Polynomial Multiplication with Contemporary Co-Processors

Beyond Kronecker, Schönhage-Strassen & Nussbaumer

Joppe W. Bos, Joost Renes and Christine van Vredendaal

NXP Semiconductors
{joppe.bos,joost.renes,christine.van.vredendaal}@nxp.com

Abstract. We discuss how to efficiently utilize contemporary co-processors used for public-key cryptography for polynomial multiplication in lattice-based cryptography. More precisely, we focus on polynomial multiplication in rings of the form $\mathbb{Z}[X]/(X^n + 1)$. We make use of the roots of unity in this ring and construct the Kronecker+ algorithm, a variant of Nussbaumer designed to combine especially well with the Kronecker substitution method, This is a symbolic NTT of variable depth that can be considered a generalization of Harvey's multipoint Kronecker substitution. Compared to straightforwardly combining Kronecker substitution with the state-of-the-art symbolic NTT algorithms Nussbaumer or Schönhage-Strassen, we halve the number or the bit size of the multiplied integers, respectively. Kronecker+ directly applies to the polynomial multiplication operations in the lattice-based cryptographic schemes Kyber and Saber, and we provide a detailed implementation analysis for the latter. This analysis highlights the potential of the Kronecker+ technique for commonly available multiplier lengths on contemporary co-processors.

Keywords: Polynomial multiplication, Kronecker substitution, Schönhage-Strassen, Nussbaumer, Co-processors.

1 Introduction

There are currently over 30 billion IoT (Internet of Things) devices installed worldwide [Maa]: this number has steadily outgrown the number of humans living on this planet and is expected to keep increasing. To secure these and many other devices, Elliptic Curve Cryptography (ECC) [Kob87, Mil86] and the Rivest–Shamir–Adleman (RSA) [RSA78] algorithm are vital components in our public-key infrastructure based on secure key exchange and digital signatures. On these embedded (IoT) devices speed is a key performance indicator. To enable them to securely and efficiently execute the complex cryptographic algorithms, many have access to dedicated hardware accelerators or so-called *co-processors*. Typically, for ECC and RSA these co-processors consist of a hardware-supported instruction set that enables the device to compute large-integer arithmetic routines efficiently.

With the steady progress in the development of a quantum computer, the security of our public-key infrastructure is being threatened. When a full-scale quantum computer would become a reality, Shor's algorithm [Sho94] is able to recover private keys used in ECC/RSA in polynomial time. To prepare for this threat, alternative cryptographic algorithms are necessary; these are called post-quantum, or quantum-safe, cryptographic algorithms. In an effort to standardize such algorithms the US National Institute of Standards and Technology (NIST) put out a call for proposals [Nat] for cryptographers to submit candidate algorithms in 2016. As of July 2020, seven out of fifteen remaining candidates are marked as finalists of which a subset is expected to be standardized in the upcoming three years.

For embedded devices the migration to completely new public-key cryptography algorithms results in multiple practical challenges. None of the seven finalists require large integer arithmetic, which is the computationally expensive operation in both ECC and RSA and exactly what is offered by existing public-key co-processors. Adding new dedicated hardware support means a significant investment for new generations of devices that cannot be started yet as it is not clear which candidate schemes will be standardized. Additionally, the design, testing and (most prominently) the migration time for these co-processors is expected to span many years.

Five of the finalist do have in common that they are so called ring-based lattice schemes. For these schemes the primary bottleneck in terms of performance is to multiply polynomials with integer coefficients: a typical example is to work with polynomials from $\mathbb{Z}[X]/(X^n+1)$ where n is a power of two. At CHES 2019, Albrecht, Hanser, Hoeller, Pöppelmann, Virdia and Wallner [AHH+18] proposed to apply Harvey's improvements [Har09] to Kronecker substitution [Kro82, Sch82] to convert polynomial multiplication to large integer multiplication and thereby unlock the potential of the already existing co-processors. This approach was explored by Wang, Gu and Yang for application to post-quantum crypto scheme Saber [WGY20].

In this paper we expand on this line of research and present a new method to realize fast polynomial arithmetic implementations on embedded devices which have access to commonly used arithmetic co-processors. We show how one can generalize Harvey's negated-evaluation-points technique such that it works in polynomial rings as frequently used in post-quantum cryptography. Our new method also be viewed as a *variant* of Nussbaumer polynomial multiplication [Nus80] combined with Kronecker substitution, which we call *Kronecker+*. It opens up the possibility to compute a *symbolic* Number Theoretic Transform (NTT) in the polynomial rings used in the post-quantum cryptographic submissions, which in turn can be computed by multiple smaller integer multiplications using Kronecker substitution in a clever way. This allows, for example, schemes such as Saber, which due to their parameter choices could not use NTTs, to compute symbolic NTTs. On contemporary co-processors this results in a more efficient polynomial multiplication, compared to existing approaches such as Schönhage-Strassen [SS71] and Nussbaumer [Nus80]. More concretely, although the overhead of the forward and backward transforms is similar to that of Nussbaumer and Schönhage-Strassen, Kronecker+ halves the number or the bitsize, respectively, of the required multiplications when compared to the aforementioned methods. In Table 1 an overview is presented of combining different approaches with Kronecker, including the new approach from this paper: Kronecker+.

In the setting of the NIST finalist Kyber (which are almost identical to Saber for our purposes), and more specifically for the parameters of Kyber-768 ($n = 256$ and $\ell = 32$), in [AHH+18] it is shown that one polynomial multiplication can be computed using the standard Kronecker substitution approach with a single multiplications of 8197 bits, and using Harvey's negated-evaluation-points technique with two integer multiplication of 4097 bits each. Kronecker+ enables further division into exactly $t = 2^\tau$ integer multiplications of $8196/t + 1$ bits each, where $\tau < 6$ is a positive integer, where the exact optimal choice of τ will strongly depend on platform-specific details.

Of course one could use asymptotically faster multiplication methods for the one or two large integer multiplications. For example, Karatsuba [KO62] replaces one multiplication of b bits with three multiplications of $b/2$ bits plus some overhead in the form of additions (or subtractions) and can be applied recursively. Moreover, r-way Toom-Cook [Too63, Coo66] generalizes this multiplication approach and replaces one b-bit multiplication with $2r - 1$ multiplications of approximately b/r bits plus some overhead for the evaluation and interpolation formula used (r = 2-way Toom-Cook is approximately equivalent to one

Table 1: Comparison of number of multiplications of certain bit length required for multiplying two polynomials with n coefficients each. Here ℓ is the parameter for Kronecker substitution (i.e., evaluating at 2^ℓ) and t specifies the depth (if applicable) of the algorithm.

Algorithm	# Muls	# Bits
Kronecker (KS1)	1	$\ell n + 1$
Harvey (KS2/KS3)	2	$\ell n/2 + 1$
Harvey (KS4)	4	$\ell n/4 + 1$
Kronecker + Karatsuba	$3^{\log t}$	$(\ell n + 1)/t$
Kronecker + Toom-Cook-t	$2t - 1$	$(\ell n + 1)/t$
Kronecker + Schönhage-Strassen	t	$2\ell n/t + t + 1$
Nussbaumer + Kronecker	$2t$	$\ell n/t + 1$
Kronecker+ (this work)	t	$\ell n/t + 1$

layer of Karatsuba). This immediately highlights the potential of our new approach: while r-way Toom-Cook can reduce one b-bit multiplication to $2r - 1$ multiplications this can be done with r multiplications using Kronecker+ (where both approaches reduce to approximately b/r-bit multiplications). A high-level overview on how to perceive our contribution in light of the many other available multiplication approaches one could try with Kronecker (something which was not done or considered in case of Schönhage-Strassen and Nussbaumer in previous work) is shown in Table 1.

It should be noted that the number of multiplications of course does not tell the full story, since each of the methods is accompanied by transformational overhead (usually in the form of additions or multiplications by small constants). However, the overhead of Kronecker+ is comparable to that of a regular NTT and of complexity $t \log t$ by employing the Cooley-Tukey butterfly approach [CT65]. In fact, because of the smaller sizes of integers that we operate on, the overhead will be smaller than for Schönhage-Strassen and Nussbaumer. Especially for small values of τ, the overhead of the transformations is even small when compared to a layer of Karatsuba of Toom-Cook.

Lastly, we would like to emphasize that this paper introduces a new method to realize fast polynomial arithmetic implementations using existing public-key co-processors: this is *not* an implementation paper. We would like to point out that there is no standardized interface to or standard functional behavior of public-key co-processors. This means each co-processor is inherently vendor specific and has different bells and whistles one could utilize. Moreover, external parties can not directly program these co-processors and hence implementation results cannot be publicly verified. The proposed polynomial multiplication technique in this paper can be applied to any co-processor which implements a large bit-size multiply-and-add accelerator and §4 gives various implementation considerations for the security engineer who wants to try this out (accompanied by Sage implementations of the high-level approach combined with lower level optimizations).

2 Preliminaries

Let $f = \sum_{i=0}^{n-1} f_i X^i$ and $g = \sum_{i=0}^{n-1} g_i X^i \in \mathbb{Z}[X]$ be two polynomials of degree less than n. In this section we describe various methods that exist in the literature to compute the multiplication $h = (f \cdot g) \bmod (X^n + 1)$. For many algorithms the reduction modulo $X^n + 1$ has little effect, as the reduction is only applied (in a straightforward manner) after the more involved multiplication in $\mathbb{Z}[X]$. However, we include it here as it is crucial for some of the polynomial multiplication algorithms under discussion (e.g., Nussbaumer and Schönhage-Strassen) and will be relevant for cryptographic applications that apply them (e.g., Kyber and Saber). For these algorithms we introduce an additional parameter t, which is a positive integer dividing n.

Note that in this section we are assuming that the polynomial multiplications is performed in (quotient rings of) $\mathbb{Z}[X]$. Most algorithms however can be applied more generally over other rings $\mathcal{R}$. In this case there is an extra assumption that multiplication by t and $2t$ are injective maps, and hence that they can be inverted. This holds of course for $\mathbb{Z}$, but does not necessarily hold for general rings $\mathcal{R}$.

All algorithms that we describe are well known, so one could argue that a detailed description is not necessary. We have opted to include it here for completeness, as they appear scattered over the literature with varying notation and level of detail. We add the relevant references in the respective sections. Moreover, presenting them in a single framework with unified notation will make it easier to introduce our own contributions (and, hopefully, highlight the elegance of its simplicity).

Finally, inspired by Harvey [Har09], we set up a small running example to ease the comparison between different (relevant) multiplication methods.

Example 1. Let f and g be the two arbitrarily chosen polynomials

$$f(X) = -3 - 3X^2 - X^3 - 2X^5 + X^6 - 3X^7 ,$$
$$g(X) = 3 + X + 3X^2 + 2X^3 - 2X^4 + 3X^5 + 3X^6 ,$$

whose coefficients can be represented with 3 bits in the interval $[-3, 3]$. The goal is to compute the product $h = (f \cdot g) \bmod X^8 + 1$, which is easily checked to be the polynomial $h(X) = 7 + 3X - 4X^2 - 15X^3 + 2X^4 - 15X^5 - 4X^6 - 21X^7$.

2.1 Polynomial Multiplication

2.1.1 Karatsuba and Toom-Cook

Karatsuba [KO62] and its generalization Toom-Cook [Too63, Coo66] are multiplication methods which are asymptotically faster compared to the schoolbook algorithm, which runs in $\mathcal{O}(N^2)$ for $N \times N \to 2N$ bit multiplication. The idea behind k-way Toom-Cook (where $k = 1$ is equal to schoolbook and $k = 2$ (essentially) to Karatsuba) is to split the single N-bit multiplication into $2k - 1$ multiplications of approximately N/k bits such that the run-time is $\mathcal{O}(N^{\log(2k-1)/\log(k)})$. This is done by evaluating the polynomials at $2k - 1$ distinct points, and performing an interpolation after having performed $2k - 1$ smaller multiplications. See [BZ07] for more details on how to optimally compute the Toom-Cook multiplication.

The 2-way ($\mathcal{O}(N^{1.585})$), 3-way ($\mathcal{O}(N^{1.465})$) or 4-way ($\mathcal{O}(N^{1.404})$) version of Toom-Cook are popular approaches to multiply medium-sized integers and have been applied in a variety of settings in cryptography. For example, in the post-quantum secure key-exchange scheme Saber [DKRV18] the 4-way version of Toom-Cook is used, followed by two applications (i.e., layers) of Karatsuba for the polynomial multiplication.

2.1.2 Fast Fourier Transform in a Finite Field

Pollard showed how to define a transform in the finite field $\mathbb{Z}_q$ of integers modulo a prime q, analogous to the discrete Fourier transform, which can be computed using a Fast Fourier Transform (FTT) algorithm [Pol71]. In cryptography this is often referred to as the Number Theoretic Transform (NTT). In this case we want to compute a polynomial product of $f, g \in \mathbb{Z}_q[X]/(X^n + 1)$, where $2n \mid (q - 1)$, so that the multiplicative group $\mathbb{Z}_q^*$ contains a principal $2n$-th root of unity ζ. We can then use the Chinese remainder theorem to construct the isomorphism

$$\begin{aligned}\mathbb{Z}_q[X]/(X^n+1) &\cong \textstyle\prod_{i=0}^{n-1}\mathbb{Z}_q[X]/(X-\zeta^{2i+1})\\ f &\mapsto (f(\zeta^1), f(\zeta^3), \dots, f(\zeta^{2n-1}))\,.\end{aligned}$$

By applying this isomorphism to f and g, their product can be reduced to to n multiplications in $\mathbb{Z}_q$. As q is typically fairly small (e.g., 12 and 23 bits for the latest versions of Kyber and Dilithium, respectively), this is not interesting for modern co-processors which are aimed at hundreds or thousands of bits for ECC or RSA. Note that similar constructions can be made with n-th principal roots of unity, requiring only that $n \mid (q-1)$, which is done for instance by Kyber. We do not elaborate on this further here.

2.1.3 Nussbaumer

This algorithm was designed in 1980 and is named after its creator [Nus80]. We base our description on those of Bernstein [Ber97, §9] and the bachelor thesis of van der Lubbe [vdL16, §3.1]. The first step is to apply the transformation

$$\begin{aligned}\Psi : \mathbb{Z}[X]/(X^n+1) &\to (\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y-X^t)\,, \\ f = \textstyle\sum_{i=0}^{n-1} f_i X^i &\mapsto \Psi(f) = \textstyle\sum_{i=0}^{t-1}\big(\sum_{j=0}^{n/t-1} f_{i+jt}Y^j\big)X^i\,.\end{aligned} \tag{1}$$

As the polynomial $\Psi(f)$ has degree less than t in X, we can trivially lift it to $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]$ and view it as a polynomial in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(X^{2t}-1)$ where the coefficients of $\Psi(f)$ for the monomials X^i for $i = t, \dots, 2t-1$ are 0. Similarly we obtain $\Psi(g)$.

As the coefficient ring $\mathbb{Z}[Y]/(Y^{n/t}+1)$ contains the $2t$-th principal root of unity $\zeta = Y^{n/t^2}$, we can apply a cyclic convolution with respect to ζ. For this we require the additional restriction on t that $t^2 \mid n$, as opposed to only $t \mid n$. More concretely, we compute

$$\Psi(h)(\zeta^i) = \Psi(f)(\zeta^i)\cdot\Psi(g)(\zeta^i)\,, \quad \text{for } i = 0, \dots, 2t-1\,, \tag{2}$$

from which we can compute $2t \cdot H_j = \sum_{i=0}^{2t-1}\Psi(h)(\zeta^i)\zeta^{-ji}$. As $\Psi(h) = \sum_{j=0}^{2t-1} H_j(Y)X^j$, we can recover h by dividing by $2t$ (recall that multiplication by $2t$ was assumed to be injective), reducing modulo $Y - X^t$ and inverting Ψ.

The main cost of the algorithm is in Equation (2). The simultaneous evaluation of $\Psi(f)$ at all the roots of unity ζ^i can be computed with a Cooley-Tukey butterfly algorithm [CT65] with complexity $\mathcal{O}(t\log t)$. The multiplications are in the ring $\mathbb{Z}[Y]/(Y^{n/t}+1)$, so consist of $2t$ multiplications of polynomials with n/t coefficients each. Of course there is a clear possibility for recursion, but such an analysis is not of interest for our purposes.

This algorithm was used in cryptographic context by van der Lubbe [vdL16], who applied it to the post-quantum crypto scheme NewHope [PAA+19] in the ring $\mathbb{Z}[X]/(X^{1024}+1)$ with $t = 32$. In that case the 1024-coefficient polynomial multiplication is reduced to 64 multiplications of polynomials with 32 coefficients each. These can in turn be computed with various methods, but on their platform of choice (Haswell) van der Lubbe demonstrates that Karatsuba is the preferred choice. Nussbaumer has also been applied in the setting of homomorphic encryption [BLLN13].

Example 2. Let f and g be as chosen in Example 1. As Nussbaumer requires that $t^2 \mid n$, the largest choice for $n = 8$ we can make is $t = 2$. Splitting the ring, we find

$$\begin{aligned}\Psi(f) &= Y^3 - 3Y - 3 + (-3Y^3 - 2Y^2 - Y)\cdot X\,,\\ \Psi(g) &= 3Y^3 - 2Y^2 + 3Y + 3 + (3Y^2 + 2Y + 1)\cdot X\,,\end{aligned}$$

as polynomials modulo $Y - X^2$. Lifting to $(Z[Y]/(Y^4+1))[X]$ and viewing them as polynomials modulo $X^4 - 1$ by setting the coefficients of X^2 and X^3 to be zero, we can apply the cyclic convolution with respect to 4-th root of unity Y^2. In other words,

$$\begin{aligned}
[\Psi(f)(1), \Psi(f)(Y^2), \Psi(f)(Y^4), \Psi(f)(Y^6)] = &\qquad [-2Y^3 - 2Y^2 - 4Y - 3, -1,\\
&4Y^3 + 2Y^2 - 2Y - 3, 2Y^3 - 6Y - 5]\,,\\
[\Psi(g)(1), \Psi(g)(Y^2), \Psi(g)(Y^4), \Psi(g)(Y^6)] = &\qquad [3Y^3 + Y^2 + 5Y + 4, 5Y^3 - Y^2 + 3Y,\\
&3Y^3 - 5Y^2 + Y + 2, Y^3 - 3Y^2 + 3Y + 6]\,.
\end{aligned}$$

Multiplying pairwise modulo Y^4+1, we get $[\Psi(h)(1), \Psi(h)(Y^2), \Psi(h)(Y^4), \Psi(h)(Y^6)]$ as

$$[-31Y^3-25Y^2-23Y+12, -5Y^3+Y^2-3Y, 11Y^3+5Y^2+7Y+6, 25Y^3-5Y^2-45Y-30]\,.$$

Inverting the transform with respect to Y^2 (i.e., applying the same convolution with Y^6), we obtain

$$[-6Y^2 - 16Y - 3, -21Y^3 - 15Y^2 - 15Y + 3, -10Y^3 - 4Y^2 + 8Y + 12, 0]\,.$$

Finally setting $Y = X^2$, viewing the tuple as the coefficients of a polynomial in X (where 0 is the coefficient of X^3) and reducing modulo X^8+1, we obtain h.

2.2 Polynomial Multiplication with Integer Multipliers

The problem of multiplying polynomials and that of integer multiplication are extremely related. The two can be linked by way of Kronecker substitution, which we first expand on in §2.2.1. This method has been further investigated by Harvey and applied to Kyber and Saber, which we explain in §2.2.2. Finally we revisit the Schönhage-Strassen algorithm in §2.2.3.

2.2.1 Kronecker Substitution

In 1882, Kronecker introduced a method to reduce computational problems related to multivariate polynomials to those related to univariate polynomials [Kro82]. A hundred years later, a similar technique was introduced by Schönhage to reduce polynomial multiplications in $\mathbb{Z}[X]$ to integer multiplication (multiplication in $\mathbb{Z}$) [Sch82]. This approach is known as the *Kronecker substitution method.*

The idea behind the method is to evaluate the polynomials at a sufficiently high two-power 2^ℓ for a positive integer ℓ, and use the resulting integers as input for a regular integer multiplication by computing $h(2^\ell) = (2^\ell) \cdot g(2^\ell)$. Finally, the resulting integer $h(2^\ell)$ is converted back to its polynomial representation h. The result is correct if the coefficients of the resulting polynomial did not "mix" with each other, i.e. if the parameter $\ell \in \mathbb{Z}$ is sufficiently large.

The main advantage of this approach, computing a polynomial multiplication by way of an integer multiplication, is that well-studied and fast implementations of asymptotic integer multiplication methods can be used. It allows contemporary co-processors containing integer-multiplication acceleration for speeding up "classical" cryptography to be re-used for the polynomial multiplications that appear in post-quantum cryptographic primitives. This was first investigated by Albrecht, Hanser, Hoeller, Pöppelmann, Virdia and Wallner [AHH+18], who used an RSA co-processor for the implementation of Kyber-768, and subsequently applied by Wang, Gu and Yang to an implementation of Saber [WGY20].

Note that [AHH+18] applies this technique to polynomial multiplication modulo X^n+1, as opposed to generic multiplication. Interestingly, although the coefficients of $f \cdot g$ and $(f \cdot g) \bmod X^n+1$ differ, their *upper and lower bound* do not. Indeed, a coefficient of $f \cdot g$ can be the sum of *at most* n products of coefficients of f and g, while a coefficient of $(f \cdot g) \bmod X^n+1$ is the sum of *exactly* n such coefficient products. Therefore the choice of ℓ remains the same regardless of reduction modulo $X^n + 1$. In particular, this implies that reduction modulo $X^n + 1$ can also be done as an intermediate step in the Kronecker domain as reduction modulo $2^{n\ell} + 1$.

Example 3. Let f and g be as chosen in Example 1. As they have (at most) 8 coefficients and they lie in the interval $[-3, 3]$, the coefficients of h (modulo $X^n + 1$) lie in the interval $[-8 \cdot 3^2, 8 \cdot 3^2]$ and can therefore be represented with $\ell = 8$ bits. Therefore, we find

$$f(2^8) = -215893506177302531\,, \quad g(2^8) = 847714908832003\,,$$

and compute the product $h(2^8) \equiv 16932392214669820680 \bmod 2^{64} + 1$. Notice that here we apply the intermediate reduction modulo $X^8 + 1$ in the Kronecker domain as reduction modulo $2^{64} + 1$. We retrieve the coefficients of h by converting to a base-256 representation. As f and g have signed coefficients in $[-3, 3]$, it is important to also take the *signed* representation

$$h(2^8) = 7 + 3 \cdot 2^8 - 4 \cdot 2^{16} - 15 \cdot 2^{24} + 2 \cdot 2^{32} - 15 \cdot 2^{40} - 4 \cdot 2^{48} - 21 \cdot 2^{56}$$

with 8-bit limbs in $[-8 \cdot 3^2, 8 \cdot 3^2]$. Interestingly, one can also apply Kronecker substitution to the intermediate multiplication in Nussbaumer (see Example 2). Evaluating $\Psi(f)(Y^i)$ and $\Psi(g)(Y^i)$ for $i = 0, 2, 4, 6$ at 2^8 leads to the tuples

$$[-33686531, -1, 67239421, 33552891]\,, \; [50398468, 83821312, 50004226, 16581382]\,,$$

which are pairwise multiplied to $[-521737972, -83821312, 184878854, 419091170]$. The multiplications are to be reduced modulo $Y^4 + 1$, and hence modulo $2^{32} + 1$ in the Kronecker domain. From here the regular Nussbaumer algorithm can be followed, with an additional final recovery to polynomial representation. These are 4 multiplications of (approximately) 32 bits each, as opposed to 1 multiplication of 64 bits for regular Kronecker.

2.2.2 Multipoint Kronecker Substitution

The size of the integers that are multiplied when applying Kronecker substitution, which impacts the efficiency of the algorithm, is strongly related to the size of ℓ. Simply put, the larger ℓ, the larger the integers and the slower the multiplication. On the other hand, ℓ needs to be at least as large as the maximum bitlength of the coefficients of h in order to recover the polynomial h from $h(2^\ell)$ correctly.

One of the main observations made by Harvey [Har09, §3.3] was that the size of ℓ can be reduced by splitting up the polynomial evaluation into two parts. Assuming for simplicity that ℓ is even, Harvey computes

$$h(2^{\ell_2}) = f(2^{\ell_2})g(2^{\ell_2})\,, \quad h(-2^{\ell_2}) = f(-2^{\ell_2})g(-2^{\ell_2})\,,$$

where $\ell_2 = \ell/2$. He then observes that

$$h^{(0)}(2^\ell) = (h(2^{\ell_2}) + h(-2^{\ell_2}))/2\,, \quad h^{(1)}(2^\ell) = (h(2^{\ell_2}) - h(-2^{\ell_2}))/(2 \cdot 2^{\ell_2})\,,$$

where $h^{(i)}$ denotes the polynomial whose j-th coefficient equals the $(2j + i)$-th coefficient of h. In other words,

$$h^{(0)}(2^\ell) = \textstyle\sum_{j=0}^{n/2-1} h_{2j} 2^{j\ell}\,, \quad \text{and} \quad h^{(1)}(2^\ell) = \textstyle\sum_{j=0}^{n/2-1} h_{2j+1} 2^{j\ell}\,.$$

The coefficients of h can therefore be recovered as the ℓ-bit limbs $h^{(0)}(2^\ell)$ and $h^{(1)}(2^\ell)$.

Denoting by $M(b)$ the cost of multiplying two b-bit integers, this approach changes the cost of the polynomial multiplication in $\mathbb{Z}[X]$ from $M(\ell n) + \mathcal{O}(\ell n)$ in the case of standard Kronecker substitution, to $2 \cdot M(\ell n/2) + \mathcal{O}(\ell n)$. Here the big-$\mathcal{O}$ terms incorporate the cost of packing and unpacking. This can lead to a significant speedup whenever the cost of multiplying is (relatively) expensive. In particular, this approach was used in the ring $\mathbb{Z}[X]/(X^{256}+1)$ with applications to Kyber (see [AHH+18]) and Saber (see [WGY20]) to speedup their respective implementations based on co-processors.

In fact, Harvey considers a second approach to split up the evaluation into *four* parts by also evaluating at the *reciprocal* $f(2^{-\ell})$, that gives rise to multiplication with a cost of $4 \cdot M(\ell n/4) + \mathcal{O}(\ell n)$. We omit the details as we do not discuss it further. In particular, it was not considered practical in the previously mentioned implementations of Kyber and Saber.

Example 4. Let f and g be as chosen in Example 1 and choose $\ell = 8$ as in Example 3. We compute

$$[f(2^4), f(-2^4)] = [3504336126, 824184061]\,,$$
$$[g(2^4), g(-2^4)] = [53355283, 47047411]\,,$$

from which we obtain $[h(2^4), h(-2^4)] \equiv [2870021177, 1290988502]$ with two multiplications modulo $2^{32}+1$. It follows that

$$h^{(0)}(2^8) = 8 + 252 \cdot 2^8 + 1 \cdot 2^{16} + 252 \cdot 2^{24}\,,\ h^{(1)}(2^8) = 4 + 241 \cdot 2^8 + 240 \cdot 2^{16} + 234 \cdot 2^{24}\,,$$

or, in signed representation, that

$$h^{(0)}(2^8) \equiv 7 - 4 \cdot 2^8 + 2 \cdot 2^{16} - 4 \cdot 2^{24}\,,\ h^{(1)}(2^8) \equiv 3 - 15 \cdot 2^8 - 15 \cdot 2^{16} - 21 \cdot 2^{24}$$

modulo $2^{32}+1$. The coefficients from h can now simply be read off. Note that this requires only 2 multiplications of (about) 32 bits each, compared to 4 for Nussbaumer combined with Kronecker.

2.2.3 Schönhage-Strassen

For the description of the Schönhage-Strassen algorithm [SS71], we base ourselves on the nice exposition of the implementation in the GMP library [GKZ07, §1] and Bernstein's paper [Ber97, §9]. We assume that the integers we multiply are outputs of Kronecker substitution of the form $F = f(2^\ell)$ and $G = g(2^\ell)$, and we want to compute their product $H = h(2^\ell)$ in $\mathbb{Z}[X]/(2^{\ell n}+1)$, i.e., modulo the polynomial modulus $(X^n + 1)$ evaluated at 2^ℓ. Interestingly, we begin by viewing the integers as polynomials by applying the map

$$\Phi : \mathbb{Z}/(2^{\ell n}+1) \to \mathbb{Z}[X]/(X^t+1)$$
$$F = \textstyle\sum_{i=0}^{t-1} F_i \cdot 2^{\ell n/t} \mapsto \Phi(F) = \sum_{i=0}^{t-1} F_i \cdot X^i\,,$$

in other words viewing the $\ell n/t$-bit limbs as coefficients of a polynomial of degree (at most) $t-1$. Note that here we can assume that $F_t = 0$, as $F = f(2^\ell)$ is a polynomial with degree at most $n-1$ evaluated at 2^ℓ, and hence is strictly smaller than $2^{n\ell}$ (and similarly for G). It can be shown that the coefficients of $\Phi(F)\Phi(G)$ can be represented with $2\ell n/t + t$ bits [GKZ07, §1], implying that it can be recovered as the unique representative of the product of $\Phi(F)$ and $\Phi(G)$ embedded in $\mathbb{Z}/(2^{2\ell n/t+t}+1)[X]/(X^t+1)$. The main observation now is that the coefficient ring for this multiplication is $\mathbb{Z}/(2^{2\ell n/t+t}+1)$, which contains a principal $2t$-th root of unity $\zeta_t = 2^{2\ell n/t^2+1}$, under the additional assumption that $t^2 \mid 2\ell n$ (note that this is weaker than Nussbaumer, which requires $t^2 \mid n$). We can use the (principal) t-th root of unity ζ_t^2 to construct a *negacyclic* convolution to reduce this multiplication to t multiplications in $\mathbb{Z}/(2^{2\ell n/t+t}+1)$, of approximately $2\ell n/t$ bits each (assuming $n \gg t$), after which we can invert Φ to recover H.

Example 5. Let f and g be as chosen in Example 1. As we require that $t^2 \mid 2\ell n = 128$, the largest choice we can take is $t = 8$. In that case

$$\Phi(F) = [254, 255, 252, 254, 255, 253, 0, 253]\,, \quad \Phi(G) = [3, 1, 3, 2, 254, 2, 3, 0]\,,$$

representing polynomials in $\mathbb{Z}[X]/(X^8 + 1)$. Applying a negacyclic convolution with 8-th root of unity $\zeta_8^2 = 64$, we obtain the tuples

$$[3191766, 12617514, 13706294, 6361802, 15707175, 16751308, 5128135, 10424123]\,,$$
$$[1893579, 12329652, 12869428, 336707, 1760443, 3940051, 4415315, 12786500]\,,$$

that are multiplied pairwise to

$$[2864000, 10297389, 10680185, 15308322, 14753371, 6584086, 650929, 5442338],$$

modulo $2^{24} + 1$ (note that 24 here is the first multiple of 8 larger than 16). Inverting the negacyclic convolution gives the tuple

$$[-66031, -65274, -4, -62734, 66295, 66799, 66806, 67813]\,.$$

Viewing these as the 8-bit limbs of an integer, we obtain the product 16932392214669820680 of F and G in $\mathbb{Z}/(2^{64} + 1)$. This can be reverted to polynomial representation by inverting the Kronecker map (see Example 3).

2.3 Public-key hardware co-processors

A typical hardware accelerator or cryptographic co-processor enhances the security and performance of hash-functions, random number generation, symmetric key or public-key cryptography. In the last category, the core of this accelerator is typically dedicated to multiplication and accumulation of large integers. One possible way of thinking about such hardware-supported instructions used to construct arbitrary length multiplication routines, is that given w-bit inputs a, b, and c it computes

$$(a \oslash c_1) \times (b \ominus c_2) + c \odot d$$

where $\oslash$, $\ominus$, and $\odot$ are optional operations with optional inputs c_1, c_2 and d. Concrete examples are

- the multiply-and-accumulate instruction present on many modern computer architectures (omitting all optional operations),
- the multiply-and-accumulate-accumulate (where $\odot$ equals the "+" operation) as present on the ARMv6 and above, and
- the ARM barrel shifter where $\ominus$ could be a shift or rotate instruction.

The multiply-and-accumulate-accumulate instruction can be used as a building block for arbitrary length multiplication and therefore also Montgomery multiplication: making this an essential building block for the most time-consuming operation in both RSA and ECC. Given the word size w this then computes $d = a \cdot b + c + d$ where all inputs are $< 2^w$ and the output is $\leq (w-1)^2 + 2(w-1) < 2^{2w}$.

Although the exact internal bit size of these co-processors is often kept secret, the word size is expected to be larger than the native word size on the embedded device (which is typically 8, 16, or even 32 bits). Typical examples of such co-processors are NXP's P71D321 [NXP], Infineon's SLE 78 [Inf], or Espressif's ESP32 [Esp]. The accompanying technical document often state that these co-processors can be used to compute RSA (often up to 4096 bits) and ECC. It should be noted that the upper bound on the number of supported bits is often due to a restriction on the available memory.

3 Kronecker+

In this section we discuss a new multiplication technique that can be viewed as a generalization of the negated-evaluation-points idea by Harvey [Har09, §2.3] and as a variant of Nussbaumer when combined with Kronecker substitution. Its main improvement with respect to [Har09] is that there is less limitation on the depth: whereas Harvey's method reduces a polynomial multiplication to two integer multiplications that are half the length compared to Kronecker substitution, we allow reducing to t multiplications of fraction $(1/t)$ of the length. For this we require that $t \mid n$ and $t \mid \ell$, which in particular implies that that $t^2 \mid 2\ell n$ (as was needed for Schönhage-Strassen). Hence t cannot be chosen completely freely, but the degree of freedom is much larger than for Harvey.

Compared to Nussbaumer we reduce the number of multiplications that are necessary. As can be seen in §2.1.3, Nussbaumer requires $2t$ multiplications of polynomials with n/t coefficients each. Applying Kronecker (i.e., evaluating at 2^ℓ) we would compute $2t$ multiplications of approximately $\ell n/t$ bits each. Instead, Kronecker+ requires only t such multiplications. The overhead of the forward and backward transformations is comparable.

3.1 An Alternative Convolution

We begin the description by revisiting the Nussbaumer algorithm, and proposing an alternative version. Initially, this will seem to serve no purpose as it does not lead to a reduced number of operations for polynomial multiplication. However, we show in §3.2 that this variant combines much better with Kronecker substitution and that Harvey's negated evaluation points technique can be considered a special case of our algorithm.

As usual, we assume that f and g are polynomials of degree (at most) $n-1$ in $\mathbb{Z}[X]/(X^n+1)$. Our alternative convolution starts identical to Nussbaumer by applying the map Ψ from Equation (1), obtaining $\Psi(f)$ and $\Psi(g)$ in the ring

$$(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y-X^t)\,.$$

This map is also used by [AHH⁺18, §4], which they refer to as "splitting the ring". They view $\Psi(f)$ and $\Psi(g)$ as degree $t-1$ polynomials in X, and multiply them through the schoolbook or Karatsuba algorithm, leading to t^2 or $3^{\log t}$ multiplications in $\mathbb{Z}[Y]/(Y^{n/t}+1)$ respectively. Alternatively, the strategy of Nussbaumer could be taken: canonically lift to $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]$ and embed in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(X^{2t}-1)$ to apply a *cyclic* convolution to $[F_0,\ldots,F_{2t-1}]$ and $[G_0,\ldots,G_{2t-1}]$ with respect to $2t$-th principal root of unity $\zeta_{2t}=Y^{n/t^2}$, where $\Psi(f)=\sum_{i=0}^{t-1}F_i(Y)X^i$ and $\Psi(g)=\sum_{i=0}^{t-1}G_i(Y)X^i$. This leads to the $2t$ multiplications

$$\Psi(h)(\zeta_{2t}^i)=\Psi(f)(\zeta_{2t}^i)\cdot\Psi(g)(\zeta_{2t}^i)\,,\quad \text{for } i=0,\ldots,2t-1\,,$$

in $\mathbb{Z}[Y]/(Y^{n/t}+1)$.

However, for Kronecker+ we deviate from both these approaches. As opposed to Nussbaumer, we only consider the length-t tuples $[F_0,\ldots,F_{t-1}]$ and $[G_0,\ldots,G_{t-1}]$ and take the principal t-th root of unity $\zeta_t=Y^{2n/t^2}$. Further, we apply weight factors X^i to the i-th element, i.e., apply a *cyclic* convolution with respect to ζ_t to the length-t tuples

$$[F_0\cdot X^0\,,\ldots,F_{t-1}\cdot X^{t-1}]\,,\ [G_0\cdot X^0\,,\ldots,G_{t-1}\cdot X^{t-1}]\,.$$

This results in the tuples

$$\left[\textstyle\sum_{i=0}^{t-1}\zeta_t^{ij}F_iX^i\right]_j\,,\ \text{and}\ \left[\textstyle\sum_{i=0}^{t-1}\zeta_t^{ij}G_iX^i\right]_j\,.$$

An interesting observation at this point is that we can combine the application of Ψ and the convolution (including weight factors) in a single step, showing that the latter tuples are

Algorithm 1 Pseudo-algorithmic *simple* description of Kronecker+.

```
Input: $f, g \in \mathbb{Z}[X]/(X^n + 1)$ for a positive integer $n$, the Kronecker parameter $\ell$ and a
    positive integer $t$ such that $t \mid \ell$ and $t \mid n$, and $M_i = 2^{2i\ell n/t^2} \cdot 2^{\ell/t}$ for $0 \le i < t$
Output: $h = \sum_{i=0}^{n-1} h_i X^i = f \cdot g \bmod X^n + 1$
1: Compute $f(M_i)$ and $g(M_i)$ for $i = 0, \ldots, t-1$.
2: Compute $h(M_i) = f(M_i) \cdot g(M_i) \bmod 2^{\ell n/t} + 1$ for $i = 0, \ldots, t-1$.
3: Compute
       $$h^{(i)} = \frac{\sum_{j=0}^{t-1} 2^{2i(t-j)\ell n/t} h(M_i)}{t \cdot 2^{i\ell/t}} \bmod 2^{\ell n/t} + 1$$
   for $i = 0, \ldots, t-1$.
4: Recover $h_{i+tj}$ as the $j$-th $\ell$-bit limb of $h^{(i)}$ for $0 \le i < t$ and $0 \le j < n/t$.
```

simply equal to $[f(\zeta_t^j \cdot X)]_j$ and $[g(\zeta_t^j \cdot X)]_j$ respectively. Although this is nice conceptually, we expect that an implementation of this algorithm would most likely separate Ψ from the convolution, making it easier to apply Cooley-Tukey-style butterflies [CT65] to the computation (see §4).

Next we perform the t multiplications

$$h(\zeta_t^i \cdot X) = f(\zeta_t^i \cdot X) \cdot g(\zeta_t^i \cdot X)\,, \text{ for } i = 0, \ldots, t-1\,. \tag{3}$$

Inverting the convolution with respect to ζ_t (including dividing by t), removing the weight factors, and possibly performing an explicit reduction modulo $Y - X^t$, gives the result $\Psi(h)$. From this we can recover h by inverting Ψ.

It should be noted at this point that the polynomials $f(\zeta_t^i \cdot X)$ and $g(\zeta_t^i \cdot X)$ do not actually lie in $\mathbb{Z}[Y]/(Y^{n/t} + 1)$, but instead still in $(\mathbb{Z}[Y]/(Y^{n/t} + 1))[X]/(Y - X^t)$. Therefore we have reduced a single multiplication in $(\mathbb{Z}[Y]/(Y^{n/t} + 1))[X]/(Y - X^t)$ to t of them. This does not make any sense from a performance perspective, and we do not suggest to use this method as described here for polynomial multiplication. However, in the next section we show that this approach has significant advantages in combination with Kronecker substitution.

3.2 Applying Kronecker

The true strength of reducing to the multiplications in Equation (3) comes from applying the (slightly modified) Kronecker substitution

$$\begin{aligned} K : (\mathbb{Z}[Y]/(Y^{n/t} + 1))[X]/(Y - X^t) &\to \mathbb{Z}/(2^{\ell n/t} + 1) \\ F = \textstyle\sum_{i=0}^{t-1} F_i(Y) \cdot X^i &\mapsto \textstyle\sum_{i=0}^{t-1} F_i(2^\ell) \cdot 2^{i\ell/t}\,. \end{aligned}$$

The mapping K maps $Y \mapsto 2^\ell$ and $X \mapsto 2^{\ell/t}$ and ensures that the map is well-defined modulo $Y - X^t$. In particular, this maps

$$\zeta_t \mapsto 2^{2\ell n/t^2}\,, \quad f(\zeta_t^i \cdot X) \mapsto f(2^{2\ell n/t^2} \cdot 2^{\ell/t})\,, \quad g(\zeta_t^i \cdot X) \mapsto g(2^{2\ell n/t^2} \cdot 2^{\ell/t})\,.$$

Hence, the multiplications in Equation (3) can be reduced to t multiplications in $\mathbb{Z}/(2^{\ell n/t} + 1)$. This means computing t multiplications of $\ell n/t + 1$ bits each instead of a single multiplication of ℓn bits. Recall that combining Nussbaumer with Kronecker substitution leads to $2t$ such multiplications. For completeness, we summarize the proposed method in Algorithm 1 which we refer to as Kronecker+.

We can now see that Algorithm 1 is a generalization of the method of Harvey [Har09, §3.3]; setting $t = 2$ and $\zeta_2 = 2^{\ell n/2} \equiv -1 \bmod 2^{\ell n/2} + 1$ in Algorithm 1 is the same as applying Harvey's approach. In fact, we generalize his method by also considering the case $t > 2$, whereas Harvey does not go beyond $t = 2$ (at least not for negated evaluation points). This generalization is made possible by the existence of t-th roots of unity in $\mathbb{Z}[Y]/(Y^{n/t}+1)$ via the map Ψ on $\mathbb{Z}[X]/(X^n+1)$, which do not exist for generic integer polynomial multiplication in $\mathbb{Z}[X]$. Of course we can always embed any integer polynomial of degree (at most) $n-1$ into a ring of the form $\mathbb{Z}[X]/(X^{2n}+1)$ and apply Ψ to reduce to t multiplications in $\mathbb{Z}[Y]/(Y^{2n/t}+1)$. This comes at the cost of approximately doubling the bitsize for the multiplications.

To illustrate our algorithm, we provide an example. Note for comparison that in Example 6 we reduce the polynomial multiplication to 8 multiplications, each of 9 bits, using a transformation very similar to Schönhage-Strassen and Nussbaumer. However, Nussbaumer (see Example 2) requires that $t^2 \mid n$ and therefore only allows $t = 2$, while even in that case needing 4 multiplications (whereas Kronecker+ only uses 2). On the other hand, Schönhage-Strassen (see Example 5) allows for $t = 8$, but reduces to 8 multiplications of 25 bits each.

Example 6. Let f and g be as chosen in Example 1, where $n = 8$, and choose $\ell = 8$ as in Example 3. Therefore we can choose $t = 8$ as well, as it divides both n and ℓ. As $n = t$, splitting the ring simply gives $\Psi(f) = f$ and $\Psi(g) = g$. Therefore, multiplying by the weights X^i we get tuples

$$[-3, 0, -3X^2, -X^3, 0, -2X^5, X^6, -3X^7], [3, X, 3X^2, 2X^3, -2X^4, 3X^5, 3X^6, 0],$$

respectively. Applying the map K that maps $X \mapsto 2^{\ell/t} = 2$ gives

$$[254, 0, 245, 249, 0, 193, 64, 130], [3, 2, 12, 16, 225, 96, 192, 0],$$

where each entry is taken modulo $2^{\ell n/t}+1 = 2^8+1$. We now take the cyclic convolution with respect to the t-th root of unity $\zeta_t \mapsto 2^{2\ell n/t^2} = 4$ modulo 2^8+1, leading to the tuples

$$[107, 228, 53, 131, 248, 161, 94, 239] \text{ and } [32, 116, 51, 47, 61, 105, 254, 129].$$

These are multiplied pairwise to give $[83, 234, 133, 246, 222, 200, 232, 248]$. Inverting the convolution leads to $[7, 6, 241, 137, 32, 34, 1, 139]$, and undoing the weights $X^i \mapsto 2^{i\ell/t} = 2^i$ in the Kronecker domain gives $[7, 3, 253, 242, 2, 242, 253, 236]$. Finally, converting to a signed representation gives the tuple $[7, 3, -4, -15, 2, -15, -4, -21]$, whose entries correspond to the coefficients of h.

3.3 Efficient Implementation

Although the description in §3.2 is nice and compact, it is not immediately clear that this can be efficiently implemented. Indeed, polynomial evaluations $f(M_i)$ and $g(M_i)$ initially appear to be of quadratic complexity in t, while multiplications by roots of unity and divisions by t and $2^{i\ell/t}$ could be costly. Therefore we comment here on the circumstances in which Kronecker+ could best be used and show that it can be used efficiently on modern (embedded) computer platforms.

Although not technically necessary for Kronecker+, division by t is most efficient to implement whenever it is a power of two $t = 2^\tau$. This means it can simply be computed as a bit shift by τ bits, which can be done very cheaply. The same holds for division by the weights $2^{i\ell/t}$. As the Kronecker parameter ℓ will commonly be chosen to be a power of two as well (since it makes it easy to implement on modern platforms), the restriction $t \mid \ell$ is equivalent to $\tau \le \log(\ell)$. In particular, for polynomials with fairly small and few coefficients for which ℓ can be chosen to be small, this implies that τ will not grow very large. For example, for Kyber in the work of [AHH+18, §5.4], $\ell = 32$ was set, which would result in an upper bound of $\tau \le 5$ for Kronecker+. We emphasize that in this case applying $\tau = 2$ would have reduced the 8192-bit multiplication to 4 multiplications of approximately

2048 bits each, which can be handled by the RSA co-processor in [AHH+18]. Instead, they resort to Harvey's method with an additional layer of schoolbook multiplication, resulting in 8 multiplications of 2048 bits each. In this case Kronecker+ therefore halves the number of required multiplications: already showing its strength compared to alternatives.

There is further reason to set t to be a power of two. Line 1 of Algorithm 1 in §3.1 can be decomposed into three steps:

1. Compute $F_i(2^\ell)$ and $G_i(2^\ell)$ for $i = 0, \dots, t-1$,
2. Compute $2^{i\ell/t} \cdot F_i(2^\ell)$ and $2^{i\ell/t} \cdot G_i(2^\ell)$ for $i = 0, \dots, t-1$,
3. Compute $f(M_i) = \sum_{j=0}^{t-1} 2^{2ij\ell n/t^2} F_j(2^\ell) 2^{j\ell/t}$ and $g(M_i) = \sum_{j=0}^{t-1} 2^{2ij\ell n/t^2} G_j(2^\ell) 2^{j\ell/t}$.

The first step is essentially free if the coefficients are positive (as they can be all be represented with less than ℓ bits), since it is just a matter of reordering the coefficients of f and g. It becomes much more complicated when the coefficients are signed, since we have to take carries into account. We elaborate on this in more detail in §4.3.1.

The second step requires shifts by $2^{i\ell/t}$, which are cheap and often even free through the use of (for example) barrel shifters. Moreover, if $i\ell/t$ is a multiple of the word size of the platform, then such shifts can be implemented by simply relabeling words. There can potentially be an additional cost by adding a reduction modulo $2^{n\ell/t} + 1$, but this has linear cost and can even be completely avoided by using lazy reduction techniques.

The main cost comes from the third step. However, the main advantage of decomposing Line 1 of Algorithm 1 in this fashion should now be clear: computing the linear combinations has naïve complexity of $\mathcal{O}(t^2)$ operations, but can instead be implemented with complexity $\mathcal{O}(t \log t)$ by using Cooley-Tukey butterflies [CT65]. These are particularly easy to implement when t is a power of two. Note that the butterfly algorithm also requires several multiplications by roots of unity $2^{2ij\ell n/t^2}$, but these are constructed to be powers of two. Therefore these operations can be done with shifts, or by relabeling words if $2ij\ell n/t^2$ is a multiple of the word size. Completely analogous statements apply to Line 3 of Algorithm 1, which is in essence an inverse convolution.

Finally, since t and ℓ are powers of two, combined with the fact that $t \mid n$ and $\ell \mid n$, makes the choice of n also being a power of two a natural one. Note in particular that there are *no* restrictions on the coefficients of f and g themselves, besides the coefficients of their product being representable with ℓ bits. For example, we do not need that $2n \mid (q-1)$ and can freely choose q according to cryptographic requirements (as opposed to algorithmic ones). See, for example, [vdL16, §5.3] for further discussion in this direction. This implies for instance that the above algorithm applies to Kyber (where $q = 3329$ is prime) as well as Saber (where $q = 2^{13}$ is a power of two).

4 Cryptographic Application

The scenarios in which the described techniques can be applied are fairly clear: whenever operations in rings of the form $\mathbb{Z}[X]/(X^n + 1)$ are used. In particular, we can consider rings of the form $\mathbb{Z}_q[X]/(X^n + 1)$ and lift the coefficients to $\mathbb{Z}$. This still leaves many of the proposals selected as finalists in the NIST standardization effort to be considered, as all lattice-based submissions use rings of this form at one point or another. In this section we focus on KEMs as the required operations are often simpler, and comparison with existing work for Kyber [AHH+18] (with $(n, q) = (256, 3329)$) and Saber [WGY20] (where $(n, q) = (256, 2^{13})$) is possible. However, we emphasize that many analogous statements can for example be made about Dilithium [DKL+18] (with $(n, q) = (256, 8380417)$).

4.1 Polynomial Multiplication in Kyber and Saber

The arithmetic core in both Kyber and Saber is to multiply a $k \times k$ matrix $\mathbf{A}$ with a $k \times 1$ vector $\mathbf{s}$, where the entries of the matrix and vectors are elements of $\mathbb{Z}_q[X]/(X^n+1)$. More concretely, the main operation is to multiply and accumulate k polynomials as $b_i = \sum_{j=0}^{k-1} a_{i,j} \cdot s_j$, which in turn has to be performed for $0 \leq i < k$. When using Kronecker substitution, from a performance point of view, it is beneficial to multiply and accumulate these k polynomial in integer representation to avoid converting them back separately (requiring only k as opposed to k^2 inverse transformations of Kronecker substitution). In order to determine the required precision in Kronecker (i.e. the parameter ℓ), the bounds on the input need to be determined. Recall from §2.2.1 that the bound ℓ is independent of the modulus X^n+1. The coefficients of the polynomial $a_{i,j}$ are uniform in $\mathbb{Z}_q$ and can be represented in the interval $[-q/2, q/2-1]$ when q is even, or $[-(q-1)/2, (q-1)/2]$ when q is odd. The coefficients of s_j are samples in $[-2, 2]$ for Kyber, and in $[-\mu, \mu]$ for Saber where $\mu = 5, 4, 3$ depending on the security level. This means that the product of 2 coefficients lies in the interval $[-3328, 3328]$ for Kyber, requiring 12.7 bits. As each coefficient of the product is an accumulation of $k \cdot 256$ of such coefficients, for a signed version of Kronecker it is sufficient to set $\ell = \lceil 12.7 + \log(k \cdot 256) \rceil$ for Kyber. This results in $\ell = 23$ for security levels 1 and 3, and $\ell = 24$ for security level 5. Completely analogously, $\ell = 25$ suffices for all security levels of Saber.

However, as also noted in [AHH+18], it is beneficial to use $\ell = 32$ such that it aligns nicely with the byte boundaries and 32-bit datatypes on modern computer architectures. Moreover, many of the required steps in Kronecker simplify significantly as we outline in the remainder of this section. As we focus on Kyber and Saber, we shall also assume that $n = 256$ in the remainder of this section. As a result, evaluating a polynomial $f \in \mathbb{Z}_q[X]/(X^{256}+1)$ at $2^{\ell/t} = 2^{32/t}$ results in integers of at most $\ell \cdot n/t = 8192/t$ bits.

4.2 Compatibility and Performance Impact

We begin with a discussion on the compatibility of Kronecker substitution with Kyber and Saber. This is easy for Saber: as all elements are sampled directly into $\mathbb{Z}_q[X]/(X^n+1)$ we can immediately apply the Kronecker+ algorithm whenever polynomial multiplication is performed.

For Kyber the situation is different. The approach taken in the NIST submission optimizes for polynomial multiplication with the proposed NTT approach described in [SAB+19]. An example is that the large Kyber matrix $\mathbf{A} \in (\mathbb{Z}_q/(X^{256}+1))^{k \times k}$, for the Kyber modulus $q = 3329$ and the parameter $k \in \{2, 3, 4\}$ depending on the parameter set, is sampled directly into the NTT domain. These design decisions have an impact on the performance of alternative approaches. When considering the Kronecker approach, the authors of [AHH+18] note that this “basically nullify all gains from a different and faster algorithm for polynomial multiplication” and decide *not* to be compatible with the Kyber specification.

We re-iterate that this comment is absolutely right. For the encryption in Kyber-768, when using Kronecker, one needs 6 additional inverse NTTs, while saving 2 NTT and 3 inverse NTT operations. Unfortunately the cost of the inverse NTT (5248 multiplications) is almost twice that of the forward NTT (2688 multiplications): hence, the additional cost due to these extra inverse NTT operation nullifies any hope of a performance optimization. In other words, no matter how fast the arithmetic co-processor, a Kyber specification-compatible implementation using a Kronecker approach will always be slower.

Fortunately, in the setting of Saber the approach for encryption and decryption is virtually the same (even the degree $n = 256$ and the number of polynomials $k = 3$ is the same as in Kyber-768) with the exception that no NTT is used. Hence, the additional cost to compute the inverse NTTs is not present and a performance speed-up can be expected. This is further investigated in §4.7 and was also considered by Wang, Gu and Yang [WGY20] on the ESP32 platform [Esp], who apply Kronecker substitution and Harvey's negated point methods to Saber.

4.3 Polynomial to Integer Representation: "Snort"

The first step of Kronecker+ (see Line 1 of Algorithm 1) is to evaluate polynomials $f, g \in \mathbb{Z}_q[X]/(X^{256}+1)$ at $\zeta_t^i \cdot 2^{\ell/t}$ for $i = 0, \ldots, t-1$, where $\zeta_t = \zeta_t(2^{32}) = 2^{16384/t^2}$ as defined in §3.2, and to reduce modulo $2^{8192/t}+1$. The analogous operation in [AHH+18, Algorithm 7] is called "snorting". As mentioned in §3.1, we split this operation up into several parts which we will discuss separately. First, we apply the map Ψ from Equation (2), which [AHH+18, §4] refers to as "splitting the ring", to obtain t polynomials of degree $t-1$ in $\mathbb{Z}[Y]/(Y^{8192/t}+1)$. This step is essentially trivial, as we simply relabel the coefficients of f and g. Then we evaluate all t polynomials at 2^{32}, which we elaborate on in §4.3.1. Only afterwards do we apply an NTT with respect to the t-th root of unity, which is discussed in §4.3.2.

4.3.1 Kronecker Substitution after Splitting the Ring

Let $\Psi(f) = \sum_{i=0}^{t-1} \left(\sum_{j=0}^{256/t-1} f_{i+jt} Y^j \right) X^i$, such that splitting the ring leads to the t polynomials

$$F_i(Y) = \sum_{j=0}^{256/t-1} f_{i+jt} Y^j, \quad \text{for } i = 0, \ldots, t-1\,,$$

that are to be evaluated at 2^{32}. It is important to distinguish the cases where the coefficients f_{i+jt} are signed, and where coefficients are unsigned. The unsigned case again leads to a very straightforward application of Kronecker substitution: since all coefficients are strictly less than 2^{32} there is no overlap between coefficients and the 32-bit limbs are simply f_{i+jt}. If the word size of the co-processor is a multiple of $\ell = 32$ (which is a plausible assumption), this means we can simply put the coefficients in the right place in the processor words. This can be done with only logical "or" operations. The same of course also applies to g.

Unfortunately, in Kyber and Saber polynomials with signed coefficients are used. More specifically, the secret keys are polynomials whose coefficients are sampled from a centered binomial distribution, i.e., from $[-\mu, \mu]$ for some small (positive) $\mu < 6$. This leads to carries in the evaluation at 2^{32}, causing additional overhead [AHH+18, §5.3]. One possible optimization to lower the number of required shift and additions is to divide the coefficients into positive and negative values and then put as many 32-bit coefficients into "free slots" of the large integer values and add / subtract them together. However, as the coefficients for these polynomials in Kyber and Saber are secret, this is difficult to accomplish in a constant-time manner without leaking the sign bits of the individual coefficients.

Alternatively, the coefficients can be converted to signed representation. As they are elements of $\mathbb{Z}_q$, we can represent $-\mu$ as the positive integer $q - \mu$ and evaluate at 2^{32}. However, this impacts the Kronecker parameter ℓ as the coefficients of the secret key would lie in the interval $[q-\mu, q+\mu]$ as opposed to $[-\mu, \mu]$. In the particular cases of Kyber and Saber this would imply that the coefficients of products can no longer be represented with 32 bits, and we have to resort to $\ell = 64$. This has significant impact on the performance. Note that this already assumes we add q to both the negative as well as the positive coefficients, as adding it to only the negatives ones could again leak information about the secret.

To remedy the impact of signed coefficients on the Kronecker parameter ℓ, we subtract q again from the coefficients. We do this *after* the Kronecker substitution to get both the advantage of not having carries, as well as not needing to increase ℓ.. As Kronecker substitution is commutative with respect to the subtraction of polynomials and integers,

$t = 2$

$$(F_0, F_1) = (F_0 + F_1, F_0 - F_1).$$

$t = 4$

$$\begin{aligned}
&(F_0, F_1, F_2, F_3) = (F_0 + F_1, F_0 - F_1, F_2 + F_3, F_2 - F_3),\\
&(F_0, F_2, F_1, F_3) = (F_0 + F_2, F_0 - F_2, F_1 + (F_3 \ll 1024), F_1 - (F_3 \ll 1024)).
\end{aligned}$$

$t = 8$

$$\begin{aligned}
&(F_0, F_1), (F_2, F_3) = (F_0 + F_1, F_0 - F_1), (F_2 + F_3, F_2 - F_3),\\
&(F_4, F_5), (F_6, F_7) = (F_4 + F_5, F_4 - F_5), (F_6 + F_7, F_6 - F_7),\\
&(F_0, F_2), (F_1, F_3) = (F_0 + F_2, F_0 - F_2), (F_1 + (F_3 \ll 512), F_1 - (F_3 \ll 512)),\\
&(F_4, F_6), (F_5, F_7) = (F_4 + F_6, F_4 - F_6), (F_5 + (F_7 \ll 512), F_5 - (F_7 \ll 512)),\\
&(F_0, F_4), (F_1, F_5) = (F_0 + F_4, F_0 - F_4), (F_1 + (F_5 \ll 256), F_1 - (F_5 \ll 256)),\\
&(F_2, F_6) = (F_2 + (F_6 \ll 512), F_2 - (F_6 \ll 512)),\\
&(F_3, F_7) = (F_3 + (F_7 \ll 768), F_3 - (F_7 \ll 768)).
\end{aligned}$$

Figure 1: Example operations required for the convolution with t-th root of unity $2^{16384/t^2}$ for choices $\tau = 1, 2, 3$, where all operations take place in the ring $\mathbb{Z}/(2^{8192/t} + 1)$. Here we write $F_i = 2^{32i/t} \cdot F_i(2^{32})$ and compute the NTT in place.

this amounts to performing the same computation. In other words,

$$F_i(2^{32}) = \sum_{j=0}^{256/t-1} f_{i+jt} \cdot 2^{32j} = \sum_{j=0}^{256/t-1} (f_{i+jt} + q) \cdot 2^{32j} - \sum_{j=0}^{256/t-1} q \cdot 2^{32j}$$

for $i = 0, \dots, t-1$. Note that these are subtractions modulo $2^{8192/t} + 1$ with the fixed polynomial $\sum_{j=0}^{256/t-1} q \cdot 2^{32j}$ that can be stored in advance. Although this involves extra additions and subtractions, these can easily be made independent of the secret key and therefore leak no information. To finalize the Kronecker subtitution step we multiply $F_i(2^{32})$ by $2^{32i/t}$ by left shifting with $32i/t$ bits.

4.3.2 Applying the NTT

Having obtained $2^{32i/t} \cdot F_i(2^{32})$ for $i = 0, \dots, t-1$, we can apply the NTT with respect to t-th root of unity $2^{16384/t^2}$ modulo $2^{8192/t} + 1$, i.e., to compute

$$f(2^{16384i/t^2} \cdot 2^{32/t}) = \sum_{j=0}^{t-1} 2^{16384ij/t^2} 2^{32j/t} F_j(2^{32}) \bmod 2^{8192/t} + 1$$

for $i = 0, \dots, t-1$. At first glance this seems to require many multiplications by the roots of unity $2^{16384ij/t^2}$. However, one can see that many of them vanish modulo $2^{8192/t} + 1$. For example, for $t = 1, 2$ we have $2^{16384ij/t^2} \equiv 1 \bmod 2^{8192/t} + 1$ for all i, j. For $t = 4$ we have $2^{16384ij/t^2} = 2^{1024ij}$ which is non-trivial modulo 2^{2048} only whenever both i and j are odd. Similarly, for larger t many of the multiplications by roots of unity vanish.

We perform the NTT computation with typical Cooley-Tukey butterflies [CT65], requiring $t \log t$ additions/subtractions [Sei18]. To make this more concrete, we summarize the NTT operations for $t = 2, 4, 8$ in Figure 1. Recall that the requirement that $t = 2^\tau \mid 32 = \ell$ implies that $\tau < 6$ in the case of Kyber and Saber. We omit $t = 16, 32$ in Figure 1 because they are tedious to write down (and read), but they are structured similarly. The operations demonstrate that the number of additions/subtractions is simply $t \log t$, as expected. In particular, it shows that the case $t = 2$ reduces to the same operations as computed by Harvey [Har09].

Although all operations here are performed modulo $2^{8192/t}+1$, a standard lazy reduction techniques can be applied to avoid any modular reductions until the very end. This will be particularly true when the co-processor has w-bit words with $w \mid (8192/t)$, in which case we need exactly $8192/(tw) + 1$ registers with a total of $8192/t + w$ bits to represent $(8192/t + 1)$-bit integers. In that case we would have $w - 1$ redundant bits that can be used for lazy reduction.

Finally we make some remarks on implementing the multiplications (i.e., bit shifts) by roots of unity. As a guiding example consider the operation $F_1 = F_1 + (F_3 \ll 1024)$ from Figure 1 for $t = 4$. Writing $F_1 = F_1^L + 2^{1024} F_1^H$ and $F_3 = F_3^L + 2^{1024} F_3^H$ as a decomposition of two (approximately) 1024-bit sequences, one can see that

$$F_1 + (F_3 \ll 1024) \equiv (F_1^L - F_3^H) + 2^{1024} \cdot (F_1^H + F_3^L) \bmod 2^{2048} + 1\,.$$

Ideally, we could implement this as two 1024-bit additions/subtractions, and therefore having no overhead over a single 2048-bit addition/subtraction. The caveat is that $F_1^L - F_3^H$ might produce a carry, which needs to be dealt with (in constant time). In the worst case the carry is subtracted (and propagated) from $2^{1024} \cdot (F_1^H + F_3^L)$ with the approximate cost of a 1024-bit subtraction, but many platforms might be able to handle this much more efficiently. More generally, bit shifts can in the worst case be implemented with $2^{8192/(2t)}$-bit additions/subtractions (on average), and be (essentially) free in the best case.

4.4 Multiplication of Integers

The multiplications in Line 2 of Algorithm 1 are technically the most straightforward operations: simply multiplying two integers modulo $2^{8192/t} + 1$. There are a few of remarks that can be made about the size of the multiplier. As integers modulo $2^{8192/t} + 1$ can be represented in $8192/t + 1$ bits, while t is a power of 2, we will need a slightly awkward size multiplier. For example, for $t = 32$ we need a 257-bit multiplier. A similar problem arises in [AHH+18], who use a multiplier that can handle integers slightly larger than 2048 bits.

A particularly interesting case is where the internal word size w of the co-processor is fairly small. As was the case for lazy reduction, we need exactly $8192/(tw) + 1$ registers with a total of $8192/t + w$ bits to represent the $(8192/t + 1)$-bit integers. As the $w - 1$ redundant bits should be more than enough to accumulate the values during the forward NTT (e.g., when $w = 64$ or $w = 128$), no reductions modulo $2^{8192/t} + 1$ are necessary before multiplying values with $8192/(tw) + 1$ limbs each.

Moreover, a typical co-processor will support multiply-and-accumulate operations that do not need require dedicated additions to accumulate the intermediate products. In that case the additions do not add to the complexity of the algorithm. Note that in the case of Kyber and Saber we perform a k-way accumulation of polynomial products, which can also be achieved with multiply-and-accumulate operations without explicit additions. On the other hand, it will be necessary to reduce the products modulo $2^{8192/t} + 1$. This can be done with t subtractions of about $8192/t$ bits each. By accumulating before reducing we require only k such reductions (as opposed to k^2).

4.5 Inverting the NTT

The inverse NTT operation is essentially the same as the forward transformation described in §4.3.2 (with inverted roots of unity). The main difference is that we do not need to split the ring, i.e., do not need an application of Ψ. Furthermore, as we only invert the result of the $k \times k$ matrix multiplication with a $k \times 1$ vector, we only apply an inverse NTT to k polynomials (as opposed to $k^2 + k$ forward transformations). This implies that the cost of the inverse is much less significant. A similar statement applies to the dot products performed in Saber and Kyber. Finally, we can implement division by $t \cdot 2^{32i/t}$ by another application of bit shifting, since $2^{-32i/t} \equiv -2^{(8192-32i)/t} \bmod 2^{8192/t} + 1$.

Algorithm 2 Sneeze-Fast with modulus $m \in \mathbb{Z}[X]$ where m is monic of degree n from [AHH+18].

```
Input: $\ell \in \mathbb{Z}$ (bit length); $G \in \mathbb{Z}/m(2^\ell)$
Output: $h = \sum_{i=0}^{n-1} h_i X^i \in \mathbb{Z}[X]/(m)$
    with $h_i \in [-2^{\ell/2}, 2^{\ell/2} - 1]$
 1: $n \leftarrow \deg(m)$
 2: $G_{-1} \leftarrow G$
 3: $c \leftarrow 0$
 4: for $i = 0, 1, \ldots, n-1$ do
 5:     $e_i \leftarrow G_{i-1} \bmod 2^\ell$
 6:     $G_i \leftarrow (G_{i-1} - e_i)/2^\ell$
 7:     $e_i \leftarrow e_i + c$
 8:     if $e_i > 2^{\ell-1}$ then
 9:         $e_i \leftarrow e_i - 2^\ell$
10:         $c \leftarrow 1$
11:     else
12:         $c \leftarrow 0$
13:     $h_i \leftarrow e_i$
14: for $i = 0, 1, \ldots n-1$ do
15:     $h_i \leftarrow h_i - m_i \cdot (G_{n-1} + c)$
```

Algorithm 3 Optimized sneeze for $m = X^{256} + 1$ ($n = 256$ and $\ell = 32$).

```
Input: $t$ integers $a_i = \sum_{j=0}^{256/t} a_{i,j} \cdot 2^{32i}$ with
    $a_{i,j} \in [0, 2^{32} - 1]$
Output: $h = \sum_{i=0}^{n-1} h_i X^i \in \mathbb{Z}[X]/(m)$ with
    $h_i \in [-2^{16}, 2^{16} - 1]$
1: for $i = 0, 1, \ldots, t-1$ do
2:     carry $\leftarrow 0$
3:     for $j = 0, 1, \ldots, 256/t - 1$ do
4:         limb $\leftarrow a_{i,j} +$ carry
5:         carry $\leftarrow$ (limb $\gg 31$) | ($a_{i,j} \gg 31$)
6:         $h_{j \cdot t + i} \leftarrow$ limb
7:     $h_i \leftarrow h_i -$ (carry $+ a_{i,256/t}$)
```

4.6 Integer to Polynomial Representation: "Sneeze"

The final step in the Kronecker substitution (see Line 3 of Algorithm 1) is to convert the integer(s) result of the inverse NTT back to the polynomial representation in $\mathbb{Z}[X]/(X^{256} + 1)$. This conversion is denoted as "sneeze" in [AHH+18] and also supports signed coefficients. The basic approach behind this algorithm can be explained most easily by considering the unsigned variant such that the output coefficients of $h = f \cdot g$ are all *positive* and when the required precision matches the computer word size: $\ell = 32$. The generic algorithm as presented in Algorithm 2 from [AHH+18] then boils down to computing nothing: just re-interpret the array of 32-bit words of the integer resulting from the inverse NTT when written down in the radix-2^{32} representation $a = \sum_i a_i \cdot 2^{32i}$, where $0 \leq a_i < 2^{32}$, as the array which represent the coefficients of $h = \sum_i h_i \cdot X^i$, where $0 \leq h_i < 2^{32}$: i.e., simply use $h_i = a_i$. For the signed setting, where the output coefficients h_i can be negative, slightly more work is needed.

In the setting of Kyber and Saber, where we assume that $\ell = 32$, this means that even for the signed version of the conversion from integer to polynomial the algorithm can be simplified significantly. For example, Line 9 checks if a 32-bit integer is greater than 2^{31} and, if so, subtracts 2^{32}: when using two's complement to represent signed integers (as is common on virtually all modern computer platforms) this operation can be omitted. It checks if the most significant bit is set in the *unsigned* representation and if so makes the number negative by setting the most significant bit in the *signed* representation. Hence, this can be done by simply reinterpreting (or casting) the integer. Moreover, the (potentially non-constant time) "if" statement to check for carries is implemented as a simple "or" operation on two bits. These optimizations are shown in our generic setting in Algorithm 3.

4.7 Performance Impact

In this section we give a crude estimate of the performance impact of Kronecker+ to determine if and when our approach becomes beneficial in practice. We present here a

Table 2: The number of required w-bit multiplication and addition instructions needed in Saber encryption and decryption using a arithmetic co-processor working on w bits and the Kronecker+ approach working with t integers.

		t					
w	op.	1	2	4	8	16	32
32	#M	792 588	399 384	202 800	104 544	55 488	31 104
	#A	0	4 902	10 507	16 663	23 579	32 015
64	#M	199 692	101 400	52 272	27 744	15 552	9 600
	#A	0	2 470	5 339	8 607	12 559	17 993
128	#M	50 700	26 136	13 872	7 776	4 800	3 456
	#A	0	1 254	2 755	4 579	7 049	10 982
256	#M	13 068	6 936	3 888	2 400	1 728	1 536
	#A	0	646	1 463	2 565	4 294	7 011

(a) Saber encryption, the Toom-Cook/Karatsuba approach uses approximately 201 156 uint16_t multiplications and 312 396 uint16_t additions.

		t					
w	op.	1	2	4	8	16	32
32	#M	198 147	99 846	50 700	26 136	13 872	7 776
	#A	0	1 806	3 871	6 139	8 687	11 795
64	#M	49 923	25 350	13 068	6 936	3 888	2 400
	#A	0	910	1 967	3 171	4 627	6 629
128	#M	12 675	6 534	3 468	1 944	1 200	864
	#A	0	462	1 015	1 687	2 597	4 046
256	#M	3 267	1 734	972	600	432	384
	#A	0	238	539	945	1 582	2 583

(b) Saber decryption, the Toom-Cook/Karatsuba approach uses approximately 50 289 uint16_t multiplications and 78 099 uint16_t additions.

simplistic analysis which will not reflect reality (at least not in all contexts) as this is heavily vendor-specific. For this comparison we use the Saber security level 3 ($n = 256$ and $k = 3$) parameter set for Round 2 of the NIST PQC Standardization. As in §4.2, we focus on the main arithmetic operations performed in encryption and decryption. In Saber the use of two-power moduli precludes NTT-like multiplication algorithms, so the designers resort to a combination of Toom-Cook and Karatsuba.

In Table 2 the number of addition and multiplication instructions are given by the arithmetic co-processor when using the Kronecker approach with $t \in \{2^0, 2^1, \ldots, 2^5\}$ and operating on $w \in \{32, 64, 128, 256\}$ bit words. Hence, this considers the word length of typical embedded ARM devices ($w = 32$), x86 platforms ($w = 64$) and wider lengths of vendor specific arithmetic co-processors. Again, these numbers for the arithmetic co-processor are overly simplistic; they ignore any load and store operations (which can be significant), ignore some of the shifts and assume the additions done in the large integer multiplications can be merged into calls to the multiply-and-accumulate instructions. Moreover, for simplicity the lowest level multiplication approach is assumed to be schoolbook irrespective of the size: this is most definitely sub-optimal for Kronecker+ for some small to medium w and t combinations but then only highlights that the cross-over point to using this technique can be even lowered. However, the whole purpose of this estimation is to give an indication if this approach makes sense in the presence of arithmetic co-processors.

For the estimates in Table 2 we use that the number of additions needed for an (inverse) transform inside Kronecker+ is $\lceil(\ell n/t+1)/w\rceil \cdot t \log t$ (see §4.3.2), while the total number of multiplications for the t integer multiplications is $t \cdot \lceil(\ell n/t+1)/w\rceil^2$ (see §4.4). The bit shifts within the transforms are counted as half-size additions of $\lceil(\ell n/(2t)+1)/w\rceil$ words each, of which there are $0, 1, 1, 5, 17, 49$ for $t = 1, 2, 4, 8, 16, 32$ respectively (see §4.3.2).

The baseline counts for Saber, stated in the captions of Table 2a and Table 2b, are obtained by counting the number of additions and multiplications (multiply-and-add instructions are counted as a single multiplication) in the optimized implementation of Saber.[1] We note that the multiplications and additions in Saber are all for 16-bit data types. These can be implemented directly on small (or large) platforms, but can be improved on for various platforms through parallelization techniques. For example, the Saber team also puts forward an implementation using the Advanced Vector Extensions (AVX) of Intel, leading to up to a 45% improvement for decapsulation [DKRV19, §5] and somewhat smaller improvements for key generation and encapsulation. Although an up to 45% reduction of cycles counts on Intel platforms is of course non-negligible, it is by no means several orders of magnitude. Therefore for a comparison with the optimized implementation we can still draw meaningful conclusions. Furthermore, on many smaller platforms (analogs of) AVX instructions will not be available. This is just one of the many examples where the choice of platform can significantly impact the efficiency of the various algorithms.

Finally, we turn to the numbers collected in Table 2. We first observe that the choice of t is an important one. As is typical for (symbolic) NTT-based approaches growing t will reduce the cost of the multiplications, but the resulting the overhead of the (inverse) transformations is non-negligible. For smaller w (e.g., $w = 32$) larger values of t will be more interesting as multiplications are relatively more costly initially. For $w = 32$ it seems that $t = 32$ is likely to lead to an improvement (for both encryption as well as decryption) even in the case where a w-word multiplication has the same cost as a w-word addition (while it is even often more costly). For larger w (e.g., $w = 256$) the crossover point is more likely to be earlier, as for example the sum of multiplications and additions is minimized for $t = 8$ for Saber encryption.

Furthermore we see that Kronecker+ can lead to significant improvements for Saber implementations. Even in the most basic scenario of standard Kronecker substitution ($t = 1$) with schoolbook multiplication, just having a larger multiplier available can significantly reduce the number of required additions and multiplications. This was also demonstrated by Wang, Gu and Yang [Esp, Table 6], who show a factor larger than $7\times$ improvement compared to the reference implementation on the ESP32 platform. Instead of directly using schoolbook multiplication, they do also employ a layer of Toom-Cook and Karatsuba. However, Table 2 reveals that Kronecker+ with larger values of t could give rise to even better performance improvements.

Remark 1. Note that a single polynomial in Saber consists of 256 coefficients that are represented in 16-bit datatypes, hence require 512 bytes of memory. Applying Kronecker+ with $\ell = 32$ leads to a representation of t integers of $8192/t$ bits each, which needs 1024 bytes of memory. Therefore memory requirements for Kronecker+ are slightly higher than for regular Saber, which could pose challenges for smaller platforms (in particular when multiple polynomials are stored simultaneously during matrix multiplication).

5 Conclusions

We introduced a more flexible way of computing polynomial multiplications in the ring $\mathbb{Z}[X]/(X^n + 1)$ that can be combined particularly well with Kronecker substitution and allows for efficient implementation using widely available arithmetic co-processors. This

[1] https://github.com/KULeuven-COSIC/SABER/tree/master/Reference_C

algorithm, which we refer to as Kronecker+, makes use of the available roots of unity by computing a symbolic NTT and can be seen as a variant of the Nussbaumer algorithm, as well as a generalization of Harvey's multipoint Kronecker substitution.

From a theoretical point of view this allows for faster polynomial multiplication in the targeted ring $\mathbb{Z}[X]/(X^n+1)$ on computer architectures with large multipliers. From a practical point of view we outline implementation considerations when contemporary co-processors are put to the task of accelerating post-quantum cryptography. We have demonstrated the potential of Kronecker+ in this setting by implementing the finalist scheme Saber using various instantiations of arithmetic co-processors.

References

[AHH$^+$18] Martin R. Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based schemes using an RSA co-processor. *IACR TCHES*, 2019(1):169–208, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7338.

[Ber97] D.J. Bernstein. Multidigit Multiplication For Mathematicians. 1997. https://cr.yp.to/papers/m3.pdf.

[BLLN13] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *Cryptography and Coding 2013*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2013.

[BZ07] Marco Bodrato and Alberto Zanoni. Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices. In Christopher W. Brown, editor, *ISSAC 2007*, pages 17–24. ACM press, July 2007.

[Coo66] S. A. Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966.

[CT65] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[DKL$^+$18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):238–268, 2018.

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 18*, volume 10831 of *LNCS*, pages 282–305. Springer, Heidelberg, May 2018.

[DKRV19] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[Esp] Espressif Systems. ESP32 Technical Reference Manual. https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

[GKZ07] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC*, 07 2007.

[Har09] David Harvey. Faster polynomial multiplication via multipoint Kronecker substitution. *Journal of Symbolic Computation*, 44(10):1502–1510, 2009.

[Inf] Infineon. SLE 78CAFX1M1SPHM. https://www.infineon.com/cms/en/product/security-smart-card-solutions/security-controllers/sle-78/sle-78cafx1m1sphm.

[KO62] Anatoly Karatsuba and Yuri Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in Physics-Doklady 7, 595–596, 1963.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[Kro82] L. Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *Journal für die reine und angewandte Mathematik*, 92:1–122, 1882.

[Maa] Gilad David Maayan. The IoT rundown for 2020: Stats, risks, and solutions. https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx.

[Mil86] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426. Springer, Heidelberg, August 1986.

[Nat] National Institute of Standards and Technology. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[Nus80] H. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980.

[NXP] NXP Semiconductors. NXP secure microcontroller SmartMX P71D321. https://www.nxp.com/docs/en/fact-sheet/P71D321.pdf.

[PAA$^+$19] Thomas Poppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[Pol71] John M. Pollard. The fast Fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374, 1971.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[SAB+19] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2019. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`.

[Sch82] Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In Jacques Calmet, editor, *Computer Algebra*, pages 3–15. Springer Berlin Heidelberg, 1982.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. `https://eprint.iacr.org/2018/039`.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[Too63] A.L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[vdL16] G. van der Lubbe. A New Hope for Nussbaumer. 2016. `https://www.cs.ru.nl/bachelors-theses/2016/Gerben_van_der_Lubbe___4389026___A_New_Hope_for_Nussbaumer.pdf`.

[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 - ACNS 2020*, volume 12146 of *LNCS*, pages 421–440. Springer, 2020.