CRYS TALS – Kyber: a CCA-secure module-lattice-based KEM

Joppe Bos
NXP
joppe.bos@nxp.com

Léo Lucas
CWI
ducas@ewi.nl

Eike Kiltz
RUB
eike.kiltz@rub.de

Tancrède Lepoint
SRI International
tancrède.lepoint@sri.com

Vadim Lyubashevsky
IBM Research - Zurich
vad@zurich.ibm.com

John M. Schanck
University of Waterloo / IQC
jschanck@uwaterloo.ca

Peter Schwabe
Radboud University
peter@cryptojedi.org

Damien Stehlé
ENS de Lyon
damien.stehle@ens-lyon.fr

ABSTRACT
Recent advances in quantum computing and the announcement by the National Institute of Standards and Technology (NIST) to define new standards for digital-signature, encryption, and key-establishment protocols increased interest in post-quantum cryptographic schemes.

This paper introduces Kyber (part of the CRYS TALS – Crypto graphic Suite for Algebraic Lattices – package that will be submitted to the NIST call for post-quantum standards), a portfolio of post-quantum cryptographic primitives built around a key-encapsulation mechanism (KEM), based on hardness assumptions over module lattices. We first introduce a CPA-secure public key encryption scheme, apply a variant of the Fujisaki–Okamoto transform to create a CCA-secure KEM, and eventually construct, in a black-box manner, CCA-secure encryption, key exchange, and authenticated-key-exchange schemes. The security of our primitives is based on the hardness of Module-LWE in the classical and quantum random oracle models, and our concrete parameters conservatively target more than 128 bits of post-quantum security.

We implemented and benchmarked the CCA-secure KEM and key exchange protocols against the ones that are based on LWE and Ring-LWE: we conclude that our schemes are not only as efficient but also feature more flexibility and security advantages over the latter schemes.

KEYWORDS
Lattice cryptography, Key encapsulation mechanism, Implementation, Module lattices, (Authenticated) key exchange, CCA security.

1 INTRODUCTION
There has been an increased interest in post-quantum cryptographic schemes triggered by recent advances in quantum computing [34] and the announcement by the National Institute of Standards and Technology (NIST) to define new standards for digital-signature, encryption, and key-establishment protocols [26]. Constructions based on the hardness of lattice problems are considered to be one of the leading candidates to replace the currently used schemes based on the believed hardness of the traditional number theoretic problems such as integer factorization and discrete logarithms.

Lattice cryptography initially gained a lot of interest in the theoretical community due to the fact that the designs for cryptographic constructions were accompanied by security proofs based on worst-case instances of lattice problems. The first lattice-based encryption scheme was proposed by Ajtai and Dwork [1]. This scheme was later simplified and improved upon by Regev in [65, 66]. One of the major achievements of Regev’s work was the introduction of an intermediate problem – the Learning With Errors (LWE) Problem – which was relatively simple to use in cryptographic constructions and asymptotically at least as hard as some standard worst-case lattice problems [23, 59].

The LWE assumption states that it is hard to distinguish from uniform the distribution (A, As + e), where A is a uniformly-random matrix in $Z_q^{m \times n}$, s is a uniformly-random vector in $Z_q^n$, and e is a vector with random “small” coefficients chosen from some distribution. Applebaum et al. [5] showed that the secret s in the LWE problem does not need to be chosen uniformly at random: the problem remains hard if s is chosen from the same narrow distribution as the errors e. Based on the idea from the NTRU cryp tosystem [43] of working with elements over polynomial rings rather than over the integers, and following a series of works on this topic [54, 56, 61, 68], Lyubashevsky et al. [55] showed that it is also hard to distinguish a variant of the LWE distribution from the uniform one over certain polynomial rings, thus defining the Ring-LWE assumption.

The combination of all of the above results finally led to the cryp tosystem in Section 3.1. Setting the parameter k to 1 and defining $R_q = Z_q[X]/(X^n + 1)$ makes the scheme a Ring-LWE cryp tosystem as originally defined in [55], whereas setting the ring $R_q$ to $Z_q$, makes the scheme an LWE-based one.2 If one sets the ring $R_q$ to some polynomial ring of dimension greater than 1 and sets k > 1, then the scheme is based on the hardness of the Module-LWE problem [22, 51]. The number of bits that can be transmitted is related to the dimension of the ring, thus using a ring $R_q$ of larger degree n allows one to transmit more bits, and this is the main reason that Ring-LWE encryption is more efficient than LWE encryption. On the other hand, having a smaller k implies more algebraic structure, making the scheme potentially susceptible to more avenues.

---

1 It should be noted that this cryptoscheme design, as well as the result from [5] applied to the Learning Parity with Noise (LPN) problem, was already present much earlier in the work of Alekhnovich [3] in which he constructed a cryptosystem based on the hardness of the LPN problem. The LWE problem is a generalization of LPN and results in more efficient cryptosystems.

2 The original cryptosystems did not include the “bit-dropping” Compress functions in key generation and encryption, but this idea was considered folklore (see [65, Sec. 2.3] for some references).
of attack. Nevertheless, at this point in time, it is unknown how to exploit the algebraic structure of Ring-LWE and concrete parameters are chosen according to the corresponding LWE problem of dimension $k \cdot n$.

This cryptosystem design was also applied to build a CPA-secure KEM by Ding et al. [35] and Peikert [60]. The main difference between this KEM and the encryption scheme is how in the parameter $v$ is defined in line 6 of the encryption algorithm (Algorithm 2). The advantage of the constructions in [35, 60] is that if one would like to construct a CPA-secure KEM transmitting a $b$-bit key, then the ciphertext is $b$ bits shorter, which is about a $3\%$ savings for typical parameters. If one wishes to construct a CCA-secure KEM, however, this advantage disappears since typical transformations from CPA-secure KEMs to CCA-secure ones implicitly go through a CPA-secure encryption scheme, which will result in adding $b$ bits to the KEM. Since in this paper we are only concerned with CCA-secure constructions, we find it simpler to start directly from the CPA-secure encryption scheme design in Section 3.

The above designs based on Ring-LWE have resulted in many recent concrete proposals accompanied by practical implementations. The instantiation presented [19] is based on Ring-LWE and was subsequently improved in [4, 52], which resulted in an experiment by Google where they used this key-exchange protocol in their Chrome Canary browser from July to November 2016 [21, 49]. Although the Ring-LWE problem results in very practical key-sizes and protocol communication, the additional algebraic structure might inspire less confidence in the underlying security. This was the motivation to study a very similar practical instantiation of a key-exchange protocol but based on LWE in [18], or to propose an efficient implementation of a CCA-secure KEM over a different ring [12].

### 1.1 Our contribution

Our main contribution is a highly-optimized instantiation of a CCA-secure KEM called Kyber, which is based on the hardness of Module-LWE. More precisely, we instantiate a CPA-secure PKE scheme Kyber.CPA in Section 3, then apply a variant of the Fujisaki-Okamoto transform to create a CCA-secure KEM Kyber in Section 4. The security reduction from the hardness of Module-LWE is tight in the random-oracle model, but non-tight is the quantum-random-oracle model [44]. From a CCA-secure KEM, one can construct, in a black-box manner, CCA-secure encryption (Kyber.Hybrid), key exchange (Kyber.KE), and authenticated-key-exchange (Kyber.AKE) schemes. Our resulting schemes are as efficient as ones that are based on Ring-LWE, but have additional flexibility and security advantages.

**Flexibility.** One of the most expensive operations in lattice-based schemes over rings is polynomial multiplication. If a scheme is based on the Ring-LWE assumption (i.e., with $k = 1$ in Algorithm 2), then if one wants to vary the security parameter related to the scheme, one would need to change the ring $R_q$ and re-implement all the ring operations. With our design, we only work over the ring $R_q = \mathbb{Z}_{7681}[X]/(X^{256} + 1)$, there is only one ring over which operations need to be optimized. Increasing and decreasing the security of the scheme can then be done simply by changing the dimension $k$ of the matrix. Our proposed conservative parameters, which we believe have very generous margins for 128-bit post-quantum security, use $k = 3$. This is the scheme we recommend using for long-term security. But if one only needs short-term security, we believe that today (and probably for the near future) one can safely use $k = 2$ for which we conservatively estimate 102-bit post quantum security. This latter parameter set will reduce the communication size of the key exchange by around $33\%$ and considerably speed up the scheme. The main building blocks of the two schemes are exactly the same, and any optimized software/hardware used for efficient multiplication in $R_q$ can be re-used.

**Security.** There have been recent attacks exploiting the algebraic structure of cyclotomic ideal lattices [15, 24, 31, 32], and others that exploit the presence of dense sub-lattices in NTRU lattices [2, 47]. In these attacks, it appears that the dimension of the module makes a big difference. In particular, the quantum attacks on finding short vectors in ideals currently do not extend to Ring-LWE [15, 24, 31, 32]. The obstacle seems to be that solutions to the shortest vector problem in ideal lattices are ring elements, whereas solutions to the Ring-LWE problem are elements in a module of dimension 2. In that respect, solutions to Module-LWE are in a module of dimension $k + 1$. Similarly, the larger module dimension also decreases the relative dimension of the dense sub-lattice, making the attack of [47] inapplicable. Based on the recent cryptanalytic progress, it therefore seems that practical attacks are less likely to appear against Module-LWE than against Ring-LWE or NTRU.

**High performance.** As we previously mentioned, the main reason that Ring-LWE is preferred to LWE in practical applications is because it allows for a larger message to be transmitted in the same amount of communication. We show that the flexibility and security improvements by moving from Ring-LWE to Module-LWE come at almost no cost. In particular, since public-key protocols only need to transmit 256 bits of information, it is unnecessary to work with rings that are greater than dimension 256 in order to be able to transmit one bit per coefficient of a ring element. Thus the key and message sizes of our protocols versus those based on Ring-LWE are not affected.

The one part where using a $k > 1$ is less efficient than $k = 1$ is when dealing with the $k \times k$ random matrix $A$. If one uses $k = 1$ and a ring of dimension $n$, then the representation of $A$ is $k^2 n = n$ elements in $\mathbb{Z}_q$. On the other hand, if one uses $k = 3$ and a ring of dimension $n/3$ (thus keeping the lattice-reduction security the same), then $A$ requires $k^2 n = 3n$ elements in $\mathbb{Z}_q$ to represent. Since the matrix $A$ is never stored, but rather expanded from some seed $\rho$ using an XOF, this disadvantage only manifests in the slight increase in the running time used in the expansion. This is to some extent mitigated because the $k^2$ entries of the matrix $A$ can be expanded independently, which enables very efficient vectorization of the XOF computation.

**Take away.** In this paper, we propose and implement a portfolio of post-quantum cryptographic primitives (CPA-secure encryption, CCA-secure KEM, CCA-secure public-key encryption, key...
exchange and authenticated key exchange) based on the hardness of Module-LWE in the classical and quantum random-oracle models. Our schemes are as efficient as the ones based on Ring-LWE, but also feature flexibility and security advantages.

**Availability of software.** We place all software described in this paper into the public domain to maximize reusability of our results. It is available for download on GitHub: https://github.com/pq-crystals/kyber.

## 2 PRELIMINARIES

All our algorithms are probabilistic. If $b$ is a string, then $a \leftarrow A(b)$ denotes the output of algorithm $A$ when run on input $b$; if $A$ is deterministic, then $a$ is fixed and we write $a \equiv A(b)$. We use the notation $b \leftarrow A(b; r)$ to make the randomness $r$ of a probabilistic algorithm $A$ explicit.

### 2.1 Cryptographic definitions

A public-key encryption scheme $\text{PKE} = (\text{KeyGen}, \text{Enc}, \text{Dec})$ is a triple of probabilistic algorithms together with a message space $\mathcal{M}$. The key-generation algorithm $\text{KeyGen}$ returns a pair $(pk, sk)$ consisting of a public key and a secret key. The encryption algorithm $\text{Enc}$ takes a public key $pk$ and a message $m \in \mathcal{M}$ to produce a ciphertext $c$. Finally, the deterministic decryption algorithm $\text{Dec}$ takes a secret key $sk$ and a ciphertext $c$, and outputs either a message $m \in \mathcal{M}$ or a special symbol $\bot$ to indicate rejection. We say that $\text{PKE}$ is $(1 - \delta)$-correct if for all messages $m \in \mathcal{M}$, we have $\Pr[\text{Dec}(sk, \text{Enc}(pk, m)) = m] \geq 1 - \delta$, where the probability is taken over $(pk, sk) \leftarrow \text{KeyGen}()$ and the random coins of $\text{Enc}$.

We recall the standard security notions for public-key encryption of indistinguishability under chosen-ciphertext and chosen-plaintext attacks (IND-CCA and IND-CPA) [64]. The advantage of an adversary $A$ is defined as $\text{Adv}_{\text{PKE}}^{\text{IND-CPA}}(A) =$

$$
\Pr \left[ \text{b' = b : } \begin{cases} (pk, sk) \leftarrow \text{KeyGen}(); \\ (m_0, m_1, s) \leftarrow A_{\text{Dec}}(c); \\ b \leftarrow \{0, 1\}; \\ c^* \leftarrow \text{Enc}(pk, m_b); \\ b' \leftarrow A_{\text{Dec}}(\cdot)(s, c^*) \end{cases} \right] - \frac{1}{2},
$$

where the decryption oracle is defined as $\text{Dec}(\cdot) := \text{Dec}(sk, \cdot)$. We further require that $|m_0| = |m_1|$ and that in the second phase $A$ is not allowed to query $\text{Dec}(\cdot)$ with the challenge ciphertext $c^*$. The advantage $\text{Adv}_{\text{PKE}}^{\text{IND-CCA}}(A)$ of an adversary $A$ is defined as $\text{Adv}_{\text{PKE}}^{\text{IND-CCA}}(A)$, with the modification that $A$ cannot query the decryption oracle.

A key-encapsulation scheme $\text{KEM} = (\text{KeyGen}, \text{Encaps}, \text{Decaps})$ is a triple of probabilistic algorithms together with a key space $\mathcal{K}$. The key-generation algorithm $\text{KeyGen}$ returns a pair $(pk, sk)$ consisting of a public key and a secret key. The encapsulation algorithm $\text{Encaps}$ takes a public key $pk$ to produce a ciphertext $c$ and a key $K \in \mathcal{K}$. Finally, the deterministic decapsulation algorithm $\text{Decaps}$ takes a secret key $sk$ and a ciphertext $c$, and outputs either a key $K \in \mathcal{K}$ or a special symbol $\bot$ to indicate rejection. We say that $\text{KEM}$ is $(1 - \delta)$-correct if $\Pr[\text{Decaps}(sk, c) = K : (c, K) \leftarrow \text{Encaps}(pk)] \geq 1 - \delta$, where the probability is taken over $(pk, sk) \leftarrow \text{KeyGen}()$ and the random coins of $\text{Encaps}$.

We recall the standard security notion for key encapsulation of indistinguishability under chosen-ciphertext attack. The advantage of an adversary $A$ is defined as $\text{Adv}_{\text{KEM}}^{\text{CCA}}(A) =$

$$
\Pr \left[ \text{b' = b : } \begin{cases} (pk, sk) \leftarrow \text{KeyGen}(); \\ b \leftarrow \{0, 1\}; \\ (c', K_0^*) \leftarrow \text{Encaps}(pk); K_1^* \leftarrow \mathcal{K}; \\ b' \leftarrow A_{\text{Decaps}}(\cdot)(pk, c', K_0^*) \end{cases} \right] - \frac{1}{2},
$$

where the $\text{Decaps}$ oracle is defined as $\text{Decaps}(\cdot) \equiv \text{Decaps}(sk, \cdot)$. We further require that $A$ is not allowed to query $\text{Decaps}(\cdot)$ with the challenge ciphertext $c'$.

In the random oracle model [9], the adversary $A$ is additionally given access to a random oracle that it can query up to $q_H$ times. If the adversary has access to a quantum computer, it is realistic to model its access to all "offline primitives" (such as hash functions) in a quantum setting. Concretely, in the quantum random oracle model [17] the adversary has access to a quantum random oracle (also called quantum accessible random oracle) that can be queried up to $q_H$ times on arbitrary quantum superpositions of input strings.

### 2.2 Rings and distributions

Let $R$ and $R_q$ denote the rings $\mathbb{Z}[X]/(X^n + 1)$ and $\mathbb{Z}_q[X]/(X^n + 1)$, respectively, where $n = 2^{2n-1}$ such that $X^n + 1$ is the $2^n$-th cyclotomic polynomial. Throughout this paper, the values of $n$, $n'$ and $q$ are 256, 9 and 7681, respectively. Regular font letters denote elements in $R$ or $R_q$ (which includes elements in $\mathbb{Z}$ and $\mathbb{Z}_q$) and bold lower-case letters represent vectors with coefficients in $R$ or $R_q$. By default, all vectors will be column vectors. Bold upper-case letters are matrices. For a vector $v$ (or matrix $A$), we denote by $v^T$ (or $A^T$) its transpose.

**Modular reductions.** For an even (resp. odd) positive integer $a$, we define $r' = r \mod a$ to be the unique element $r'$ in the range $0 < r' \leq a/2$ (resp. $-a/2 < r' \leq a/2$) such that $r' = r \mod a$. For any positive integer $a$, we define $r' = r \mod a$ to be the unique element $r'$ in the range $0 \leq r' < a$ such that $r' = r \mod a$. When the exact representation is not important, we simply write $r \mod a$.

**Rounding.** For an element $x \in \mathbb{Q}$ we denote by $\lfloor x \rceil$ rounding of $x$ to the closest integer with ties being rounded up.

**Sizes of elements.** For an element $w \in \mathbb{Z}_q$, we write $\|w\|_{\infty} = \text{max} |w_0, w_1, \ldots, w_q|$ to mean $|w \mod \mathbb{Z}_q|$. We now define the $\ell_\infty$ and $\ell_2$ norms for $w = w_0 + w_1 X + \ldots + w_{n-1} X^{n-1} \in R$:

$$
\|w\|_{\infty} = \text{max} |w_0|, \|w_2| = \sqrt{|w_0|^2 + \ldots + |w_{n-1}|^2},
$$

Similarly, for $w = (w_1, \ldots, w_k) \in \mathbb{R}^k$, we define

$$
\|w\|_{\infty} = \text{max} |w_1|, \|w_2| = \sqrt{|w_1|^2 + \ldots + |w_k|^2}.
$$

**Distributions.** For a set $S$, we write $z \leftarrow S$ to denote that $z$ is chosen uniformly at random from $S$. If $S$ is a probability distribution, then this denotes that $z$ is chosen according to the distribution $S$.

**Extendable output function.** Suppose that Sam is an extendable output function, that is a function on bit strings in which the output can be extended to any desired length. If we would like Sam to take as input $x$ and then produce a value $y$ that is distributed according to distribution $S$ (or uniformly over a set $S$), we write
$y \sim S := \text{Sam}(x)$. It is important to note that this procedure is completely deterministic: a given $x$ will always produce the same $y$. For simplicity, we assume that the output distribution of $\text{Sam}$ is perfect, whereas in practice $\text{Sam}$ will be implemented using random oracles and produces an output that is statistically close to the perfect distribution.

**Binomial distribution.** We define the centered binomial distribution $B_n$ for some positive integer $n$ as follows:

$$\text{Sample } (a_0, a_1, \ldots, a_n, b_1, \ldots, b_n) \leftarrow \{0,1\}^{2n} \text{ and output } n \sum_{i=1}^{n} (a_i - b_i).$$

If $v$ is an element of $R$, we write $v \sim \beta_q$ to mean that $v \in R$ is generated from a distribution where each of its coefficients is generated according to $B_q$. Similarly, a $k$-dimensional vector of polynomials $v \in R^k$ can be generated according to the distribution $\beta_q^k$.

**Compression and Decompression.** We now define a function $\text{Compress}_q(x,d)$ that takes an element $x \in \mathbb{Z}_q$ and outputs an integer in $\{0, \ldots, 2^d - 1\}$, where $d < \lceil \log_2(q) \rceil$. We furthermore define a function $\text{Decompress}_q$ such that

$$x' = \text{Decompress}_q(\text{Compress}_q(x,d),d) \quad (1)$$

is an element close to $x$ — more specifically

$$|x' - x \mod q| \leq B_q \approx \left\lceil \frac{q}{2^d+1} \right\rceil.$$

The functions satisfying these requirements are defined as:

$$\text{Compress}_q(x,d) = \left\lfloor (2^d/q) \cdot x \mod 2^d \right\rfloor,$$

$$\text{Decompress}_q(x,d) = \left\lfloor (q/2^d) \cdot x \right\rfloor.$$

If $x'$ is a function of $x$ as in Eq. (1), then for a randomly chosen $x \leftarrow \mathbb{Z}_q$, the distribution of

$$x' - x \mod q$$

is almost uniform over the integers of magnitude at most $B_q$. In particular, this distribution has equal weight over integers of magnitude at most $B_q - 1$ and has a smaller weight on the integer(s) of magnitude $B_q$.

When $\text{Compress}_q$ or $\text{Decompress}_q$ is used with $x \in R_q$ or $x \in R_q^k$, the procedure is applied to each coefficient individually.

The main reason for defining $\text{Compress}_q$ and $\text{Decompress}_q$ functions is to be able to discard some low-order bits in the public key and the ciphertext which do not have much effect on the correctness of decryption — thus making the parameters smaller. The $\text{Compress}_q$ function is also used in one other place where its intuitive purpose is not to “compress”. In line 3 of the decryption procedure (Algorithm 3), the function is used to decrypt to a 1 if $v - s^T u$ is closer to $[q/2]$ than to 0, and decrypt to a 0 otherwise.

### 2.3 Module-LWE

Let $k$ be a positive integer parameter. The hard problem underlying the security of our schemes is Module-LWE. It consists in distinguishing uniform samples $(a_i, b_i) \leftarrow R_q^k \times R_q$ from samples $(a_i, b_i) \in R_q^k \times R_q$ where $a_i \leftarrow R_q^k$ is uniform and $b_i = a_i^T s + e_i$ with $s \leftarrow \beta_q^k$ common to all samples and $e_i \leftarrow \beta_q$ fresh for every sample. More precisely, for an algorithm $A$, we define $\text{Adv}^\text{mLWE}_{m,k,\eta}(A) = \left| \Pr [ b' = 1 : A \leftarrow R_q^{mk} ; (s,e) \leftarrow \beta_q^k \times \beta_q^m ; b = As + e, b' = A(A,b) ] - \Pr [ b' = 1 : A \leftarrow R_q^{mk} ; b \leftarrow R_q^m, b' = A(A,b) ] \right|.$

### 3 KYBER’S IND-CPA-SECURE ENCRYPTION

Let $k, d_1, d_2, d_3$ be positive integer parameters, and recall that $n = 256$. Let $M = \{0,1\}^{256}$ denote the message space, where every message $m \in M$ can be viewed as a polynomial in $R$ with coefficients in $\{0,1\}$. Consider the public-key encryption scheme Kyber. CPA = (KeyGen, Enc, Dec) as described in Algorithms 1 to 3.

**Algorithm 1 Kyber.CPA.KeyGen(): key generation**

1. $\rho, \sigma \leftarrow \{0,1\}^{256}$
2. $A \sim \text{pkk}^k := \text{Sam}(\rho)$
3. $(s,e) \sim \beta_q^k \times \beta_q^k := \text{Sam}(\sigma)$
4. $t := \text{Compress}_q(As + e, d_1)$
5. return $(pk := (t,\rho), sk := s)$

**Algorithm 2 Kyber.CPA.Enc(pk = (t,\rho), m \in M): encryption**

1. $r \leftarrow \{0,1\}^{256}$
2. $t := \text{Decompress}_q(t, d_1)$
3. $A \sim \text{pkk}^k := \text{Sam}(\rho)$
4. $(r,e_1, e_2) \sim \beta_q^k \times \beta_q^k \times \beta_q := \text{Sam}(r)$
5. $u := \text{Compress}_q(A^T r + e_1, d_4)$
6. $v := \text{Compress}_q(t^T r + e_2 + \left\lfloor \frac{q}{2} \right\rfloor \cdot m, d_5)$
7. return $c := (u,v)$

**Algorithm 3 Kyber.CPA.Dec(sk = s, c = (u,v)): decryption**

1. $u := \text{Decompress}_q(u, d_4)$
2. $v := \text{Decompress}_q(v, d_5)$
3. return $\text{Compress}_q(v - s^T u, 1)$

**Correctness.** We show below the correctness of the encryption scheme described in Algorithms 1 to 3. We will select parameters in Section 6 to make the decryption error negligible, i.e., that Kyber.CPA is $(1 - \delta)$-correct with $\delta < 2^{-128}$.

**Theorem 3.1.** Let $k$ be a positive integer parameter. Let $s, e, r, e_1, e_2$ be random variables that have the same distribution as in Algorithms 1 and 2. Also, let $e_1 \sim \psi_{d_1}^k, e_2 \sim \psi_{d_2}^k, c_u \sim \psi_{d_4}^k, c_v \sim \psi_{d_5}^k$, be distributed according to the distribution $\psi$ defined as follows:

Let $\psi_d^k$ be the following distribution over $R$:

\[ \psi_d^k := \frac{1}{|R_d^k|} \sum_{x \in R_d^k} \delta_{(x,h(x))} \]
1. Choose uniformly-random \( y \leftarrow R^k \)
2. return \( \{ y - \text{Decompress}_q(\text{Compress}_q(y, d), d) \} \mod^* q \)

Denote
\[
\delta = \Pr \left[ \left\| e^T r + e_2 + c_v - s^T e_1 + c_T^r r + s^T c_u \right\|_\infty \geq \lceil q/4 \rceil \right].
\]

Then Kyber.CPA is \((1 - \delta)-\text{correct})\.

Remark 3.2. We provide with our software a Python script that allows to compute a tight upper bound on \( \delta \); the parameter set we recommend for Kyber in Table 1 yields \( \delta = 2^{-142} \).

Proof. The value of \( t \) in line 6 of Algorithm 2 is:
\[
t = \text{Decompress}_q(\text{Compress}_q(\{ As + e, d_i \}, d_i) = As + e + c_v,
\]
for some \( c_v \in R^k \). The value of \( u \) in Algorithm 3 is
\[
u = \text{Decompress}_q(\text{Compress}_q(\{ AT + e_1, d_u \}, d_u) = AT + e_1 + c_v,
\]
for some \( c_v \in R^k \). And the value of \( v \) is
\[
u = \text{Decompress}_q(\text{Compress}_q(\{ v + [q/2] \cdot m, d_v \}, d_v)
= t^T r + e_2 + [q/2] \cdot m + c_v,
= (As + e + c_v)^T r + e_2 + [q/2] \cdot m + c_v,
= (As + e)^T r + e_2 + [q/2] \cdot m + c_v + c_T^r r,
\]
for some \( c_v \in R^k \). In all of the above, we can safely assume that the values \( c_v \), \( c_v \), and \( c_v \) are distributed according to the distribution \( \psi \) defined in the Theorem statement. The reason is that all of these are of the form \( \{ y - \text{Decompress}_q(\text{Compress}_q(y, d), d) \} \mod^* q \) where \( y \) is pseudo-random based on the hardness of Module-LWE.

Using the above, we obtain
\[
u - s^T u = e^T r + e_2 + [q/2] \cdot m + c_v + c_T^r r - s^T r + s^T c_u
\]
If \( \left\| e^T r + e_2 + c_v - s^T e_1 + c_T^r r + s^T c_u \right\|_\infty < \lceil q/4 \rceil \), then we can write \( v - s^T u = w + [q/2] \cdot m \) where \( \left\| w \right\|_\infty < \lceil q/4 \rceil \). Define \( m^r = \text{Compress}_q(\{ v - s^T u, 1 \} \), We then know that
\[
\lceil q/4 \rceil \geq \left\| v - s^T u - [q/2] \cdot m^r \right\|_\infty = \left\| w + [q/2] \cdot m - [q/2] \cdot m^r \right\|_\infty.
\]
By the triangle inequality and the fact that \( \left\| w \right\|_\infty < \lceil q/4 \rceil \), we obtain
\[
\left\| [q/2] \cdot (m - m^r) \right\|_\infty < 2 \cdot \lceil q/4 \rceil,
\]
which (for all odd \( q \)) implies that \( m = m^r \), and proves the correctness of Kyber.CPA.

\[ \square \]

Security. We prove that the encryption scheme defined above is IND-CPA secure under the Module-LWE hardness assumption.

Theorem 3.3. For any adversary \( A \), there exists an adversary \( B \) such that \( \text{Adv}^{\text{CPA}}_{\text{Kyber.CPA}}(A) \leq 2 \cdot \text{Adv}^{\text{mwe}}_{k+1, k, q}(B) \).

Proof. Let \( A \) be an adversary that is executed in the IND-CPA security experiment which we call game \( Go_0 \), i.e., \( \text{Adv}^{\text{PKE}}_{\text{Kyber.CPA}}(A) = | \text{Pr}[b = b' \text{ in game } Go_0] - 1/2 | \). In game \( Go_1 \), the value \( t' := As + e \) which is used in KeyGen is substituted by a uniform random value. It is possible to verify that there exists an adversary \( B \) with the same running time as that of \( A \) such that \( | \text{Pr}[b = b' \text{ in game } Go_1] - | \text{Pr}[b = b' \text{ in game } Go_1] | \leq \text{Adv}^{\text{mwe}}_{k+1, k, q}(B) \leq \text{Adv}^{\text{mwe}}_{k+1, k, q}(B) \). In game \( Go_2 \), the values \( u' := AT + e_1 \) and \( v' := t^T r + e_2 \) used in the generation of the challenge ciphertext are simultaneously substituted with uniform random values. Again, there exists an adversary \( B \) with the same running time as that of \( A \) with \( | \text{Pr}[b = b' \text{ in game } Go_1] - | \text{Pr}[b = b' \text{ in game } Go_1] | \leq \text{Adv}^{\text{mwe}}_{k+1, k, q}(B) \). Note that in game \( Go_2 \), the value \( v \) from the challenge ciphertext is independent of bit \( b \) and therefore \( \text{Pr}[b = b' \text{ in game } Go_2] = 1/2 \). Collecting the probabilities yields the required bound.

\[ \square \]

4 THE CCA-SECURE KEM

Let \( G: \{0,1\}^* \rightarrow \{0,1\}^{32 \times 256} \) and \( H: \{0,1\}^* \rightarrow \{0,1\}^{256} \) be hash functions. Consider the public-key key encapsulation mechanism Kyber = (KeyGen, Encaps, Decaps) as described in Algorithms 1, 4 and 5, where KeyGen is the same as the one of Kyber.CPA from the previous section, with the difference that \( sk \) also contains \( pk = (r, t) \) and a secret 256-bit random value \( z \). It is obtained by applying a KEM variant [44] of the Fujisaki–Okamoto transform [37, 69] to the Kyber.CPA encryption scheme. Note that we make explicit the randomness \( r \) in the Enc algorithm.

Algorithm 4 Kyber.Encaps\((pk = (r, t))\)
1. \( m \leftarrow \{0,1\}^{256} \)
2. \( (K, r, d) := G(pk, m) \)
3. \( (u, v) := \text{Kyber.CPA.Enc}(\{ r, t \}, m; r) \)
4. \( c := (u, v, d) \)
5. \( K := H(K, c) \)
6. return \( (c, K) \)

Algorithm 5 Kyber.Decaps\((sk = (s, z, \rho, t), c = (u, v, d))\)
1. \( m' := \text{Kyber.CPA.Dec}(s, (u, v)) \)
2. \( (\tilde{K}', \tilde{r}', \tilde{d}') := G(pk, m') \)
3. \( (u', v') := \text{Kyber.CPA.Enc}(\{ \rho, t \}, m'; \tilde{r}') \)
4. if \( (u, v, d) = (u', v', d') \) then
5. \( K := H(\tilde{K}', c) \)
6. else
7. \( K := H(z, c) \)
8. end if

We stress that Kyber.Decaps never returns \( \bot \). Instead, in case re-encryption fails, it returns a pseudo-random key \( K := H(z, c) \), where \( z \) is a random, secret seed.

Correctness. If Kyber.CPA is \((1 - \delta)-\text{correct})\) and \( G \) is a random oracle, then Kyber is \((1 - \delta)-\text{correct})\) [44].

Security. The following concrete security statement proves Kyber’s CCA-security when the hash functions \( G \) and \( H \) are modeled as random oracles. We provide the concrete security bounds from [44] which considers the KEM variant of the FO transformation and also takes a non-zero correctness error \( \delta \) into account.

Theorem 4.1. For any adversary \( A \) that makes at most \( q_G \) many queries to random oracle \( H \), at most \( q_C \) many queries to random oracle \( G \), and \( q_D \) queries to the decryption oracle, there exists an adversary \( B \) such that
\[
\text{Adv}_{\text{Kyber}}^{\text{CCA}}(A) \leq 3 \cdot \text{Adv}_{\text{Kyber.CPA}}^{\text{CPA}}(B) + q_H \cdot \delta + \frac{2q_G + q_H + 1}{2^{256}}.
\]
Note that the security bound is tight. In particular, in combination with Theorems 3.1 and 3.3 we obtain a tight reduction from the Module-LWE hardness assumption. We remark that there exists an alternative security reduction from the weaker notion of ONE-WAY CPA-security [44] of Kyber. CPA is, however, not tight as it loses a multiplicative factor \( q_G \).

The value \( d \) in Kyber’s ciphertexts is not necessary for the security proof in case \( H \) and \( G \) are modeled as standard random oracles. However, it is crucial for a proof in the quantum random oracle model. Concretely [44, 69] proved that Kyber is CCA secure in the quantum random oracle model, provided that Kyber.CPA is CPA-secure. Again, we provide the concrete bound from [44].

**Theorem 4.2.** For any quantum adversary \( A \) that makes at most \( q_H \) many queries to quantum random oracle \( H \), at most \( q_G \) many queries to quantum random oracle \( G \), and at most \( q_D \) many (classical) queries to the decryption oracle, there exists a quantum adversary \( B \) such that

\[
\text{Adv}^{\text{cca}}_{\text{Kyber}}(A) \leq 4qH \sqrt{qD \cdot qH \cdot \delta + qG \cdot \text{Adv}^\text{cpa}_{\text{Kyber},\text{CPA}}(B)}.
\]

Unfortunately, as it is common for proofs in the quantum random oracle model, the above security bound is far from tight and therefore can only serve as an asymptotic indication of Kyber’s CCA-security in the quantum random oracle model.

**Hashing \( pk \) into \( \hat{K} \).** The Kyber CCA transform is essentially the transform from [44, 69], with one small tweak: we hash the public key \( pk \) into \( \hat{K} \). This tweak has two effects. First, it makes the KEM contributory; the shared key \( K \) does not depend only on input of one of the two parties. The second effect is a multi-target protection. Consider an attacker who searches through many values \( m \) to find one that is “likely” to produce a failure during decryption. Such a decryption failure of a legitimate ciphertext would leak some information about the secret key. In the pre-quantum setting this attack approach is doomed because of the negligible failure probability \( \delta \). In a post-quantum setting, the attacker could use Grover’s algorithm to search for such an \( m \). However, the attacker is then facing the problem to encode “likely to produce a decryption failure” in the Grover oracle. This is equivalent to identifying noise vectors that are likely to have a large inner product with \((s, e)\); probably the best strategy is to search for \( m \) that produce noise vectors of large norm. Even though we believe this attack approach is unlikely to result in any better performance than a brute-force Grover search of the 256-bit shared key \( K \), hashing \( pk \) into \( \hat{K} \) ensures that an attacker would not be able to use precomputed values \( m \) against multiple targets.

**CCA-secure public-key encryption.** We remark that a CCA-secure public-key encryption scheme can be obtained by combining the CCA-secure KEM Kyber with any CCA-secure symmetric encryption scheme [33] (aka. DEM). We describe the resulting hybrid encryption scheme Kyber.Hybrid in Appendix A.

## 5 KEY EXCHANGE PROTOCOLS

Let Kyber = (KeyGen, Encaps, Decaps) be the IND-CCA secure KEM from the previous section. Figure 1 describes the Kyber key exchange protocol Kyber.KE obtained as a direct application of the key encapsulation mechanism. In key exchange constructions using a KEM, it is common to hash the “view” of each participant (i.e., all received and sent messages) into the final key. In Kyber, the public key \( pk \) is hashed into the “pre-key” \( \hat{K} \) and the ciphertext is hashed into the final key \( K \); hence the shared key obtained in a key exchange already includes the complete “view” of each participant.

**Figure 1:** Kyber.KE – Key Exchange protocol using the Kyber = (KeyGen, Encaps, Decaps) key encapsulation mechanism.

**Authenticated key exchanges protocols.** Note that the protocol of Fig. 1 by itself only provides security against passive adversaries (and in particular fails to protect against man-in-the-middle attacks). Let \( H : \{0,1\}^* \rightarrow \{0,1\}^{256} \) be a hash function. Figure 2 describes our one-sided (unilateral) authenticated key exchange protocol Kyber.UAKE in which party \( P_1 \) knows the static (long-term) key of party \( P_2 \), and Fig. 3 describes our authenticated key-exchange protocol Kyber.AKE where each party knows the static (long-term) key of the other party.

**Figure 2:** Kyber.UAKE – One-sided authenticated key exchange protocol using Kyber, where \( P_1 \) knows the static public key of \( P_2 \).

The shared key derived at the end of the above protocols not only depends on the ephemeral key and ciphertext \((pk, c)\), but also on the static (long-term) keys \( pk_2 \) and associated ephemeral ciphertexts \( c_i \) (where \( i = 2 \) and \( i = 1, 2 \) respectively).

Our authenticated key-exchange protocols follow a generic construction from any CCA-secure encryption scheme. Concretely,
We decided to fix \( d_u = d_t = 11 \), which unifies compression of public keys and the “key component” \( v \) of the ciphertext.

**Core-SVP hardness.** To analyze the security of Kyber, we follow the methodology introduced in [4, Sec. 6.1]. This means that we assume that the best way to solve the Module-LWE problem underlying Kyber is to treat it as a general LWE problem. Moreover we consider the primal and dual attacks to be the only known attacks relevant to our parameter sets. After optimizing the parameters for the primal attack with respect to the success criteria of [4, Sec. 6.3], we find that the attack would invoke BKZ with blocksize 610 to 615 (depending on whether one uses the primal or dual attack). The cost of BKZ with blocksize 610 is dominated by a polynomial number of calls to a dimension 610 SVP solver. Suppressing this polynomial number of SVP calls and all subexponential factors in the cost of the best known quantum algorithm for SVP [48, Sec. 14.2.10], this implies a cost of \( > 2^{161} \) operations in the quantum RAM model. According to this very conservative analysis, Kyber offers 161 bits of security against the best known quantum attacks targeting the underlying lattice problem.

**LWE security vs. LWR security.** The analysis in the previous paragraph considers only the noise introduced by addition of “noise polynomials” sampled from \( \beta_p \); it does not take into account the additional uniform noise introduced by rounding. Various recent proposals for lattice-based cryptosystems rely only on this noise from rounding. See for example the schemes proposed in [12] and [28]. In some sense, the rounding induces a deterministic noise, and it is not clear how this determinism affects concrete security. On one hand, there are reductions between LWR and LWE, but they involve substantial losses, especially in the ring setting [8]. In particular, they do not apply when the number of dropped bits is so small. On the other hand, there is to our knowledge, no attack that performs better on LWR than LWE with the corresponding uniform noise. (See [16, Sec. 4] for a reduction from LWR to LWE with corresponding uniform noise.) Yet it is not clear to us that LWR has received any dedicated attention from cryptanalysts, and we therefore prefer to remain conservative and estimate security of Kyber only based on the LWE noise.

**Resistance to hybrid attacks.** Several schemes [12, 42] are potentially vulnerable to a hybrid attack [38, 45], mixing lattice reduction techniques with Meet-in-the-Middle combinatorial search. This attack is particularly difficult to analyze, and recent work [70] suggests that it is often not as competitive as previously thought. We note that this attack is especially relevant when secrets and errors are ternary and sparse, which is not the case for our design.

**Algebraic attacks.** The main novelty of our design is in the use of Module-LWE rather than Ring-LWE. One of the motivations for this change is to move further away from the recently uncovered weaknesses of ideal lattices [15, 24, 31, 32] — yet without the cost of using completely unstructured LWE. The work of [32] mentions obstacles towards a quantum attack on Ring-LWE from their new techniques, but nevertheless suggests using Module-LWE, as it plausibly creates even more obstacles.

**Scaling security and performance.** A particularly attractive feature of Module-LWE (as compared to LWE or Ring-LWE) is, that scaling security only needs marginal changes to existing, possibly
highly optimized implementations. Specifically, the only parameters that need to change to scale security (and performance) of Kyber, are \( k \) and \( \eta \); note that optimized code for polynomial arithmetic is not affected by changing those parameters. Table 2 lists one “paranoid” parameter set aiming at security similar to NewHope (using dimension \( n \cdot k = 1024 \)) and one “light” parameter set that might become interesting for the 96-bit security level, or, with a tighter security analysis for the 128-bit security level, if continued effort in cryptanalysis does not produce significantly better attacks.

The Core-SVP hardness analysis against the best known quantum attacks yields 218 bits of security for the paranoid parameter set and 102 bits of security for the light parameter set.

A note on passively secure KEMs. We note that in order to support the CCA transformation, we need a negligible (in the cryptographic sense) failure probability. Previous proposals like NewHope [4] or Frodo [18] are designed to only achieve passive security and can live with much higher failure probabilities (\( \approx 2^{-60} \) for NewHope and \( 2^{-38.9} \) for the recommended parameter set of Frodo). If one were to optimize a passively secure KEM from Module-LWE, one could reduce the rounding parameters \( d_i \) and \( d_k \) to \( d_i = d_k = 10 \) to further reduce public-key size (to 992 bytes) and ciphertext size (to 1088 bytes) while increasing the failure probability (to \( 2^{-71.9} \)).

7 IMPLEMENTATION

In this section we give all the remaining details of our implementations of Kyber and report on performance of subroutines. Both implementations are fully protected against timing attacks. All cycle counts in this section were obtained on one core of an Intel Core-i7 4770K (Haswell) with hyperthreading and TurboBoost turned off running at 3.5 GHz. They are median cycle counts over 1000 measurements.

7.1 Primitives and encodings

Sections 3 and 4 introduce Kyber in abstract terms without fixing concrete instantiations of the functions H, G, and Sam, and without fixing encodings of messages. This subsection details concrete instantiations of these building blocks.

Symmetric primitives. The main symmetric building blocks are the two hash functions H and G, a function that accepts as input the public seed \( \rho \) and generates the uniform matrix \( A \in \mathbb{R}^{k \times k} \), and a function that accepts as input a secret seed \( r \) and generates as output noise polynomials sampled from \( \beta_q \). Note that in passively secure KEMs like BCNS [19], NewHope [4], or Frodo [18], the choice of how noise polynomials are sampled is a local decision: implementations on different platforms can choose whatever PRNG is the best option on the respective platform. This is also true for noise generation in Kyber’s key generation, but, because of the CCA transform, is no longer true for noise generation in encapsulation.

We decided to instantiate all hash functions with the extendable output function SHAKE-128, standardized in FIPS 202 [58]. For the expansion of (public and secret) seeds we use the domain-separated version cSHAKE-128, that has recently been standardized in FIPS 800-185 [46]. All cSHAKE-128 domain separators in Kyber are 2 bytes long; we will denote them in the following as \((i,j)\), where \( i \) is the byte at the lower address. With this choice, all symmetric primitives in Kyber rely on the same underlying primitive, namely the Keccak-f1600 permutation. The only exception is that for key generation, different implementations are free to use whatever PRNG is offering the best performance and security on their respective platform.

We are aware that another choice of symmetric primitives would yield somewhat better performance on most platforms. For example, we could have decided to use SHA256 for all hashes (with output extension for \( G \) via MGFI; see [57, App. B.2.1]), and AES in counter mode for the expansion of seeds. This choice would certainly be faster on platforms with hardware AES and SHA256 support. However, on platforms without hardware support, AES implementations are notorious for timing-attack vulnerabilities. Furthermore, as pointed out in [4, Sec. 3], the use of a PRG (which AES in counter mode is), is not helpful to argue security, because in the generation of A, the input is public, whereas security of a PRG is only given for secret inputs.

Other possible choices of primitives that would yield better performance are the ChaCha20 stream cipher [10] that has recently been standardized for TLS [50] or the BLAKE2X extendable output function [6]. Unfortunately, neither of these functions has received a lot of cryptanalytic attention, yet, so we prefer to stick to the conservative choice of SHAKE-128, which was standardized after years of cryptanalytic scrutiny through the course of the SHA-3 competition.

The NTT domain. Computing the discrete Fourier transform on elements from \( \mathbb{R}_q \) can be done with methods analogous to the fast Fourier transform [29], except that operations on coefficients are defined in a finite field [62]. This is often referred to as the number theoretic transform (NTT). Before being able to define the expansion of the seed \( \rho \) into the matrix \( A \), we need to define the NTT domain of polynomials. Let \( \omega = 3844 \in \mathbb{Z}_q \) and \( \psi = \sqrt{\omega} = 62 \), where \( \psi \) is chosen as the smallest element of multiplicative order \( 2^9 \) in \( \mathbb{F}_q^* = \mathbb{Z}_q^* \).

For a polynomial \( g = \sum_{i=0}^{255} g_i X^i \in \mathbb{R}_q \), define the polynomial \( \hat{g} \) in NTT domain as

\[
\text{NTT}(g) = \hat{g} = \sum_{i=0}^{255} \hat{g}_i X^i, \quad \text{with} \quad \hat{g}_i = \sum_{j=0}^{255} \psi^i g_{i+j} \mod q.
\]

The inverse NTT\(^{-1}\) of the function NTT is essentially the same as the computation of NTT, except that it uses \( \omega^{-1} \mod q = 6584 \), multiplies by powers of \( \psi^{-1} \mod q = 1115 \) after the summation, and also multiplies each coefficient by the scalar \( n^{-1} \mod q = 7651 \), so that

\[
\text{NTT}^{-1}(\hat{g}) = g = \sum_{i=0}^{255} g_i X^i, \quad \text{with} \quad g_i = n^{-1} \psi^{-i} \sum_{j=0}^{255} \hat{g}_{i+j} \mod q.
\]

For two polynomials \( f, g \in \mathbb{R}_q \), the product \( fg \) can be computed as NTT\(^{-1}\{\text{NTT}(f) \circ \text{NTT}(g)\} \), where \( \circ \) denotes the point-wise multiplication.

Generation of A. Generation of the matrix \( A = (a_{i,j}) \in \mathbb{R}_q^{k \times k} \) receives as input the public seed \( \rho \). To generate the entry \( a_{i,j} \in \mathbb{R}_q \), we first expand \( \rho \) through cSHAKE-128 with the 2-byte domain separator \((i,j)\). The output of this expansion is considered a stream.
of 16-bit little-endian integers. On this sequence of 16-bit integers we run rejection sampling as follows: first set the upper 3 bits of the integers to zero, then use the resulting integer as coefficient for $a_{i,j}$ if it is smaller than $q$, otherwise discard it and move to the next 16-bit integer. Fill the polynomial $a_{i,j}$ starting from the constant coefficient moving to the coefficient belonging to $X^{n-1}$. The resulting polynomial $a_{i,j}$ is assumed to be in NTT domain. Note that this generation of $A$ exhibits $k^2$-way parallelism for the expansion of $\rho$.

**Generation of noise polynomials.** Noise polynomials in Kyber are sampled from $\beta_i$. To obtain such a noise polynomial we first expand a seed to an array of $n = 256$ uniformly random bytes $(r_0, \ldots, r_{255})$. We then generate coefficient $e_i$ of a noise polynomial $e = \sum_{i=0}^{255} e_iX^i$ by subtracting the Hamming weight of the most significant nibble of $r_i$ from the Hamming weight of the least significant nibble of $r_i$. As stated above, the choice of how the 256 uniformly random bytes are generated during key generation is a local, platform-dependent choice. During encapsulation we again use cSHAKE-128 with 2-byte domain separators to expand the 32-byte secret key to 256 bytes. To generate $r = (r_0, r_1, r_2)$ we use domain separators $(0,0)$, $(1,0)$, and $(2,0)$; to generate $e_1 = (e_{1,0}, e_{1,1}, e_{1,2})$ we use domain separators $(3,0)$, $(4,0)$, and $(5,0)$; and to generate $e_2$ we use domain separator $(6,0)$. Note that $r$ is used as input to a multiplication, which is computed via the fast negacyclic NTT operation outlined above. In order to compute the NTT inplace, we would first have to bit-reverse the order of coefficients in $r$. As in NewHope, we omit this bitreversal, and instead assume the coefficients of $r$ to be in bit-reversed order.

**Inputs to $G$ and $H$.** The description of the CCA transform includes the public key $pk$ as input of $G$ and includes the ciphertext $c = (u, v, d)$ as input of $H$. In the implementation we instead include $H(pk)$ and $H(c)$. The two additional hashes may seem redundant, but simplify implementation with a non-incremental API for $G$ and $H$. Furthermore using $H(pk)$ instead of $pk$ as input to $G$ enables a small speedup for decapsulation at the cost of a slightly increased secret-key size as explained in the next paragraph.

**Encoding of keys and ciphertexts.** In NewHope, polynomials in public keys and the ciphertext are in NTT domain; in Kyber all polynomials sent over the channel are in normal domain. This is necessary for the compression through rounding (see Section 3) to work.

A Kyber public key is a tuple $(t, \rho)$, where $t$ is a vector of three polynomials with 256 11-bit coefficients each, and $\rho$ is a 32-byte seed. We encode the polynomials in compressed little-endian format to fit it in $(256 \cdot 11)/8 = 352$ bytes, concatenate the compressed three polynomials and finally concatenate $\rho$ to obtain public keys of $3 \cdot 352 + 32 = 1088$ bytes.

A Kyber secret key is a vector of three polynomials in NTT domain with 256 13-bit coefficients each. We store these polynomials in compressed little-endian format resulting in a total of $(3 \cdot 256 \cdot 13)/8 = 1248$ bytes. For re-encapsulation during decapsulation we additionally need the public key, which we simply concatenate and store as part of the secret key. Finally, we also concatenate $H(pk)$ to avoid having to compute this hash during decapsulation and concatenates the 32 bytes of the value $z$ that is used to compute the pseudo-random returned key when re-encapsulation fails. This results in a total size of $1248 + 1088 + 32 + 32 = 2400$ bytes for the secret key.

A Kyber ciphertext is a 3-tuple $(u, v, d)$, where $u$ is a vector of three polynomials with 256 11-bit coefficients each, $v$ is a polynomial with 256 3-bit coefficients, and $d$ is a 32-byte hash. Using the same compressed little-endian format for polynomials as for keys we obtain ciphertexts with a total size of $3 \cdot 352 + (3 \cdot 256)/8 + 32 = 1184$ bytes.

**Size-speed tradeoffs.** It is possible to use different tradeoffs between secret-key size and decapsulation speed. If secret-key size is critical, it is of course possible to not store $H(pk)$ and also to not store the public key as part of the secret key but instead recompute it during decapsulation. Furthermore, not keeping the secret key in NTT domain makes it possible to compress each coefficient to only 5 bits, resulting in a total size of only 320 bytes for the three polynomials. Finally, as all randomness in key generation is generated from two 32-byte seeds, it is also possible to only store these seeds and re-run key generation during decapsulation.

In the other direction, if secret-key size does not matter very much and decapsulation speed is critical, one might decide to store the expanded matrix $A$ as part of the secret key and avoid recomputation from the seed $\rho$ during the re-encapsulation part of decapsulation.

All performance results reported in the following assume the secret-key format described in the previous paragraph; i.e., with polynomials in NTT domain, including the public key and $H(pk)$, but not including $A$.

### 7.2 Reference implementation

Kyber’s reference implementation in C follows much in the spirit of the NewHope reference implementation described in [4, Sec. 7.2]. In particular, it only relies on 16-bit and 32-bit integer arithmetic (outside of Keccak) and uses the same combination of short Barrett reductions and Montgomery reductions to accelerate the NTT computation. One consequence of the modulus $q = 7681$ is that the short Barrett reduction becomes slightly more efficient; an unsigned 16-bit integer $a$ can be reduced to an unsigned integer $r$ between 0 and 11768 and congruent modulo $q$ via the following three operations:

| n   | k   | q   | $\eta$ | $(d_0, d_2, d_4)$ | $\delta$ | pq sec. | $|pk|$ in bytes | $|c|$ in bytes |
|-----|-----|-----|--------|-----------------|--------|---------|----------------|----------------|
| Paranoid | 256 | 4   | 7681   | (11, 3, 11)   | $2^{-145}$ | 218     | 1440           | 1536         |
| Light  | 256 | 2   | 7681   | (11, 3, 11)   | $2^{-169}$ | 102     | 736            | 832          |

Table 2: Alternative parameter sets for higher and lower security levels.
We reduce after level 1 (i.e., after 2 levels), then again after level \( q \) allows us to reduce coefficients modulo \( q \). Vectorized Keccak. This representation results in a very fast computation as first described in [41, Sec. 3.2] and also used for NewHope in [4]. Essentially, our implementation of the NTT follows the same approach as [41] and [4], except that we carefully optimize for \( n = 256 \) and \( q = 7681 \). Specifically, we merge levels 0–4 and then merge levels 5–7 to reduce load and store operations. The 13-bit modulus \( q \) allows us to reduce coefficients modulo \( q \) only every 3 levels. We reduce after level 1 (i.e., after 2 levels), then again after level 4, and finally after level 7. To make best use of the 53-bit radix of double-precision floating-point numbers, we precompute powers of \( \omega \) in the range \([-q/2, q/2]\) and also reduce to this range inside the NTT. Only the last modular reduction goes back to unsigned representation. One NTT takes 1992 cycles; an NTT\(^{-1}\) operation includes a bit reversal and takes 2632 cycles.

We also use vectorized double-precision floating point arithmetic for pointwise multiplication and polynomial addition and subtraction.

Vectorized Keccak. As mentioned earlier, Keccak has a reputation of not being particularly fast in software. One reason is that Keccak is very hard to vectorize; in fact, according to the eBACS benchmarks, the fastest implementation of Keccak on Intel Haswell processors is the non-vectorized “simple64” implementation.

The picture changes drastically if a protocol can compute multiple independent streams of SHA-3, SHAKE, or cSHAKE on inputs and outputs of the same length. More specifically, the Keccak code package [14] includes an implementation for AVX2 that computes 4 independent streams in parallel. We make use of this 4-way parallel implementation in the expansion of \( \rho \) involved in the generation of the matrix \( A \) and also in the generation of noise polynomials during encapsulation. Specifically, for the generation of \( A \), we generate 8 streams of uniformly random 16-bit numbers via two calls to this function, leaving only one sequential SHAKE-128 call. In encapsulation we generate 8 arrays of 256 uniformly random bytes via two calls to 4-way parallel cSHAKE-128 and discard one of those arrays. The speedup from vectorized Keccak is crucial: compared to NewHope, Kyber needs to generate more than twice as many uniformly random polynomial coefficients, yet, with 34304 cycles, generation of the matrix \( A \) is about as fast as generation of the equivalent value \( a \) in NewHope.

Rejection sampling. Part of the generation of \( A \) is rejection sampling on the stream of 16-bit integers produced by the cSHAKE-128 expansion. We adopt the fast vectorized approach described in [40] for this task. One difference is that we do not need to first conditionally subtract \( q \) four times; we simply eliminate the upper 3 bits of each 16-bit integer in a 256-bit vector through one mask instructions and then compare to a constant vector filled with 16-bit copies of \( q \).

7.3 AVX2 implementation

Modern 64-bit Intel processors feature the AVX2 vector-instruction set that supports operations on 256-bit vectors that can be interpreted as vectors of 8 single-precision or 4 double-precision floating-point numbers, or as vectors of integers of various sizes. These AVX2 instructions were also used for the optimized implementation of the optimized NewHope software described in [4, Sec. 7].

Polynomial arithmetic. For polynomial arithmetic we represent polynomials as arrays of double-precision floating-point numbers. This representation results in a very fast NTT computation as first described in [41, Sec. 3.2] and also used for NewHope in [4]. Essentially, our implementation of the NTT follows the same approach as [41] and [4], except that we carefully optimize for \( n = 256 \) and \( q = 7681 \). Specifically, we merge levels 0–4 and then merge levels 5–7 to reduce load and store operations. The 13-bit modulus \( q \) allows us to reduce coefficients modulo \( q \) only every 3 levels. We reduce after level 1 (i.e., after 2 levels), then again after level 4, and finally after level 7. To make best use of the 53-bit radix of double-precision floating-point numbers, we precompute powers of \( \omega \) in the range \([-q/2, q/2]\) and also reduce to this range inside the NTT. Only the last modular reduction goes back to unsigned representation. One NTT takes 1992 cycles; an NTT\(^{-1}\) operation includes a bit reversal and takes 2632 cycles.

We also use vectorized double-precision floating point arithmetic for pointwise multiplication and polynomial addition and subtraction.

Vectorized Keccak. As mentioned earlier, Keccak has a reputation of not being particularly fast in software. One reason is that Keccak is very hard to vectorize; in fact, according to the eBACS benchmarks, the fastest implementation of Keccak on Intel Haswell processors is the non-vectorized “simple64” implementation.

The picture changes drastically if a protocol can compute multiple independent streams of SHA-3, SHAKE, or cSHAKE on inputs and outputs of the same length. More specifically, the Keccak code package [14] includes an implementation for AVX2 that computes 4 independent streams in parallel. We make use of this 4-way parallel implementation in the expansion of \( \rho \) involved in the generation of the matrix \( A \) and also in the generation of noise polynomials during encapsulation. Specifically, for the generation of \( A \), we generate 8 streams of uniformly random 16-bit numbers via two calls to this function, leaving only one sequential SHAKE-128 call. In encapsulation we generate 8 arrays of 256 uniformly random bytes via two calls to 4-way parallel cSHAKE-128 and discard one of those arrays. The speedup from vectorized Keccak is crucial: compared to New Hope, Kyber needs to generate more than twice as many uniformly random polynomial coefficients, yet, with 34304 cycles, generation of the matrix \( A \) is about as fast as generation of the equivalent value \( a \) in NewHope.

Rejection sampling. Part of the generation of \( A \) is rejection sampling on the stream of 16-bit integers produced by the cSHAKE-128 expansion. We adopt the fast vectorized approach described in [40] for this task. One difference is that we do not need to first conditionally subtract \( q \) four times; we simply eliminate the upper 3 bits of each 16-bit integer in a 256-bit vector through one mask instructions and then compare to a constant vector filled with 16-bit copies of \( q \).

7.4 Flexibility of Kyber

One possible use of Kyber is for ephemeral key exchange, for example in TLS 1.2 as illustrated by [19] and by Google’s post-quantum TLS experiment [21] with NewHope. Indeed, the experiment concluded that they “did not find any unexpected impediment to deploying something like NewHope” [49] and Kyber features performances close to the one of New Hope but with smaller sizes.

However, the CCA security of Kyber makes it a much more versatile tool. Not only is it possible to cache ephemeral keys for some time (which would be a security disaster for BCNS, Frodo, or NewHope), we can also use it for classical IND-CCA key encryption of messages of arbitrary length [33] (cf. the hybrid CCA-secure scheme of Appendix A) and for authenticated key exchange protocols, as described in Fig. 3. The Kyber software package includes implementations of the unilaterally authenticated key exchange Kyber.UAKE described in Fig. 2 and the mutually authenticated key exchange Kyber.AKE described in Fig. 3.

8 PERFORMANCE RESULTS AND COMPARISON

In this section we report on the performance of our standalone implementations of Kyber, Kyber-based authenticated key exchange, and an integration of Kyber within the Open Quantum Safe (OQS) framework\(^2\) [67].

8.1 Standalone Kyber

In Table 3 we give performance results of the standalone implementations of Kyber and compare them to results from the literature on lattice-based KEMs, key-exchange protocols, and encryption schemes. We compiled the Kyber software with gcc-4.9.2 with optimization flags -O3 -march=native -O3 -fomit-frame-pointer -fwrapv -msse2avx -maxv2 -march=corei7-avx with flags -march=corei7-avx -msse2avx -maxv2 -march=corei7-avx, except for the non-vectorized implementation of Keccak, which we compile with clang-3.5.10 with flags -march=native -O3 -fomit-frame-pointer -fwavp -Qunused-arguments.

To give an indication of security levels obtained by the different schemes we include the core-SVP hardness estimation (“Sec. estim.”) following the approach from [4]. Note that this estimate does not say anything about the applicability of hybrid or algebraic attacks.

8.2 Kyber-based authenticated key exchanges

To illustrate one use case of Kyber and to establish a data point for high-performance post-quantum authenticated key exchanges, the Kyber software package includes implementations of Kyber.AKE and Kyber.UAKE. The performance in terms of message sizes and CPU cycles (for our AVX2 optimized software) is summarized in

---

\(^2\)Note that one can easily combine KEMs (e.g., Kyber with a pre-quantum KEM) by hashing the shared secret keys together.

\(^3\)https://www.openquantumsafe.org
Table 3: Comparison of lattice-based KEMs and public-key encryption. Benchmarks were performed on an Intel Core i7-4770K (Haswell) if not indicated otherwise. Cycles are stated for key generation (K), encapsulation/encryption (E), and decapsulation/decryption (D). Bytes are given for secret keys (sk), public keys (pk), and ciphertexts (c). The column “ct?” indicates whether the software is running in constant time, i.e., with protection against timing attacks.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Sec. estim.</th>
<th>Prob.</th>
<th>ct?</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Passively secure KEMs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCNS [19]</td>
<td>78</td>
<td>Ring-LWE</td>
<td>yes</td>
<td>$K$: $\approx 2,477,958$</td>
<td>sk: 4096</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$E$: $\approx 3,995,977$</td>
<td>pk: 4096</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$D$: $\approx 481,937$</td>
<td>c: 4224</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$E$: 110,986</td>
<td>pk: 18,24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$D$: 19,422</td>
<td>c: 20,48</td>
</tr>
<tr>
<td>Frodo [18] (recommended parameters)</td>
<td>130</td>
<td>LWE</td>
<td>yes</td>
<td>$K$: $\approx 2,938,000$</td>
<td>sk: 11,280</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$E$: $\approx 3,484,000$</td>
<td>pk: 11,296</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$D$: $\approx 338,000$</td>
<td>c: 11,288</td>
</tr>
<tr>
<td><strong>CCA-secure KEMs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$E$: &gt; 51,488</td>
<td>pk: 1232</td>
</tr>
<tr>
<td>spLWE-KEM [27] (128-bit PQ parameters)</td>
<td>128</td>
<td>spLWE</td>
<td>?</td>
<td>$K$: $\approx 336,700$</td>
<td>sk: ?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$E$: $\approx 813,800$</td>
<td>pk: ?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$D$: $\approx 785,200$</td>
<td>c: 804</td>
</tr>
<tr>
<td>Kyber (this paper) (C reference)</td>
<td>161</td>
<td>Module-LWE</td>
<td>yes</td>
<td>$K$: 276,720</td>
<td>sk: 2368</td>
</tr>
<tr>
<td>Kyber (this paper) (AVX2 optimized)</td>
<td>161</td>
<td>Module-LWE</td>
<td>yes</td>
<td>$K$: 77,892</td>
<td>sk: 2400</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$K$: 11,965</td>
<td>pk: 1088</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$D$: 125,736</td>
<td>c: 1184</td>
</tr>
<tr>
<td><strong>CCA-secure public-key encryption</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NTRUEncrypt ees743ep1[42]</td>
<td>159</td>
<td>NTRU</td>
<td>no</td>
<td>$K$: 1,194,816</td>
<td>sk: 1,120</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$E$: 57,440</td>
<td>pk: 1,027</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$D$: 110,604</td>
<td>c: 980</td>
</tr>
<tr>
<td>Lizard [28] (recommended parameters)</td>
<td>128</td>
<td>LWE+LWR</td>
<td>no</td>
<td>$K$: 97,573,000</td>
<td>sk: 466,944</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$E$: $\approx 35,050$</td>
<td>pk: 2,031,616</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$D$: $\approx 80,840$</td>
<td>c: 1072</td>
</tr>
</tbody>
</table>

- a According to the conservative "best known quantum attack" estimates from [1].
- b Benchmarked on a 2.6GHz Intel Xeon E5 (Sandy Bridge).
- c The NTRU Prime paper reports benchmarks only for polynomial multiplication.
- d Benchmarked on “PC (Macbook Pro) with 2.6GHz Intel Core i5”.
- f As reported by the software from https://github.com/LizardOpenSource/Lizard_c, compiled with gcc-6.3 with flags -O3 -fomit-frame-pointer -msse2avx -mavx2 -march=native on Intel Core i7-4770K.
- g Unlike our scheme, the paper reports secret-key size without the public key required for decryption in the Targhi-Unruh transform.
- h Sizes used by the software; those could be compressed by a factor 1.6, incurring only small computational overhead.
- i According to the conservative ‘best known quantum attack’ estimates from [1], with appropriate adaptations (balanced lattice attacks [28, Sec. 4.2]).
- j The problem underlying NTRU Prime is subtly different than in NTRU; it uses a different ring than commonly used in NTRU and uses deterministic noise.
Table 6: Timings for post-quantum key exchanges (C reference implementations) in Open Quantum Safe.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Operations (µs)</th>
<th>$P_1$ (start)</th>
<th>$P_2$</th>
<th>$P_1$ (end)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIDH [30]</td>
<td>15 015</td>
<td>33 530</td>
<td>14 241</td>
<td></td>
</tr>
<tr>
<td>Frodo [18]</td>
<td>3 436</td>
<td>4 027</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>BCNS [19]</td>
<td>1 087</td>
<td>1 729</td>
<td>178</td>
<td></td>
</tr>
<tr>
<td>NewHope [53]</td>
<td>60</td>
<td>104</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>Kyber</td>
<td>77</td>
<td>100</td>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. The only paper describing an implementation of lattice-based authenticated key exchange that we are aware of is [71]. Our software outperforms the results of [71] by more than two orders of magnitude.

8.3 Integration with OQS

The Open Quantum Safe (OQS) aims at supporting the development and prototyping of quantum-resistant cryptography. In particular, it proposes a common API for post-quantum key exchange algorithms, making it easy to integrate new algorithms and compare to alternatives. We integrated the reference C implementation of our CCA-secure KEM Kyber in OQS library, l1boqs, and ran the benchmarking command ./test_kex --bench; the results are provided in Table 5

Note that the timings differ slightly from those of Table 3. Indeed, l1boqs provides reference implementations for randomness sampling, AES, Keccak and ChaCha20; our reference implementation has been minimally modified to be compatible with those API. Also, only the reference C implementation of NewHope is (as of today) integrated to l1boqs and not the AVX2-optimized version listed in Table 3. Finally, all implementations have been compiled with the same flags, which does not necessarily reflect the full potential of every implementation. Note that, once integrated to l1boqs, it is easy to integrate Kyber to OQS’s fork of OpenSSL 1.0.2.7

ACKNOWLEDGMENTS

The authors would like to thank Isis Lovecruft for suggesting the name Kyber and Andreas Hülse for very helpful discussions.

This work is supported by a Veni Innovational Research Grant from NWO under project number 639.021.645. This work is supported by Canada’s NSERC CREATE program. IQC is supported in part by the Government of Canada and the Province of Ontario. Eike Kiltz was supported by the ERC Consolidator Grant ERC-2013-CoG-615073-ERCC. Vadim Lyubashevsky was supported by the sNSF ERC Transfer Starting Grant CRETTP2-166734-FELICITY and the H2020 Project SafeCrypto. The last author was supported by the ERC Starting Grant ERC-2013-Syg-335086-LATTAC.

REFERENCES


---

7 https://github.com/open-quantum-safe/openssl


A THE CCA-SECURE ENCRYPTION SCHEME

We use the canonical way proposed by Cramer and Shoup to compose Kyber, our secure key encapsulation mechanism (KEM), with a secure one-time symmetric-key encryption (SKE, or DEM) scheme [33]. We call Kyber.Hybrid the resulting hybrid encryption scheme.

On the choice of a symmetric encryption scheme. Any SKE scheme that is (one-time) secure against chosen-ciphertext attacks and with key space $\mathcal{K} = \{0, 1\}^{256}$ can be combined with our key encapsulation mechanism Kyber. Typical examples include AES-OCB, AES-GCM or ChaCha20-Poly1305. Depending on one’s application and architecture, different needs and choices for the symmetric encryption scheme are possible; we decide in this paper to not restrict ourselves to a specific application nor to a specific cipher. Additionally to the previously mentioned ciphers, several submissions to the Caesar competition for authenticated encryption are serious candidates for SKE.

Description of Kyber.Hybrid. We describe the public-key hybrid encryption scheme Kyber.Hybrid = (KeyGen, Enc, Dec) in Algorithms 6 to 8, assuming a SKE $(E, D)$ where the encryption algorithm $E$ takes as input a key in $\mathcal{K} = \{0, 1\}^{256}$ and a message in $\{0, 1\}^*$ and outputs a ciphertext, and where the decryption algorithm $D$ takes as input a key and a ciphertext and outputs a message (or the rejection symbol $\bot$).

Algorithm 6 Kyber.Hybrid.KeyGen()
1: $(pk := (\rho, t), sk := (s, \rho, t)) \leftarrow$ Kyber.KeyGen()
2: return $(pk, sk)$

Algorithm 7 Kyber.Hybrid.Enc($pk = (\rho, t), m$)
1: $(c, K) \leftarrow$ Kyber.Encaps($pk$)
2: $c' := E(K, m)$
3: return $c'' := (c, c')$

Algorithm 8 Kyber.Hybrid.Dec($sk = (s, z, \rho, t), c'' = (c, c')$)
1: $K :=$ Kyber.Decaps($sk, c$)
2: return $m := D(K, c')$

Correctness and security. The correctness and security of our hybrid encryption scheme Kyber.Hybrid follow from those of the KEM and the chosen SKE [33, Th. 5].