leq n\ell$.
2. A "`for`" loop called n times. For the three cases, Shift&Add computes:
   - $\frac{\delta}{2}$ additions or subtractions between two integers of size $n\ell$.
3. Radix conversion of the coefficients.

Then Shift&Add complexity is $E(n) + n(S(n\ell, \ell) + \frac{\delta}{2}A(n\ell, n\ell))$. □

4.2.4 Complexities comparison

Let $f(x)$ and $g(x)$ be polynomial of degree $n-1$. Assume that the coefficients of $g(x)$ belong to $\{-\frac{\delta}{2},\ldots,0,\ldots,\frac{\delta}{2}\}$ and that for all $i \in \{0,\ldots,2n-1\}, |(f(x)g(x))_i| \leq 2^{\ell-1}$. To choose the most efficient algorithm for polynomial multiplication we need to compare the three following complexities, depending on the component specification.

- Shift&Add: $E(n) + n(S(n\ell,\ell) + \frac{\delta}{2}A(n\ell,n\ell)))$.
- Kronecker substitution: $2E(n) + M(n\ell,n\ell)$.
- Kronecker substitution variant: $E(n) + n(M(n\ell,k) + S(n\ell,\ell) + A(n\ell,n\ell))$.

In Section 5.2, we explain how to instantiate the different basic complexities in order to compare the above estimations. We focus our study on the execution time and do not provide memory consumption estimates.

4.3 Time-memory trade-offs

The amount of RAM in embedded devices can be very limited. However, some devices allow a larger RAM consumption which can be utilize to speed-up our algorithms.

Polynomial representation. In Section 2 we describe our compact polynomial representation. This representation is useful to optimize our memory consumption but not to access to the polynomial coefficients. The evaluation and radix conversion require a lot of accesses to the coefficients, thus representing these coefficients as a machine word (e.g 32-bit) improves significantly the performance of these algorithms. Moreover, for some components, using a machine word representation allows to replace shift by arithmetic of pointers.

Precomputation. In our context, polynomial multiplication is between $f(x)$ which is a random polynomial over R_q and $g(x)$ which has coefficients in $\{-\frac{\delta}{2},\ldots,0,\ldots,\frac{\delta}{2}\}$, where δ is close to 0. Then, we can precompute $\frac{\delta}{2}-1$ multiples of $f(x)$: $2\times f(x),\ldots,\frac{\delta}{2}\times f(x)$ to reduce the number of operation to one addition/subtraction and one shift of each iteration of Shift&Add loop "`for`".

Positive case. As mentioned above, one of the polynomial has negative coefficients. That implies to handle carry propagation for the evaluation, radix conversion or the subdivision. The carry propagation requires an important amount of software implementation to be handled. However, KSV and Shift&Add can perform the polynomial multiplication without negative coefficient and then without carry propagation. Indeed, as the cost of a supplementary evaluation/storage of $-f(x) \mod q$, which is the computation $q-f_i$ for all the coefficients of f, KSV (resp. Shift&Add) multiplies (resp. add) $f(x)$ when the coefficient of $g(x)$ is positive and $-f(x) \mod q$ when the coefficient is negative.

4.4 Polynomial subdivisions

RSA/ECC coprocessors perform large integers arithmetic with data in buffer whose size has a fixed limit. In our algorithms, after evaluation of a polynomial, the resulting integer is generally too large to fit in these buffers. In this case, a subdivision is performed on the polynomials before evaluation. Let $f(x) = f_I + f_S x^{n/2}$ and $g(x) = g_I + g_S x^{n/2}$, where f_I,f_S, g_I and g_S have degree $< n/2$. We consider the following methods to subdivide:

Naive: $f(x)g(x) = f_I g_I + (f_I g_S + f_S g_I)x^{n/2} + f_S g_S$

Karatsuba: $f(x)g(x) = f_I g_I + ((f_I + f_S)(g_I + g_S) - f_I g_I - f_S g_S)x^{n/2} + f_S g_S x^n$

Karatsuba performs fewer multiplications at the cost of extra additions, subtractions and memory usage. Depending on the coprocessor specification, it can be slower than the naive subdivision.

Impact on ℓ Subdividing d times divides the value of n by 2^d. However, for the naive subdivision it does not reduce the value of $\ell = \lfloor \log_2(n\beta\delta) \rfloor + 1$. Indeed, after recombination the bitsize result is the same as a multiplication without subdivision.

Karatsuba requires the multiplication $(f_I + f_S)(g_I + g_S)$, which increases the value of $\ell = \lfloor \log_2(n\beta\delta) \rfloor + 1$. In fact, let f_I, f_S have coefficients in $\{0, \ldots, \beta\}$ and g_I, g_S have coefficients in $\{-\frac{\delta}{2}, \ldots, \frac{\delta}{2}\}$. Then, $(f_I + f_S)$ have coefficients in $\{0, \ldots, 2\beta\}$ and $(g_I + g_S)$ have coefficients in $\{-\delta, \ldots, \delta\}$. Thus, $\ell' = \lfloor \log_2(\frac{n}{2} \cdot 2\beta \cdot 2\delta) \rfloor + 1 = \ell + 1$. More generally, if d subdivisions are required, then $\ell' = \ell + d$ must be used instead of ℓ.

Impact on the complexities Subdividing allows to perform a large integer multiplication by few multiplications, additions and subtractions on smaller integers. KS can require one more subdivision than KSV and Shift&Add to avoid a lot of load, store and addition in software. Hence, to determine the most efficient algorithm, with the requirement of d subdivisions for KSV and Shift&Add and d' subdivisions for KS, we only need to compare the following complexities, depending on the component specification.

- Shift&Add: $E(n) + x^d \left(\frac{y}{x} A(\frac{n}{2^d}\ell, \frac{n}{2^d}\ell) + \frac{n}{2^d}(S(\frac{n}{2^d}\ell, \ell) + \frac{\delta}{2} A(\frac{n}{2^d}\ell, \frac{n}{2^d}\ell))\right)$.
- Kronecker substitution: $2E(n) + x^{d'} \left(\frac{y}{x} A(\frac{n}{2^d}\ell, \frac{n}{2^d}\ell) + M(\frac{n}{2^{d+1}}\ell, \frac{n}{2^{d+1}}\ell)\right)$.
- Kronecker substitution variant: $E(n) + x^d(\frac{y}{x} A(\frac{n}{2^d}\ell, \frac{n}{2^d}\ell) + \frac{n}{2^d}(M(\frac{n}{2^d}\ell, k) + S(\frac{n}{2^d}\ell, \ell) + A(\frac{n}{2^d}\ell, \frac{n}{2^d}\ell)))$.

The values x and y are, respectively, the number of sub-multiplications and the number of additions (or subtractions) required by the subdivision method. Hence, for the naive subdivision $x = 4$ and $y = 3$ and for Karatsuba $x = 3$ and $y = 6$.

In this paper we use the naive or Karatsuba to subdivide but the subdivision can be achieved with other methods.

5 Assessment

5.1 Context

We evaluate three lattice-based algorithms: Saber, LAC and Kyber. They have been submitted to the NIST PQC standardization [DKRV19, XYD+19, ABD+19]. Saber and Kyber passed the round 2 and they are finalists of the round 3. LAC did not passed the NIST 2nd round but won the Chinese cryptographic competition and thus remains relevant to study.

5.1.1 Parameters

In the following, the Kyber round 1 specifications are considered for Kyber512R1 and Kyber1024R1, in order to compare our results with the previous work in [AHH+19]. For the other schemes, round 2 specifications are considered. Our results come from a device with dedicated hardware coprocessor for large-integer operations.

In the following results we consider the multiplication of $f(x)$ by $g(x)$ over $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, where:

1. $f(x)$ is of degree $n-1$ with coefficients in $\{0,\ldots,q-1\}$.
2. $g(x)$ is of degree $n-1$ with coefficients in $\{\frac{-\delta}{2},\ldots,\frac{\delta}{2}\}$.
3. the evaluation point is 2^ℓ, where the value of ℓ is given by Proposition 1.

Parameters for each candidate are represented in the following table.

Set/Param	n	q	δ	ℓ
Kyber512R1	256	7681	10	25
Kyber1024R1	256	7681	6	24
KyberR2	256	3329	4	22
Light Saber	256	2^{13}	10	25
Fire Saber	256	2^{13}	6	24
Lac128	512	251	2	18
Lac256	1024	251	2	19

5.1.2 Target

Assessments are done on a smart card component. Due to intellectual properties reasons, the name or a full description of this component cannot be given. However, details on our analysis are given, allowing it to be reproduced on any component embedding a coprocessor designed for large integer arithmetic. This component is used in real-life products like bank cards, passports, secure elements, etc. It embed hardware accelerators for asymmetric cryptography computations, including large-integer arithmetic.

In the sequel, this chip is referred as "Component A". It has a 32-bit architecture. Its asymmetric coprocessor can handle 2048-bit operands. The addition of two 2048-bit integers is done in less than ten cycles, while the multiplication takes several thousand cycles. Since the addition is several thousand times cheaper than the multiplication, Shift&Add is expected to be faster than Kronecker Substitution when one operand has coefficients close to 0.

The coprocessor can also multiply a coefficient to a 2048-bit operand, that is of interest with Kronecker Substitution Variant. Then, KSV is expected to be faster than Shift&Add and Kronecker Substitution when one operand has small coefficients but not too close to 0.

5.2 From theory to practice: a methodology

In this section, we propose a methodology to determine, with a minimal amount of implementation, which polynomial multiplication algorithm is the fastest on a given component and for a given set of parameters. This is done by measuring the timings of basic operations (integer multiplication, addition and subtraction, shift and evaluation) and plugging them into the complexities from Section 4. Hence, this ease the algorithm choice without coding all the multiplication algorithms. This can help a developer to quickly decide which algorithm is the best choice.

We detail the methodology for Kyber512R1 parameters on component A. We focus on the comparison between Kronecker Substitution and Shift&Add and between Kronecker Substitution Variant and Shift&Add.

Component A coprocessor can handle operands of about 2048 bits. For Kyber512R1 parameters $n=256$ and $\ell=25$, the polynomials after evaluation are of size $n\ell=6400$ bits. Thus, we subdivide it to perform our algorithms (see Section 4.4).

On this component, the naive subdivision is more efficient than Karatsuba. For Kronecker Substitution we subdivide until the result $f(x)g(x)$ can fit in the coprocessor. That requires 3 subdivisions to get $\frac{n}{2^3}\ell=800$ bitsize per operand. For Shift&Add and KSV, we subdivide until the subdivisions of polynomial $f(x)$ can fit in the coprocessor.

That requires 2 subdivisions of size 1600-bit. Then, KS requires one more subdivision than KSV and Shift&Add to avoid a lot of load, store and addition in software. Indeed, KS result doubles the size of the operand while each iteration of KSV and Shift&Add increases only by ℓ the result size, which can be handled without an important amount of software manipulation.

Naive subdivision transforms large arithmetic operations to 4^d smaller arithmetic operations, where d is the required depth. This leads to the following expressions for Shift&Add, KSV and KS complexities:

$$\begin{aligned} n(S(n\ell,\ell)+\delta A(n\ell,n\ell))+E(n) &= 4^2\left(\frac{n}{2^2}[\frac{\delta}{2}A(1600,1600)+S(1600,\ell)]\right) \\ &\quad + 3A(1600,1600)+E(n) \\ E(n)+n(M(n\ell,k)+S(n\ell,\ell)+A(n\ell,n\ell)) &= 4^2\left(\frac{n}{2^2}[M(1600,\ell)+S(1600,\ell)+A(1600,1600)]\right) \\ &\quad + 3A(1600,1600)+E(n) \\ M(n\ell,n\ell)+2E(n) &= 4^3M(800,800)+16A(1600,1600)+2E(n) \end{aligned}$$

The value $3A(1600,1600)$ and $16A(1600,1600)$ are due to the recombination of the naive subdivision but they are negligible. Let $C(S\&A) = \frac{n}{2^2}(\frac{\delta}{2}A(1600,1600)+S(1600,\ell))+3A(1600,1600)$. We measure the execution time corresponding to $C(S\&A)$ as a reference. Then we measure $E(n)$, $M(800,800)$ and $M(1600,\ell)+S(1600,\ell)+A(1600,1600)$ and express them in terms of $C(S\&A)$.These measurements are obtained with an emulator. We get that:

$$\begin{aligned} E(n) &\simeq 1.34\times C(S\&A) \\ M(800,800) &\simeq 0.20\times C(S\&A) \\ \frac{n}{2^2}(M(1600,\ell)+S(1600,\ell)+A(1600,1600)) &\simeq 0,86\times C(S\&A). \end{aligned}$$

It follows the following estimations for Shift&Add, KS and KSV:

$$\begin{aligned} n(S(n\ell,\ell)+\frac{\delta}{2}A(n\ell,n\ell))+E(n) &\simeq 17.34\times C(S\&A) \\ M(n\ell,n\ell)+2E(n) = 4^3M(800,800)+2E(n) &\simeq 15.48\times C(S\&A) \\ E(n)+n(M(n\ell,k)+S(n\ell,\ell)+A(n\ell,n\ell)) = 4^2(0.86\times C(S\&A))+E(n) &\simeq 15.1\times C(S\&A) \end{aligned}$$

Hence, in this configuration, KSV is expected to be the fastest algorithm for these parameters.

Following this methodology, we get the following expected ratios between Kronecker Substitution and Shift&Add and between KSV and Shift&Add, all measurements being obtained with emulators.

Table 1: Expected ratio based on basic operations performances for component A

	KS/S&A	KSV/S&A
Kyber512R1	0.89	0.87
Kyber1024R1	1.17	1.1
KyberR2	1.26	1.2
Light Saber	0.89	0.87
Fire Saber	1.17	1.1
Lac128	1.45	1.37
Lac256	1.41	1.42

5.3 Experiments

To estimate the gain provided by the RSA/ECC accelerator, we measure the performance of the following:

- algorithms relying on hardware coprocessors (KS, KSV, Shift&Add)
- software implementation of schoolbook multiplication (Saber, Lac)
- software implementation of NTT (Kyber)

The NTT implementation used for Kyber512R1 and Kyber1024R1 is detailed in [LN16]. For KyberR2, the reference implementation of Kyber round 2 is used [ABD+19]. We measure NTT performance with the requirement of frequency transformations of both polynomials (w/ NTT(A)) or with only one frequency transformation (w/o NTT(A)).

Our results are obtained by computing a complete polynomial multiplication over R_q without any optimization of the reduction modulo q and $x^n + 1$. The following assessments are done using an emulator.

Practical results are given in Table 2. As expected regarding Table 1, Shift&Add is the fastest algorithm for 5/7 parameter sets, for which the coefficients of one of the polynomial are close to 0. Moreover, as attempted for the 2 other parameters sets, KSV is the fastest algorithm.

For LAC software naive multiplication, we report the result for a C implementation and for an optimized version in assembly.

Note that the theoretical ratios from Table 1 are not the exact same ratios between the practical results. This is because radix conversion and reduction over R_q are not taken into account in the theoretical complexity (see Section 4), while these operations are part of the timings in Table 2. Nevertheless, the fastest algorithm is always the expected one and proves that our methodology introduced in Section 5.2 is relevant.

In all cases, algorithms using hardware accelerators are much faster than the software one, even for NTT based multiplications.

Table 2: Polynomial multiplication over R_q cycle count on component A

Param/Algo	KS	KSV	SA	NTT	Naive
Kyber512R1	588k	**556k**	594k	1139k w/ NTT(A) 793k w/o NTT(A)	N/A
Kyber1024R1	572k	539k	**500k**	1139k w/ NTT(A) 793k w/o NTT(A)	N/A
KyberR2	535k	512k	**441k**	998k w/ NTT(A) 704k w/o NTT(A)	N/A
Light Saber	580k	**546k**	585k	N/A	11691k
Fire Saber	563k	530k	**493k**	N/A	11440k
Lac128	1594k	1586k	**1285k**	N/A	15560k 1683k (asm)
Lac256	6209k	6310k	**4980k**	N/A	62340k 7494k (asm)

6 Conclusion

In this paper, we pursue the work initiated in [AHH+19]. We propose two additional methods to compute polynomial multiplications using a RSA/ECC accelerator, when one

operand has small coefficients. We introduce a methodology to compare the different methods, based on the implementation on a minimal set of basic functions. This methodology can help to quickly choose the most efficient multiplication algorithm on a given platform, depending on the cryptosystem parameters.

Then, we assess the polynomial multiplication algorithms for several NIST KEM candidates on a smart card chip. The practical results are consistent with the expected behavior from our methodology. In our study, using a RSA/ECC coprocessor always leads to faster polynomial multiplications than software implementations, including reference implementations of NTT. In addition, for this component, our proposed multiplications are faster than the Kronecker Substitution described in [AHH+19].

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

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber, 2019. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions.

[AHH+19] Martin R Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based Schemes Using an RSA Co-Processor. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 169–208, 2019.

[BSI] BSI. Migration zu Post-Quanten-Kryptografie - Handlungsempfehlungen des BSI.

[CCA+20] Sreeja Chowdhury, Ana Covic, Rabin Yu Acharya, Spencer Dupee, Fatemeh Ganji, and Domenic Forte. Physical Security in the Post-quantum Era: A Survey on Side-channel Analysis, Random Number Generators, and Physically Unclonable Functions. *arXiv preprint arXiv:2005.04344*, 2020. https://arxiv.org/abs/2005.04344.

[DFA+20] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches. Cryptology ePrint Archive, Report 2020/795, 2020. https://eprint.iacr.org/2020/795.

[DKRV19] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber, 2019. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1857–1874, 2017.

[fCR18] Chinese Association for Cryptography Research. National cryptographic algorithm design competition, 2018. Available at https://www.cacrnet.org.cn/site/content/838.html.

[fCR20] Chinese Association for Cryptography Research. Lac won first prize of the national cryptographic algorithm design competition, 2020. Available at https://m.cacrnet.org.cn/site/content/854.html.

[JWBJRCvV20] Joppe W. Bos, Joost Renes, and Christine van Vredendaal. Polynomial Multiplication with Contemporary Co-Processors: Beyond Kronecker, Schönhage-Strassen & Nussbaumer. Cryptology ePrint Archive, Report 2020/1303, 2020. https://eprint.iacr.org/2020/1303.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *International Conference on Cryptology and Network Security*, pages 124–139. Springer, 2016.

[MAA+20] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, Angela Y Robinson, Daniel C Smith Tone, and Jacob Alperin Sheriff. Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. Technical report, National Institute of Standards and Technology, July 2020.

[Moo16] Dustin Moody. Post-Quantum Cryptography NIST's Plan for the Future, 2016. Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/pqcrypto-2016-presentation.pdf.

[MOP08] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, volume 31. Springer Science & Business Media, 2008.

[Nus82] Henri J. Nussbaumer. *Number Theoretic Transforms*, pages 211–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-Secure and Masked Ring-LWE Implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 142–174, 2018.

[RdCR+16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively Homomorphic Ring-LWE Masking. In *Post-Quantum Cryptography*, pages 233–244. Springer, 2016.

[RJH+18] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Side-channel Assisted Existential Forgery Attack on Dilithium-A NIST PQC candidate., 2018. https://eprint.iacr.org/2018/821.

[RRDC+16] Oscar Reparaz, Sujoy Sinha Roy, Ruan De Clercq, Frederik Vercauteren, and Ingrid Verbauwhede. Masking Ring-LWE. *Journal of Cryptographic Engineering*, 6(2):139–153, 2016.

[Sho97] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.

[SRB20] Sujoy Sinha Roy and Andrea Basso. High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):443–466, Aug. 2020.

[WGY20] Bin Wang, Xiaozhuo Gu, and Yingshan Yang. Saber on ESP32. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security*, pages 421–440, Cham, 2020. Springer International Publishing.

[XYD+19] Lu Xianhui, Liu Yamin, Jia Dingding, Xue Haiyang, He Jingnan, and Zhang Zhenfei. LAC: Lattice-based Cryptosystems, 2019. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions.

[YZMZBY+20] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. A High-performance Hardware Implementation of Saber Based on Karatsuba Algorithm. Cryptology ePrint Archive, Report 2020/1037, 2020. https://eprint.iacr.org/2020/1037.