Key Encapsulat**I**o**N** and Encryption base**D** on Latt**I**ces

Rachid El Bansarkhani
QuantiCor Security & TU-Darmstadt

Document generated with the help of `pqskeleton` version .

Contents

1 Introduction

Lattices as mathematical objects have been studied by early mathematicians such as Gauss or Dirichlet due to its extremely rich combinatorial structure appearing in many areas of mathematics. But in the last 2 decades they have also extensively been utilized in cryptography to build powerful cryptosystems, where the security stems from the worst-case hardness of well studied lattice problems.

Beside the NTRU assumption the main computational assumptions exploited to build those cryptosystems are the hardness of the problems LWE/SIS [1, 17, 13], ring-LWE/ring-SIS [12, 15, 7, 14] and recently also MLWE/MSIS [11], which are equipped with security guarantees based on worst-case lattice problems. However, the efficiency of cryptosystems increases by imposing more structure. Thus, one almost only finds ring instantiations of the respective schemes for use in practice.

The decision problems are widely used to build lattice-based encryption schemes, where the public key and ciphertexts can be represented as LWE instances $\mathbf{A} \cdot \mathbf{s} + \mathbf{e}$. CPA-security is thus obtained almost for free.

We propose trapdoor-based encryption schemes, where the message is injected into the error term. By use of the trapdoor, the secret vector and error terms are recovered during decryption and are thus available for inspection. In many encryption schemes, this is indeed not possible. Since the message is embedded in the error term, small expansion factors can be realized at competitive parameters, i.e. we can encrypt more keys or data per (small) ciphertext bit (e.g. for sign-then-encrypt). Furthermore, it can easily be transformed to ensure CCA security. This work is based on [2, 16].

2 General algorithm specification (part of 2.B.1)

Parameter Definition	
n	power of two
$x^n + 1$	cyclotomic polynomial
$\mathbb{Z}[x]$	set of polynomials with integer coefficients
$\mathbb{Z}_b[x]$	set of polynomials with integer coefficients modulo b
$\mathcal{R}$	$\mathbb{Z}[x]/\langle x^n + 1\rangle$
$\mathcal{R}_b$	$\mathbb{Z}_b[x]/\langle x^n + 1\rangle$
$\mathcal{R}_b^d$	set of d polynomials from $\mathcal{R}_b$
q	modulus
ℓ	module rank
k	$\log q$
$[x]$	represents a polynomial in $\mathcal{R}$ with all coefficients equal to x.
$\lfloor x \rceil$	rounds x to the nearest integer.
$\mathbf{g}$	gadget $\mathbf{g} = \lfloor q/2 \rfloor$
$\mathbf{g}'$	gadget $\mathbf{g} = 2^{k-2}$ used for higher security levels
rsec	coefficient range $\{-\mathsf{rsec}, \ldots, \mathsf{rsec} - 1\}$ of the secrets and error used
p	$p = \mathsf{rsec}$ for simplicity
t	number of truncated bits per coefficient of the public key
$\mathbf{A} \in \mathcal{R}_q^{\ell \times \ell}$	public uniform matrix
$\mathbf{r} \in \mathcal{R}^{\ell}$	secret key
$\bar{\mathbf{p}} \in \mathcal{R}_q^{\ell}$	decompressed public key
$\bar{\mathbf{b}} \in \mathcal{R}_q^{\ell}$	compressed public key
λ	security parameter
δ	decryption failure
μ	seed of size 2λ for $\mathbf{A} \in \mathcal{R}_q^{\ell \times \ell}$
γ	seed of size 2λ for $\mathbf{r}, \mathbf{r}'$
$\mathsf{MLWE}_{x,y,z}$	MLWE instances over a module of rank x with y samples having uniform errors in $\{-z, \ldots, z-1\}$
Secret key size	$n\ell \log 2p + n\ell(k-t) + 2\lambda$ bits
Public key size	$n\ell(k-t) + 2\lambda$ bits
Message size	$n(\ell+1)\log 2p$ bits

2.1 Trapdoor based Encryption $\mathsf{Kindi}_{\mathsf{CPA}}$ with Uniform Errors

In this section, we describe our CPA-secure Module-LWE/SIS based encryption scheme $\mathsf{KINDI}_{\mathsf{CPA}}$. It is based on the works [2, 16] and employs trapdoors in order to recover the secret vector and error term from Module-LWE instances. In fact, the scheme embeds the message into the error term and further encrypts a random string (similar to a KEM) in the secret vector, which can be exploited as a key for a symmetric key cipher. We note that our encryption scheme can be seen in some sense as a "simplified" KEM, where $\mathbf{c} = \mathsf{Kindi}_{\mathsf{CPA}}.\mathsf{Encrypt}(\mathsf{pk}, \mathsf{msg}) = (s_1 \leftarrow \mathcal{R}_2, \mathsf{Encrypt}(\mathsf{pk}, \mathsf{msg}||s_1, \mathsf{G}(s_1)), \mathsf{msg}||s_1 \leftarrow \mathsf{Decrypt}(\mathsf{sk}, \mathbf{c})$ and $\mathsf{Kindi}_{\mathsf{CPA}}.\mathsf{Decrypt}$ just outputs msg. One part of the message, namely the random string, is always hashed with a random oracle in order to deterministically generate the secret and error term. The encryption engine $\mathsf{Encrypt}(\cdot)$ thus coincides with the deterministic encryption scheme in [9], if msg is for instance set to $\mathbf{0}$. In our KEM construction we need s_1 in order to finally deduce the key, thus we take $s_1 \leftarrow \mathcal{R}_2$ out of $\mathsf{Kindi}_{\mathsf{CPA}}.\mathsf{Encrypt}$.

We now give a specification of the parameter space and the algorithms.

We note that for the sake of completeness of the specification we the following decryption engine to output the message and the seed. However, this is only the case when it is used for the CCA-secure KEM construction as a subroutine since the seed is then used to derive the session key. However, when the scheme is used as a regular CPA-secure encryption scheme s_1 is certainly not part of the output as it is not a message. Thus, all of our implementations in the original submission packages to NIST and also the current version never output the seed s_1.

2.1.1 Parameter Space and Notation

We operate with the rings $\mathcal{R} = \mathbb{Z}[x]/\langle x^n + 1\rangle$ and $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, where $n, q = 2^k$ are powers of two and k is a positive integer. In general, we define $\mathcal{R}_b := \mathbb{Z}_b[x]/\langle x^n + 1\rangle$ for some positive integer b. Furthermore, we introduce the gadget polynomial $\mathbf{g} = \lfloor q/2 \rfloor$ with all coefficients being zero except for the constant. For $q = 2^k$, we have $\mathbf{g} = 2^{k-1}$. By ℓ we denote the module rank and the message size per coefficient amounts to $\log_2 \mathsf{rsec}$ bits. Let λ denote the bit security level and define $p := \mathsf{rsec}$ for simplicty. In the implementation, we use rsec instead. By $[x]$ we denote the polynomial with all coefficients equal to x.

2.1.2 Secret and Public Keys

Two seeds of size 2λ bits are generated, where $\lambda \geq 128$. The first seed μ is used to generate the public matrix $\mathbf{A} \in \mathcal{R}_q^{\ell \times \ell}$ by use of a $\mathsf{PRNG} \in \{\mathsf{Shake128}, \mathsf{Shake256}\}$, which consists of ℓ^2 uniformly distributed ring elements modulo q. This seed is public. The second seed γ of size 2λ bits is secret and serves to generate the private key $\mathbf{r} \in \mathcal{R}^\ell$ and the error term $\mathbf{r}'$ each consisting of ℓ ring elements with coefficients sampled uniformly at random from $\{-\mathsf{p}, \ldots, \mathsf{p}-1\}$. In particular, Shake_p generates uniform random integers from $\{0, \ldots, 2p-1\}$ with Shake and substracts p. The uncompressed public key part $\mathbf{b}$ is a module-LWE instance $\mathbf{b} = \mathbf{A} \cdot \mathbf{r} + \mathbf{r}' \in \mathcal{R}_q^\ell$. The public key thus consists of $\mathsf{pk} := (\bar{\mathbf{b}}, \mu)$ and the secret key $\mathsf{sk} := (\gamma, \bar{\mathbf{b}}, \mu)$ also contains the public key required for the decryption engine when recovering the secret and error terms.

Algorithm 1: $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{KeyGen}(1^n, p, k, t, \ell)$:

$\gamma, \mu \leftarrow \{0,1\}^{2\lambda}$
$\mathbf{A} \in \mathcal{R}_q^{\ell \times \ell} \leftarrow \mathsf{Shake}(\mu)$
$\mathbf{r}, \mathbf{r}' \in \mathcal{R}_q^\ell \leftarrow \mathsf{Shake}_p(\gamma)$
$\mathbf{b} = \mathbf{A} \cdot \mathbf{r} + \mathbf{r}'$
$\bar{\mathbf{b}} = \mathsf{Compress}(\mathbf{b}, t)$
$\mathsf{pk} := (\bar{\mathbf{b}}, \mu),\ \mathsf{sk} := (\gamma, \bar{\mathbf{b}}, \mu)$
return $(\mathsf{pk}, \mathsf{sk})$

Compressing the public key just requires to truncate the least t significant bits. Thus, if the public key is uniform random, then so the compressed one within the defined range. If

Algorithm 2: $\mathsf{Compress}(\mathbf{x} \in \mathcal{R}_q^{\ell+1}, t \in \mathbb{N})$:

1 Truncate the t least significant bits of each coefficient in $\mathbf{x}$.
2 $\bar{\mathbf{b}} = \lfloor \mathbf{x}/2^t \rfloor \in \mathcal{R}_{2^{k-t}}^{\ell+1}$.
3 **return** $\bar{\mathbf{b}}$

$q > 2$ is prime (or odd), we can compress by $\mathsf{Compress}(\mathbf{x} \in \mathcal{R}_q^{\ell+1}, t \in \mathbb{N}) = \bar{\mathbf{b}} = \lfloor (\mathbf{x}/q) \cdot 2^{k-t} \rceil$, where $k := \lceil \log q \rceil$. In this case, we have in general a t bit compression, since q is represented by k bits.

2.1.3 Encryption

If $\mathsf{coins} = \perp$ (required for the KEM) the encryptor samples a secret binary polynomial s_1 with coefficients from $\{0, 1\}$ uniformly at random, otherwise coins already contains s_1 (for the KEM). First, the matrix $\mathbf{A} \in \mathcal{R}_q^{\ell}$ is deterministically generated from μ. Subsequently, the public key is retrieved and decompressed via multiplication with 2^t. We note that for a prime modulus q, decompression is accomplished by $\lfloor (\bar{\mathbf{b}}/2^{k-t}) \cdot q \rceil$, where $k := \lceil \log q \rceil$. The random binary polynomial s_1 is extended via the random oracle G implemented as Shake to $\ell - 1$ uniform random polynomials $(\mathbf{s}_2, \ldots, \mathbf{s}_\ell)$ with coefficients in the range $\{0, \ldots, 2p-1\}$, one random polynomial $\bar{\mathbf{s}}_1$ in the range $\{0, \ldots, p-1\}$, i.e. one bit per coefficient less than the other polynomials, a bit string of size $n(\ell+1)\log 2p$ bits, and ℓ polynomials with coefficients in the range $\{-2^{t-1}, \ldots, 2^{t-1}-1\}$. To obtain a secret polynomial $\mathbf{s}_1$ over the full range $\mathcal{R}_{2p}$, we shift $\bar{\mathbf{s}}_1$ by one bit and add s_1. The message msg is xored with the uniform random string $\bar{u}$ and finally split into $n \log 2p$ bit chunks that are encoded as polynomials $\mathbf{u}_i$ from $\mathcal{R}_{2p}^n$ with $\log 2p$ bits per entry. The error term is just $\mathbf{u}$ centered around zero, i.e. each coefficient is translated by p. Since the least t bits of the public key are truncated, we add to each decompressed public key polynomial $\bar{\mathbf{p}}_i$ the polynomial $\mathbf{w}_i$ such that $\mathbf{p}$ is uniform random, i.e. $\mathbf{p}_i$ equals to $2^t \cdot \bar{\mathbf{b}}_i + \mathbf{w}_i$ except for $\mathbf{p}_1$ which additionally contains $\mathbf{g}$ for decryption. The ciphertext is computed as a module-LWE instances with the centered secret $\mathbf{s} = (\mathbf{s}_1 - [p], \ldots, \mathbf{s}_\ell - [p])$. To enable recovery of s_1 we adjust the last ciphertext sample via subtraction of $\mathbf{g} \cdot [p]$, which vanishes modulo q for $\mathbf{g} = 2^{k-1}$ and $p = 2^x$ with $x \geq 1$.

For $n = 256$ at a post quantum security level of 256 bits, we need $s_1 \leftarrow \mathcal{R}_4$ and $\mathbf{g} = 2^{k-2}$. The coefficients are recovered with the alternative subroutine $\mathsf{Recover}'$.

Theorem 2.1 *In the random oracle model, assume that there exists a PPT-adversary $\mathcal{A}$ against the scheme, then there exists a reduction $\mathcal{D}$ that breaks $\mathsf{MLWE}_{\ell,\ell+1,p}$ such that $\mathsf{Adv}^{\mathsf{CPA}}_{\mathsf{Kindi}(\mathcal{A})} \leq 3\mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell+1,p}(\mathcal{D})$.*

Proof. We proceed in a sequence of hybrids. Thus, let $\mathcal{H}_0$ be the real $\mathsf{IND-CPA}$ game. In $\mathcal{H}_1$, the MLWE instance $\mathbf{b} = \mathbf{A} \cdot \mathbf{r} + \mathbf{e}$ in the key generation step is changed to a uniform random value. Since $q = 2^k$, the $k-t$ most significant bits of the coefficients in $\mathbf{b}$ form the coefficients of the compressed public key. Hence, $\bar{\mathbf{b}}$ remains uniform random in $\mathcal{R}_{2^{k-t}}^{\ell}$. If there exists

Algorithm 3: $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}(\mathsf{pk}, \mathsf{msg} = \{0,1\}^{n(\ell+1)\log 2p}; \mathsf{coins} = \perp \text{ or } s_1 \in \mathcal{R}_2)$:

$s_1 \leftarrow \mathcal{R}_2$
$\mathbf{A} \leftarrow \mathsf{Shake}(\mu)$
$\bar{\mathbf{p}} = \mathsf{Decompress}_1(\bar{\mathbf{b}}, t)$
$\bar{\mathbf{w}}, \bar{u}, \bar{\mathbf{s}}_1, (\mathbf{s}_2, \ldots, \mathbf{s}_\ell) \in \mathcal{R}_{2^t}^{\ell} \times \{0,1\}^{n(\ell+1)\log 2p} \times \mathcal{R}_p \times \mathcal{R}_{2p}^{\ell-1} \leftarrow \mathsf{G}(s_1) := \mathsf{Shake}(s_1)$
$\mathbf{w} = (\bar{\mathbf{w}}_1 - [2^{t-1}], \ldots, \bar{\mathbf{w}}_\ell - [2^{t-1}])$
$\mathbf{p} = (\bar{\mathbf{p}}_1 + \mathbf{w}_1 + \mathbf{g}, \bar{\mathbf{p}}_2 + \mathbf{w}_2, \ldots, \bar{\mathbf{p}}_\ell + \mathbf{w}_\ell)$
$\mathbf{s} = (s_1 + 2 \cdot \bar{\mathbf{s}}_1 - [p], \mathbf{s}_2 - [p], \ldots, \mathbf{s}_\ell - [p])^\top$
$u = \bar{u} \oplus \mathsf{msg}$
$\mathbf{u} = \mathsf{Encode}(u)$
$\mathbf{e} = (\mathbf{u}_1 - [p], \ldots, \mathbf{u}_\ell - [p])^\top,\ e = \mathbf{u}_{\ell+1} - [p]$
$(\mathbf{c}, c)^\top = (\mathbf{A}^\top \cdot \mathbf{s} + \mathbf{e}, \mathbf{p} \cdot \mathbf{s} + \mathbf{g} \cdot [p] + e) \in \mathcal{R}_q^{\ell+1}$
return $(\mathbf{c}, c)$

Algorithm 4: $\mathsf{Decompress}(\mathbf{x} \in \mathcal{R}_q^{\ell+1}, t \in \mathbb{N})$:

$\mathbf{b} = 2^t \cdot \mathbf{x}$

Algorithm 5: $\mathsf{Encode}(u \in \{0,1\}^{n(\ell+1)\log 2p})$:

Pack $\log 2p$ bits of u into each coefficient of $\mathbf{u}_i \in \mathcal{R}$ for $1 \le i \le \ell + 1$.
Each $\mathbf{u}_i \in \mathcal{R}$ contains $n \cdot \log 2p$ bits.
$\mathbf{u} = (\mathbf{u}_1, \ldots, \mathbf{u}_{\ell+1})$
return u

an adversary that can distinguish the hybrids $\mathcal{H}_0$ and $\mathcal{H}_1$, then there exists a reduction $\mathcal{D}_0$ that can distinguish $\mathsf{MLWE}_{\ell,\ell,p}$ from uniform such that $\mathsf{Adv}_{\mathcal{H}_0,\mathcal{H}_1}(\mathcal{D}_0) \le \mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell,p}(\mathcal{D}_0)$. In the hybrid $\mathcal{H}_2$, the elements $\bar{\mathbf{w}}, \bar{u}, \bar{\mathbf{s}}_1, (\mathbf{s}_2, \ldots, \mathbf{s}_\ell)$ are replaced by uniform random elements (RO) such that $\mathbf{e}, \mathbf{e}_{\ell+1}, \mathbf{s}$ are again uniform random. We note that $\mathbf{p} \in \mathcal{R}_q^\ell$ in hybrid $\mathcal{H}_2$ is uniform random too even after compression/decompression due to the additional polynomial $\mathbf{w}_i := \bar{\mathbf{w}}_i - [2^{t-1}]$ with uniform random coefficients from $\{-2^{t-1}, \ldots, 2^{t-1} - 1\}$. Thus, we obtain $\mathsf{Adv}_{\mathcal{H}_1,\mathcal{H}_2}(\mathcal{D}_1) \le \mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell,p}(\mathcal{D}_1)$ for the chosen parameters. Finally, in $\mathcal{H}_3$ the ciphertexts $\mathbf{c} = \mathbf{A}^\top \cdot \mathbf{s} + \mathbf{e}$ and $c = \mathbf{p} \cdot \mathbf{s} + \mathbf{g} \cdot [p] + e$ are replaced by uniform random elements. If there exists an adversary that can distinguish the hybrids $\mathcal{H}_2$ and $\mathcal{H}_3$, then there exists a reduction $\mathcal{D}_2$ that can distinguish $\mathsf{MLWE}_{\ell,\ell+1,p}$ from uniform such that $\mathsf{Adv}_{\mathcal{H}_2,\mathcal{H}_3}(\mathcal{D}_2) \le \mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell+1,p}(\mathcal{D}_2)$.

We now analyze the advantage of an adversary in $\mathcal{H}_0$, which is given by

$$\begin{aligned} \mathsf{Adv}_{\mathcal{H}_0}(\mathcal{A}) := \mathsf{Adv}^{\mathsf{CPA}}_{\mathsf{Kindi}}(\mathcal{A}) &= |P[b = b' \text{ in } \mathcal{H}_0] - 1/2| \\ &\le \mathsf{Adv}_{\mathcal{H}_0,\mathcal{H}_1}(\mathcal{D}) + \mathsf{Adv}_{\mathcal{H}_1,\mathcal{H}_2}(\mathcal{D}) + \mathsf{Adv}_{\mathcal{H}_2,\mathcal{H}_3}(\mathcal{D}) \le 3\mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell+1,p}(\mathcal{D}) \,. \end{aligned}$$

2.1.4 Decryption

The decryption engine works similar to the encryption engine. First, the least significant bit of the coefficients of $\mathbf{s}_1$ are recovered via $s_1 = \mathsf{Recover}(\mathbf{v}) \in \mathcal{R}_2$ and $\mathbf{v} = c - \mathbf{c} \cdot \mathbf{r}^\top = 2^{k-1}\mathbf{s}_1 + \mathbf{d} \bmod q = 2^{k-1}s_1 + \mathbf{d} \bmod q$ for some small $\|\mathbf{d}\|_\infty$. This recovery function has also been used for instance in [4] avoiding if-else checks. From s_1 the vectors $\bar{u}$ and $\mathbf{s}_i$ are derived. We obtain $(\mathbf{u}_1 - [p], \ldots, \mathbf{u}_{\ell+1} - [p]) = (\mathbf{e}, e) = (\mathbf{c} - \mathbf{A}^\top \cdot \mathbf{s}, c - \mathbf{p} \cdot \mathbf{s}) \bmod q$. The decoder $\mathsf{Decode}(\mathbf{u})$ maps the set of polynomials with coefficients in the range $[0, 2p]$ to a bit string such that the bit string $\mathsf{msg} = \mathsf{Decode}(\mathbf{u}) \oplus \bar{u}$ returns the message. The second output value s_1 is necessary for the KEM.

Algorithm 6: $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Decrypt}(\mathsf{sk}, (\mathbf{c}, c))$:

$\mathbf{A} \leftarrow \mathsf{Shake}(\mu)$
$\mathbf{r} \leftarrow \mathsf{Shake}_p(\gamma)$
$\bar{\mathbf{p}} = \mathsf{Decompress}(\bar{\mathbf{b}}, t)$
$\mathbf{v} = c - \mathbf{c} \cdot \mathbf{r}^\top$
$s_1 = \mathsf{Recover}(\mathbf{v}) \in \mathcal{R}_2$
$\bar{\mathbf{w}}, \bar{u}, \bar{\mathbf{s}}_1, (\mathbf{s}_2, \ldots, \mathbf{s}_\ell) \in \mathcal{R}_{2^t}^\ell \times \{0,1\}^{n(\ell+1)\log 2p} \times \mathcal{R}_p \times \mathcal{R}_p^{\ell-1} \leftarrow \mathsf{Shake}(s_1)$
$\mathbf{w} = (\bar{\mathbf{w}}_1 - [2^{t-1}], \ldots, \bar{\mathbf{w}}_\ell - [2^{t-1}])$
$\mathbf{p} = (\bar{\mathbf{p}}_1 + \mathbf{w}_1 + \mathbf{g}, \bar{\mathbf{p}}_2 + \mathbf{w}_2, \ldots, \bar{\mathbf{p}}_\ell + \mathbf{w}_\ell)$
$\mathbf{s} = (s_1 + 2 \cdot \bar{\mathbf{s}}_1 - [p], \mathbf{s}_2 - [p], \ldots, \mathbf{s}_\ell - [p])^\top$
$(\mathbf{e}, e) = (\mathbf{u}_1 - [p], \ldots, \mathbf{u}_{\ell+1} - [p]) = (\mathbf{c} - \mathbf{A}^\top \cdot \mathbf{s}, c - \mathbf{p} \cdot \mathbf{s}) \bmod q$
$\mathsf{msg} = \mathsf{Decode}(\mathbf{u}) \oplus \bar{u}$
return (msg, s_1)

Algorithm 7: $\mathsf{Decode}(\mathbf{u} \in \mathcal{R}_{2p}^{\ell+1})$:

Concatenate the least significant $\log 2p$ bits of all coefficients in $\mathbf{u}$ into u.
return $u \in \{0,1\}^{n(\ell+1)\log 2p}$

Algorithm 8: $\mathsf{Recover}(\mathbf{v} \in \mathcal{R}_q)$:

Let v_i be in $\{0, \ldots, q-1\}$.
For $i = 1$ to n **do**
    $b_i = \lfloor v_i / 2^{k-1} \rceil \bmod 2$
return $b \in \mathcal{R}_2$

Algorithm 9: $\mathsf{Recover}'(\mathbf{v} \in \mathcal{R}_q)$:

Let v_i be in $\{0, \ldots, q-1\}$.
For $i = 1$ to n **do**
    $b_i = \lfloor v_i / 2^{k-2} \rceil \bmod 4$
return $b \in \mathcal{R}_4$

Theorem 2.2 *Let the coefficients of* $\mathbf{r}_j, \mathbf{r}'_j, \mathbf{s}_j - [p]$ *and* $\mathbf{e}_i = \mathbf{u}_i - [p]$ *be uniformly distributed from* $\{-p, \ldots, p-1\}$ *for* $1 \leq j \leq \ell$ *and* $1 \leq i \leq \ell + 1$. *Further, let the coefficients of* $\mathbf{w}_j$ *be sampled uniformly at random from* $\{-2^{t-1}, \ldots, 2^{t-1} - 1\}$ *for* $1 \leq j \leq \ell + 1$. *Then, for*

$$\delta := P[\|\mathbf{x}^\top \cdot \mathbf{s} + e - \mathbf{e} \cdot \mathbf{r}^\top\|_\infty \geq q/4]$$

the algorithm is $(1 - \delta)$ *correct, where* $\mathbf{x} = \mathbf{w}^\top + \mathsf{Decompress}(\mathsf{Compress}(\mathbf{A} \cdot \mathbf{r} + \mathbf{r}')) - \mathbf{A} \cdot \mathbf{r}$.

Proof. We choose the parameters p and q such that s_1 is correctly recovered. Let $\mathbf{s} = (\mathbf{s}_1 - [p], \ldots, \mathbf{s}_\ell - [p])$, then

$$\begin{aligned}
\|\mathbf{v} - 2^{k-1} s_1\|_\infty &= \|c - \mathbf{c}^\top \cdot \mathbf{r} - 2^{k-1} s_1\|_\infty \\
&= \|\mathbf{p} \cdot \mathbf{s} + \mathbf{g} \cdot [p] + e - (\mathbf{A}^\top \cdot \mathbf{s} + \mathbf{e})^\top \cdot \mathbf{r} - 2^{k-1} s_1\|_\infty \\
&= \|\mathbf{g} \cdot (\mathbf{s}_1 - [p]) + \mathbf{g} \cdot [p] + e + \mathbf{x}^\top \cdot \mathbf{s} - \mathbf{e}^\top \cdot \mathbf{r} - 2^{k-1} s_1\|_\infty \\
&= \|\mathbf{x}^\top \cdot \mathbf{s} + e - \mathbf{e}^\top \cdot \mathbf{r}\|_\infty < q/4 \,.
\end{aligned}$$

We note that in case $p = 2^\kappa$ for $\kappa > 0$, then the term $\mathbf{g} \cdot [p]$ vanishes and is not needed in the computation. We note that in case $\mathbf{g} = \lfloor q/2 \rfloor$ for a prime modulus $q > 2$ the difference $\mathbf{g} \cdot \|\mathbf{s}_1 - 2^{k-1} s_1\|_\infty \leq 2p$ would be very small and not vanish completely.

For $n = 256$ and $\lambda = 256$ (key size 2λ bits resisting Grover's search), we have $\mathbf{g} = 2^{k-2}$. We define the correctness requirement with respect to the bound $q/8$ rather than $q/4$, i.e.

$$\delta := P[\|\mathbf{x}^\top \cdot \mathbf{s} + e - \mathbf{e} \cdot \mathbf{r}^\top\|_\infty \geq q/8] \,.$$

2.2 Trapdoor-based CCA-secure KEM $\mathsf{KINDI}_{\mathsf{KEM}}$ with Uniform Errors

The key encapsulation mechanism $\mathsf{KINDI}_{\mathsf{KEM}}$ has the same parameter space as $\mathsf{KINDI}_{\mathsf{CPA}}$. We adopt the transformation [9] in order to realize a KEM by our construction. In fact, we already indicated in Section 2.1 that some of the transformations are already encompassed in our construction. Thus, the construction gets very simple.

Algorithm 10: $\mathsf{QEncaps}(\mathsf{pk})$:

$m \leftarrow \mathsf{M}$
$\mathbf{c} := \mathsf{Enc}(\mathsf{pk}, m, \mathsf{G}(m))$
$d := \mathsf{H}(m)$
$K := \mathsf{H}'(m, \mathbf{c})$
return $(K, \mathbf{c}, d)$

Algorithm 11: $\mathsf{QDecaps}(\mathsf{sk}, \mathbf{c})$:

$m' \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathbf{c})$
$\mathbf{c}' := \mathsf{Enc}(\mathsf{pk}, m', \mathsf{G}(m'))$
if $\mathbf{c}' = \mathbf{c} \wedge \mathsf{H}(m') = d$
    return $K := \mathsf{H}'(m', \mathbf{c})$
else
    return $K := \mathsf{H}'(s, \mathbf{c})$

The generic construction secure in the quantum random oracle model is given by the following two algorithms, where $\mathsf{G}, \mathsf{H}, \mathsf{H}'$ denote random oracles.

We state the theorem for tight security, when the computation and check of d is omitted. For that we combine the security implications [9] $\mathsf{IND-CPA} \implies \mathsf{OW-PCVA}$ and $\mathsf{OW-PCVA} \implies \mathsf{IND-CCA}$.

Theorem 2.3 *Let* M *denote the message space. Furthermore, for any* $\mathsf{IND-CCA}$ *adversary that makes* q_{G} *queries to the random oracle* G*,* q_{H} *queries to the random oracle* H*, and* q_D *queries to the decapsulation oracle, there exists an* $\mathsf{IND-CPA}$ *adversary such that*

$$\mathsf{Adv}^{\mathsf{IND-CCA}}_{\mathsf{KEM}}(\mathcal{B}) \leq q_G \cdot \delta + \frac{2 \cdot q_G + q_H}{|\mathsf{M}|} + 3\mathsf{Adv}^{\mathsf{IND-CPA}}_{\mathsf{PKE}}(\mathcal{A}) \tag{1}$$

and the running time of A *is about that of* B*.*

This reduction is tight. Thus, we can tightly reduce it from $\mathsf{MLWE}_{\ell,\ell+1,p}$.

For security in the quantum random oracle model, which requires d, there is an alternative theorem in [9].

2.3 Key Generation

The key generation step just outputs the keys of $\mathsf{KINDI}_{\mathsf{CPA}}$.

Algorithm 12: $\mathsf{KINDI}_{\mathsf{CCA-KEM}}.\mathsf{KeyGen}(1^n, p, k, t, \ell)$:

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{KeyGen}(1^n, p, k, t, \ell)$
return $(\mathsf{pk}, \mathsf{sk})$

2.4 Encapsulation

The encapsulation mechanism slightly differs from the generic construction. We do not need to input $\mathsf{G}(s_1)$ but rather just s_1. In fact, the encryption engine $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}$ does this implicitly within the algorithm as it applies $\mathsf{G}(s_1)$ to deterministically deduce the secret and error polynomials. At the same time s_1 is encrypted (see $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}$). As in the generic construction, we compute the key $K \in \{0,1\}^{2\lambda}$ and $d \in \{0,1\}^{2\lambda}$. Due to the fact that KINDI has a large message container, we can also encrypt d and send a ciphertext that is as large as in $\mathsf{KINDI}_{\mathsf{CPA}}$. Finally, the ciphertext is output. We implement the different random oracles as $\mathsf{H}(s_1) := \mathsf{Shake}(s_1||\mathsf{padding}), \mathsf{H}'(s_1, \mathbf{c}) := \mathsf{Shake}(s_1, \mathbf{c})$ and $\mathsf{G}(s_1) := \mathsf{Shake}(s_1)$, where we use a one byte $\mathsf{padding} = 4$.

Algorithm 13: $\mathsf{KINDI}_{\mathsf{CCA-KEM}}.\mathsf{Encaps}(\mathsf{pk})$:

$s_1 \leftarrow \{0,1\}^n$
$d \leftarrow \mathsf{H}(s_1)$
$(\mathbf{c}, c)^\top \leftarrow \mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}(\mathsf{pk}, d; s_1)$
$K \leftarrow \mathsf{H}'(s_1, (\mathbf{c}, c))$
return $(K, (\mathbf{c}, c))$

We note that by setting $d = \mathbf{0}$ (or something deterministic) we obtain the CCA-secure scheme in the random oracle model. For the quantum random oracle variant, d is retrieved during decryption.

2.5 Decapsulation

The decapsulation mechanism implicitly performs many steps of the generic construction within $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Decrypt}$. For instance, it is not required to encrypt s_1' again once recovered from the ciphertext as we prove below. It is only necessary, to check that the value $d = \mathsf{H}(s_1')$ is equal to the recovered value d' . In case, the check is correct the key is deduced, otherwise it outputs a random key for some uniform random $s \in \{0,1\}^{2\lambda}$.

In the following lemma we show that it suffices to only check $d' = d$ in order to satisfy the conditions from [9] for key decapsulation.

Lemma 2.4 *If $d' = d$ is satisfied, then $s_1' = s_1$ and $(\mathbf{c}, c) = \mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}(\mathsf{pk}, d; s_1')$.*

Proof. If $d' = d$, then $\mathsf{G}(s_1) = \mathsf{G}(s_1')$, which means that s_1' has been correctly recovered. As

Algorithm 14: $\mathsf{KINDI}_{\mathsf{CCA-KEM}}.\mathsf{Decaps}(\mathsf{sk}, (\mathbf{c}, c))$:

$(d', s_1') \leftarrow \mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Decrypt}(\mathsf{sk}, (\mathbf{c}, c))$
if $d' = d := \mathsf{H}(s_1')$
    return $\mathsf{H}'(s_1', (\mathbf{c}, c))$
else
    return $\mathsf{H}'(s, (\mathbf{c}, c))$

a result, we have that $\mathbf{c}, c$ is uniquely generated from $\mathbf{s}$ and $\mathbf{u} = \mathsf{Encode}(\bar{u} \oplus d)$ with

$$\bar{\mathbf{w}}, \bar{u}, \bar{\mathbf{s}}_1, (\mathbf{s}_2, \ldots, \mathbf{s}_\ell) \leftarrow \mathsf{G}(s_1')$$

and $\mathbf{s}_1 = s_1 + 2\bar{\mathbf{s}}_1$.

We stress that it is possible to just check that $d' = \mathbf{0}$, if in the random oracle model d is set to $\mathbf{0}$ during encapsulation (see Section 5.5).

Furthermore, we note that if $\mathsf{KINDI}_{\mathsf{CPA}}$ is $(1 - \delta)$ correct, then so is the resulting $\mathsf{KINDI}_{\mathsf{CCA-KEM}}$.

3 List of parameter sets (part of 2.B.1)

3.1 Parameter set $\mathsf{encrypt/KINDI-256-3-4-2}$

Public key encryption with $n = 256,\ \ell = 3,\ p = 4,\ t = 2$ and $q = 2^{14}$.

3.2 Parameter set $\mathsf{encrypt/KINDI-512-2-2-2}$

Public key encryption with $n = 512,\ \ell = 2,\ p = 2,\ t = 2$ and $q = 2^{13}$.

3.3 Parameter set $\mathsf{encrypt/KINDI-512-2-4-1}$

Public key encryption with $n = 512,\ \ell = 2,\ p = 4,\ t = 1$ and $q = 2^{14}$.

3.4 Parameter set $\mathsf{encrypt/KINDI-256-5-2-2}$

Public key encryption with $n = 256,\ \ell = 5,\ p = 2,\ t = 2$ and $q = 2^{14}$.

3.5 Parameter set $\mathsf{encrypt/KINDI-512-3-2-1}$

Public key encryption with $n = 512,\ \ell = 3,\ p = 2,\ t = 1$ and $q = 2^{13}$.

3.6 Parameter set $\mathsf{kem/KINDI-256-3-4-2}$

Key encapsulation mechanism with $n = 256,\ \ell = 3,\ p = 4,\ t = 2$ and $q = 2^{14}$.

3.7 Parameter set $\mathsf{kem/KINDI-512-2-2-2}$

Key encapsulation mechanism with $n = 512,\ \ell = 2,\ p = 2,\ t = 2$ and $q = 2^{13}$.

3.8 Parameter set $\mathsf{kem/KINDI-512-2-4-1}$

Key encapsulation mechanism with $n = 512,\ \ell = 2,\ p = 4,\ t = 1$ and $q = 2^{14}$.

3.9 Parameter set $\mathsf{kem/KINDI-256-5-2-2}$

Key encapsulation mechanism with $n = 256,\ \ell = 5,\ p = 2,\ t = 2$ and $q = 2^{14}$.

3.10 Parameter set $\mathsf{kem/KINDI-512-3-2-1}$

Key encapsulation mechanism with $n = 512,\ \ell = 3,\ p = 2,\ t = 1$ and $q = 2^{13}$.

4 Design rationale (part of 2.B.1)

We propose a simple and highly efficient trapdoor-construction, where the public key $\mathbf{B}$ represents an MLWE instance endowed with a trapdoor $\mathbf{T}$. Roughly spoken, ciphertexts are generated as MLWE instances $\mathbf{B}^\top \cdot \mathbf{s} + \mathbf{e}$, where $\mathbf{s}$ and $\mathbf{e} = \mathbf{u} \oplus \mathsf{msg}$ are vectors of uniform random polynomials. The message is simply xored to the error polynomials such that large amounts of data can be encrypted at very competitive parameters, for instance useful in sign-then-encrypt scenarios or for the transmission of encrypted key bundles etc. Different to other proposals, the decryption engine can recover all $\mathbf{s}, \mathbf{e} = \mathbf{u} \oplus \mathsf{msg}$ and thus msg by means of the trapdoor $\mathbf{T}$. This further allows to inspect the secret and error polynomials and use all of the information stored therein. Our proposal not only encrypts arbitrary messages, but also outputs by construction a uniform random string s_1 for free that can act as a key for a symmetric key cipher as required in a KEM. In other words, the random coins used to encrypt the message can be recovered by use of the trapdoor.

4.1 Implementation of Polynomial Multiplication and the FFT/NTT

For multiplication of polynomials we make extensive use of the Fast Fourier Transformation, which is a divide-and-conquer algorithm and outputs the result in $O(\log n)$ steps. We precompute tables containing powers of the complex root of unity used for the FFT. Our AVX2 optimized variant processes 4 coefficients at once.

Different to our previous submission we adapted the FFT following the work [18]. The work suggests a variant for the FFT/NTT, wich can be applied in a straightforward manner. By this modification we omit the use of the costly bit-reversal function, which in turn improves the speed significantly. Instead of using the bit-reversal function, the forward and inverse FFT have been redesigned to match [18]. In addition, due the linearity of the FFT we reduce the number of the inverse FFT. To this end, we use the following equation

$$\sum_i \mathsf{FFT}^{-1}(\mathsf{FFT}(\mathbf{a}_i) \circ \mathsf{FFT}(\mathbf{b}_i)) = \mathsf{FFT}^{-1}(\sum_i \mathsf{FFT}(\mathbf{a}_i) \circ \mathsf{FFT}(\mathbf{b}_i)),$$

a fact that is simple but effective.

In case the NTT is used for a prime modulus $q \equiv 1 \bmod 2n$, one can use the framework of [18] directly.

4.2 Practical Instantiation of Random Oracles

We choose to implement random oracles with the FIPS 202 standardized Shake. It is also used in other lattice-based schemes such as Frodo and Kyber. The matrix $\mathbf{A}$ is generated by use of a $\mathsf{PRNG} \in \{\mathsf{Shake128}, \mathsf{Shake256}\}$ and a uniform random input string μ of size 2λ bits. In fact, we only use Shake256 except for one parameter set namely $n = 256$ and $\ell = 3$. For the optimized variants we use the Keccak code package[1] that allows via AVX2 to compute 4 independent streams of random values on 4 inputs of the same length. Each input $\mathsf{input}_i = \mu||i$ is obtained by the seed concatenated with a one byte number $0 \leq i \leq 3$ resulting in independent uniform random streams. Thus, we do not store $\mathbf{A}$ but rather derive it from μ each time we need it. Since we work modulo 2^k, each k bit chunk is considered as a little endian integer representing one coefficient. Similarly, we generate uniform secrets and errors just from $\mathsf{Shake}(s_1)$. Our choice for p to be a power of two allows us to proceed as with the matrix $\mathbf{A}$ taking the required bits from Shake for $\bar{\mathbf{w}}, \bar{u}, \bar{\mathbf{s}}_\mathbf{1}$ and $\mathbf{s}_2, \ldots, \mathbf{s}_\ell$. For the random oracle G we use the same padding scheme in our optimized variant. The message is xored to $\bar{u}$ prior to encoding.

However, for the computation of $d \in \{0,1\}^{2\lambda}$ in the KEM we append 4 to s_1 before invoking $\mathsf{H} := \mathsf{PRNG}$. We implement $\mathsf{H}' := \mathsf{PRNG}$ without any padding in the reference implementation. For the optimized variant we split the large ciphertext into 4 inputs and invoke Shake outputting 4 streams of size 2λ bits each. The outputs are subsequently concatenated with s_1 serving as input to one regular Shake call.

4.3 Constant Time Implementation

The scheme execution does not depend on the secrets. This includes the error recovery subroutine and all the operations involved. We heavily rely on shake as a pseudorandom number generator, which outputs random bits in constant time. In the previous submission, we used rounding functions of the C math library, which makes use of branches. This observation has also been made in [10]. However, in the current version the error recovery subroutine is implemented by use of only elementary operations in C such as Ands, shifts and Ors without any secret-dependent branches.

[1] https://keccak.team/

4.4 Further Implementation Details

We mark the end of a message by a one byte padding. Modulo $q = 2^k$ operations are obtained almost for free as it just requires to take the k least significant bits.

The ciphertexts, compressed public keys and secret keys are stored in little endian format. The $k - t$ bit coefficients of the compressed public key are appended to each other before the seed μ is concatenated to the resulting string. For the secret key we proceed similarly.

The scheme can also be implemented with a prime modulus such that the NTT is applicable. In this case, the compression and decompression function slightly differ from the $q = 2^k$ case as pointed out in the respective sections.

5 Detailed performance analysis (2.B.2)

5.1 Description of platform

We implemented both our CPA/CCA secure schemes on a machine that is specified by an Intel Core i5-6200U processor (Skylake) operating at 2.3GHz and 8GB of RAM running on one core. We used Ubuntu 17.10 64-bit (Kernel 4.13.0-17) and gcc version 7.2.0 with compilation flags

- Reference: `-fomit-frame-pointer -Ofast -march=native`
- AVX Version: `-fomit-frame-pointer -Ofast -msse2avx -mavx2 -march=native`

5.2 Time

The following measurements are for `kem` and `encrypt`. The difference in running times between `kem` and `encrypt` stems from 3 additional invocations of Shake for `kem`. We took the average over 1 Mio measurements.

`Kindi-256-3-4-2`:

- Reference Implementation:
  - Key generation in cycles: 111416
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(130204,142817)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(158467,177532)
- AVX Implementation:
  - Key generation in cycles: 68319

  - Encryption/Encaps in cycles: (encrypt,kem)=(78787,93652)
  - Decryption/Decaps in cycles: (encrypt,kem)=(93257,113653)

We note that in case we use Shake256 key generation increases by about 3000-6000 cycles, encryption by about 3000-8000 cycles, encaps by about 8000-10000, decryption by 5000-6000 cycles and decaps by about 7000-10000 cycles (for the reference implementation and AVX implementation). The decryption failure rate is here $\delta = 2^{-192}$. In the AVX implementation, encryption is carried out at a speed of 320 cycles per message byte or 68 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 396 cycles per message byte or 84 cycles per ciphertext byte.

Kindi-512-2-2-2:

- Reference Implementation:
  - Key generation in cycles: 118600
  - Encryption/Encaps in cycles: (encrypt,kem)=(153807,178471)
  - Decryption/Decaps in cycles: (encrypt,kem)=(206709,228405)
- AVX Implementation:
  - Key generation in cycles: 75651
  - Encryption/Encaps in cycles: (encrypt,kem)=(96450,122521)
  - Decryption/Decaps in cycles: (encrypt,kem)=(120017,146319)

The decryption failure rate is here $\delta = 2^{-284}$. In the AVX implementation, encryption is carried out at a speed of 373 cycles per message byte or 57 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 488 cycles per message byte or 75 cycles per ciphertext byte.

Kindi-512-2-4-1:

- Reference Implementation:
  - Key generation in cycles: 126369
  - Encryption/Encaps in cycles: (encrypt,kem)=(158942,190550)
  - Decryption/Decaps in cycles: (encrypt,kem)=(209795,232604)
- AVX Implementation:
  - Key generation in cycles: 82020
  - Encryption/Encaps in cycles: (encrypt,kem)=(97694,127527)
  - Decryption/Decaps in cycles: (encrypt,kem)=(129559,159846)

The decryption failure rate is here $\delta = 2^{-165}$. In the AVX implementation, encryption is carried out at a speed of 248 cycles per message byte or 53 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 324 cycles per message byte or 69 cycles per ciphertext byte.

`Kindi-256-5-2-2`:

- Reference Implementation:
  - Key generation in cycles: 268648
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(296470,323689)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(344806,374417)
- AVX Implementation:
  - Key generation in cycles: 160556
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(173906,195571)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(198187,221439)

The decryption failure rate is here smaller than $\delta = 2^{-216}$.In the AVX implementation, encryption is carried out at a speed of 731 cycles per message byte or 104 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 857 cycles per message byte or 122 cycles per ciphertext byte.

`Kindi-512-3-2-1`:

- Reference Implementation:
  - Key generation in cycles: 223902
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(268237,299651)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(341753,365118)
- AVX Implementation:
  - Key generation in cycles: 140641
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(164223,197927)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(194572,240839)

The decryption failure rate is here $\delta = 2^{-276}$. In the AVX implementation, encryption is carried out at a speed of 502 cycles per message byte or 77 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 636 cycles per message byte or 97 cycles per ciphertext byte.

5.3 Space

The secret key, public key and ciphertext sizes can be computed straightforwardly. The sizes of the CPA-secure scheme and the KEM are equal except for the ciphertext size of the KEM, which is larger by 2λ. In the following, we indicate the size for the $\mathsf{KINDI}_{\mathsf{CPA}}$.

The ciphertext size is $n(\ell+1)k/8$ bytes, whereas the public key pk amounts to $(n\ell(k-t)+2\lambda)/8$ bytes including the seed μ for the matrix $\mathbf{A}$. The secret key amounts to $(n\ell(k-t)+4\lambda)/8$ bytes including the size of the public key. The message size for encryption amounts to $n\ell(\log 2p)/8-1$ bytes, where one byte is used for the padding. Thus, we obtain the following.

`Kindi-256-3-4-2`:

- Ciphertext size: 1792 bytes
- Public key size: 1184 bytes
- Secret key size: 1248 bytes
- Message size: 383 bytes
- Message expansion factor: 4.7

`Kindi-512-2-2-2`:

- Ciphertext size: 2496 bytes
- Public key size: 1456 bytes
- Secret key size: 1584 bytes
- Message size: 383 bytes
- Message expansion factor: 6.5

`Kindi-512-2-4-1`:

- Ciphertext size: 2688 bytes
- Public key size: 1728 bytes
- Secret key size: 1856 bytes
- Message size: 575 bytes
- Message expansion: 4.7

`Kindi-256-5-2-2`:

- Ciphertext size: 2688 bytes
- Public key size: 1984 bytes
- Secret key size: 2112 bytes
- Message size: 383 bytes
- Message expansion factor: 7

`Kindi-512-3-2-3`:

- Ciphertext size: 3328 bytes
- Public key size: 2368 bytes
- Secret key size: 2496 bytes
- Message size: 511 bytes
- Message expansion: 6.5

5.4 How parameters affect performance

The main parameters governing the performance and security level of the schemes are n, ℓ, q and $p = \mathsf{rsec}$. For increasing parameters n, p or ℓ the security of the overall system is increased while simultaneously decreasing the performance level via n and ℓ or increasing the secret key size at a higher decyption failure rate via p. For increasing q and all other parameters being fixed, the decryption failure rate and the security of the system decrease while the ciphertext and public key sizes increase.

5.5 Optimizations

We generated the secret key from the secret seed γ of size 2λ bits during decryption. In pricipal, it is also possible to generate public keys just by use of the secret seed γ and the public seed μ. If in applications, the running time is of interest, then all keys $\mathbf{r}$, $\bar{\mathbf{b}}$, and the matrix $\mathbf{A}$ are stored rather than the seeds. In case, key sizes are more important than running time, then one may store only the seeds and generate the respective keys during decryption or decapsulation. Furthermore, it is possible to compress the ciphertext in case the message container is not fully exhausted, i.e. one can compress the coefficients of $\mathbf{c}_i$ if the respective error terms do not contain message bits. For the simplicity of our construction, we did not include these modifications. As already explained in the encapsulation mechanism, we can avoid the transmission of d in plain and instead check that $d' = \mathsf{H}(s_1')$, where s_1' and d' are recovered during decryption.

6 Expected strength (2.B.4) in general

6.1 Security definitions

The KEM is designed for IND-CCA2 security and PKE ensures CPA security. See Section 7 for quantitative estimates of the security of specific parameter sets.

6.2 Rationale

See Section 8 for an analysis of known attacks. This analysis also presents the rationale for these security estimates.

7 Expected strength (2.B.4) for each parameter set

7.1 Parameter set $\mathsf{encrypt/KINDI-256-3-4-2}$

Classical security	PQ-security	Category
181	164	3

7.2 Parameter set $\mathsf{encrypt/KINDI-512-2-2-2}$

Classical security	PQ-security	Category
229	207	5

7.3 Parameter set $\mathsf{encrypt/KINDI-512-2-4-1}$

Classical security	PQ-security	Category
255	232	5

7.4 Parameter set $\mathsf{encrypt/KINDI-256-5-2-2}$

Classical security	PQ-security	Category
270	251	5

7.5 Parameter set $\mathsf{encrypt/KINDI-512-3-2-1}$

Classical security	PQ-security	Category
365	330	5

7.6 Parameter set kem/KINDI − 256 − 3 − 4 − 2

Classical security	PQ-security	Category
181	164	3

7.7 Parameter set kem/KINDI − 512 − 2 − 2 − 2

Classical security	PQ-security	Category
229	207	5

7.8 Parameter set kem/KINDI − 512 − 2 − 4 − 1

Classical security	PQ-security	Category
255	232	5

7.9 Parameter set kem/KINDI − 256 − 5 − 2 − 2

Classical security	PQ-security	Category
270	251	5

7.10 Parameter set kem/KINDI − 512 − 3 − 2 − 1

Classical security	PQ-security	Category
365	330	5

8 Analysis of known attacks (2.B.5)

We give a summary of the most relevant attacks against our MLWE based encryption schemes. To this end, MLWE instances are considered as regular LWE instances of dimension $\ell \cdot n$ with $(\ell + 1) \cdot n$ samples. To date, there exist no better cryptanalytic algorithms to attack MLWE for concrete parameters than the ones on regular LWE. The best way to attack our encryption schemes is to mount a key recovery attack or to consider attacks against the ciphertext. Since we chose the same parameter for the ciphertext and public key, we need only to consider attacks against the ciphertext since it further contains an additional ring sample as compared to the public key. We apply the conservative methodology of [3] in order to estimate the attack complexity or to choose reasonable parameters. Currently, the best way to attack the system is carried out with the primal and dual attacks using BKZ. This lattice reduction algorithm reduces the basis of the lattice using polynomial calls to an SVP oracle in a smaller dimension. For the computation of the attack complexity only one call to the SVP oracle is

taken into account. All other factors are also removed leading to very conservative estimates.

In the classical setting the best-known attack bound is $2^{0.292b}$ deduced from lattice sieving whereas in the post-quantum setting the SVP solver requires $2^{0.265b}$ with quantum sieving. Here b denotes the block size. The best plausible security estimates rely on building lists of size $2^{0.2075b}$.

The primal attack on our cryptosystem consists in finding a unique solution $(\mathbf{s}, \mathbf{e}, 1)$ for the SIS instance $[\ \mathbf{P}^\top \mid \mathbf{I} \mid\ -\mathbf{c}] \cdot \mathbf{x} \equiv \mathbf{0} \bmod q$ for $\mathbf{x} \in \mathbb{Z}^{n(2\ell+1)+1}$ and pk considered as a matrix $\mathbf{P} = [\ \mathbf{A} \mid \mathbf{p}^\top] \in \mathbb{Z}_q^{n\ell \times n(\ell+1)}$ and $\mathbf{c} \in \mathbb{Z}_q^{n(\ell+1)}$.

For the dual attack the attacker tries to find a short vector in the dual lattice that is employed to distinguish MLWE samples from uniform random samples in order to break decision-MLWE.

We do not need to take (quantum) hybrid attacks [8] into account as they often only get significant for sparse binary or trinary secrets and errors, which is never the case for the chosen parameters. Those attacks are based on Howgrave-Graham's Hybrid Attack combining lattice reduction with guessing techniques such as brute-force or meet-in-the-middle attacks.

Algebraic attacks such as finding short generators do not apply in our setting as the parameters required for a successful attack are far from being practical [5, 6].

9 Advantages and limitations (2.B.6)

The encryption scheme KINDI is a simple and flexible trapdoor-based encryption scheme, which by use of the trapdoor allows to retrieve back the error term and secret key from Module-LWE based ciphertexts. By this, it is possible to inspect all the constituents, if they comply with the allowed parameters in order to detect, for instance, inadmissible error terms. Furthermore, lattice-based trapdoor constructions are used in many areas of cryptography, not only for encryption or KEMs. Thus, KINDI may serve as a basis for new primitives. For instance, when using a slightly modified $\mathsf{KINDI}_{\mathsf{CPA}}$ in combination with a random oracle tag $\mathsf{mac} = H(\mathbf{s}, \mathbf{e})$, we already obtain a scheme that can be employed in CCA2-secure scenarios, since an adversary needs to know the unique inputs in order compute mac or differently spoken a correct ciphertext requires already to show knowledge of all its inputs via the mac.

In addition, KINDI allows to encrypt huge amount of data at once resulting in low message expansion factors as compared to other proposals since the error serves to transport the message. This is particularly interesting for sign-then-encrypt scenarios or for the transport of key bundles etc. For appropriate parameters signatures (uniform or Gauss) could also act as the error term, in this case the encryption scheme needs not to compute $\bar{u}$. Our proposal always by construction encrypts both a uniform random key s_1 and arbitrary messages.

Thus, it inherently tends to act as a KEM. Due to this, we see that many steps from [9] are already implicit in our CPA-secure construction resulting in very small overhead. In fact, even the generation of s_1 in $\mathsf{KINDI}_{\mathsf{CCA-KEM}}$ is implicitly accomplished in the encryption engine. Due to uniform random secrets and error vectors generated by SHAKE and the applied operations our implementations are constant-time.

There exist a wide range of parameters for various security levels. Increasing the parameters allows to encrypt more data at once without loosing efficiency. Due to small parameters and for primes q close to some $y \cdot 2^k$ for a suitable k, we can even use the NTT transform very efficiently applicable in constrained devices preferring to operate over integers than doubles.

Our $\mathsf{KINDI}_{\mathsf{CCA-KEM}}$ can easily be deployed into the TLS protocol as shown by Google for `NewHope` or in constrained devices or can be transformed into an authenticated key exchange protocol using known transformations.

References

[1] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM.

[2] Rachid El Bansarkhani, Özgür Dagdelen, and Johannes A. Buchmann. Augmented learning with errors: The untapped potential of the error term. *IACR Cryptology ePrint Archive*, 2014:733, 2014.

[3] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1006–1018, 2016.

[4] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 553–570, 2015.

[5] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. *IACR Cryptology ePrint Archive*, 2015:313, 2015.

[6] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-svp. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, pages 324–348, 2017.

[7] Henri Gilbert, editor. *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*. Springer, 2010.

[8] Florian Göpfert, Christine van Vredendaal, and Thomas Wunderer. A hybrid lattice basis reduction and quantum search attack on LWE. In *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, pages 184–202, 2017.

[9] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. *IACR Cryptology ePrint Archive*, 2017:604, 2017.

[10] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on cortex-m4 to speed up nist pqc candidates. Cryptology ePrint Archive, Report 2018/1018, 2018. https://eprint.iacr.org/2018/1018.

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

[12] Vadim Lyubashevsky and Daniele Micciancio. Generalized compact knapsacks are collision resistant. *Electronic Colloquium on Computational Complexity (ECCC)*, (142), 2005.

[13] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 372–381, 2004.

[14] Chris Peikert, Oded Regev, and Noah Stephens-Davidowitz. Pseudorandomness of ring-lwe for any ring and modulus. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 461–473, 2017.

[15] Chris Peikert and Alon Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. *Electronic Colloquium on Computational Complexity (ECCC)*, (158), 2005.

[16] El Bansarkhani Rachid. Lara - a design concept for lattice-based encryption. Cryptology ePrint Archive, Report 2017/049, 2017. https://eprint.iacr.org/2017/049.

[17] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93, 2005.

[18] Gregor Seiler. Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.