Implementing RLWE-based Schemes Using an RSA Co-Processor

Martin R. Albrecht[1], Christian Hanser[2], Andrea Hoeller[2], Thomas Pöppelmann[3], Fernando Virdia[1], Andreas Wallner[2*]

[1] Information Security Group, Royal Holloway, University of London, UK
[2] Infineon Technologies Austria AG
[3] Infineon Technologies AG, Germany
martin.albrecht@royalholloway.ac.uk,
Christian.Hanser@infineon.com,
Andrea.Hoeller@infineon.com,
Thomas.Poeppelmann@infineon.com,
Fernando.Virdia.2016@rhul.ac.uk,
Andreas.Wallner@infineon.com

Abstract. We repurpose existing RSA/ECC co-processors for (ideal) lattice-based cryptography by exploiting the availability of fast long integer multiplication. Such co-processors are deployed in smart cards in passports and identity cards, secured microcontrollers and hardware security modules (HSM). In particular, we demonstrate an implementation of a variant of the Module-LWE-based Kyber Key Encapsulation Mechanism (KEM) that is tailored for high performance on a commercially available smart card chip (SLE 78). To benefit from the RSA/ECC co-processor we use Kronecker substitution in combination with schoolbook and Karatsuba polynomial multiplication. Moreover, we speed-up symmetric operations in our Kyber variant using the AES co-processor to implement a PRNG and a SHA-256 co-processor to realise hash functions. This allows us to execute CCA-secure Kyber768 key generation in 79.6 ms, encapsulation in 102.4 ms and decapsulation in 132.7 ms.

Keywords: learning with errors · smart card · implementation

1 Introduction

The development of an efficient quantum order-finding algorithm by Shor [Sho97] invalidated the quantum hardness of factoring and discrete logarithms in Abelian groups. Since then, there has been a growing effort to develop new public-key encryption and signature algorithms that can resist cryptanalysis using large-scale general quantum computers. The resulting constructions are referred to as "quantum safe" or "post-quantum". Popular families are code-based, multivariate, isogeny-based and lattice-based cryptography.

In 2016 the US National Institute of Standards and Technology (NIST) started a several year long process to standardise post-quantum cryptographic schemes [Nat16]. Furthermore, the European Telecommunications Standards Institute (ETSI) created a

*The research of Albrecht was supported by EPSRC grant "Bit Security of Learning with Errors for Post-Quantum Cryptography and Fully Homomorphic Encryption" (EP/P009417/1) and by the European Union PROMETHEUS project (Horizon 2020 Research and Innovation Program, grant 780701). The research of Virdia was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1). The research of Hanser, Hoeller, Pöppelmann and Wallner was supported by European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM).

IACR Transactions on Cryptographic Hardware and Embedded Systems Vol. 0, No.0, pp.1—40, DOI:XXXXXXXX

quantum-safe cryptography working group [CCD+15] and in 2016 Google conducted its first post-quantum cryptography at-scale test [Lan16]. Whatever we may think of the timeline or even plausibility of the arrival of general quantum computers, quantum-safe cryptography is gaining momentum.

From a practical perspective, two crucial requirements are efficiency and ease of deployment of newly proposed schemes. Indeed, submissions to the NIST process are encouraged to provide optimised software implementations aimed at general purpose microprocessors. However, implementations of quantum-safe schemes are also required in constrained (often embedded) environments such as microcontrollers or smart cards.

In the smart-card setting, low-power general purpose 16 or 32-bit CPUs are commonly augmented by cryptographic co-processors capable of executing Diffie-Hellman key exchanges, encryptions or signatures based on RSA or elliptic curves. As such, these cryptographic co-processors come equipped with an integer multiplier capable of handling multiplication (and addition) in $\mathbb{Z}_N$ for $\log_2 N \approx 2048$.

Contribution. In this work, we repurpose existing cryptographic co-processors to accelerate lattice-based cryptography. For this we make use of variants of Kronecker substitution combined with low-degree polynomial arithmetic. Using this strategy, we manage to implement a variant of the Kyber Key Encapsulation Mechanism (KEM) [SAB+17] using the Kyber768 parameter set promising 161 bits of security.[1] Our various implementations target a commercially available smart card (SLE 78 with 16 Kbyte RAM) and its RSA, AES, and SHA-256 co-processor. To evaluate Kronecker substitution we implement standard Kronecker substitution (KS1) together with Karatsuba-based polynomial multiplication and Kronecker with negated evaluation points (KS2) [Har09] using schoolbook-based polynomial multiplication. We compare our results with an implementation of Kyber and NewHope on the same target device that are not utilising large integer multiplication on the co-processor, implementations of RSA as well as related work. In summary, our work provides evidence that lattice-based post-quantum cryptography can be competitive with RSA on contactless high-security 16-bit smart cards with only limited RAM when RSA, AES and SHA-2 co-processors are used.

Approach & outline. The key computational task in {Ring, Module}-LWE encryption/decryption is to evaluate

$$\textsc{MulAdd}\big(a(x),\, b(x),\, c(x),\, f(x),\, q\big) \coloneqq a(x) \cdot b(x) + c(x) \bmod (f(x),\, q)$$

for polynomials $a(x), b(x), c(x) \in \mathbb{Z}_q[x]/(f(x))$. In this work, we realise the MulAdd gadget using a combination of a variant of Kronecker substitution [VZGG13, p. 245] and low-degree polynomial arithmetic in the spirit of Schönhage's trick [Sch77]. Kronecker substitution is a well-known and well-utilised technique in computer algebra to reduce polynomial multiplication to integer multiplication. Briefly, we start from standard Kronecker substitution by considering $a(2^\ell) \cdot b(2^\ell) + c(2^\ell) \bmod f(2^\ell)$ where e.g. $a(2^\ell)$ represents the integer obtained by evaluating $a(x)$ at 2^ℓ for some sufficiently big integer ℓ. However, for typical parameter choices, e.g. those of Kyber or NewHope [ADPS16], this strategy produces integers too large for our hardware multiplier to handle. Thus, in Section 3 we apply a variant of Harvey [Har09] to our use-case. Harvey proposed Kronecker variants which permit to half the required bitsize of the integers being multiplied at the cost of doubling the number of multiplications. This provides a worthwhile trade-off for medium-sized integers where quasi-linear integer multiplication algorithms are not yet competitive. However, in our context Harvey's technique on its own still does not suffice to

[1] We stress that our variant of Kyber is not interoperable with Kyber as specified in [SAB+17]. The main differences are choices for symmetric functions and that Kyber explicitly requires the usage of the Number Theoretic Transform (NTT), which we cannot realise efficiently with our approach.

reduce the integer operands to match our hardware multiplier. Thus, we utilise (low-degree) polynomial arithmetic on top. Overall, we obtain an implementation which computes the IND-CCA Kyber768 decapsulation in $8 \cdot (3^2 + 3 + 3) = 120$ modular multiplications of 2049-bit numbers. In contrast, decrypting 2048-bit RSA requires roughly $2 \cdot 1.5 \cdot 1024 = 3072$ multiplications of 1024-bit numbers in Chinese Remainder Theorem (CRT) representation.[2] We describe our implementation in detail in Section 5, discuss performance in Section 6 and finish with a discussion in Section 7.

Large modulus LWE. In lattice-based cryptography, noisy variants of Kronecker substitution have been used to show various polynomial-time equivalences. In [BLP+13] a reduction from n-dimensional LWE with modulus q to 1-dimensional LWE with modulus q^n is provided using

$$A \coloneqq \sum_{i=0}^{n-1} a_i q^i, \quad S \coloneqq \sum_{i=0}^{n-1} s_i q^{n-i-1}, \quad A \cdot S \bmod q^n \approx \langle \vec{a}, \vec{s} \rangle \cdot q^{n-1}. \tag{1}$$

This reduction is extended to the Approximate-GCD problem in [CS15]. In [CLT13], a variant of the Approximate-GCD problem is defined for realising fully homomorphic encryption which permits to pack several plaintext bits into one big integer using the CRT. The reduction from [BLP+13] is extended in [AD17] to a reduction from Module-LWE to large modulus Ring-LWE and a dimension-halving, modulus squaring self-reduction of Ring-LWE. In [Chu17], it is noted that given $A \coloneqq \sum_{i=0}^{n-1} a_i q^i$, $S \coloneqq \sum_{i=0}^{n-1} s_i q^i$ and $c(x) = a(x) \cdot s(x) \bmod x^n + 1$, we have

$$A \cdot S \bmod (q^n + 1) \approx_s \sum_{i}^{n-1} c_i q^i$$

where $\approx_s$ means $\approx$ in each "slot" defined by multiples of q. This observation then gives rise to the I-RLWE problem, which also permits packing several plaintext bits into one large integer. In [Chu17], a reduction from Ring-LWE to I-RLWE is given, but this reduction does not consider the noise distribution, only its size.[3] In [Ham17], a variant of I-RLWE over a pseudo-Mersenne field is given to instantiate an MLWE KEM. Similarly, [AJPS17] can be considered as an integer variant of NTRU.

Post-quantum cryptography on microcontrollers. Microcontrollers and embedded processors usually have only very limited amount of available RAM, space to store program code and operate with relatively simple 8-, 16-, or 32-bit processor architectures. They are sometimes also referred to as *constrained devices* and are mostly used in embedded applications where low energy consumption, reduced device costs, and other aspects like real-time capabilities are required. Such requirements are commonly not fulfilled by computer systems or powerful System-on-Chips (SoC) with external non-volatile memory or RAM. A special class of constrained devices are smart cards or chip cards which are used in electronic banking, secured identification (e.g. passports or national ID cards), authentication, or transport and ticketing applications. Smart cards are usually equipped with protection mechanisms against a wide range of invasive or non-invasive attacks and they often feature dedicated accelerators to speed-up and to protect cryptographic operations

[2] Of course, this metric does not account for the cost of embedding of polynomials into integers as well as additional operations required in lattice-based cryptography, like randomness sampling or expensive CCA transformations. Moreover, the data structures in RSA are much smaller than in lattice-based cryptography so that transfers between CPU and co-processors with internal memory appropriate to hold RSA-2048 base, exponent, modulus and result have much less impact on performance.

[3] We note, though, that according to all known cryptanalysis results for public-key *encryption* based on LWE, the noise distribution does not play a significant role if it provides enough entropy.

(e.g. AES, ECC or RSA). Most commercial chip cards are certified according to Common Criteria[4] and evaluated in a laboratory.

The implementation of post-quantum cryptography on constrained devices is an active research area. Most works in the literature focus on performance but from a practical standpoint RAM consumption, code footprint and maintainability of the code-base are also important metrics. Examples of PQC implementations are works that deal with multivariate signatures [CHT12], code-based encryption [vMOG15] and hash-based signatures [HRS16]. In the area of lattice-based cryptography, examples are an implementation of NTRU [BSJ15], an implementation of BLISS signatures on 32-bit ARM [OPG14], an implementation of CPA-secure public-key encryption based on Ring-LWE on an 8-bit AVR [LPO+17a] and 32-bit ARM [dCRVV15]. An implementation of the NewHope key exchange protocol which is similar to Ring-LWE encryption is given in [AJS16]. In [KMRV18] an implementation of Saber is provided which is an MLWE-based KEM that does not rely on the NTT for polynomial multiplication.

Similarly, the protection of lattice-based cryptography against side-channel attacks has already been explored. An implementation of a masked decryption of Ring-LWE CPA-secure PKE is described in [RdCR+16] and an implementation of a CCA-secure and masked variant is given in [OSPG18]. A masked implementation of the GLP signature scheme is provided in [BBE+18]. What has received comparably less attention in the literature so far are flexible cryptographic co-processors for lattice-based cryptography in the spirit of RSA or ECC co-processors (cf. [SBPV07]) and instruction set extensions (cf. a multiply-accumulate instruction [Wen13]).

2 Preliminaries

For $x \in \mathbb{R}$, we write $\lceil x \rfloor$ to mean the closest integer to x (where $\lceil y + \frac{1}{2} \rfloor := y+1$ for $y \in \mathbb{Z}$). For $a, b \in \mathbb{Z}$, we write $a \bmod^{(+)} b$ for the unique integer $\hat{a} \equiv a \bmod b$ such that $0 \leq \hat{a} < b$. We write $a \bmod^{(-)} b$ for the unique integer $\hat{a} \equiv a \bmod b$ such that $-b/2 \leq \hat{a} < b/2$. We extend this definition to tuples, vectors, matrices and polynomials a over $\mathbb{Z}$ component-wise. We also write $[a]_b$ for $a \bmod^{(+)} b$. We often write $\{a, \ldots, b\}$ to mean the set $[a, b] \cap \mathbb{Z}$.

We write

$$[\![\text{condition}]\!] := \begin{cases} 1 & \text{if condition is true,} \\ 0 & \text{if condition is false.} \end{cases}.$$

Unless stated otherwise, we let $R = \mathbb{Z}[x]/(x^n + 1)$ where n is a power of 2, and let $R_q = R/(q)$ for some positive integer q. We let R^k (resp. R_q^k) be a ring module of dimension k over R (resp. R_q). Throughout we identify equivalence classes in R_q with their representative polynomial with coefficients $\bmod^{(-)} q$, and its lifted versions in R and in $\mathbb{Z}[x]$.

Given a set S and a probability distribution D over S we write $s \xleftarrow{r} D$ to mean $s \in S$ sampled according to D using coins r, and $s \xleftarrow{r} S$ to mean $s \in S$ sampled uniformly random from S with coins r. We may omit coins, in which case we write $\xleftarrow{\$}$. We denote by $\vec{v}$ tuples in S^k, and use capital letters for matrices $\vec{M} \in S^{k \times k}$. We write $t \leftarrow v$ to assign value v to variable t inside algorithms. Inside algorithms, we refer to the i^{th} entry in an array a as $a^{(i)}$. If a variable v gets overwritten as part of a loop (e.g. $v \leftarrow v/2$), we may refer to the variable v after step i of the loop as $v^{[i]}$ (e.g. $v^{[i]} \leftarrow v^{[i-1]}/2$).

2.1 Hard problems

All schemes considered in this work relate to the Learning with Errors problem [Reg09].

[4] See `http://www.commoncriteriaportal.org/products/#IC`.

Definition 1 (LWE [Reg09])**.** Let n, q be positive integers, χ be a probability distribution on $\mathbb{Z}$ and $\vec{s}$ be a secret vector in $\mathbb{Z}_q^n$. We denote by $L_{\mathbf{s},\chi}$ the probability distribution on $\mathbb{Z}_q^n \times \mathbb{Z}_q$ obtained by choosing $\vec{a} \in \mathbb{Z}_q^n$ uniformly at random, choosing $e \in \mathbb{Z}$ according to χ and considering it in $\mathbb{Z}_q$, and returning $(\vec{a}, c) = (\vec{a}, \langle \vec{a}, \vec{s} \rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$.
Decision-LWE is the problem of deciding whether pairs $(\vec{a}, c) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ are sampled according to $L_{\mathbf{s},\chi}$ or the uniform distribution on $\mathbb{Z}_q^n \times \mathbb{Z}_q$.
Search-LWE is the problem of recovering $\vec{s}$ from $(\vec{a}, c) = (\vec{a}, \langle \vec{a}, \vec{s} \rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ sampled according to $L_{\mathbf{s},\chi}$.

Decision- and Search-LWE are polynomial-time equivalent [Reg09]. Since LWE leads to public-key sizes at least quadratic in the security parameter, many schemes are based on its ring variant, aptly called Ring-LWE [LPR10] or "RLWE" in short. Below, we give the definition of Polynomial-LWE [SSTX09] (or "PLWE") which is equivalent to the RLWE definition for power-of-two cyclotomic rings. For e.g. prime cyclotomic ring these two definitions are not equivalent, i.e. the geometry of the error distribution changes somewhat. However, as is common in the literature, we will abuse notation and refer to PLWE as RLWE.

Definition 2 (RLWE [SSTX09, LPR10])**.** Let n, q be positive integers, χ be a probability distribution on $\mathbb{Z}$ and s be a secret polynomial in R_q. We denote by $L_{s,\chi}$ the probability distribution on $R_q \times R_q$ obtained by choosing $a \in R_q$ uniformly at random, choosing $e \in R$ by sampling each of its coefficients according to χ and considering it in R_q, and returning $(a, c) = (a, a \cdot s + e) \in R_q \times R_q$.
Decision-RLWE is the problem of deciding whether pairs $(a, c) \in R_q \times R_q$ are sampled according to $L_{s,\chi}$ or the uniform distribution on $R_q \times R_q$.
Search-RLWE is the problem of recovering s from $(a, c) = (a, a \cdot s + e) \in R_q \times R_q$ sampled according to $L_{s,\chi}$.

The decision and search variants of RLWE are polynomial-time equivalent for cyclotomic rings [LPR10]. The increased efficiency of RLWE compared to LWE is achieved by adding algebraic structure. Informally, each RLWE sample can be viewed as n correlated LWE samples. While, so far, no algorithm is known which exploits this additional structure, some designs hedge against such attacks by considering instances which require the attacker to find short vectors in a lattice of larger module rank [CDW17, SAB+17]. In particular, Module-LWE (or "MLWE") interpolates between the plain and the ring variants of LWE.

Definition 3 (MLWE [LS15])**.** Let n, q, k be positive integers, χ be a probability distribution on $\mathbb{Z}$ and $\vec{s}$ be a secret module element in R_q^k. We denote by $L_{\mathbf{s},\chi}$ the probability distribution on $R_q^k \times R_q$ obtained by choosing $\vec{a} \in R_q^k$ uniformly at random, choosing $e \in R$ by sampling each of its coefficients according to χ and considering it in R_q, and returning $(\vec{a}, c) = (\vec{a}, \langle \vec{a}, \vec{s} \rangle + e) \in R_q^k \times R_q$.
Decision-MLWE is the problem of deciding whether pairs $(\vec{a}, c) \in R_q^k \times R_q$ are sampled according to $L_{\mathbf{s},\chi}$ or the uniform distribution on $R_q^k \times R_q$.
Search-MLWE is the problem of recovering $\vec{s}$ from $(\vec{a}, c) = (\vec{a}, \langle \vec{a}, \vec{s} \rangle + e) \in R_q^k \times R_q$ sampled according to $L_{\mathbf{s},\chi}$.

Again, the search and the decision variants of this problem are polynomial-time equivalent [LS15, Thm. 4.7].

2.2 Kyber

A recent construction relying on the MLWE problem is the Kyber Key Encapsulation Mechanism. Kyber has been submitted to the NIST PQC standardisation process [SAB+17] and a variant is also published as an academic paper [BDK+17]. It is defined by an intermediate

IND-CPA secure Public-Key Encryption (PKE) scheme which is then transformed to an IND-CCA secure KEM using a generic transform [HHK17].[5] We note that Kyber unambiguously refers to the IND-CCA secure KEM, i.e. [SAB+17] does not formally propose a public-key encryption scheme nor a KEM which only claims IND-CPA security.

Definition 4 (Simplified Kyber.CPA following [BDK+17]; cf. [SAB+17])**.** For $n = 256$ let $k, n, q, \eta, d_t, d_u, d_v$ be positive integers. Let $\mathcal{M} = \{0,1\}^n$ be the plaintext space, where each message $m \in \mathcal{M}$ can be seen as a polynomial in R with coefficients in $\{0,1\}$. Define the functions

$$\textsc{Compress}_q(x,d) \coloneqq \lceil (2^d/q) \cdot x \rfloor \bmod^{(+)} 2^d\,,$$
$$\textsc{Decompress}_q(x,d) \coloneqq \lceil (q/2^d) \cdot x \rfloor,$$

let χ be a centered binomial distribution with support $\{-\eta, \ldots, \eta\}$, and let χ_n be the distribution of polynomials of degree n with entries independently sampled from χ. Define the public-key encryption scheme Kyber.CPA = (Kyber.CPA.Gen, Kyber.CPA.Enc, Kyber.CPA.Dec) as in Algorithms 1, 2 and 3.

$(\rho, \sigma) \xleftarrow{\$} \{0,1\}^{256} \times \{0,1\}^{256}$;
$\vec{A} \xleftarrow{\rho} R_q^{k \times k}$;
$(\vec{s}, \vec{e}) \xleftarrow{\sigma} \chi_n^k \times \chi_n^k$;
$\vec{t} \leftarrow \textsc{Compress}_q(\vec{A}\vec{s} + \vec{e}, d_t)$;
return $pk_{CPA} \coloneqq (\vec{t}, \rho)$, $sk_{CPA} \coloneqq \vec{s}$;

Algorithm 1: Kyber.CPA.Gen.

Input: $pk_{CPA} = (\vec{t}, \rho)$
Input: $m \in \mathcal{M}$
Input: $r \xleftarrow{\$} \{0,1\}^{256}$
$\vec{t} \leftarrow \textsc{Decompress}_q(\vec{t}, d_t)$;
$\vec{A} \xleftarrow{\rho} R_q^{k \times k}$;
$(\vec{r}, \vec{e}_1, e_2) \xleftarrow{r} \chi_n^k \times \chi_n^k \times \chi_n$;
$\vec{u} \leftarrow \textsc{Compress}_q(\vec{A}^T\vec{r} + \vec{e}_1, d_u)$;
$v \leftarrow \textsc{Compress}_q(\langle \vec{t}, \vec{r} \rangle + e_2 + \lceil \frac{q}{2} \rfloor \cdot m, d_v)$;
return $c \coloneqq (\vec{u}, v)$;

Algorithm 2: Kyber.CPA.Enc.

Input: $sk_{CPA} = \vec{s}$
Input: $c = (\vec{u}, v)$
$\vec{u} \leftarrow \textsc{Decompress}_q(\vec{u}, d_u)$;
$v \leftarrow \textsc{Decompress}_q(v, d_v)$;
return $\textsc{Compress}_q(v - \langle \vec{s}, \vec{u} \rangle, 1)$;

Algorithm 3: Kyber.CPA.Dec.

In Kyber, the parameters that define the base ring R_q are fixed at $n = 256$ and $q = 7681$. The parameters that define key and ciphertext compression are also fixed and set to $d_u = 11, d_v = 3$ and $d_t = 11$. The three different security levels are obtained by different choices of k and η. All relevant Kyber parameters are summarised in Table 1.

[5] We note that [SAB+17] does not include the Targhi-Unruh tag.

Table 1: Parameters proposed to NIST for instantiating Kyber KEM.

Parameter Set	n	q	k	η	q. bit-sec.	NIST level	failure	$\|pk\|$	$\|sk\|$	$\|ctxt\|$
Kyber512	256	7681	2	5	102	1	2^{-145}	736	1632	800
Kyber768	256	7681	3	4	161	3	2^{-142}	1088	2400	1152
Kyber1024	256	7681	4	3	218	5	2^{-169}	1440	3168	1504

Sizes of the public key ($|pk|$), secret key ($|sk|$), and ciphertext ($|ctxt|$) are given in bytes.

The performance of an implementation of Kyber depends highly on the speed of the polynomial multiplication algorithm and the performance of the PRNG instantiations as a large number of pseudo random data is required when generating $\vec{A} \xleftarrow{\rho} R_q^{k\times k}$ or when sampling noise from χ_n^k. Regarding operations in R_q, Kyber.CPA.Gen requires the computation of k^2 multiplications and $(k-1)k+k$ additions (line 4 of Algorithm 1). For encryption as defined in Kyber.CPA.Enc, k^2 multiplications and $(k-1)k+k$ additions (line 4 of Algorithm 2) as well as k multiplications and $(k-1)+2$ additions (line 5) are needed. The decryption routine Kyber.CPA.Dec can be implemented with k multiplications and $k-1+1$ additions (line 3 of Algorithm 3). Note that Kyber specifies a Number Theoretic Transform (NTT). The NTT allows to implement a fast polynomial multiplication by computing $c = \mathrm{NTT}^{-1}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b))$ for $a, b, c \in R_q$, where $\circ$ denotes coefficient-wise multiplication. Kyber exploits that the NTT is a one-to-one map and assumes that randomly sampled polynomials in $\vec{A}$ are already in the transformed domain. Thus, an implementation using a different multiplication algorithm than the NTT would have to apply an inverse transformation first and then use the polynomial multiplication algorithm of its choice to stay compatible with the original specification.

Given two hash functions $G\colon \{0,1\}^* \to \{0,1\}^{2\times 256}$ and $H\colon \{0,1\}^* \to \{0,1\}^{256}$, Kyber is obtained from Kyber.CPA using a Fujisaki-Okamoto style transform from [HHK17] as shown in Algorithms 4, 5, 6. Within Kyber.Decaps a re-encryption has to be computed whose result is compared to the received ciphertext. Thus Kyber specifies exactly how to generate the uniformly random matrix $\vec{A}$ as well as polynomials from the error distribution χ_n from a seed. For this the authors of Kyber have chosen different instantiations from the SHA3 family (SHAKE-128, SHAKE-256, SHA3-256 and SHA3-512).

1 $((\vec{t}, \rho), \vec{s}) \leftarrow$ Kyber.CPA.Gen() ;
2 $z \xleftarrow{\$} \{0,1\}^{256}$;
3 $h \leftarrow H((\vec{t}, \rho))$;
4 **return** $pk \coloneqq (\vec{t}, \rho)$, $sk \coloneqq (\vec{s}, \vec{t}, \rho, h, z)$;

Algorithm 4: Kyber.Gen.

Input: $pk = (\vec{t}, \rho)$
1 $m \xleftarrow{\$} \{0,1\}^{256}$;
2 $m \leftarrow H(m)$;
3 $(\hat{K}, r) \leftarrow G(m, H(pk))$;
4 $(\vec{u}, v) \leftarrow$ Kyber.CPA.Enc$(pk, m; r)$;
5 $c \leftarrow (\vec{u}, v)$;
6 $K \leftarrow H(\hat{K}, H(c))$;
7 **return** (c, K) ;

Algorithm 5: Kyber.Encaps.

Input: $sk = (\vec{s}, \vec{t}, \rho, h, z)$
Input: $c = (\vec{u}, v)$
$m' \leftarrow$ KYBER.CPA.DEC$(\vec{s}, (\vec{u}, v))$;
$(\hat{K}', r') \leftarrow G(m', h)$;
$(\vec{u}', v') \leftarrow$ KYBER.CPA.ENC$(pk, m'; r')$;
if $(\vec{u}', v') = (\vec{u}, v)$ **then**
  $K \leftarrow H(\hat{K}', H(c))$;
else
  $K \leftarrow H(z, H(c))$;
end
return K ;

Algorithm 6: KYBER.DECAPS.

2.3 Target platform

We use an Infineon SLE78CLUFX5000 chip card[6] with 16 Kbyte RAM and 500 Kbyte NVM which features a 16-bit CPU running at 50 MHz. The target chip is equipped with common peripherals (watchdog, timers), internal security functions and encryption procedures, a True Random Number Generator (TRNG), as well as a symmetric co-processor to accelerate AES, a co-processor to compute SHA-256 and an asymmetric co-processor for RSA and ECC acceleration. The chip allows contact-based as well as contactless operation where it is powered by a field generated by a common smart card reader. It is intended for use in applications like passports, identity cards, access control or payment cards (e.g. banking, value or credit cards). A similar target device from the SLE 78 family has previously been used to implement hash-based XMSS signatures [HBB13] or eta pairings [GK15].

The asymmetric co-processor on the SLE78CLUFX5000 allows fast basic long number calculations on integers slightly larger than 2048 bits (addition, subtraction, integer multiplication, modular multiplication). In practice it is mainly used by cryptographic libraries for RSA and ECC. However, for an earlier generation smart card (Infineon SLE66P) Garcia and Seifert describe an implementation of AES on the modular arithmetic co-processor [GS02].

As there is no standard for RSA/ECC co-processors our low-level implementation is certainly vendor specific. However, the general approach described in Section 3 and Section 4 should be transferable to a large number of devices as most other smart card vendors appear to use similar approaches. Additional devices that could profit from our work could be server systems like the IBM PCIe Cryptographic Coprocessor[7] or existing FPGA-based RSA/ECC accelerator cards or RSA/ECC accelerator IP.

3 Kronecker

Kronecker substitution is a classical technique in computer algebra for reducing polynomial arithmetic to large integer arithmetic, cf. [VZGG13, p. 245] and [Har09]. The fundamental idea behind this technique is that univariate polynomial and integer arithmetic are identical except for carry propagation in the latter. Thus, coefficients are simply packed into an integer in such a way as to terminate any possible carry chain. For example, say, we want to multiply two polynomials $f(x) \coloneqq x + 2$ with $g(x) \coloneqq 3x + 4$ in $\mathbb{Z}[x]$. We may write

[6] We refer to `https://www.infineon.com/cms/de/product/security-smart-card-solutions/security-controllers/sle-78/` for more information on the SLE 78 family.

[7] See `http://www-304.ibm.com/jct01003c/common/ssi/ShowDoc.wss?docURL=/common/ssi/rep_sm/1/649/ENUS4767-_h01/index.html`.

$f(100) = 100 + 2 = 102$ and $g(100) = 300 + 4 = 304$. Multiplying gives $102 \cdot 304 = 31008$ or $3x^2 + 10x + 8$. In implementations, we use powers of two as evaluation points since this permits efficient "packing" (polynomial to integer) and "unpacking" (integer to polynomial) using only cheap bit shifts.

In this work, we employ Kronecker substitution for computing

$$\textsc{MulAdd}\big(a(x),\, b(x),\, c(x),\, f(x)\big) \coloneqq a(x) \cdot b(x) + c(x) \bmod f(x)$$

with all polynomials having signed coefficients from different ranges.

In more detail, we first pack the polynomials into integers A, B, C, F using Algorithm 7 (Snort). We then compute $D \coloneqq A \cdot B + C \bmod F$. Finally, we unpack D to $d(x)$ using Algorithm 8 (Sneeze). We note that our packing/unpacking algorithms are straight-forward adaptations of standard Kronecker packing/unpacking to the signed case, cf. [Har09]. We give high-level, proof-of-concept Sage [S+17] implementations for the algorithms in this section in Appendix B.

Lemma 1 establishes the correctness of this procedure. While correctness of Kronecker substitution is well-established and signed coefficients are usually easily covered by bit shifts [Har09], we give a complete proof of correctness and in particular the required precision in order to maintain the same error as in Kyber, since faithful re-encryption is required for standard IND-CCA transforms such as the one in [HHK17] utilised by Kyber. On the other hand, loosening this requirement, permits to decrease precision (the parameter ℓ below) and hence to improve performance.

Input: $g \in \mathbb{Z}[x]$
Input: $f \in \mathbb{Z}[x]$
Input: bitlength ℓ
return $g(2^\ell) \bmod^{(+)} f(2^\ell)$;

Algorithm 7: Snort(g, f, ℓ).

Input: $G \in \{0, \ldots, f(2^\ell) - 1\}$
Input: $f \in \mathbb{Z}[x]$, monic
Input: bitlength ℓ
$n \leftarrow \deg(f)$;
$G^{[-1]} \leftarrow G$;
for $i = 0, 1, \ldots, n-1$ **do** `// step` i
  $e^{(i)} \leftarrow G^{[i-1]} \bmod^{(+)} 2^\ell$;
  $G^{[i]} \leftarrow \left(G^{[i-1]} - e^{(i)}\right)/2^\ell$;
  if $e^{(i)} > 2^{\ell-1}$ **then** `// negative coefficient`
    $e^{(i)} \leftarrow e^{(i)} - 2^\ell$;
    $G^{[i]} \leftarrow G^{[i]} + 1$;
  end
  $r^{(i)} \leftarrow e^{(i)}$;
end
for $i = 0, 1, \ldots, n-1$ **do** $r^{(i)} \leftarrow r^{(i)} - f_i\, G^{[n-1]}$; `// subtract` $\mathtt{b} \cdot f(x)$
return $\{r^{(i)}\}_{i=0}^{n-1}$;

Algorithm 8: Sneeze(G, f, ℓ).

Lemma 1. *Let* $a, b, c \in \mathbb{Z}[x]$ *such that* $a = \sum_{i=0}^{n-1} a_i x^i$, $b = \sum_{i=0}^{n-1} b_i x^i$, $c = \sum_{i=0}^{n-1} c_i x^i$

with $a_i \in \{-\alpha, \ldots, \alpha\}$, $b_i \in \{-\beta, \ldots, \beta\}$, *and* $c_i \in \{-\gamma, \ldots, \gamma\}$. *Let*

$$d \coloneqq \sum_{i=0}^{n-1} d_i\, x^i \equiv a \cdot b + c \bmod f$$

with $d_i \in \{-\delta, \ldots, \delta\}$, *where* $\delta > 0$ *depends on* $\alpha, \beta, \gamma, n, f$ *and* f *is monic of degree* n *such that* $f(2^\ell) > 2^{n\ell} - 1$. *Let* $\varphi \coloneqq \max_{i<n} |f_i|$, *and let* $\ell > \log_2(\delta + \varphi) + 1$ *be an integer (e.g.* $\ell = \lceil \log_2(\delta + \varphi + 1) \rceil + 1$*).*

If

$$A \coloneqq \textsc{Snort}(a,\, f,\, \ell),$$
$$B \coloneqq \textsc{Snort}(b,\, f,\, \ell),$$
$$C \coloneqq \textsc{Snort}(c,\, f,\, \ell),$$

and

$$D \coloneqq A \cdot B + C \bmod^{(+)} f(2^\ell),$$

then $\textsc{Sneeze}\,(D,\, f,\, \ell)$ *returns* $\{r^{(i)}\}_{i=0}^{n-1}$ *where* $r^{(i)} = d_i$ *for* $i \in \{0, \ldots, n-1\}$.

Corollary 1 (Power of two cyclotomic)**.** *Let* α, β, γ *be as above, let* n *be a power of* 2*, and let* $f(x) = x^n + 1$. *Let* $\delta \coloneqq n\alpha\beta + \gamma$. *Then Lemma 1 applies.*

Proof. See Appendix A. □

Corollary 2 (Prime cyclotomic)**.** *Let* α, β, γ *be as above, let* $n = p - 1$ *where* p *is prime, and let* $f = \sum_{i=0}^{n} x^i$. *Let* $\delta \coloneqq (2n-1)\alpha\beta + \gamma$. *Then Lemma 1 applies.*

Proof. See Appendix A. □

Proof of Lemma 1. We need to uniquely encode any possible d as an integer modulo $f(2^\ell)$. Since the coefficients d_i are ℓ bits long, and we need to store n of them, this means that we require $f(2^\ell) > 2^{n\ell} - 1$.

When Sneeze is called, we set

$$G^{[-1]} \coloneqq D = A \cdot B + C \bmod^{(+)} f(2^\ell).$$

Since $d \equiv a \cdot b + c \bmod f$, it follows by explicit computation that

$$G^{[-1]} = D = \sum_{i=0}^{n-1} d_i\, 2^{\ell i} + \mathtt{b}\, f(2^\ell)$$

where the last equality is over the integers, for some $\mathtt{b} \in \mathbb{Z}$. Given that

$$\left| \sum_{i=0}^{n-1} d_i\, 2^{\ell i} \right| \leq \delta\, \frac{2^{n\ell} - 1}{2^\ell - 1} \leq (2^{\ell-1} - 1)\, \frac{2^{n\ell} - 1}{2^\ell - 1} < 2^{n\ell - 1},$$

the assumption that $f(2^\ell) > 2^{n\ell} - 1 > 2^{n\ell-1}$ implies that $\mathtt{b} \in \{0, 1\}$.

The main computation in Sneeze is done between lines 3 and 11, hence we define some conditions on the output of that loop and prove they hold by induction.

Claim. *After step* $i \in \{0, \ldots, n-1\}$, *we have*

$$r^{(i)} = d_i + \mathtt{b}\, f_i \tag{2}$$

and

$$G^{[i]} = \sum_{j=i+1}^{n-1} d_j\, 2^{\ell(j-i-1)} + \mathtt{b} \sum_{j=i+1}^{n} f_j\, 2^{(j-i-1)\ell} \tag{3}$$

Assume Conditions 2, 3 hold for step $i-1 \geq 0$. We start by assigning

$$\begin{aligned} e^{(i)} &= G^{[i-1]} \bmod^{(+)} 2^\ell \\ &= \sum_{r=i}^{n-1} d_r \, 2^{\ell(r-i)} + \mathtt{b} \sum_{j=i}^{n} f_j \, 2^{(j-i)\ell} \bmod^{(+)} 2^\ell \\ &= d_i + \mathtt{b}\, f_i + \mathtt{t}_i \, 2^\ell \end{aligned}$$

for some $\mathtt{t}_i \in \mathbb{Z}$ such that $e^{(i)} \in \{0, \ldots, 2^\ell - 1\}$. Similar to before, by definition of ℓ and the fact that $\mathtt{b} \in \{0, 1\}$, we have

$$|d_i + \mathtt{b}\, f_i| \leq \delta + \varphi < 2^{\ell-1} \text{ for all } i \in \{0, \ldots, n-1\} \tag{4}$$

Hence $\mathtt{t}_i \in \{0, 1\}$ for $i < n$. We then set

$$\begin{aligned} G^{[i]} &= \frac{G^{[i-1]} - e^{(i)}}{2^\ell} \\ &= \sum_{r=i+1}^{n-1} d_r \, 2^{\ell(r-i-1)} + \mathtt{b} \sum_{j=i+1}^{n} f_j \, 2^{(j-i-1)\ell} - \mathtt{t}_i \end{aligned}$$

and balance $e^{(i)} \pmod{2^\ell}$. By the size consideration made in Inequality 4, this amounts to subtracting $\mathtt{t}_i 2^\ell$ from $e^{(i)}$. We keep account of this subtraction by adding back $\mathtt{t}_i$ to $x^{(i)}$. Finally, we assign $r^{(i)} \leftarrow e^{(i)}$. Hence Conditions 2, 3 hold for step $i \geq 1$. Similarly, we can see that Conditions 2, 3 also hold for step $i = 0$, proving the claim.

By Condition 3, after step $i = n-1$ we have $G^{[n-1]} = \mathtt{b} < 2^\ell$, which would become the coefficient of an n^{th} power of x in d. Line 12 takes care of reducing this modulo f, which results in assigning

$$r^{(i)} \leftarrow r^{(i)} - f_i \, G^{[n-1]} = d_i + \mathtt{b}\, f_i - f_i \, \mathtt{b} = d_i \quad \text{for all } i < n,$$

completing the proof. □

Since operating on $G^{[i]}$ involves integer arithmetic on $n\ell$ bit integers, we may modify Algorithm 8 to correct carries on $e^{(i)}$ in order to avoid executing line 8 of Algorithm 8. This variant of the algorithm is given as Algorithm 9. Note that with this change the only large integer operations are division with remainder modulo 2^ℓ and thus cheap, while the final output of the algorithm is the same.

The proof of Lemma 1 can be directly adapted to the MLWE setting where we let

$$\left\{ a_i = \sum_{j=0}^{n-1} a_{i,j} x^j \right\}_{i=1}^{\kappa}, \quad \left\{ b_i = \sum_{j=0}^{n-1} b_{i,j} x^j \right\}_{i=1}^{\kappa}, \quad c = \sum_{j=0}^{n-1} c_j x^j$$

with $a_{i,j} \in \{-\alpha, \ldots, \alpha\}$, $b_{i,j} \in \{-\beta, \ldots, \beta\}$, and $c_j \in \{-\gamma, \ldots, \gamma\}$ and want to compute $\sum_{i=1}^{\kappa} a_i \cdot b_i + c \pmod f$, by letting

$$\ell > \log_2(\kappa(\delta - \gamma) + \gamma + \varphi) + 1.$$

Overall, we arrive at the following corollary.

Corollary 3 (KyberMulAdd)**.** *Let $\{a_i, b_i\}_{i=1}^{\kappa}, c \subset \mathbb{Z}[x]$, be as above, with $\alpha = \left\lfloor \frac{q}{2} \right\rfloor$, and $\beta = \gamma = \eta$, and let $f = x^n + 1$. Let*

$$\ell > \log_2 \left(\kappa n \left\lfloor \frac{q}{2} \right\rfloor \eta + \eta + 1 \right) + 1$$

be an integer. Let $A_i \coloneqq \textsc{Snort}(a_i, f, \ell)$, $B_i \coloneqq \textsc{Snort}(b_i, f, \ell)$, $C \coloneqq \textsc{Snort}(c, f, \ell)$, and $D \coloneqq \vec{A} \cdot \vec{B} + C \bmod^{(+)} f(2^\ell)$. Then $\textsc{Sneeze}(D, f, \ell)$ returns $d \coloneqq \sum_{i=1}^{\kappa} a_i \cdot b_i + c \pmod f$.

Remark 1. From $d \in R$, the result in R_q can be obtained by coefficient-wise modular reduction.

Input: $G \in \{0, \ldots, f(2^\ell) - 1\}$
Input: $f \in \mathbb{Z}[x]$, monic
Input: bitlength ℓ
$n \leftarrow \deg(f)$;
$G^{[-1]} \leftarrow G$, $c \leftarrow 0$;
for $i = 0, 1, \ldots, n-1$ **do** `// step` i
  $e^{(i)} \leftarrow G^{[i-1]} \bmod^{(+)} 2^\ell$;
  $G^{[i]} \leftarrow \left(G^{[i-1]} - e^{(i)}\right) / 2^\ell$;
  $e^{(i)} \leftarrow e^{(i)} + c$;
  if $e^{(i)} > 2^{\ell-1}$ **then** $e^{(i)} \leftarrow e^{(i)} - 2^\ell$, $c \leftarrow 1$ **else** $c \leftarrow 0$;
  $r^{(i)} \leftarrow e^{(i)}$;
end
for $i = 0, 1, \ldots, n-1$ **do** $r^{(i)} \leftarrow r^{(i)} - f_i \left(G^{[n-1]} + c\right)$; `// subtract` $\mathtt{b} \cdot f(x)$
return $\{r^{(i)}\}_{i=0}^{n-1}$;

Algorithm 9: SNEEZE-FAST(G, f, ℓ). Same as SNEEZE, but avoiding large integer arithmetic for carry propagation.

3.1 Compact Kronecker

In [Har09], David Harvey presents two improved packing techniques for Kronecker substitution, reducing integer sizes at the cost of performing more multiplications: KS2 or "negated evaluation points" evaluates at $\left(2^\ell, -2^\ell\right)$ and KS3 or "reciprocal evaluation points" evaluates at $\left(2^\ell, 2^{-\ell}\right)$. Each technique halves the required integer bit size at the cost of performing two multiplications. Note that integer arithmetic is super-linear (e.g. Karatsuba multiplication is used for medium-sized inputs and has a cost of $3^{\log_2 L}$ for integers of size L, see below) and thus this trade-off produces a noticeable speed-up. The two techniques are orthogonal and can be combined, which reduces bit sizes by a factor of four at the cost of increasing the number of multiplications to four. The combined algorithm is referred to as KS4.

The KS2 algorithm proceeds as follows. Assume $a(x), b(x)$ are such that their product $c(x) \coloneqq a(x) \cdot b(x)$ has positive coefficients bounded by $2^{2\ell}$. Let

$$\begin{aligned} c^{(+)} \coloneqq \quad & c(2^\ell) = a(2^\ell) \cdot b(2^\ell) && = \sum_{[i]_2=0} c_i \, 2^{i\ell} + \sum_{[i]_2=1} c_i \, 2^{i\ell} \\ c^{(-)} \coloneqq \quad & c(-2^\ell) = a(-2^\ell) \cdot b(-2^\ell) && = \sum_{[i]_2=0} c_i \, 2^{i\ell} - \sum_{[i]_2=1} c_i \, 2^{i\ell} \end{aligned}$$

Then, we can recover the even coefficients of $c(x)$ from

$$c^{(+)} + c^{(-)} = c(2^\ell) + c(-2^\ell) = 2 \sum_{[i]_2=0} c_i \, 2^{i\ell}$$

and the odd coefficients from

$$c^{(+)} - c^{(-)} = c(2^\ell) - c(-2^\ell) = 2 \cdot 2^\ell \sum_{[i]_2=1} c_i \, 2^{(i-1)\ell}$$

since the sum and the difference cancel out either the even or the odd powers. The coefficients can be either read directly with care to their offset, or dividing the above quantities by the appropriate power of 2 over the integers.

The KS2 algorithm is compatible with arithmetic modulo $f = x^n + 1$, when n is even. When doing this over $\mathbb{Z}_{f(2^\ell)}$ some care must be taken since reducing $c^{(\cdot)}$ modulo $f(2^\ell)$ may change its parity. In such case the coefficients can be recovered by either multiplying $c^{(+)} + c^{(-)}$ by $2^{-1} \bmod^{(+)} f(2^\ell)$ and $c^{(+)} - c^{(-)}$ by $2^{-\ell-1} \bmod^{(+)} f(2^\ell)$, or multiplying both quantities by a desired power of 2 modulo $f(2^\ell)$ and reading the coefficients with the appropriate offset. Packing and unpacking are identical to standard Kronecker substitution, i.e. the proof of Lemma 1 applies directly when working with such an f.

On the other hand, adapting packing and unpacking to combine the KS3 algorithm with modular reduction is somewhat more involved, requiring a fair amount of careful bit shifting. Implementing this strategy would roughly half the number of multiplications required at the cost of a more involved packing/unpacking algorithm. However, since our packing and unpacking routines already take time comparable to the actual multiplications they facilitate and since our target platform does not have efficient bit-shift operations, we did not attempt an implementation of KS3 or KS4.

4 Splitting the ring

Commercially available multipliers are usually capable of evaluating $(x, y, z) \mapsto x \cdot y \pmod z$ where $\log x, \log y, \log z < m$ for some fixed value of m which may be lower than what is required to apply Lemma 1 directly. In fact, for typical parameter sizes of lattice-based cryptography and of RSA, this is expected to be the case. Thus, in this section – where we focus on $f = x^n + 1$ with n a power of two – we explain our strategy for utilising these "too small" multipliers.

Let $a(x)$, $b(x)$, $c(x)$ be polynomials of degree $< n$ as defined in Lemma 1 and let ℓ be the packing length used, we want to compute $a(x) \cdot b(x) + c(x) \pmod{f(x)}$. So far we have considered two ways of doing this. First, we can pack every coefficient of each polynomial individually in a large enough buffer, say of length ℓ, and then directly compute the result using polynomial arithmetic. Alternatively, we can use Lemma 1 and evaluate $a(2^\ell) \cdot b(2^\ell) + c(2^\ell) \bmod^{(+)} (2^{n\ell} + 1)$ packing all the coefficients of each polynomial at once in a buffer of length $n\ell + 1$, and then unpack the final result. A third option consists of interpolating between these two methods by combining Kronecker substitution with (typically low-degree) polynomial arithmetic in order to shorten the lengths of the multiplier's inputs. This approach is similar to fast integer multiplication algorithms by Schönhage [Sch77] or Nussbaumer [Nus80] applying an FFT.

The idea is the following. Say we have

$$a(x) = a_0 + a_1\,x + a_2\,x^2 + a_3\,x^3 \quad \text{and} \quad b(x) = b_0 + b_1\,x + b_2\,x^2 + b_3\,x^3$$

and we want to compute $a(x) \cdot b(x) \pmod{x^4 + 1}$, i.e.

$$\begin{aligned}(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)\,x^3 + (a_2b_0 + a_1b_1 + a_0b_2 - a_3b_3)\,x^2 \\ +(a_1b_0 + a_0b_1 - a_3b_2 - a_2b_3)\,x \;\; + \;\; a_0b_0 - a_3b_1 - a_2b_2 - a_1b_3\end{aligned}$$

but we have a multiplier that would only let us work modulo $x^2 + 1$ given the ℓ required by Lemma 1. Letting $y = x^2$, we can write $a(x, y) = a^{(0)}(y) + a^{(1)}(y)\,x$ where

$$a^{(0)}(y) = a_0 + a_2\,y \quad \text{and} \quad a^{(1)}(y) = a_1 + a_3\,y,$$

and similarly for $b = b(x, y)$. Then, computing $a(x, y) \cdot b(x, y) \pmod{y^2 + 1}$ can be

accomplished by packing $A^{(\cdot)} = \textsc{Snort}(a^{(\cdot)})$, $B^{(\cdot)} = \textsc{Snort}(b^{(\cdot)})$, and multiplying

$$\begin{aligned}\hat{C}(x) &\coloneqq a(x, 2^\ell) \cdot b(x, 2^\ell) \bmod^{(+)} 2^{2\ell} + 1 \\ &= (A^{(0)} + A^{(1)}\, x) \cdot (B^{(0)} + B^{(1)}\, x) \bmod^{(+)} 2^{2\ell} + 1.\end{aligned}$$

We obtain

$$\begin{aligned}&(a_1b_1 - a_3b_3 + (a_3b_1 + a_1b_3)\, y)\, x^2 + a_0b_0 - a_2b_2 + (a_2b_0 + a_0b_2)\, y \\ &+(a_1b_0 + a_0b_1 - a_3b_2 - a_2b_3 + (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)\, y)\, x\end{aligned}$$

if we were to unpack the coefficients of $\hat{C}(x)$. Note that the coefficients on the second line match our target, but the coefficients on the first line do not (they are not grouped correctly and the signs do not necessarily match). This can be corrected by using the identity $y = x^2$ and thus rewriting $x^2 \to y$ and reducing again modulo $y^2 + 1$. From our intermediate representation $\hat{C}(x) = \hat{C}_0 + \hat{C}_1\, x + \hat{C}_2\, x^2$, this can be done by defining $C(x) = C_0 + C_1\, x$ with

$$C_0 \coloneqq \hat{C}_0 + (2^\ell \cdot \hat{C}_2 \bmod^{(+)} 2^{2\ell} + 1) \bmod^{(+)} 2^{2\ell} + 1 \quad \text{and} \quad C_1 = \hat{C}_1,$$

and then unpacking $C(x)$ to obtain $a \cdot b \pmod{x^4 + 1}$.

More generally, this can be formally described as follows. Let $n = m \cdot \omega$. Given a polynomial $p(x) = \sum_{i=0}^{n-1} p_i x^i$ of degree $< n$, we can set $y = x^m$, and then rewrite p as

$$\begin{aligned}p(x, y) &= \left(p_0 + p_{0+m}\, y + \cdots + p_{0+(\omega-1)m}\, y^{\omega-1}\right)\, x^0 \\ &+ \left(p_1 + p_{1+m}\, y + \cdots + p_{1+(\omega-1)m}\, y^{\omega-1}\right)\, x^1 \\ &+ \ldots \\ &+ \left(p_{m-1} + p_{m-1+m}\, y + \cdots + p_{m-1+(\omega-1)m}\, y^{\omega-1}\right)\, x^{m-1} \\ &= p^{(0)}(y) + p^{(1)}(y)\, x + \cdots + p^{(m-1)}(y)\, x^{m-1}\end{aligned}$$

where we write $p^{(i)}(y) \coloneqq \sum_{j=0}^{\omega-1} p_{i+jm}\, y^j$, polynomials in y of degree $< \omega$ (i.e. $p^{(i)} \leftarrow \textsc{Fr}(p, m, i)$, cf. Algorithm 10). The idea is to pack each $p^{(i)}$, $p \in \{a, b, c\}$, into buffers $P^{(i)} \coloneqq p^{(i)}(2^\ell) \bmod^{(+)} (2^{\omega\ell} + 1)$ of length $\omega\ell + 1$, and then evaluate

$$a(x, 2^\ell) \cdot b(x, 2^\ell) + c(x, 2^\ell) \bmod^{(+)} (2^{\omega\ell} + 1),$$

where $p(x, 2^\ell) \equiv \sum_{i=0}^{m} P^{(i)}\, x^i$. By Lemma 1, the integer modulo operation will act on the coefficients as reduction modulo $y^\omega + 1 \equiv x^n + 1 \pmod{y - x^m}$ would.

Working with polynomials $a(x, y)$, $b(x, y)$, the resulting polynomial $a(x, y) \cdot b(x, y)$ will be a linear combination of monomials of the form $y^i\, x^j$. If we were to substitute $x^m = y$ back now, we would obtain monomials of degree $\geq n$ every time that $im + j \geq n$, which we do not want. Furthermore, depending on how we index the $y^i\, x^j$ in our code, we may be in need of combining ("grouping") constant coefficients from different monomials $y^i\, x^j \neq y^r\, x^s$ mapping to the same power of x.

To better see what adjustments need to be done to the resulting polynomial in x, we look at $a(x, y) \cdot b(x, y) \pmod{y^\omega + 1}$ in detail.

$$\begin{aligned}a(x, y) \cdot b(x, y) &= \sum_{i,r=0}^{m-1} a^{(i)}(y)\, b^{(r)}(y)\, x^{i+r} \\ &= \sum_{i,r=0}^{m-1} \sum_{j,s=0}^{\omega-1} a_{i+jm}\, b_{r+sm}\, y^{j+s}\, x^{i+r} \\ &\equiv \sum_{i,r=0}^{m-1} \sum_{j,s=0}^{\omega-1} (-1)^{[\![j+s \geq \omega]\!]} a_{i+jm}\, b_{r+sm}\, y^{[j+s]_\omega}\, x^{i+r} \pmod{y^\omega + 1}\end{aligned}$$

Given that $y^{[j+s]_\omega}\, x^{i+r} \equiv x^{m\cdot[j+s]_\omega + i + r} \pmod{y - x^m}$, we can see that after reducing modulo $y^\omega + 1$ it will be necessary to further reduce modulo $y - x^m$ whenever $m \cdot [j+s]_\omega + i + r \geq n$, which can happen only if $i + r \geq m$. We do this by sending any monomial $y^j\, x^i$ where $i \geq m$ to $y^{j+1}\, x^{i-m} \pmod{y^\omega + 1}$, or equivalently by mapping monomials x^i with $i \geq m$ to $2^\ell\, x^{i-m}$, as done in Line 11 of Algorithm 11. This also takes care of groupings. Then, we can simply Sneeze every coefficient to obtain the final result. The full procedure results in Algorithms 10 and 11.

A possible optimisation could be that of choosing ℓ more aggressively. Indeed, we only ever need to pack polynomials of degree ω, and hence we could use this value in place of n. This would save $\approx \log m$ bits per packed coefficient while still being able to perform the reduction modulo $y^\omega + 1 \equiv 2^{\omega\ell} + 1$, overall resulting in a saving of size $\approx \omega \log m$ per packed polynomial $p^{(i)}(y)$. In this case one would need to unpack the $P^{(i)}$ before the second reduction and final grouping, and handle these afterwards on the CPU.

Input: polynomial $g \in R$
Input: step size m, dividing n
Input: offset o
1 $\omega \leftarrow n/m$;
2 **return** $\sum_{j=0}^{\omega} g_{m\cdot j+o}\, x^j$;
Algorithm 10: Ff(g, m, o). Return a new polynomial containing every mth coefficient of g, starting at offset o.

At the heart of Algorithm 11 is polynomial multiplication of two, typically low-degree, polynomials in line 8. A straightforward choice to realise this multiplication is schoolbook multiplication. This has quadratic complexity but is a simple algorithm. Another natural option is Karatsuba multiplication. In its simplest form, the algorithm computes the product $a + b \cdot x$ and $c + d \cdot x$ in $\mathbb{Z}[x]$ by computing the products $t_0 = a \cdot c$, $t_1 = b \cdot d$ and $t_2 = (a + b) \cdot (c + d) = ac + ad + bc + bd$ and outputting $t_0 + (t_2 - t_0 - t_1) \cdot x + t_2 x^2$. It has a cost of $3^{\lceil \log_2 L \rceil}$ multiplications for degree $L - 1$ polynomials. We note that finding better multiplication formulas for larger degrees is an active area of research [Mon05, FH07, CÖ10, BDEZ12].

5 Implementation

Using the strategies outlined in Sections 3 and 4, we are now ready to fix an implementation of Kyber and the KyberMulAdd gadget (see Corollary 3) using a big integer multiplier. We focus on the Kyber768 parameter set (or more precisely a variant) and implement it on the Infineon SLE 78 (SLE78CLUFX5000) equipped with an RSA, an AES and a SHA-256 co-processor and 16 Kbyte RAM. All our software is native code and written in C and assembly language.

5.1 Description of Kyber using Kronecker

First we provide a description of our variant of Kyber.CPA that takes into account Kronecker substitution. The algorithms closely resemble the implementation on our target device and include certain optimisations for performance and reduction of memory consumption.

In Algorithm 12 we describe our implementation of Kyber.CPA.Gen[8] and follow the notation of [SAB+17] where appropriate. The sampling of a uniform polynomial $a_{i,j} \in \vec{A}$ is done by Parse(XOF$(\rho||i||j)$) for a random seed $\rho \in \{0,1\}^{256}$ using an

[8] Instead of using SHA3-512 to hash the randomness, we directly take the output from the on-chip TRNG using the TRNG$(\cdot)$ function; see below.

Input: polynomial $a(x) \in R$
Input: polynomial $b(x) \in R$
Input: bitlength ℓ
Input: width parameter ω, dividing n
1 $f \leftarrow x^{\omega} + 1$;
2 $m \leftarrow n/\omega$;
`// construct polynomials` $A(x), B(x)$ `of degree` $< m$
3 **for** $i = 0, 1, \ldots, m-1$ **do**
4 $\quad A_i \leftarrow \textsc{Snort}(\textsc{Ff}(a(x),\, m,\, i),\, f,\, \ell)$;
5 $\quad B_i \leftarrow \textsc{Snort}(\textsc{Ff}(b(x),\, m,\, i),\, f,\, \ell)$;
6 **end**
7 $F \leftarrow 2^{\omega\ell} + 1$;
`// polynomial multiplication modulo integer` F
8 $\hat{C}(x) \leftarrow A(x) \cdot B(x) \bmod^{(+)} F$;
`// construct polynomial` $C(x)$ `of degree` $< m$
9 $C_{m-1} \leftarrow \hat{C}_{m-1}$;
10 **for** $i = 0, 1, \ldots, m-2$ **do**
11 $\quad C_i \leftarrow \hat{C}_i + \left(2^{\ell} \cdot \hat{C}_{m+i} \bmod^{(+)} F\right) \bmod^{(+)} F$;
12 **end**
`// construct tuple` $\hat{c}$ `of polynomials` $\hat{c}_i$ `each of degree` $< \omega$
13 **for** $i = 0, 1, \ldots, m-1$ **do**
14 $\quad \hat{c}_i \leftarrow \textsc{Sneeze}\,(C_i,\, f,\, \ell)$;
15 **end**
`// construct polynomial` $c(x)$ `of degree` $< n$
16 **for** $i = 0, 1, \ldots, \omega-1$ **do**
17 $\quad$ **for** $j = 0, \ldots, m-1$ **do**
18 $\quad\quad c_{m \cdot i + j} \leftarrow (\hat{c}_j)_i$;
19 $\quad$ **end**
20 **end**
21 **return** $c(x)$;

Algorithm 11: $a(x) \cdot b(x) \bmod x^n + 1$ using an integer multiplier capable of performing modular multiplication of integers up to $\omega\ell + 1$ bits.

Extendable Output Function (XOF) denoted as XOF$(\cdot)$. The sampling of a secret or noise polynomial in R_q is described by CBD(PRF(σ, N)) where CBD stands for centred binomial distribution and where PRF is a pseudorandom function (PRF) that takes a random seed $\sigma \in \{0,1\}^{256}$ and an integer N for domain separation. In [SAB+17] it is specified that PRF(σ, N) = SHAKE-256(σ, N) and that XOF = SHAKE-128.

With regard to arithmetic, it is easy to see that $s_0, \ldots, s_{k-1}$ are used k times each, when computing $\vec{A} \cdot \vec{s}$. Thus a straightforward optimisation is to pack them into a big integer only once. This resembles some similarity to the NTT, where it is also possible to achieve speedups by the very simple observation that polynomials that are used several times have to be transformed into the NTT domain only once. To obtain more control over the usage of Snort and Sneeze, which is already integrated into the high-level gadget KyberMulAdd, we split KyberMulAdd into sub-functions. The $\hat{C} = \textsc{MulAddSingle}(A, B, C)$ function takes as input $A = \textsc{Snort}(a), B = \textsc{Snort}(b), C = \textsc{Snort}(c)$ for $a, b, c \in R_q$ and computes $\hat{D}(x) \leftarrow A(x) \cdot B(x) + C(x) \bmod^{(+)} F$ as specified in line 8 of Algorithm 11. The $D = \textsc{FinalEll}(\hat{D})$ function takes $\hat{D}$ and constructs the polynomial $D(x)$ of degree $< m$ (line 11 of Algorithm 11) by multiplying by 2^{ℓ}. To save stack memory we do not generate the full matrix $\vec{A}$ but only one coefficient after the other. All in all, our approach to

key generation requires $k^2 + 2k$ calls to Snort, k^2 big integer multiplications realised by MulAddSingle and k calls to Sneeze as well as FinalEll.

$\rho \xleftarrow{\$} \text{TRNG}()$; // $\rho \in \{0,1\}^{256}$ sampled from internal TRNG
$\sigma \xleftarrow{\$} \text{TRNG}()$; // $\sigma \in \{0,1\}^{256}$ sampled from internal TRNG
$N \leftarrow 0$;
// Sample $\vec{s}$ and transform to $\vec{S}$
for $i = k-1, k-2, \ldots, 0$ **do**
  $s_{tmp} \leftarrow \text{CBD}(\text{PRF}(\sigma, N))$;
  $N \leftarrow N + 1$;
  $S_i \leftarrow \text{Snort}(s_{tmp})$;
end
// Compute $\vec{A}\vec{s} + \vec{e}$
for $i = 0, 1, \ldots, k-1$ **do**
  $e \leftarrow \text{CBD}(\text{PRF}(\sigma, N))$;
  $N \leftarrow N + 1$;
  $\hat{T} \leftarrow \text{Snort}(e)$;
  for $j = 0, 1, \ldots, k-1$ **do**
    $a_{tmp} \leftarrow \text{Parse}(\text{XOF}(\rho||i||j))$;
    $A_{tmp} \leftarrow \text{Snort}(a_{tmp})$;
    $\hat{T} \leftarrow \text{MulAddSingle}(A_{tmp}, S_j, \hat{T})$;
  end
  $T \leftarrow \text{FinalEll}(\hat{T})$;
  $t_i \leftarrow \text{Sneeze}(T)$;
end
$pk \leftarrow \text{Encode}_{d_t}(\text{Compress}_q(\vec{t}, d_t)||\rho)$;
$sk \leftarrow \text{Encode}_{13}(\vec{s} \bmod^{(+)} q)$;
return $pk_{CPA} \coloneqq pk$, $sk_{CPA} \coloneqq sk$;

Algorithm 12: Kyber.CPA.Imp.Gen

CPA-secure Kyber encryption is described in Algorithm 13 where the computation of $\vec{A}^T\vec{r} + \vec{e}_1$ can be realised in the same way as the key generation procedure by packing each polynomial of $\vec{r}$ into $\vec{R}$ only once and with on-the-fly generation of polynomials of $\vec{A}$ to save stack memory. The only difference is that we initialise $\hat{U}_{tmp}$ with on-the-fly sampled and packed error polynomials $e_i \in \vec{e}_1$ before computing the k scalar products. For $\langle \vec{t}, \vec{r} \rangle + e_2 + \lceil \frac{q}{2} \rfloor \cdot m$ we sample e_2 by $e \leftarrow \text{CBD}(\text{PRF}(\sigma, N))$, set $\hat{V} \leftarrow \text{Snort}(e + \bar{m})$ and then compute the scalar product in a loop with $\hat{V} \leftarrow \text{MulAddSingle}(R_i, T_{tpm}, \hat{V})$. All in all, Kyber.CPA.Imp.Enc requires $k^2 + 3k + 1$ calls to Snort, $k^2 + k$ big integer multiplications by MulAddSingle and $k + 1$ calls to Sneeze as well as FinalEll.

In Algorithm 14 we describe CPA-secure Kyber decryption. The implementation of the scalar product to compute $\langle \vec{s}, \vec{u} \rangle$ follows the approach from encryption. To reuse MulAddSingle and to save code needed for a subtraction gadget we first negate v by computing $\hat{V} \leftarrow \text{Snort}(-v)$ and then negate the final result again as $\text{Compress}_q(-v, 1)$ to obtain $v - \langle \vec{s}, \vec{u} \rangle$. We need $2k + 1$ calls of Snort, k big integer multiplications by MulAddSingle and one call to Sneeze as well as FinalEll.

5.2 Implementation of Kyber on SLE 78

We now give details of our implementation of CPA and CCA-secure Kyber768 (thus $k = 3$) on the SLE 78 that are independent of the chosen approach for packing and big integer

Input: $m \in \mathcal{M}$
Input: pk_{CPA}
$\vec{t}, \rho \leftarrow \text{DECODE}_{d_t}(pk_{CPA})$;
$\vec{t} \leftarrow \text{DECOMPRESS}_q(\vec{t})$;
$N \leftarrow 0$;
// Sample MLWE secret $\vec{r}$ and transform to $\vec{R}$
for $i = k-1, k-2, \ldots, 0$ **do**
  $r_{tmp} \leftarrow \text{CBD}(\text{PRF}(\sigma, N))$;
  $N \leftarrow N + 1$;
  $R_i \leftarrow \text{SNORT}(r_{tmp})$;
end
// Compute $\vec{A}^T\vec{r} + \vec{e_1}$
for $i = 0, 1, \ldots, k-1$ **do**
  $e \leftarrow \text{CBD}(\text{PRF}(\sigma, N))$;
  $\hat{U}_{tmp} \leftarrow \text{SNORT}(e)$;
  $N \leftarrow N + 1$;
  for $j = 0, 1, \ldots, k-1$ **do**
    $a_{tmp} \leftarrow \text{PARSE}(\text{XOF}(\rho||i||j))$;
    $A_{tmp} \leftarrow \text{SNORT}(a_{tmp})$;
    $\hat{U}_{tmp} \leftarrow \text{MULADDSINGLE}(A_{tmp}, R_j, \hat{U}_{tmp})$;
  end
  $U_{tmp} \leftarrow \text{FINALELL}(\hat{U}_{tmp})$;
  $u_i \leftarrow \text{SNEEZE}(U_{tmp})$;
end
// Compute $\langle \vec{t}, \vec{r} \rangle + e_2$
$\bar{m} \leftarrow \text{ENCODEMSG}(m)$;
$e \leftarrow \text{CBD}(\text{PRF}(\sigma, N))$;
$e \leftarrow e + \bar{m}$;
$\hat{V} \leftarrow \text{SNORT}(e)$;
for $i = 0, 1, \ldots, k-1$ **do**
  $T_{tpm} \leftarrow \text{SNORT}(t_i)$;
  $\hat{V} \leftarrow \text{MULADDSINGLE}(R_i, T_{tpm}, \hat{V})$;
end
$V \leftarrow \text{FINALELL}(\hat{V})$;
$v \leftarrow \text{SNEEZE}(V)$;
// Encode ciphertext
$c_1 \leftarrow \text{ENCODE}_{d_u}(\text{COMPRESS}_q(\vec{u}, d_u))$;
$c_2 \leftarrow \text{ENCODE}_{d_v}(\text{COMPRESS}_q(v, d_v))$;
return $c \coloneqq (c_1||c_2)$;

Algorithm 13: KYBER.CPA.IMP.ENC

multiplication (see Section 5.3 and Section 5.4). All our implementations are not fully compatible with the specification as Kyber is explicitly defined with a specific NTT and assumes that the pseudorandom polynomials of $\vec{A}$ are already output by the sampler in the NTT domain.

To expand randomness into a longer bitstream, Kyber originally specifies the use of various instances from the SHA3 family as PRNG (originally, XOF is SHAKE-128 and PRF is SHAKE-256). We implemented one version of the samplers that is compatible with the specification where SHAKE-128 and SHAKE-256 are realised in software. Hardware acceleration is not possible as our target device does not have a SHA3 hardware accelerator.

Input: $c \coloneqq (c_1||c_2)$
Input: sk_{CPA}
$\vec{s} \leftarrow \textsc{Decode}_{13}(sk_{CPA})$;
$\vec{u} \leftarrow \textsc{Decompress}_q(\textsc{Decode}_{d_u}(c_1))$;
$v \leftarrow \textsc{Decompress}_q(\textsc{Decode}_{d_v}(c_2))$;
$\hat{V} \leftarrow \textsc{Snort}(-v)$;
// Compute $v - \langle \vec{s}, \vec{u} \rangle$
for $i = 0, 1, \ldots, k-1$ **do**
  $U_{tmp} \leftarrow \textsc{Snort}(u_i)$;
  $S_{tmp} \leftarrow \textsc{Snort}(s_i)$;
  $\hat{V} \leftarrow \textsc{MulAddSingle}(S_{tmp}, U_{tmp}, \hat{V})$;
end
$V \leftarrow \textsc{FinalEll}(\hat{V})$;
$v \leftarrow \textsc{Sneeze}(V)$;
return $\textsc{Encode}_1(\textsc{Compress}_q(-v, 1))$;

Algorithm 14: Kyber.CPA.Imp.Dec

The SHA3 implementation written in C has been optimised to some extent with assembly to remove obvious performance bottlenecks introduced by the compiler. Additionally, we have implemented a (non-compatible) Kyber variant that is using AES-256 in counter mode to implement XOF and PRF. A similar approach has been used by Google in their NewHope experiment where the constant polynomial a was also sampled using AES [Lan16]. Even though there are some theoretical concerns [ADPS16], this approach appears to be secure in practice. When AES-256 is chosen as PRNG we can rely on the AES co-processor of the SLE78CLUFX5000 and do not need to implement AES in software.

A difference that is not noticeable by a user is that we, as previously mentioned, do not hash the randomness provided to key generation due to the availability of a TRNG. The hashing of the input randomness in the Kyber specification is intended as a protection against leakage of the internal state of a random number generator. However, on our target device we have access to a certified RNG with appropriate post-processing and thus expensive computation of SHA3-512 is unnecessary.

The implementations of CBD, Parse, Encode, Decode and $\textsc{Decompress}_q$ follow the C reference implementation and are not particularly optimised using assembly. Our implementation of CCA-secure Kyber using the FO transformation is denoted as Kyber.CCA.Imp.Gen for key generation, Kyber.CCA.Imp.Enc for encapsulation and Kyber.CCA.Imp.Dec for decapsulation and we straightforwardly follow Algorithm 4 to 6. The main additional operations demanded by the CCA conversion are the computation of hash functions to implement random oracles. In one version of our implementation we follow the specification where H is using SHA3-256 and G is using SHA3-512 and where SHA3 is implemented in software. Additionally, we implemented a variant where H is realised by the MAC-based scheme HKDF [Kra10] using a SHA-256 co-processor and where H is realised by a call to SHA-256. The usage of HKDF is necessary as the output of G has to be longer than a single SHA-256 hash.

5.3 Realisation of KyberMulAdd with KS1

The KyberMulAdd gadget consists of the functions Snort, MulAddSingle, FinalEll, and Sneeze. In case of KS1 (standard Kronecker substitution) parameters $(\omega, m) = (64, 4)$ can be used (see Algorithm 11). Then 64 coefficients can be packed into one integer and it is possible to perform polynomial arithmetic modulo $x^4 + 1$. When aiming for minimal size we could have used 25 bits of precision per coefficient and thus $64 \cdot 25 = 1600$

bits in total. However, to simplify the packing algorithm we have chosen 32 bits per coefficient (thus $\ell = 32$) which leads to integers of $64 \cdot 32 = 2048$ bits. This way no shifts by arbitrary integers are required as everything is immediately word aligned in Snort. This provides a performance advantage as the SLE 78 needs one cycle for each shift to the right or left. Moreover, the big integer multiplier is relatively fast and thus the tradeoff between simpler packing/unpacking and slightly larger integer coefficients turned out to be favorable. However, on different platforms this may not be the case. An issue that costs some performance is the correct handling of carry bits caused by negative coefficients in Snort.

For a single big integer multiplication in MulAddSingle we use the RSA co-processor on the SLE78CLUFX5000 which has five registers of length slightly larger than 2048-bit. In a simplified model it is able to compute additions of two registers in 8 cycles while a multiplication with modular reduction takes roughly 9,300 cycles. However, not all registers are general purpose. One register is a working register that contains the result of a computation and is not directly accessible from the CPU. Another register is needed to store the modulus when performing operations modulo p. Thus three registers are available for temporary results or operands. Naturally, for an integer multiplication modulo $\log_2 p = 2048$, two registers are already occupied with operands.

For KS1 with parameters $(\omega, m) = (64, 4)$ and $\ell = 32$ one option to realise the polynomial multiplication $\hat{C}(x) \leftarrow A(x) \cdot B(x) \bmod^{(+)} F$ for $A, B, C \in \mathbb{Z}_p$ with $p = F = 2^{\omega\ell} + 1 = 2^{2048} + 1$ described in line 8 of Algorithm 11 would be schoolbook multiplication. As we have to do polynomial arithmetic modulo $x^4 + 1$ this would lead to $4^2 = 16$ multiplications in $\mathbb{Z}_p$ due to the quadratic complexity of schoolbook multiplication. To reduce the number of multiplications we have chosen Karatsuba multiplication for our KS1 implementation of the MulAddSingle function, which leads to 9 multiplications, 17 additions and 16 subtractions in $\mathbb{Z}_p$. These numbers include additions or subtractions required for the modulo $x^4 + 1$ operation. In general, Karatsuba multiplication leads to a large number of additions as a trade-off for fewer multiplications. An approach where the additions are executed on the RSA co-processor would be possible but requires a lot of transfers. We thus decided to exploit the ability to run the co-processor and the CPU in parallel. While the RSA co-processor executes a modular multiplication we compute long integer additions in parallel on the CPU. This can easily be achieved by the appropriate rearrangement of multiplication and addition/subtraction operations in the Karatsuba formula. For simplicity, we give a sort example for $a(x) = a_0 + a_1\, x$ and $b(x) = b_0 + b_1\, x$. A polynomial multiplication can be computed with Karatsuba as $a(x)b(x) = a_0b_0 + ((a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0)\, x + a_1b_1\, x^2$. Here some additions can be performed in parallel to multiplications where $T_1 = a_1 \cdot b_1$ and $T_2 = b_1 + b_0$ is computed in parallel, then $T_3 = a_0 \cdot b_0$ and $T_4 = a_1 + a_0$, then $T_5 = T_2 \cdot T_4$ and $T_6 = T_1 + T_3$. Final additions and computations are $T_7 = x^2 \cdot T_1$, $T_8 = T_5 - T_6$, $T_9 = x \cdot T_8$, $T_{10} = T_7 + T_9$, and $T_{11} = T_3 + T_{10}$ where $a(x)b(x) = T_{11}$. Note in our specific case also some additions or subtractions caused by the modulo $x^4 + 1$ operation are also hidden behind multiplications. For the remaining additions and subtractions we make use of the co-processor. To save cycles for transfers we store the result of several additions/subtractions in one register of the co-processor so that we only have to transfer values into the co-processor and then read out the final result. The FinalEll function (see line 10 of Algorithm 11) requires 3 multiplications by 2^ℓ. They are implemented on the co-processor using a special command that allows fast shifting by 32 bits and are thus relatively cheap.

5.4 Realisation of KyberMulAdd with KS2

The KyberMulAdd gadget can also be implemented for KS2 (compact Kronecker) with parameters $(\omega, m) = (128, 2)$. Compact Kronecker would allow to pack 128 coefficients into two big integers with 13 bits per coefficients. With 13 bits of precision per coefficient

$13 \cdot 128 = 1664$ bits would be required in total. However, similarly to KS1 we use 16 bits for easier packing/unpacking and end up with integers of size of $16 \cdot 128 = 2048$ bits ($\ell = 16$). Computations are then performed on two polynomials modulo $x^2 + 1$. This leads to $2 \cdot 2^2 = 8$ multiplications in $\mathbb{Z}_p$ for $p = F = 2^{\omega\ell} + 1$ when using schoolbook multiplication. With Karatsuba a reduction to $2 \cdot 3 = 6$ multiplication would be possible. As the difference between Karatsuba and schoolbook is small we use schoolbook multiplication to implement KS2. This allows us to store partial products during schoolbook multiplication in the free register of the RSA co-processor. This way we can perform additions with the RSA co-processor and save time as we do not have to retrieve every result from the co-processor into the memory.

5.5 Realisation of MulAdd for other RLWE-based schemes

In a similar fashion to Sections 5.3 and 5.4, one could choose to implement MulAdd for RLWE-based schemes working with polynomials of higher degree. For example, NewHope [AAB+17] proposes the following set of parameters (NewHope512) targetting Category 1 security [Nat16]: $n = 512, q = 12289, \eta = 8, f = x^n + 1$. Lemma 1 suggests using $\ell = 26$ (resp. $\ell = 13$) bits of precision per coefficient for use with KS1 (resp. KS2). To further improve packing and unpacking performance, we consider $\ell = 32$ (resp. $\ell = 16$), which results in parameters $(\omega, m) = (64, 8)$ with $32 \cdot 64 = 2048$ bits per polynomial coefficient (resp. $(\omega, m) = (128, 4)$ with $16 \cdot 128 = 2048$ bits per polynomial coefficient), assuming our integer multiplier supports inputs of length $2048 + 1$ bits. Schoolbook multiplication would then requrie $8^2 = 64$ (resp. $2 \cdot 4^2 = 32$) multiplications in $\mathbb{Z}_p$ for $p = F = 2^{\omega\ell} + 1$, while recursively applying Karatsuba would result in $3^{\log_2 8} = 27$ (resp. $2 \cdot 3^{\log_2 4} = 18$) multiplications. While this procedure gives us a rough estimate of the cost of implementing NewHope512 using different strategies, it does not take into account concrete implementation issues such as the size and number of registers available in the CPU, the number of additions required, or the possible speedups from running light operations on the CPU while waiting for the modular multiplier to return a result for each multiplication.

6 Performance and comparison

In this section we describe the performance of our Kyber768 implementation on the SLE 78 and compare our results to related work. All cycle counts are averages of several runs and have been measured on a cycle accurate FPGA-based emulator.

6.1 Implementation performance

In Table 2 we provide cycle counts of our implementation of Kyber768, its variants, and selected sub-functions. The results show similar performance for the KS1 and KS2 approach in Kyber.CPA.Imp with a small advantage for KS2. The explanation is that KS1 with Karatsuba requires only a single multiplication more than KS2 with schoolbook. The additional additions necessary for Karatsuba in KS1 can effectively be hidden by running them in parallel with the RSA co-processor and Snort for KS1 is roughly twice as fast than for KS2. However, this is only a conclusion for the particular parameters using the specific co-processor. KS1 and KS2 might lead to very different results in case our approach would be used to implement a scheme like NewHope where n is much larger than in Kyber. Cycle counts for CBD and Parse show that usage of the AES co-processor provides a significant speedup compared to the SHA3 software implementation. For CBD the difference is a factor of 300 and for Parse even a factor of 945. With more optimization of the SHA3 software, e.g. by writing it fully in assembly, it might be possible to reduce this to some extent. An additional advantage is that the AES co-processor already implements some

countermeasures against physical attacks. Such attacks are not the focus of our work but a secured PRNG would be easier to realise with the AES co-processor than by using a shared SW implementation of SHA3 (see [OSPG18] where this necessity is discussed and performance of a shared SHA3 is given). With roughly $\approx 376,000$ cycles used for sampling in KYBER.CPA.IMP.GEN ($\approx 9 \times$ PARSE $+ 6 \times$ CBD) and roughly $\approx 407,000$ cycles used in KYBER.CPA.IMP.ENC ($\approx 9 \times$ PARSE $+ 7 \times$ CBD) the sampling requires only about 10 percent of the overall runtime. Additionally, in Table 3 we have computed the sum of cycles based on the calls to measured subfunctions for KS1. This gives an overview what amount of cycles can be associated to each operation. In all three functions the most cycles are contributed by MULADDSINGLE and SNEEZE. They would be a natural target for further optimization.

Compared to a Kyber768 implementation that is using the NTT as specified in [SAB+17] on the SLE 78 in software, our approach of using the co-processor to compute the KYBERMULADD gadget provides an advantage. On the SLE 78 a single $n = 256$ NTT costs 997,691 cycles. The computation of KYBERCPA.ENC for $k = 3$ requires 10 calls to the NTT[9] which alone would account for roughly $10 \cdot 997{,}691 \approx 10.0$ million cycles plus additional overhead from pointwise multiplication and addition.

In case one would want to make our implementation compatible with Kyber as specified in [SAB+17] in terms of NTT usage and still use the KYBERMULADD gadget we would have to perform k^2 inverse NTTs and then use our multiplication algorithm. This would add roughly $3^2 \cdot 997{,}691 \approx 9.0$ million cycles to GEN and ENC when executed on the CPU. It would basically nullify all gains from a different and faster algorithm for polynomial multiplication.

All in all, when our Kyber variant that is using the AES co-processor (i.e. AES-HW) is run on our target device with an average clock frequency of 50 MHz we can execute KYBER.CPA.IMP.GEN in 72.5 ms, KYBER.CPA.IMP.ENC in 94.9 and KYBER.CPA.IMP.DEC in 28.4 ms.

For the CCA variant the decryption becomes slower due to the re-encryption but the additional overhead of the hash functions H and G is rather low when the SHA-256 co-processor is used (HW-SHA-256) to compute SHA-256 and HKDF with HMAC-SHA-256. When H and G are instantiated with SHA3 implemented in software (SW-SHA3) a significant portion of the computation is now attributed to SHA3. In comparison we can execute KYBER.CCA.IMP.GEN in 79.6 ms (2,903 ms with SW-SHA3), KYBER.CCA.IMP.ENC in 102.4 ms (571.2 ms with SW-SHA3) and KYBER.CCA.IMP.DEC in 132.7 ms (394.0 ms with SW-SHA3). An implementation of Kyber that is fully compatible with the specification [SAB+17] would not achieve practical performance mainly due to the slow SHA3 PRNG performance and to a lesser extent due to the slower NTT in software. Of course, further low-level optimization of SHA3 and the NTT could change this picture to some extent.

6.2 Comparison with related work

In Table 4 we provide a comparison of our results with related work on similar target platforms. However, it should be noted that such a comparison will always lack precision as many parameters of published implementations differ in terms of cryptographic (post-quantum) bit-security level, implementation security level, exact variant of a scheme, CPU architecture, maximum clock frequency of the device, or availability of specific accelerators. Moreover, only limited information is available about most smart card platforms and those platforms are often not available without signing non-disclosure agreements. It is also clear that the requirements for a certified contactless high security controller, where most

[9] See [SAB+17, Algorithm 5] where 3 NTTs are required to transform $\vec{r}$, 3 inverse NTTs are applied to $\vec{A} \circ \vec{r}$, 3 inverse NTTs are needed to transform $\vec{t}$ and 1 inverse NTT is then needed to obtain v.

Table 2: Performance of our work on the SLE 78 target device in clock cycles.

Operation	Cycles
Snort (KS1)	31,017
Sneeze (KS1)	295,730
MulAddSingle (KS1)	201,767
FinalEll (KS1)	28,381
Snort (KS2)	70,015
Sneeze (KS2)	295,331
MulAddSingle (KS2)	186,652
FinalEll (KS2)	90,728
NTT ($n = 256$, in SW)	997,691
Pointwise-Multiplication ($n = 256$, in SW)	356,549
CBD(PRF(σ, N)) (Software-SHA3)	9,341,406
CBD(PRF(σ, N)) (Hardware-AES)	31,068
Parse(XOF($\rho\|\|i\|\|j$)) (Software-SHA3)	19,934,170
Parse(XOF($\rho\|\|i\|\|j$)) (Hardware-AES)	21,081
Kyber.CPA.Imp.Gen (HW-AES: PRF/XOF; KS1)	3,953,224
Kyber.CPA.Imp.Enc (HW-AES: PRF/XOF; KS1)	5,385,598
Kyber.CPA.Imp.Dec (KS1)	1,382,963
Kyber.CPA.Imp.Gen (HW-AES: PRF/XOF; KS2)	3,625,718
Kyber.CPA.Imp.Enc (HW-AES: PRF/XOF; KS2)	4,747,291
Kyber.CPA.Imp.Dec (KS2)	1,420,367
Kyber.CCA.Imp.Gen (HW-AES: PRF/XOF; HW-SHA-256: H; KS2)	3,980,517
Kyber.CCA.Imp.Enc (HW-AES: PRF/XOF; HW-SHA-256: G, H; KS2)	5,117,996
Kyber.CCA.Imp.Dec (HW-AES: PRF/XOF; HW-SHA-256: G, H; KS2)	6,632,704
Kyber.CCA.Imp.Gen (HW-AES: PRF/XOF; SW-SHA3: H; KS2)	14,512,691
Kyber.CCA.Imp.Enc (HW-AES: PRF/XOF; SW-SHA3: G, H; KS2)	18,051,747
Kyber.CCA.Imp.Dec (HW-AES: PRF/XOF; SW-SHA3: G, H; KS2)	19,702,139

computations are done using co-processors, are expected to lead to different CPU designs or low-level implementations than that for a high performance embedded microcontroller.

As we use an RSA co-processor for lattice-based cryptography, a natural target for a comparison is RSA. The cycle counts given in Table 4 for co-processor supported RSA on our SLE 78 target device are based on the data sheet. With an average clock frequency of 50 MHz on the SLE 78, RSA encryption can be executed in 6 ms while RSA decryption with CRT needs 120 ms. In comparison with our work this shows that our Kyber implementation is two orders of magnitude slower for encryption but performs decryption with similar speed. In case RSA is not used with CRT our Kyber decryption even outperforms RSA. However, it should be noted that the RSA cycle counts do not account for padding like Optimal Asymmetric Encryption Padding (OAEP) which is often used to achieve CCA2 security for RSA. However, they include countermeasures against physical attacks (e.g. exponent blinding or message blinding, see [FWA+13]) while our implementation does not.

Publicly available information on the performance of RSA and ECC on various smart cards running the JavaCard platform can be found in works like [DRHM17, SNS+16], the Bachelor's thesis of Kvašňovský [Kva16] as well as in the JCAlgTest project[10]. Across the selected cards, the runtime for an RSA2048 encryption function call is in the range from 8 to 74 ms while RSA2048 decryption takes between 426 to 2,927 ms and 140 to 1,569 ms when using the Chinese Remainder Theorem (CRT). On-card key generation

[10] See `https://www.fi.muni.cz/~xsvenda/jcalgtest/comparative-table.html`.

Table 3: Called functions, number of calls, clock cycles, and final sum of clock cycles.

Kyber.CPA.Imp.Gen (KS1)			
Function	Calls	Cycles per function	Product
CBD(PRF(σ, N)) (HW-AES)	6	31,068	186,408
Parse(XOF($\rho\|\|i\|\|j$)) (HW-AES)	9	21,081	189,729
Snort	15	31,017	465,255
MulAddSingle	9	201,767	1,815,903
Sneeze	3	295,730	887,190
FinalEll	3	28,381	85,143
Encode/Decode	-	-	400,226
			= 4,029,854

Kyber.CPA.Imp.Enc (KS1)			
Function	Calls	Cycles per function	Product
CBD(PRF(σ, N)) (HW-AES)	7	31,068	217,476
Parse(XOF($\rho\|\|i\|\|j$)) (HW-AES)	9	21,081	189,729
Snort	19	31,017	589,515
MulAddSingle	12	201,767	2,421,204
Sneeze	4	295,730	1,182,920
FinalEll	4	28,381	113,524
Encode/Decode	-	-	676,453
			= 5,390,629

Kyber.CPA.Imp.Dec (KS1)			
Function	Calls	Cycles per function	Product
CBD(PRF(σ, N)) (HW-AES)	0	31,068	0
Parse(XOF($\rho\|\|i\|\|j$)) (HW-AES)	0	21,081	0
Snort	4	31,017	217,119
MulAddSingle	3	201,767	605,301
Sneeze	1	295,730	295,730
FinalEll	1	28,381	28,381
Encode/Decode	-		365,175
			= 1,511,706

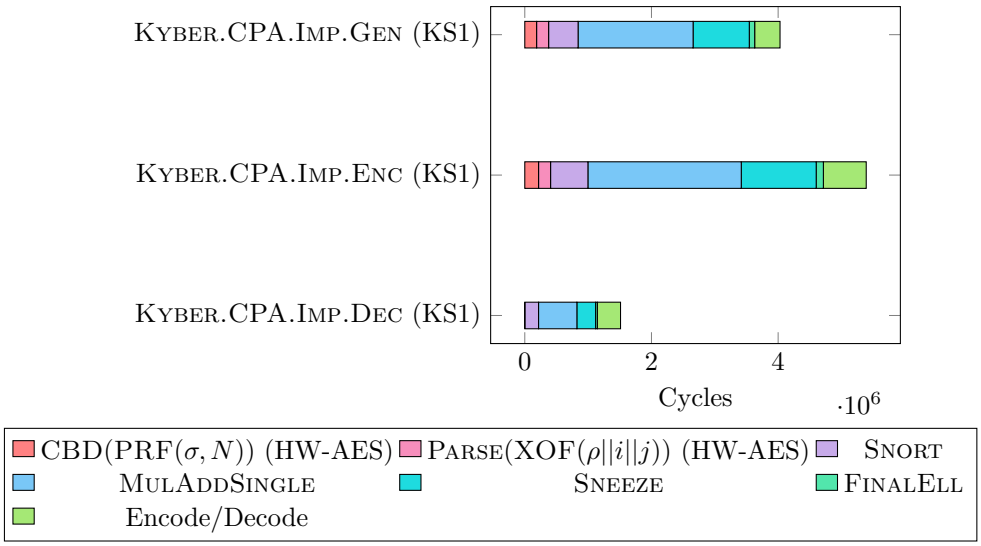

Figure 1: Total cycle counts per function from Table 3.

for RSA2048 is a complex process with a variable runtime due to the required primality testing and takes between 6,789 and 44,143 ms. There is also a certain overhead by the JavaCard platform compared to a pure native implementation as well as overhead from various countermeasures against physical attacks.

For comparison with other post-quantum schemes we have ported the reference implementation of ephemeral/CPA-secure NewHope with $n = 1024$ claiming 255-bits of post-quantum security onto our target device. To obtain a fair comparison we also changed the internal PRNG to use the co-processor-based AES in counter-mode and we removed costly randomness hashing in the key generation. With these modifications the main bottleneck in NewHope is the computation of NTTs. When comparing CPA-secure NewHope implementation (claimed 255-bit security level) with our CPA-secure Kyber (claimed 161-bit security level) in an ephemeral key setting[11], we achieve a factor of 6 better performance for Alice (Gen+Dec) and a factor of 7 better performance for Bob (Enc). Note that the implementation of our variant of Kyber that is not using the NTT would most likely lead a loss of performance on other platforms. However, the implementation of Saber on ARM given in [KMRV18] shows that high performance is also possible without using the NTT when parameters are chosen accordingly.

Most modern general purpose ARM-based microcontroller platforms (e.g. Cortex-M) have the advantage of a 32-bit architecture and are equipped with a single-cycle or few-cycle multiplier (optional in Cortex-M0). Thus good performance can be expected for most arithmetic operations, e.g. the inner loop of the NTT. Open-source implementations of Kyber768 and NewHope1024 targeting general purpose ARM controllers are available through the *mupq* project [va18]. It can be seen that in comparison with such a different class of devices our CCA-secure Kyber768 implementation of Gen and Enc is slower than CCA-secure Kyber768 on ARM using the NTT.

7 Conclusion and future work

In this work we have shown that fast post-quantum cryptography is feasible on current smart card platforms. On a commercially available device it is possible to obtain a significant speedup of the arithmetic of lattice-based cryptography by reusing already existing co-processors dedicated to the acceleration of RSA or ECC. Our work can thus be used by the industry for a possibly smoother migration towards PQC, by reusing already existing and available hardware. Our work also shows that the NTT might not always be the superior polynomial multiplication algorithm.[12] This seems to be a worthwhile consideration in the context of the NIST standardisation process where some schemes made the NTT part of their definition. Moreover, our results show that the performance of lattice-based schemes on particular embedded devices highly depends on the speed of the underlying PRNG. It might be worthwhile to consider constructions that make use of PRNGs based on AES instead of SHA3 due to the better availability of (secured) AES hardware acceleration on smart cards or constrained devices in general. The same argument applies to the instantiation of hash functions using SHA-256.

With regard to the optimisation of our particular Kyber implementation, a possible next step is an implementation on an ARM-based smart card or embedded secure element equipped with an ECC/RSA co-processor. On such an architecture the comparison to standard microcontroller-based implementations of PQC (e.g. [vMOG15, DHH+15, OSPG18]) would be much easier. Additionally, it is an open question how much speedup ECC/RSA co-processors will actually provide on ARM platforms equipped with a single-

[11] Of course, a better target for comparison would be Kyber1024 with 218-bit security but an implementation on SLE 78 is not available as we focus on Kyber768.

[12] See also an NTT-related discussion on the NIST PQC mailing list: `https://groups.google.com/a/list.nist.gov/forum/#!topic/pqc-forum/r9R7OJT6x_c`.

Table 4: Comparison of our work with other PKE or KEM schemes on various microcontroller platforms in clock cycles.

Scheme	Target	Gen	Enc	Dec
Kyber768[a] (CPA; our work)	SLE 78	3,625,718	4,747,291	1,420,367
Kyber768[b] (CCA; our work)	SLE 78	3,980,517	5,117,996	6,632,704
RSA-2048[c]	SLE 78	-	$\approx$ 300,000	$\approx$ 21,200,000
RSA-2048 (CRT)[d]	SLE 78	-	$\approx$ 300,000	$\approx$ 6,000,000
Kyber768 (CPA+NTT)[e]	SLE 78	$\approx$ 10,000,000	$\approx$ 14,600,000	$\approx$ 5,400,000
NewHope1024[f]	SLE 78	$\approx$ 14,700,000	$\approx$ 31,800,000	$\approx$ 15,200,000
Kyber768[g]	ARM	1,200,351	1,497,789	1,526,564
NewHope-1024[h]	ARM	1,168,224	1,738,922	298,877
CPA-RLWE-512[i]	AVR	-	1,975,806	553,536
CCA-RLWE-1024[j]	ARM	2,669,559	4,176,68	4,416,918
Saber[k]	ARM	1,147,000	1,444,000	1,543,000
QC-MDPC[l]	ARM	-	7,018,493	42,129,589
Curve25519[m]	MSP	5,941,784	11,883,568	5,941,784
Curve25519[n]	ARM	3,589,850	7,179,700	3,589,850

[a] CPA-secure Kyber variant using the AES co-processor to implement PRF/XOF and KS2 on SLE 78 @ 50 MHz.

[b] CCA-secure Kyber variant using the AES co-processor to implement PRF/XOF, the SHA-256 co-processor to implement G and H and KS2 on SLE 78 @ 50 MHz.

[c] RSA-2048 encryption with short exponent and decryption without CRT and with countermeasures on SLE 78 @ 50 MHz. Extrapoliation based on data-sheet.

[d] RSA-2048 decryption with short exponent and decryption with CRT and countermeasures on SLE 78 @ 50 MHz. Extrapoliation based on data-sheet.

[e] Extrapolation of cycle counts of CPA-secure Kyber768 based on our implementation assuming usage of the AES co-processor to implement PRF/XOF and a software implementation of the NTT with 997,691 cycles for an NTT on SLE 78 @ 50 MHz.

[f] Reference implementation of constant time ephemeral NewHope key exchange ($n = 1024$) [ADPS16] modified to use the AES co-processor as PRNG on SLE 78 @ 50 MHz.

[g] Kyber768 from *mupq* project [va18] on ARM Cortex-M4F (STM32).

[h] Constant time ephemeral NewHope key exchange (n=1024) [ADPS16] from [AJS16] on ARM Cortex-M0 (STM32) @ 48 MHz.

[i] Constant time CPA-secure RLWE-encryption [LP11] (RLWEenc-IIa with $n = 512$) from [LPO+17b] on 8-bit ATxmega128A1 @ 32 MHz.

[j] CCA-secure RLWE-encryption [LP11] ($n = 1024$) from [OSPG18] on ARM Cortex-M4F (STM32) @ 168 MHz. With first order masking decryption is 25,334,493 cycles.

[k] Saber [DKRV17] from [KMRV18] on ARM Cortex-M4F (STM32F4) @ 168 MHz. Parameters provide 180-bit of quantum-security.

[l] CPA-secure QC-MPDC public-key encryption [MTSB12] from [vMOG15] on ARM Cortex-M4F (STM32F407) @ 168 MHz. Parameters provide 80-bit pre-quantum security level.

[m] Elliptic curve Diffie-Hellman using Curve25519 [Ber06] from [DHH+15] on 16-bit MSP430X @ 16 MHz. For simplification we report the cost of one point multiplication (PM) in Gen, two PMs in Enc and one PM in Dec.

[n] Elliptic curve Diffie-Hellman using Curve25519 [Ber06] from [DHH+15] on ARM Cortex @ 48 MHz. Reporting as in [l].

cycle multiplier. Here it is also worth to consider that on an ARM processor Snort, Sneeze, and software-based big integer addition are also expected to be significantly faster due to the more efficient instruction set and larger word size, while the CPU and the co-processor could still execute in parallel.

From the algorithmic side, in the case of the KS1, $\omega = 64$ implementation of Kyber we currently require $\ell \geq 25$ bits of precision, and hence opted for using 32 bits. By using the considerations made in Section 4 about swapping ω for n in the formula for computing ℓ, we could get down to $\ell \geq 23$, making it possible to save some memory at the cost of a more complex unpacking.

In a more general direction it appears interesting to investigate whether a performance advantage can be obtained with schemes specifically designed with the constraints of the big integer multiplier in mind such as ThreeBears [Ham17] or Mersenne-75683917 [AJPS17]. However, we note that these schemes use integer sizes too large for direct handling with our co-processor. In contrast, MLWE-based schemes immediately allow for a piece-wise approach. Thus, another interesting target for implementation could be an MLWE-based scheme that is parameterised with a power-of-two modulus q, e.g. SABER [DKRV17] which permits to efficiently implement the strategy from Equation (1). For example, a viable choice could be a prime-cyclotomic ring for $n = 167$ with 2^{13} such that each ring element fits directly into a co-processor register. Another approach would be a Kyber instantiation with a smaller prime modulus q, as we do not have to choose q in a way that a fast NTT exists. Moreover, our results naturally transfer over to the Dilithium signature scheme and an implementation on the SLE 78 is a natural next step. However, parameters have to be adapted for Dilithium, as it uses a larger modulus $q = 8380417$. Another interesting question is whether it is possible to efficiently use RSA/ECC co-processors to implement the NTT by treating the big integer multiplier as a vector processor using smart packing of coefficients or a variant of Kronecker substitution.

References

[AAB+17] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe Thomas Pöppelmann, and Douglas Stebila. Newhope. Technical report, National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.

[AD17] Martin R. Albrecht and Amit Deo. Large modulus ring-LWE ≥ module-LWE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 267–296. Springer, Heidelberg, December 2017.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16*, pages 327–343. USENIX Association, 2016.

[AJPS17] Divesh Aggarwal, Antoine Joux, Anupam Prakash, and Mikos Santha. Mersenne-756839. Technical report, National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. Newhope on ARM cortex-m. In *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016*, pages 332–349, 2016.

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 354–384. Springer, Heidelberg, April / May 2018.

[BDEZ12] Razvan Barbulescu, Jérémie Detrey, Nicolas Estibals, and Paul Zimmermann. Finding optimal formulae for bilinear maps. In Ferruh Özbudak and Francisco Rodríguez-Henríquez, editors, *Arithmetic of Finite Fields*, volume 7369 of *Lecture Notes in Computer Science*, pages 168–186. Springer Berlin Heidelberg, 2012.

[BDK+17] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS - kyber: a cca-secure module-lattice-based KEM. *IACR Cryptology ePrint Archive*, 2017:634, 2017. to appear in IEEE European Symposium on Security and Privacy 2018, EuroS&P 2018.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006.

[BLP+13] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 575–584. ACM Press, June 2013.

[BSJ15] Ahmad Boorghany, Siavash Bayat Sarmadi, and Rasool Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. *ACM Trans. Embed. Comput. Syst.*, 14(3):42:1–42:25, April 2015.

[CCD+15] Matthew Campagna, Lidong Chen, Dr Özgür Dagdelen, Jintai Ding, Jennifer K. Fernick, Nicolas Gisin, Donald Hayford, Thomas Jennewein, Norbert Lütkenhaus, Michele Mosca, Brian Neill, Mark Pecen, Ray Perlner, Grégoire Ribordy, John M. Schanck, Dr Douglas Stebila, Nino Walenta, William Whyte, and Dr Zhenfei Zhang. ETSI whitepaper: Quantum safe cryptography and security. `http://www.etsi.org/images/files/ETSIWhitePapers/QuantumSafeWhitepaper.pdf`, June 2015.

[CDW17] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-SVP. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 324–348. Springer, Heidelberg, April / May 2017.

[CHT12] Peter Czypek, Stefan Heyse, and Enrico Thomae. Efficient implementations of MQPKS on constrained devices. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 374–389. Springer, Heidelberg, September 2012.

[Chu17] Gu Chunsheng. Integer version of ring-LWE and its applications. Cryptology ePrint Archive, Report 2017/641, 2017. `http://eprint.iacr.org/2017/641`.

[CLT13] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 476–493. Springer, Heidelberg, August 2013.

[CÖ10] Murat Cenk and Ferruh Özbudak. On multiplication in finite fields. *Journal of Complexity*, 26(2):172–186, 2010.

[CS15] Jung Hee Cheon and Damien Stehlé. Fully homomophic encryption over the integers revisited. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 513–536. Springer, Heidelberg, April 2015.

[dCRVV15] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of ring-LWE encryption. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015*, pages 339–344, 2015.

[DHH+15] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Des. Codes Cryptography*, 77(2-3):493–514, 2015.

[DKRV17] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber. Technical report, National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.

[DRHM17] Petr Dzurenda, Sara Ricci, Jan Hajny, and Lukas Malina. Performance analysis and comparison of different elliptic curves on smart cards. In *International Conference on Privacy, Security and Trust (PST)*, 2017. to appear, see `https://www.ucalgary.ca/pst2017/files/pst2017/paper-39.pdf`.

[FH07] Haining Fan and M. Anwar Hasan. Comments on "five, six, and seven-term karatsuba-like formulae". *IEEE Trans. Computers*, 56(5):716–717, 2007.

[FWA+13] Dirk Feldhusen, Guntram Wicke, Arnold Abromeit, Lex Schoonen, and Zertifizierungsstelle BSI˙Minimum requirements for evaluating side-channel attack resistance of rsa, dsa and diffie-hellman key exchange implementations. Technical report, German Federal Office for Information Security - BSI, 1 2013. See `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_BSI_guidelines_SCA_RSA_V1_0_e_pdf.pdf?__blob=publicationFile&v=1`.

[GK15] Peter Günther and Volker Krummel. Implementing cryptographic pairings on accumulator based smart card architectures. In *Mathematical Aspects of Computer and Information Sciences - 6th International Conference, MACIS 2015, Berlin, Germany, November 11-13, 2015, Revised Selected Papers*, pages 151–165, 2015.

[GS02] Antonio Valverde Garcia and Jean-Pierre Seifert. On the implementation of the Advanced Encryption Standard on a public-key crypto-coprocessor. In *Fifth Smart Card Research and Advanced Application Conference, CARDIS '02*, pages 135–146, 2002.

[Ham17] Mike Hamburg. Three bears. Technical report, National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.

[Har09] David Harvey. Faster polynomial multiplication via multipoint kronecker substitution. *J. Symb. Comput.*, 44(10):1502–1510, 2009.

[HBB13] Andreas Hülsing, Christoph Busold, and Johannes Buchmann. Forward secure signatures on smart cards. In Lars R. Knudsen and Huapeng Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 66–80. Springer, Heidelberg, August 2013.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Heidelberg, November 2017.

[HRS16] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS - computing a 41 KB signature in 16 KB of RAM. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part I*, volume 9614 of *LNCS*, pages 446–470. Springer, Heidelberg, March 2016.

[KMRV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM CCA-secure module lattice-based key encapsulation on ARM. Cryptology ePrint Archive, Report 2018/682, 2018. `https://eprint.iacr.org/2018/682`.

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, August 2010.

[Kva16] Rudolf Kvašňovský. The detailed performance analysis of JavaCard cryptographic smartcards, 2016.

[Lan16] Adam Langley. CECPQ1 results, Nov 2016.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, Heidelberg, February 2011.

[LPO+17a] Zhe Liu, Thomas Pöppelmann, Tobias Oder, Hwajeong Seo, Sujoy Sinha Roy, Tim Güneysu, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. High-performance ideal lattice-based cryptography on 8-bit AVR microcontrollers. *ACM Trans. Embedded Comput. Syst.*, 16(4):117:1–117:24, 2017.

[LPO+17b] Zhe Liu, Thomas Pöppelmann, Tobias Oder, Hwajeong Seo, Sujoy Sinha Roy, Tim Güneysu, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. High-performance ideal lattice-based cryptography on 8-bit AVR microcontrollers. *ACM Trans. Embedded Comput. Syst.*, 16(4):117:1–117:24, 2017.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, Jun 2015.

[Mon05] Peter L. Montgomery. Five, six, and seven-term karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.

[MTSB12] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. MDPC-McEliece: New McEliece variants from moderate density parity-check codes. Cryptology ePrint Archive, Report 2012/409, 2012. http://eprint.iacr.org/2012/409.

[Nat16] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the Post-Quantum Cryptography standardization process. http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf, December 2016.

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

[OPG14] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. Beyond ECDSA and RSA: lattice-based digital signatures on constrained devices. In *The 51st Annual Design Automation Conference 2014, DAC '14*, pages 110:1–110:6, 2014.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure masked Ring-LWE implementations. *IACR TCHES*, 2018(1):142–174, 2018. https://tches.iacr.org/index.php/TCHES/article/view/836.

[RdCR+16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively homomorphic ring-lwe masking. In *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 233–244, 2016.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):1–40, Sep 2009.

[S+17] William Stein et al. *Sage Mathematics Software Version 8.0*. The Sage Development Team, 2017. http://www.sagemath.org.

[SAB+17] Peter Schwabe, Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehle. Crystals-kyber. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[SBPV07] Kazuo Sakiyama, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. HW/SW co-design for public-key cryptosystems on the 8051 micro-controller. *Computers & Electrical Engineering*, 33(5-6):324–332, 2007.

[Sch77] Arnold Schönhage. Schnelle multiplikation von polynomen über körpern der charakteristik 2. *Acta Informatica*, 7(4):395–398, Dec 1977.

[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.

[SNS+16] Petr Svenda, Matús Nemec, Peter Sekan, Rudolf Kvasnovský, David Formánek, David Komárek, and Vashek Matyás. The million-key question - investigating the origins of RSA public keys. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 893–910, 2016.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 617–635. Springer, Heidelberg, December 2009.

[va18] various authors. Post-quantum crypto library for the ARM Cortex-M4. Website, 2018. accessed April 2018, see `https://github.com/mupq/pqm4`.

[vMOG15] Ingo von Maurich, Tobias Oder, and Tim Güneysu. Implementing QC-MDPC McEliece encryption. *ACM Trans. Embedded Comput. Syst.*, 14(3):44:1–44:27, 2015.

[VZGG13] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2013.

[Wen13] Erich Wenger. A lightweight atmega-based application-specific instruction-set processor for elliptic curve cryptography. In *Lightweight Cryptography for Security and Privacy - Second International Workshop, LightSec 2013*, pages 1–15, 2013.

A Cyclotomic gadgets

In this Appendix we prove Corollaries 1 and 2.

of Corollary 1. We need to verify that

$$f(2^\ell) > 2^{n\ell} - 1 \tag{5}$$

and that

$$d_i \in \{-\delta, \ldots, \delta\} \tag{6}$$

Condition 5 holds since $f(2^\ell) = 2^{n\ell} + 1$. Condition 6 follows by explicitly evaluating

$$d(x) = \sum_{i=0}^{n-1} d_i\, x^i \coloneqq a(x) \cdot b(x) + c(x) \pmod{x^n + 1}$$

which implies that

$$d_i = \sum_{[j+k]_n = i} -1^{[\![j+k \geq n]\!]}\, a_j\, b_k + c_i$$

and hence

$$\max_{\substack{\{a_j\}_j, \{b_k\}_k, \\ \{c_m\}_m}} |d_i| \leq n\alpha\beta + \gamma =: \delta.$$

□

Lemma 2. *Let $a = \sum_{i=0}^{n-1} a_i\, x^i$, $b = \sum_{i=0}^{n-1} b_i\, x^i$ with a_i, $b_i \in \mathbb{Z}$, and let $f = \sum_{i=0}^{n} x^i$. Let $c_i \coloneqq \sum_{j+k=i} a_j b_k$ such that $c \coloneqq \sum_{i=0}^{2n-2} c_i\, x^i = a \cdot b$ and let $d \coloneqq \sum_{i=0}^{n-1} d_i\, x^i \equiv c \pmod f$. Then*

$$d = \sum_{i=0}^{n-3} (c_i - c_n + c_{i+n+1})\, x^i + (c_{n-2} - c_n)\, x^{n-2} + (c_{n-1} - c_n)\, x^{n-1}$$

and each d_i is a sum of at most $2n-1$ terms of the form $a_j b_k$.

Proof. Let $f^{(m)} \coloneqq \sum_{i=0}^{m} x^i$ (it follows that $f \equiv f^{(n)}$). Since a and b have degree $< n$, we know that we need to reduce modulo f only the powers x^{i+n} for $i = 0, \ldots, n-2$ of c. For $i \geq 1$ we have

$$\begin{aligned} x^{i+n} &\equiv x^i(x^n - f^{(n)}(x)) \pmod f \\ &= -x^i(f^{(n-1)}) \\ &= -x^{i-1}(xf^{(n-1)}) \\ &= -x^{i-1}(f^{(n)} - 1) \\ &\equiv x^{i-1} \pmod f, \end{aligned}$$

while for $i = 0$, $x^n \equiv -f^{(n-1)} \pmod f$. Hence, we can write

$$\begin{aligned} c &= \sum_{i=0}^{2n-2} c_i\, x^i \\ &= \sum_{i=0}^{n-1} c_i\, x^i + c_n\, x^n + \sum_{i=0}^{n-3} c_{n+i+1}\, x^{n+i+1} \\ &\equiv \sum_{i=0}^{n-1} c_i\, x^i - c_n \sum_{i=0}^{n-1} x^i + \sum_{i=0}^{n-3} c_{n+i+1}\, x^i \pmod f \\ &\equiv \sum_{i=0}^{n-3} (c_i - c_n + c_{n+i+1})\, x^i + (c_{n-2} - c_n)\, x^{n-2} + (c_{n-1} - c_n) x^{n-1} \pmod f \end{aligned}$$

where each c_i is a sum of $\#\{(j,k) \in [0,n-1]^2 \cap \mathbb{Z}^2 \mid j+k=i\} = n - |i-n+1|$ terms $a_j b_k$.

Hence,

$$d = \sum_{i=0}^{n-3} (c_i - c_n + c_{n+i+1})\, x^i + (c_{n-2} - c_n)\, x^{n-2} + (c_{n-1} - c_n) x^{n-1}$$

where by explicit computation d_{n-1} is a sum of $2n-1$ terms $a_j b_k$, d_{n-2} is a sum of $2n-2$ such terms and, for $i \leq n-3$, d_i has $3n - |i-n+1| - |n-n+1| - |n+i+1-n+1| = 2n-2$ such terms. □

of Corollary 2. We need to verify that

$$f(2^\ell) > 2^{n\ell} - 1 \tag{7}$$

and that

$$d_i \in \{-\delta, \ldots, \delta\} \tag{8}$$

Condition 7 holds since $f(2^\ell) = 2^{n\ell} + 2^{(n-1)\ell} + \cdots + 1$. Condition 8 follows by explicitly evaluating

$$d = \sum_{i=0}^{n-1} d_i\, x^i \equiv a \cdot b + c \pmod{f}$$

using Lemma 2, which implies that

$$\max_{\substack{\{a_j\}_j, \{b_k\}_k, \\ \{c_m\}_m}} |d_i| \leq (2n-1)\alpha\beta + \gamma =: \delta.$$

□

B Proof of Concept

Our high-level proof-of-concept implementation is written in SageMath [S+17].

```
# -*- coding: utf-8 -*-
"""
Kyber using big integer arithmetic - proof-of-concept

.. note :: Run tests as ``sage -t test.py``

"""
from sage.all import parent, ZZ, vector, PolynomialRing, GF
from sage.all import log, ceil, randint, set_random_seed, random_vector, matrix, floor


def BinomialDistribution(eta):
    r = 0
    for i in range(eta):
        r += randint(0, 1) - randint(0, 1)
    return r


def balance(e, q=None):
    """
    Return a representation of `e` with elements balanced between `-q/2` and `q/2`

    :param e: a vector, polynomial or scalar
    :param q: optional modulus, if not present this function tries to recover it from `e`

    :returns: a vector, polynomial or scalar over/in the integers
    """
    try:
        p = parent(e).change_ring(ZZ)
        return p([balance(e_, q=q) for e_ in e])
    except (TypeError, AttributeError):
        if q is None:
            try:
                q = parent(e).order()
            except AttributeError:
                q = parent(e).base_ring().order()
        e = ZZ(e)
        e = e % q
        return ZZ(e-q) if e>q//2 else ZZ(e)
```

```


# Kyber (sort of)

class Kyber:

    n = 256
    q = 7681
    eta = 4
    k = 3
    D = staticmethod(BinomialDistribution)
    f = [1]+[0]*(n-1)+[1]
    ce = n

    @classmethod
    def key_gen(cls, seed=None):
        """Generate a new public/secret key pair

        :param cls: Kyber class, inherit and change constants to change defaults
        :param seed: seed used for random sampling if provided

        .. note :: Resembles Algorithm 1 of the Kyber paper.

        """
        n, q, eta, k, D = cls.n, cls.q, cls.eta, cls.k, cls.D

        if seed is not None:
            set_random_seed(seed)

        R, x = PolynomialRing(ZZ, "x").objgen()
        Rq = PolynomialRing(GF(q), "x")
        f = R(cls.f)

        A = matrix(Rq, k, k, [Rq.random_element(degree=n-1) for _ in range(k*k)])
        s = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        e = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        t = (A*s + e)  % f  # NOTE ignoring compression

        return (A, t), s

    @classmethod
    def enc(cls, pk, m=None, seed=None):
        """IND-CPA encryption sans compression

        :param cls: Kyber class, inherit and change constants to change defaults
        :param pk: public key
        :param m: optional message, otherwise all zero string is encrypted
        :param seed: seed used for random sampling if provided

        .. note :: Resembles Algorithm 2 of the Kyber paper.

        """
        n, q, eta, k, D = cls.n, cls.q, cls.eta, cls.k, cls.D

        if seed is not None:
            set_random_seed(seed)

        A, t = pk
```

```
        R, x = PolynomialRing(ZZ, "x").objgen()
        f = R(cls.f)

        r  = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        e1 = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        e2 = R([(D(eta)) for _ in range(n)])

        if m is None:
            m = (0,)

        u = (r*A + e1) % f  # NOTE ignoring compression
        u.set_immutable()
        v = (r*t + e2 + q//2 * R(list(m))) % f  # NOTE ignoring compression
        return u, v

    @classmethod
    def dec(cls, sk, c, decode=True):
        """IND-CPA decryption

        :param cls: Kyber class, inherit and change constants to change defaults
        :param sk: secret key
        :param c: ciphertext
        :param decode: perform final decoding

        .. note :: Resembles Algorithm 3 of the Kyber paper.

        """
        n, q = cls.n, cls.q

        s = sk
        u, v = c

        R, x = PolynomialRing(ZZ, "x").objgen()
        f = R(cls.f)

        m = (v - s*u) % f
        m = list(m)
        while len(m) < n:
            m.append(0)

        m = balance(vector(m), q)

        if decode:
            return cls.decode(m, q, n)
        else:
            return m

    @staticmethod
    def decode(m, q, n):
        """Decode vector `m` to `\{0,1\}^n` depending on distance to `q/2`

        :param m: a vector of length `\leq n`
        :param q: modulus

        """
        return vector(GF(2), n, [abs(e)>q/ZZ(4) for e in m] + [0 for _ in range(n-len(m))])

    @classmethod
    def encap(cls, pk, seed=None):
        """IND-CCA encapsulation sans compression or extra hash

        :param cls: Kyber class, inherit and change constants to change defaults
        :param pk: public key
        :param seed: seed used for random sampling if provided

        .. note :: Resembles Algorithm 4 of the Kyber paper.

        """
        n = cls.n

        if seed is not None:
            set_random_seed(seed)

        m = random_vector(GF(2), n)
        m.set_immutable()
        set_random_seed(hash(m))  # NOTE: this is obviously not faithful

        K_ = random_vector(GF(2), n)
        K_.set_immutable()
        r = ZZ.random_element(0, 2**n-1)

        c = cls.enc(pk, m, r)

        K = hash((K_, c))  # NOTE: this obviously isn't a cryptographic hash
        return c, K

    @classmethod
    def decap(cls, sk, pk, c):
        """IND-CCA decapsulation

        :param cls: Kyber class, inherit and change constants to change defaults
        :param sk: secret key
        :param pk: public key
        :param c: ciphertext

        .. note :: Resembles Algorithm 5 of the Kyber paper.

        """
        n = cls.n

        m = cls.dec(sk, c)
        m.set_immutable()
        set_random_seed(hash(m))  # NOTE: this is obviously not faithful
```

```
        K_ = random_vector(GF(2), n)
        K_.set_immutable()
        r = ZZ.random_element(0, 2**n-1)

        c_ = cls.enc(pk, m, r)

        if c == c_:
            return hash((K_, c))  # NOTE: this obviously isn't a cryptographic hash
        else:
            return hash(c)  # NOTE ignoring z
```

```
class MiniKyber(Kyber):
    """
    Tiny parameters for testing.
    """
    n = 8
    q = 127
    eta = 1
    k = 1
    f = [1]+[0]*(n-1)+[1]
    ce = n


class Nose:
    """
    Snorting (packing) and sneezing (unpacking).
    """

    @staticmethod
    def snort(g, f, p):
        """
        Convert vector `g` in `\ZZ^n` with coefficients bounded by `p/2` in absolute value to
        integer `\bmodp f(p)`.

        :param g: a vector of length `n`
        :param f: a minpoly
        :param p: base

        :returns: an integer mod `f(p)`
        """
        return g.change_ring(ZZ)(p) % f(p)

    @staticmethod
    def sneeze(G, f, p):
        """Convert integer `G \bmodl f(p)` to vector of integers

        :param G: an integer `\bmodl f(p)`
        :param f: a minpoly
        :param p: base

        """
        assert(G >= 0 and G < f(p))
        n = f.degree()
        c = 0
        r = []
        for i in range(n):
            e = G % p
            G -= e
            e += c
            G = G//p
            c = int(e > p//2)
            e -= c*p
            r.append(e)

        for i in range(n):
            r[i] -= f[i]*(G+c)

        return r[:n]

    @staticmethod
    def proof_sneeze(G, f, p):
        """Convert integer `G \bmod f(p)` to vector of integers

        :param G: an integer `\bmod f(p)`
        :param f: a minpoly
        :param p: base

        """
        assert(G >= 0 and G < f(p))
        n = f.degree()
        r = []
        for i in range(n):
            e = G % p
            G -= e
            G = G//p
            if e > p//2:
                e -= p
                G += 1
            r.append(e)

        for i in range(n):
            r[i] -= f[i]*G

        return r[:n]

    @classmethod
    def prec(cls, scheme):
        """
        Return `\log_2(k ce eta (q-1)/2 + (q-1)/2 + 1) + 1`

        1. eta q/2 is the upper bound on the product in absolute value
```

```
        2. We add ce such products during modular reduction
        3. We add up k such numbers when doing inner products
        4. We add a number of size eta in absolute value
        5. The modular reduction of the integer multiplier might add +/- max_i(|f_i|) to balance the output
        6. One sign bit

        """
        eta, q, k, f, ce = scheme.eta, scheme.q, scheme.k, scheme.f, scheme.ce
        l = log(k*ce*floor(q/ZZ(2))*eta + eta + max([abs(fi) for fi in f]) + 1, 2) + 1
        return l

    @classmethod
    def muladd(cls, scheme, a, b, c, l=None):
        """
        Compute `a \cdot b + c mod f` using big-integer arithmetic

        :param cls: Skipper class
        :param scheme: Scheme class, inherit and change constants to change defaults
        :param a: vector of polynomials in `ZZ_q[x]/(x^n+1)`
        :param b: vector of polynomials in `ZZ_q[x]/(x^n+1)`
        :param c: polynomial in `ZZ_q[x]/(x^n+1)`
        :param l: bits of precision

        """
        R, x = PolynomialRing(ZZ, "x").objgen()
        k, f = scheme.k, R(scheme.f)

        if l is None:
            l = ceil(cls.prec(scheme))

        A = vector(R, k, [cls.snort(a[j], f, 2**l) for j in range(k)])
        B = vector(R, k, [cls.snort(b[j], f, 2**l) for j in range(k)])
        C = cls.snort(c, f, 2**l)
        F = f(2**l)
        D = (A*B + C) % F
        d = cls.sneeze(D % F, f, 2**l)
        return R(d)
```

```
# Skipper

class Skipper4(Nose):
    """
    Kyber using big integer arithmetic

    IND-CPA Decryption in 30 multiplication of (64 \cdot 25 =) 1600-bit integers.

    - Degree 4 polynomial multiplication
    - Standard signed Kronecker substitution to pack 64 coefficients into one integer.

    """

    @staticmethod
    def ff(v, offset, start=0):
        """Fast-forward through vector `v` in ``offset`` sized steps starting at ``start``

        :param v: vector
        :param offset: increment in each step
        :param start: start offset

        """
        p = parent(v)
        return p(list(v)[start::offset])

    @classmethod  # TODO: n vs 2n expansion factor # TODO: tempted of getting rid of this
    def prec(cls, kyber):
        """
        Return `\log_2(k n eta (q-1)/2 + (q-1)/2 + 1) + 1`

        1. eta q/2 is the upper bound on the product in absolute value
        2. We add n such products during modular reduction          # TODO: n vs 2n
        3. We add up k such numbers when doing inner products
        4. We add a number of size eta in absolute value
        5. The modular reduction of the integer multiplier might add +/- max_i(|f_i|) to balance the output
        6. One sign bit

        """
        n, eta, q, k, f = kyber.n, kyber.eta, kyber.q, kyber.k, kyber.f
        l = log(k*n*floor(q/ZZ(2))*eta + eta + max([abs(fi) for fi in f]) + 1, 2) + 1
        return l

    @classmethod
    def muladd(cls, kyber, a, b, c, l=None):
        """
        Compute `a \cdot b + c` using big-integer arithmetic

        :param cls: Skipper class
        :param kyber: Kyber class, inherit and change constants to change defaults
        :param a: vector of polynomials in `ZZ_q[x]/(x^n+1)`
        :param b: vector of polynomials in `ZZ_q[x]/(x^n+1)`
        :param c: polynomial in `ZZ_q[x]/(x^n+1)`
        :param l: bits of precision

        """
        m, k = 4, kyber.k
        w = kyber.n//m
        R, x = PolynomialRing(ZZ, "x").objgen()
        f = R([1]+[0]*(w-1)+[1])

        if l is None:
            # Could try passing degree w, but would require more careful
            # sneezing
            l = ceil(cls.prec(kyber))
```

```
        R = PolynomialRing(ZZ, "x")
        x = R.gen()

        A = vector(R, k, [sum(cls.snort(cls.ff(a[j], m, i), f, 2**l) * x**i
                              for i in range(m))
                          for j in range(k)])

        C = sum(cls.snort(cls.ff(c, m, i), f, 2**l) * x**i for i in range(m))

        B = vector(R, k, [sum(cls.snort(cls.ff(b[j], m, i), f, 2**l) * x**i
                              for i in range(m))
                          for j in range(k)])

        F = f(2**l)

        # MUL: k * 3^2 (Karatsuba for length 4)
        # % F here is applied to the 64-coeff-packs.
        # k comes from len(A) = len(B) = k, each constrains
        # a deg 4 poly needing (recursive) karatsuba => 9
        W = (A*B + C) % F

        # MUL: 3
        # specific trick for how we multiply degree n = 256 polys
        # the coefficients from above need readjustment
        # here doing 2**l *  is basically doing y * !!! and if this wraps around
        # it takes care of the - in front
        W = sum((W[0+i] + (2**l * W[m+i] % F))*x**i for i in range(m-1)) + W[m-1]*x**(m-1)

        D = [cls.sneeze(W[i] % F, f, 2**l) for i in range(m)]

        d = []
        for j in range(w):
            for i in range(m):
                d.append(D[i][j])

        return R(d)

    @classmethod
    def enc(cls, kyber, pk, m=None, seed=None, l=None):
        """IND-CPA encryption sans compression

        :param kyber: Kyber class, inherit and change constants to change defaults
        :param pk: public key
        :param m: optional message, otherwise all zero string is encrypted
        :param seed: seed used for random sampling if provided

        """
        n, q, eta, k, D = kyber.n, kyber.q, kyber.eta, kyber.k, kyber.D

        if seed is not None:
            set_random_seed(seed)

        A, t = pk

        R = PolynomialRing(ZZ, "x")

        r  = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        e1 = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        e2 = R([(D(eta)) for _ in range(n)])

        if m is None:
            m = (0,)

        u = vector(R, [cls.muladd(kyber, r, A.column(i), e1[i], l=l) for i in range(k)])
        u.set_immutable()
        v = cls.muladd(kyber, r, t, e2 + q//2 * R(list(m)), l=l)
        return u, v

    @classmethod
    def dec(cls, kyber, sk, c, l=None, decode=True):
        """Decryption.

        :param kyber: Kyber class, inherit and change constants to change defaults
        :param sk: secret key
        :param c: ciphertext
        :param l: bits of precision
        :param decode: perform final decoding

        """
        n, q = kyber.n, kyber.q

        u, v = c
        s = sk

        m = -cls.muladd(kyber, s, u, -v, l=l)
        m = balance(vector(m), q)
        if decode:
            return kyber.decode(m, q, n)
        else:
            return m


class Skipper2Negated(Skipper4):
    """
    Kyber using big integer arithmetic

    IND-CPA Kyber Decryption in 20 multiplications of (128 \cdot 13 =) 1664-bit integers.

    - Degree 2 polynomial multiplication
    - Negated, signed Kronecker substitution to pack 128 coefficients into one integer.
    """

    @classmethod
```

```
    def prec(cls, kyber):
        """
        Return half the precision required by ``Skipper4``.

        :param kyber: Kyber class, inherit and change constants to change defaults

        """
        return Skipper4.prec(kyber)/ZZ(2)

    @classmethod
    def muladd(cls, kyber, a, b, c, l=None):
        """
        Compute `a \cdot b + c` using big-integer arithmetic

        :param cls: Skipper class
        :param kyber: Kyber class, inherit and change constants to change defaults
        :param a: vector of polynomials in `ZZ_q[x]/(x^n+1)`
        :param b: vector of polynomials in `ZZ_q[x]/(x^n+1)`
        :param c: polynomial in `ZZ_q[x]/(x^n+1)`
        :param l: bits of precision

        """
        m, k = 2, kyber.k
        w = kyber.n//m
        R, x = PolynomialRing(ZZ, "x").objgen()
        f = R([1]+[0]*(w-1)+[1])
        g = R([1]+[0]*(w//2-1)+[1])

        if l is None:
            l = ceil(cls.prec(kyber))

        R = PolynomialRing(ZZ, "x")
        x = R.gen()

        Ap = vector(R, k, [sum(cls.snort(cls.ff(a[j], m, i), f, 2**l) * x**i for i in range(m))
                           for j in range(k)])
        An = vector(R, k, [sum(cls.snort(cls.ff(a[j], m, i), f, -2**l) * x**i for i in range(m))
                           for j in range(k)])

        Cp = sum(cls.snort(cls.ff(c, m, i), f,  2**l) * x**i for i in range(m))
        Cn = sum(cls.snort(cls.ff(c, m, i), f, -2**l) * x**i for i in range(m))

        Bp = vector(R, k, [sum(cls.snort(cls.ff(b[j], m, i), f, 2**l) * x**i for i in range(m))
                           for j in range(k)])
        Bn = vector(R, k, [sum(cls.snort(cls.ff(b[j], m, i), f, -2**l) * x**i for i in range(m))
                           for j in range(k)])

        F = 2**(w * l) + 1

        # MUL: 2 * k * 3
        Wp = (Ap*Bp + Cp) % F
        Wn = (An*Bn + Cn) % F

        We = (Wp+Wn) % F
        Wo = (Wp-Wn) % F

        Wo, We = (sum((Wo[0+i] + (2**l * We[m+i] % F))*x**i for i in range(m-1)) + Wo[m-1]*x**(m-1)) % F, \
                 (sum((We[0+i] + (2**l * Wo[m+i] % F))*x**i for i in range(m-1)) + We[m-1]*x**(m-1)) % F

        _inverse_of_2_mod_F = F - 2**(w*l-1)
        _inverse_of_2_to_the_l_plus_1_mod_F = F - 2**(w*l-1-1)
        We = (We * _inverse_of_2_mod_F) % F
        Wo = (Wo * _inverse_of_2_to_the_l_plus_1_mod_F) % F

        D =  [cls.sneeze(We[i] % F, g, 2**(2*l)) for i in range(m)]
        D += [cls.sneeze(Wo[i] % F, g, 2**(2*l)) for i in range(m)]

        d = []

        for j in range(w//2):
            for i in range(2*m):
                d.append(D[i][j])

        return R(d)
```