LIMA-1.1 : A PQC Encryption Scheme

Principal Submitter: Nigel P. Smart
Email: nigel.paul.smart@gmail.com
Telephone: ++32 163 79 231
Organization: KU Leuven.
Address: COSIC, Departement Elektrotechniek (ESAT), Kasteelpark Arenberg 10 - bus 2440, 3001 Leuven, Belgium.

Auxillary Submitters: Martin R. Albrecht (Royal Holloway, University of London), Yehuda Lindell (Bar-Ilan University), Emmanuela Orsini (KU Leuven), Valery Osheter (Unbound Tech), Kenneth G. Paterson (Royal Holloway, University of London), Guy Peer (Unbound Tech).

Name of Inventors/Developers: Martin R. Albrecht (Royal Holloway, University of London), Yehuda Lindell (Bar-Ilan University), Emmanuela Orsini (KU Leuven), Valery Osheter (Unbound Tech), Kenneth G. Paterson (Royal Holloway, University of London), Guy Peer (Unbound Tech), Nigel P. Smart (KU Leuven).

Name of the Owner: Nigel P. Smart

Signature of Submitter:

Backup Point of Contact: Martin R. Albrecht
Email: martin.albrecht@royalholloway.ac.uk
Telephone: +44 1784 414922
Organization: Royal Holloway, University of London.
Address: Information Security Group, Royal Holloway, University of London, Egham Hill, Egham, Surrey, TW20 0EX, United Kingdom.

Contents

Chapter 1

Discussion and Design Rationale

Note: This is Version 1.1 of the LIMA specification which includes a number of optimizations and minor corrections over the first specification. We also include different parameter choices, which are enabled by some of our minor modifications.

We introduce LIMA (LattIce MAthematics), a set of lattice-based public-key encryption and key encapsulation mechanisms, offering chosen plaintext security and chosen ciphertext security options. LIMA mixes conservative, standard and boring design choices with some efficiency improvements and flexibility. These factors are exhibited in its genesis: it is based on the ring variant [LPR10] of the LWE problem [Reg05] and on the encryption construction in [LP11]. We use the Fujisaki-Okamoto transform [FO99] to obtain an IND-CCA secure public-key encryption scheme. Our IND-CCA key encapsulation mechanism (KEM) is obtained via a transform of Dent [Den03]. This provides improved communication efficiency over using our IND-CCA public-key encryption scheme directly as a KEM; we also give a tight security proof for our IND-CCA KEM.

Thus our basic building blocks use highly respected and well studied cryptographic components. Our preference for "boring and simple" is illustrated by the fact that while our construction is efficient, other constructions such as [BDK+17] achieve higher encryption and decryption speeds. However, run times for lattice-based schemes are already generally faster than current public-key schemes and thus we view optimizing run times as being less important compared to simplicity.

Mathematical Structure

For efficiency we use a ring variant. Encryption based on Ring-LWE has been extensively studied in the literature [BG14, LPR10, LPR13, CMV+15]. The core component is essentially the Lyubashevsky *et al.* scheme [LPR10], or (equivalently) the BGV [BGV14] homomorphic encryption scheme, with the message being in the upper bits as opposed to the lower bits. This is sometimes referred to as the FV scheme [FV12].[1] The scheme also bears some resemblance to the scheme in [Pei14], although we adopt a completely different approach to ciphertext compression and producing IND-CCA variants. A lot of prior research has been conducted not only into the basic scheme, but also into implementation aspects [LSR+15, RVM+14], including protecting against side-channel attacks [RRVV15].

We opted for the ring variant of LWE to reduce the ciphertext size and to increase performance compared to LWE. In between Ring-LWE and LWE sits the Module-LWE [LS15] problem. As recently shown in [AD17], this problem is polynomial-time equivalent to Ring-LWE with a large modulus. Such large modulus Ring-LWE instances are distinguished from our Ring-LWE instances by the module-rank required to solve them. That is, while the Ring-LWE problem considered here requires one to find unusually short vectors in a module of rank three, this dimension is bigger in constructions such as [BDK+17]. At present, it is unclear if any loss of security is implied by using smaller module rank for ranks > 1.

[1] We do not use any homomorphic properties in this document, we just mention this to show the theoretical pedigree and prior analysis of our approach.

Name	Ring Variant	Functionality	Security
LIMA-2p-Enc-CPA	Two-Power	Encryption	IND-CPA
LIMA-sp-Enc-CPA	Safe-Prime	Encryption	IND-CPA
LIMA-2p-KEM-CPA	Two-Power	Key Encapsulation	IND-CPA
LIMA-sp-KEM-CPA	Safe-Prime	Key Encapsulation	IND-CPA
LIMA-2p-Enc-CCA	Two-Power	Encryption	IND-CCA
LIMA-sp-Enc-CCA	Safe-Prime	Encryption	IND-CCA
LIMA-2p-KEM-CCA	Two-Power	Key Encapsulation	IND-CCA
LIMA-sp-KEM-CCA	Safe-Prime	Key Encapsulation	IND-CCA

Table 1.1: The different LIMA schemes.

We offer two forms of ring to use within the LIMA system:

- Power-of-two Cyclotomics: These are relatively well studied and admit very efficient implementations.
- Safe-Prime Cyclotomics: These are almost as good as power-of-two cyclotomics in terms of the ring geometry, but are slightly less efficient.

The reasons for offering the second option for the ring are twofold. Firstly, using safe-prime cyclotomics allows us to create a more flexible parameter space in terms of the ring dimension. Secondly, recall that a prime p is called "safe" if $p = 2 \cdot p' + 1$ for another prime p'. This implies that there are only two non-trivial subfields of the cyclotomic field $\mathbb{Q}[X]/\Phi_p(X)$. By contrast, the presence of subfield towers in the power-of-two case raises the question of whether this could leave open routes for attack such as those in [CJL16a, ABD16]. While it was shown soon after that towers of subfields are not required for these attacks to succeed [KF17], it appears prudent to hedge against possible future attacks regardless. For example, [BBdV+17] shows that, for some fields, the presence of subfields can lead to a much easier lattice problem. We stress, however, that none of these subfield attacks are currently applicable to Ring-LWE. In summary, for added flexibility and to allow a more conservative choice of field, we give the *option* of using safe-prime cyclotomics.

The use of safe-primes appears to mitigate the possibility of attacks via subfields, but it also avoids the complex ring geometry associated with picking fields with large Galois group, such as the choice made in [BCLvV16]. Thus, we believe that using safe-prime fields offers a nice compromise between efficiency and protection against potential future attacks. In addition, using safe-prime cyclotomics allows us, with a little extra work, to be able to still use FFTs to evaluate the main polynomial products involved in the operation of our schemes. Thus, we think safe prime cyclotomic fields offer a good compromise between the benefits of the use of two power cyclotomics and fields defined by polynomials with large Galois groups.

We stress that we consider our schemes instantiated with the power-of-two rings to be secure, and they are more efficient in this case. We only offer the safe-prime variant if subfield attacks are a concern, and to offer additional flexibility in terms of parameter choices.

Cryptographic Structure

We offer IND-CPA and IND-CCA variants of a public-key encryption scheme. The IND-CCA variant is based on the Fujisaki-Okamoto transform [FO99]. Our choice of this methodology for achieving IND-CCA security is that it permits a tight security proof. We also offer IND-CPA and IND-CCA variants of a KEM construction, with the IND-CCA variant being built using a construction of Dent [Den03, Table 5] applied to our IND-CPA public-key encryption scheme. Dent's transformation did not have a tight security proof, so in [AOP+17a] we also provide a new proof that is tight in our specific context. We note that a related but more generic tight reduction was recently given in [HHK17].

Ignoring parameter sizes for the moment, this document therefore defines eight possible configurations for LIMA, as listed in Table 1.1.

Scheme	N	q
LIMA-2p	512	18433
LIMA-2p	1024	40961
LIMA-2p	2048	40961
LIMA-sp	1018	12521473
LIMA-sp	1306	48181249
LIMA-sp	1822	44802049
LIMA-sp	2062	16900097

Table 1.2: Parameter sets for use with LIMA.

The first two schemes in Table 1.1 should not be used (without care) in any application. The third and fourth schemes should also be used with care, but are included here as they are mentioned in the NIST call as being potentially desirable in some key exchange applications.

All the LIMA schemes are derived from a base encryption algorithm Enc-CPA-Sub which can result in decryption failures. In particular Enc-CPA-Sub could make a choice of randomness that produces in a ciphertext which results in a decryption failure (or even decrypts to the wrong message). This is a standard problem in lattice based schemes and can cause problems in operation and in security proofs. Our parameters are chosen to ensure that this happens with negligible probability.

We define seven parameter sets to be used in conjunction with the eight different LIMA schemes. Two of these parameter sets are in the "power-of-two" setting and four in the "safe-prime" setting. This yields a total of 24 different schemes, with this flexibility enabling scaling to larger messages spaces and/or increased security levels. The seven parameter sets are shown in Table 1.2

Note that for LIMA-sp parameter sets, the larger value of q is to enable the FFT to be efficiently computed via Bluestein's algorithm. This places some requirements on the value of q. Using the FFT also means that ring multiplication can be more easy implemented on highly parallel processors such as GPUs. Another option in the LIMA-sp case would have been to abandon the usage of the FFT methods and use smaller values of q. This, however, would have come at the expense of an estimated slowdown by a factor of about two.[2]

Random Seeds

All random seeds/coins for our algorithms are passed through a NIST approved XOF, from NIST SP 800 185 [NIS16]. This means *we do not* pass the seeds/coins through a DRBG first, as recommended by NIST (in the FAQ related to this call).

The XOF output is used to generate random finite field elements, symmetric keys, and (importantly for us) samples from a distribution somewhat close to a Gaussian distribution. For this latter task we adopt a method suggested in [ADPS16], which is both efficient (in software and hardware) and constant-time.

Ciphertext Compression

All ciphertexts are compressed in the sense of ignoring *coefficients* of the "message carrying component" which contains no information about the message. Further compression can be obtained by Huffman encoding the ring elements, and by removing bytes from the ciphertext carrying component.

In the case of our IND-CPA KEM, a further form of compression is possible by utilizing the reconciliation approach of [Pei14]. We did not use this for two reasons. Firstly, it is only applicable to the IND-CPA KEM.[3]

[2] In this case we would recommend using Karatsuba multiplication to multiply the ring elements, as opposed to coordinate wise multiplication in the FFT domain.

[3] In [Pei14] a method of obtaining an IND-CCA secure KEM is given, but it is less efficient than ours.

Secondly, there is some concern in the community over patenting of reconciliation mechanisms, and we wanted to avoid uncertainties arising from unresolved issues relating to the licensing of patents.

Intellectual Property

To our knowledge, the algorithms contained in this proposal are not covered by any intellectual property constraints.

The implementations provided in this proposal were written by Valery Osheter, Guy Peer and Nigel P. Smart.

Chapter 2

Specification

In this chapter, we slowly develop our specification from the bottom up.

2.1 Random Numbers, Hashing, XOFs and KDFs

To ensure efficient implementation, and to avoid multiple low level components (e.g. block ciphers and hash functions) we build all of our symmetric key components out of the SHA-3 hash function. Our constructions makes use of the following two functions, which are modelled as random oracles in our proofs of security:

- A function which takes a string and produces an essentially unbounded "random" string of bytes from this input string. This function will be used as the basis of all of the sampling needed in our algorithms, i.e. all the "Gaussian samples" and other random values, in ways which we will describe in detail below. Technically this function is an XOF (Extendable Output Function), and for our purposes we will use KMAC256, which is based on the SHA-3 hash function and defined in NIST SP 800 185 [NIS16].
- A function similar to that above, but which will only produce an output of fixed length. This fixed length output is intended to be used as a key in a higher level application. Technically, this function is a KDF (Key Derivation Function), and again we will use KMAC256 for this purpose.

2.1.1 KMAC256

We will model the KMAC256 algorithm via the following API for the purposes of this document, for details see [NIS16]. When called in the form $\mathsf{XOF} \leftarrow \mathsf{KMAC}(\mathsf{key}, \mathsf{data}, 0)$ the output is an XOF object, and when called in the form $K \leftarrow \mathsf{KMAC}(\mathsf{key}, \mathsf{data}, L)$ the output is a string of L bits in length. In both cases the input is a key key (of length at least 256 bits), a (one-byte) data string data, and a length field L in bits. The data string data will be used as a diversifier in our application, and it will correspond to the *domain separation* field in the KMAC standard. We use the empty string for the *input* field in the KMAC standard. Thus different values of data will specify different uses of the KMAC construction within our algorithms. In particular we will use single byte values of data only, as follows:

- $\mathsf{data} = 0x00$: Use a KDF.
- $\mathsf{data} = 0x01$: Use in LIMA key generation as an XOF.
- $\mathsf{data} = 0x02$: Use in LIMA Enc-CPA as an XOF.
- $\mathsf{data} = 0x03$: Use in LIMA Enc-CCA as an XOF.
- $\mathsf{data} = 0x04$: Use in LIMA Encap-CPA as an XOF.
- $\mathsf{data} = 0x05$: Use in LIMA Encap-CCA as an XOF.

In the case when $L = 0$ we shall let $a \leftarrow \mathsf{XOF}[n]$ denote the process of obtaining n bytes from the XOF object returned by the call to KMAC.

$\underline{\mathsf{KDF}^{[n]}(k)}$

In particular, our KDF can be derived as follows. Given a key k and desired output length n (in bytes) the KDF is then defined by:

1. $K \leftarrow \mathsf{KMAC}(k, 0x00, n)$
2. Output K.

2.1.2 Random Field Elements

At various points we will need to select an element uniformly at random from a finite field $\mathbb{F}_q$, where q is a prime, or we will need to produce vectors of such elements uniformly at random. These are defined in the following two operations, which assume a XOF has already been set up as above.

If $\mathbf{x}$ and $\mathbf{y}$ are two vectors of the same length over $\mathbb{F}_q$, we let $\mathbf{x} \otimes \mathbf{y}$ denote their Schur product, i.e. the coordinatewise product, over $\mathbb{F}_q$. Similarly we let $\mathbf{x} \oplus \mathbf{y}$ denote the coordinate-wise sum mod q.[1]

$\underline{a \xleftarrow[\mathsf{XOF}]{} \mathbb{F}_q}$

We consume twice as many random bits as q has from our XOF in order to obtain a value with a distribution that is close to uniform.

1. $s \leftarrow \mathsf{XOF}[\ 2 \cdot \lceil \log_{256} q \rceil\ ]$.
2. Convert s to an integer (msb is the left most bit).
3. $a \leftarrow s \pmod q$.
4. Output a.

$\underline{\mathbf{a} \xleftarrow[\mathsf{XOF}]{} \mathbb{F}_q^n}$

1. For i from 1 to n do
   (a) $a_i \xleftarrow[\mathsf{XOF}]{} \mathbb{F}_q$.
2. Output $\mathbf{a}$.

2.1.3 Random Approximate Discrete Gaussians

We define a distribution χ_σ from which the coefficients of an element in our ring R (to be defined later) will be drawn. This distribution is an approximation to the discrete Gaussian distribution with standard deviation σ and mean $\mu = 0$. There are various methods to approximately sample from such a distribution, for example Knuth-Yao or Box-Muller. We instead use a coarse approximation which can be realized by a constant time algorithm, and which is also suitable for implementation in hardware. In particular, we use the method of approximating a Discrete Gaussian via a centred binomial distribution, as suggested in [ADPS16], parametrizd by a value B. Starting with $2 \cdot B + 2$ random bits (b_i, b'_i) for $i = 0, \ldots, B$, one samples from χ_σ by computing

$$\sum_{i=0}^{B} (b_i - b'_i)$$

[1] We also use $\oplus$ to define exclusive-or, the difference in the two uses of this symbol should be clear from the context.

which is a binomial distribution centred on zero and with standard deviation $\sqrt{(B+1)/2} \approx \sigma$. Thus this distribution will approximate the discrete Gaussian distribution with standard deviation σ. In our scheme we select $B = 19$, and so we obtain $\sigma \approx 3.16$, and we require 40 bits (i.e. 5 bytes) of randomness per sample.

GenerateGaussianNoise$_{\mathsf{XOF}}(\sigma)$:

1. $B \leftarrow \lceil 2 \cdot \sigma^2 \rfloor - 1 = 19$. (Here and throughout $\lceil \cdot \rfloor$ denotes rounding of the argument to the nearest integer.)
2. $t = \mathsf{XOF}[5]$; interpret t as a bit string of length 40 in the natural way.
3. $s \leftarrow 0$.
4. For $i = 0$ to B do
   (a) $s \leftarrow s - t[2 \cdot i] + t[2 \cdot i + 1]$.
5. Return s.

2.1.4 Checking

To aid the implementor we provide in the `KAT` directory a file `XOF-KAT.txt` which will allow the implementor to check the above methods for generating random field elements and elements from an approximate discrete Gaussian. The KAT values are given in terms of the input string to the XOF, the associated data item, and then the corresponding outputs.

2.2 Cyclotomic Rings

As discussed at the beginning, our cryptographic system will come in two variants, depending on which type of cyclotomic field one chooses to use.

- **LIMA-2p: Power-of-Two:** In this variant N is a power of two and q is a prime such that $q \equiv 1 \pmod{2 \cdot N}$. We have $M = 2 \cdot N$ and $N = \phi(M)$, thus $q \equiv 1 \pmod{M}$. In this case we define the rings as
$$R = \mathbb{Z}[X]/(X^N + 1), \quad R_2 = \mathbb{Z}_2[X]/(X^N + 1), \text{ and } R_q = \mathbb{Z}_q[X]/(X^N + 1).$$
Note that $\Phi_M(X) = X^N + 1$ in this case.
- **LIMA-sp: Safe-Prime:** In this variant we select p to be a "safe prime", i.e. a prime such that $p = 2 \cdot p' + 1$ for another prime p'. We let e denote the smallest integer such that $2^e > 2 \cdot p$ and we let q be a prime such that $q \equiv 1 \pmod{2^e \cdot p}$. In this case we define the rings as
$$R = \mathbb{Z}[X]/(X^N + X^{N-1} + \cdots + X + 1), \quad R_2 = \mathbb{Z}_2[X]/(X^N + X^{N-1} + \cdots + X + 1),$$
and
$$R_q = \mathbb{Z}_q[X]/(X^N + X^{N-1} + \cdots + X + 1),$$
where $N = p - 1 = 2 \cdot p'$. Note that $\Phi_p(X) = X^{p-1} + X^{p-2} + \cdots + X + 1 = X^N + X^{N-1} + \cdots + X + 1$ in this case.

Elements of these rings are degree $(N-1)$ polynomials with coefficients from $\mathbb{Z}, \mathbb{Z}_2, \mathbb{Z}_q$, respectively. Equivalently, these are represented as vectors of length N, with elements in the $\mathbb{Z}, \mathbb{Z}_2, \mathbb{Z}_q$, respectively.

2.2.1 Roots of Unity

Our schemes use a fixed set of roots of unity. We define these using the following function:

RootOfUnity(m, q):

1. $a \leftarrow 1$.
2. Do
   (a) $a \leftarrow a + 1$.
   (b) $\alpha \leftarrow a^{(q-1)/m} \pmod q$.
   (c) $\beta \leftarrow \Phi_m(\alpha) \pmod q$.
3. While $(\beta \neq 0)$.
4. Output α.

Note that if m is a power of two then $\Phi_m(X) = X^{m/2} + 1$ and if $m = 2 \cdot p$ is twice a prime p then $\Phi_m(X) = X^{2 \cdot p-2} + X^{2 \cdot p-4} + \cdots + X^2 + 1 = \Phi_p(X^2)$.

We use the above procedure to derive the following roots of unity:

LIMA-2p: By construction $\mathbb{F}_q$ contains a $(2 \cdot N)$-th root of unity, so we let

$$\alpha_0 \leftarrow \mathsf{RootOfUnity}(2 \cdot N, q).$$

Given α_0, we set $\alpha_1 = 1/\alpha_0 \pmod q$. We also set $\beta_0 = 1/N \pmod q$. For our two parameter sets in this setting, we have the following values for $\alpha_0, \alpha_1, \beta_0$:

N	q	α_0	α_1	β_0
512	18433	7673	3935	18397
1024	40961	20237	16233	40921
2048	40961	18088	14056	40941

LIMA-sp: By construction $\mathbb{F}_q$ contains a 2^e-th root of unity, and a $2 \cdot p$-th root of unity. We define

$$\begin{aligned}
\alpha_0 &\leftarrow \mathsf{RootOfUnity}(2 \cdot p, q), \\
\alpha_1 &\leftarrow 1/\alpha_0 \pmod q, \\
\beta_0 &\leftarrow \mathsf{RootOfUnity}(2^e, q), \\
\beta_1 &\leftarrow 1/\beta_0 \pmod q.
\end{aligned}$$

For our four parameter sets in this setting we have the following values of $\alpha_0, \alpha, \beta_0, \beta_1$:

N	q	e	α_0	α_1	β_0	β_1
1018	12521473	11	1561269	8501297	9597006	10910567
1306	48181249	12	30019814	39013233	5599915	28280508
1822	44802049	12	43213195	19941338	8284672	1121361
2062	16900097	13	12381941	15641966	213248	7202243

2.2.2 Representation

For each ring elment $\mathbf{a} \in R_q$ we represent it as a polynomial of degree $N - 1$, which we also think of as a vector of N elements in $\mathbb{F}_q$. To store/transmit such elements on-the-wire we use an ordering of the coefficients in which the coefficient corresponding to the smallest coefficient is to the left (i.e. transmitted first). Thus we encode

$$a_0 + a_1 X + \cdots + a_{N-1} \cdot X^{N-1}$$

as the array of integers $(a_0, a_1, \ldots, a_{N-1})$. See Section 2.8 for how we encode data in more detail.

BV-2-RE($\mathbf{b}$):

Given a byte array $\mathbf{b}$ we produce a ring element, where each coefficient corresponds to a *bit* of $\mathbf{b}$, as follows, We let $\mathbf{b} = (b_0, \ldots, b_{t-1})$ where $8 \cdot t < N$.

1. $\mathbf{a} \leftarrow 0$.
2. $d \leftarrow 0$.
3. For $i = 0, \ldots, t-1$
   (a) $t \leftarrow b_i$
   (b) For $i = 0, \ldots, 7$
      i. $\mathbf{a} \leftarrow \mathbf{a} + (t \wedge 1) \cdot X^d$.
      ii. $t \leftarrow t \gg 1$.
      iii. $d \leftarrow d + 1$.
4. Return $\mathbf{a}$.

The inverse operation we denote by $\mathbf{b} \leftarrow$ RE-2-BV($\mathbf{a}$), where in this case a is assumed to have coefficients in $\{0, 1\}$.

2.2.3 Truncation

To aid bandwidth efficiency, we sometimes truncate a ring element to a vector of integers modulo q, of smaller size. Given a ring element $\mathbf{a} \in R_q$ of the form

$$\mathbf{a} = a_0 + a_1 \cdot X + \cdots + a_{N-1} \cdot X^{N-1},$$

we define, for $1 \leq T \leq N$,

$$\mathsf{Trunc}(\mathbf{a}, T) = a_0 + a_1 \cdot X + \cdots + a_{T-1} \cdot X^{T-1}.$$

2.3 Fourier Transforms Over Cyclotomic Rings

The reason for defining the restrictions on q in the previous section, and for fixing specified roots of unity, is so that we can define the following Fourier Transform algorithms. These enable us to perform arithmetic in the above rings via coordinate-wise additions and multiplications.

FFT$_{sub-1}(\mathbf{x}, n, \tau)$:

This algorithm takes as input a vector $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_{n-1})$ of length n (a power of two), and a root of unity τ and performs the following recursive steps:

1. If $n = 1$ then return $\mathbf{x}$.
2. $\mathbf{y} \leftarrow (x_0, x_2, x_4, \ldots)$.
3. $\mathbf{z} \leftarrow (x_1, x_3, x_5, \ldots)$.
4. $\beta \leftarrow \tau^2 \pmod q$.
5. $\mathbf{y} \leftarrow \mathsf{FFT}_{sub-1}(\mathbf{y}, n/2, \beta)$,
6. $\mathbf{z} \leftarrow \mathsf{FFT}_{sub-1}(\mathbf{z}, n/2, \beta)$.
7. $\omega \leftarrow \tau$.

8. For i from 0 to $n/2 - 1$
   (a) $s \leftarrow \omega \cdot z_i \pmod{q}$.
   (b) $t \leftarrow y_i$.
   (c) $x_i \leftarrow t + s \pmod{q}$.
   (d) $x_{i+n/2} \leftarrow t - s \pmod{q}$.
   (e) $\omega \leftarrow \omega \cdot \beta \pmod{q}$.
9. Return $\mathbf{x}$.

$\mathsf{FFT}_{sub-2}(\mathbf{x}, n, \tau)$**:**

This algorithm takes as input a vector $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_{n-1})$ of length n (a power of two), and a root of unity τ and performs the following recursive steps, which are identical to those above, except for the way the final output is produced:

1. If $n = 1$ then return $\mathbf{x}$.
2. $\mathbf{y} \leftarrow (x_0, x_2, x_4, \ldots)$.
3. $\mathbf{z} \leftarrow (x_1, x_3, x_5, \ldots)$.
4. $\beta \leftarrow \tau^2 \pmod{q}$.
5. $\mathbf{y} \leftarrow \mathsf{FFT}_{sub-2}(\mathbf{y}, n/2, \beta)$,
6. $\mathbf{z} \leftarrow \mathsf{FFT}_{sub-2}(\mathbf{z}, n/2, \beta)$.
7. $\omega \leftarrow 1$.
8. For i from 0 to $n/2 - 1$
   (a) $s \leftarrow \omega \cdot z_i \pmod{q}$.
   (b) $t \leftarrow y_i$.
   (c) $x_i \leftarrow t + s \pmod{q}$.
   (d) $x_{i+n/2} \leftarrow t - s \pmod{q}$.
   (e) $\omega \leftarrow \omega \cdot \tau \pmod{q}$.
9. Return $\mathbf{x}$.

$\mathsf{BFFTData}()$**:**

For the LIMA-sp fields we will be using Bluestein's FFT algorithm and as such we will require a number of items of precomputed data. We first define three arrays with the following ranges

$$\mathsf{powers}[0 \ldots 1][0 \ldots p-1],$$
$$\mathsf{powersi}[0 \ldots 1][0 \ldots p-1],$$
$$\mathbf{bd}[0 \ldots 1][0 \ldots 2^e - 1].$$

We will refer to the (i, j)-th element of powers by $\mathsf{powers}_{i,j}$ etc, and the i-th row of $\mathbf{bd}$ by $\mathbf{bd}_i$. We initialize these data structures as follows:

1. $\mathsf{powersi}_{0,0} \leftarrow 1/2^e \pmod{q}$.
2. $\mathsf{powersi}_{1,0} \leftarrow 1/(2^e \cdot p) \pmod{q}$.

3. For r from 0 to 1 do
   (a) $\mathsf{powers}_{r,0} \leftarrow 1$.
   (b) $\mathbf{bd}_{r,p-1} = 1$.
   (c) For i from 1 to $p-1$ do
      i. $s \leftarrow i^2 \pmod{2 \cdot p}$.
      ii. $\mathsf{powers}_{r,i} \leftarrow (\alpha_r)^s \pmod{q}$.
      iii. $\mathsf{powersi}_{r,i} \leftarrow \mathsf{powers}_{r,i} \cdot \mathsf{powersi}_{r,0} \pmod{q}$.
      iv. $b \leftarrow (\alpha_{1-r})^s \pmod{q}$.
      v. $\mathbf{bd}_{r,p-1+i} \leftarrow b$.
      vi. $\mathbf{bd}_{r,p-1-i} \leftarrow b$.
   (d) $\mathbf{bd}_{r,i} \leftarrow 0$ for $i = 2 \cdot p - 1$ to $2^e - 1$.
   (e) $\mathbf{bd}_r \leftarrow \mathsf{FFT}_{sub-2}(\mathbf{bd}_r, 2^e, \beta_0)$.

BFFT($\mathbf{a}, r$):

This algorithm performs both the forward and the backward directions for the FFT in the LIMA-sp field variant. The input in both cases is a vector $\mathbf{a}$ of length p, indexed from zero, the output is a vector of length p, again indexed from zero. The forward direction is implemented by setting $r = 0$, and the backward direction is implemented by setting $r = 1$. In the forward direction the $(p-1)$-th element in the input array $\mathbf{a}$ is equal to zero, whereas in the backward directtion the first element in the array $\mathbf{a}$ is equal to zero. Thus (essentially) the input has size $p-1$, and not p; but for ease of exposition we think of it as being of size p.

1. Define $\mathbf{x}$ as a vector indexed by 0 to $2^e - 1$, initialized to zero, and $\mathbf{b}$ as a vector indexed by 0 to $p-1$.
2. $x_i \leftarrow \mathsf{powers}_{r,i} \cdot a_i \pmod{q}$ for $i = 0, \ldots, p-1$.
3. $\mathbf{x} \leftarrow \mathsf{FFT}_{sub-2}(\mathbf{x}, 2^e, \beta_0)$
4. $x_i \leftarrow x_i \cdot \mathbf{bd}_{r,i} \pmod{q}$ for $i = 0, \ldots, 2^e - 1$.
5. $\mathbf{x} \leftarrow \mathsf{FFT}_{sub-2}(\mathbf{x}, 2^e, \beta_1)$
6. $b_i \leftarrow x_{i+p-1} \cdot \mathsf{powersi}_{r,i} \pmod{q}$ for $i = 0, \ldots, p-1$.
7. Return $\mathbf{b}$.

2.3.1 Fourier Transforms

We can now define the Fourier Transform algorithms themselves.

FFT(f):

We can think of the polynomial $f \in R_q$ given by $f = f_0 + f_1 \cdot X + f_2 \cdot X^2 + \cdots + f_{N-1} \cdot X^{N-1}$ as the vector $\mathbf{f} = (f_0, f_1, \ldots, f_{N-1}) \in \mathbb{F}_q^N$. To compute the FFT of f we then perform the steps:

LIMA-2p:

1. $\mathbf{x} \leftarrow \mathsf{FFT}_{sub-1}(\mathbf{f}, N, \alpha_0)$.
2. Return $\mathbf{x}$.

LIMA-sp:

1. $\mathbf{y} \leftarrow \mathsf{BFFT}(\mathbf{f}, 0)$.
2. $x_i \leftarrow y_{i+1}$ for $i = 0, \ldots, p-2$.
3. Return $\mathbf{x}$.

$\mathsf{FFT}^{-1}(\mathbf{x})$:

To invert the above operation we perform the following steps:

LIMA-2p:

1. $\gamma \leftarrow \alpha_1^2 \pmod{q}$.
2. $\mathbf{f} \leftarrow \mathsf{FFT}_{sub-2}(\mathbf{x}, N, \gamma)$.
3. $\delta \leftarrow \beta_0$.
4. For i from 0 to $N-1$ do
   (a) $f_i \leftarrow f_i \cdot \delta \pmod{q}$.
   (b) $\delta \leftarrow \delta \cdot \alpha_1 \pmod{q}$.
5. Return $\mathbf{f}$.

LIMA-sp:

1. $y_0 \leftarrow 0$.
2. $y_{i+1} \leftarrow x_i$ for $i = 0, \ldots, p-2$.
3. $\mathbf{f} \leftarrow \mathsf{BFFT}(\mathbf{y}, 1)$.
4. $f_i \leftarrow f_i - f_{p-1} \pmod{q}$ for $i = 0, \ldots, p-1$.
5. Return $\mathbf{f}$.

Note that step 4 immediately above performs reduction modulo $\Phi_p(X)$.

2.3.2 Checking

Again, since this is a relatively complex set of operations, we provide a KAT file in the KAT directory called `FFT-KAT.txt` to enable implementors to check they are getting the same output from the FFT routines for specific inputs.

2.4 IND-CPA Public-Key Encryption

In this section, we define an IND-CPA public-key encryption scheme which will encrypt messages of size at most $\lfloor N/8 \rfloor$ bytes in length. In the next sub-sections, we will use elements of this IND-CPA encryption scheme to build our IND-CCA encryption scheme (which will encrypt messages of $\lfloor N/8 \rfloor - 32$ bytes in length) and our IND-CPA and IND-CCA key encapsulation mechanisms. We define

$$\Delta_q = \left\lfloor \frac{q}{2} \right\rfloor.$$

$\mathsf{KeyGen}(\mathbf{seed}_0)$:

Key generation proceeds as follows, where the input $\mathbf{seed}_0$ is a string containing *at least* 256 bits of entropy.

1. $\mathsf{XOF} \leftarrow \mathsf{KMAC}(\mathbf{seed}_0, 0x01, 0)$
2. $a = (a_0, \ldots, a_{N-1}) \xleftarrow[\mathsf{XOF}]{} \mathbb{F}_q^N$. Interpret a as an element of R_q.
3. For $i = 0$ to $N-1$ do $s_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.

4. For $i = 0$ to $N-1$ do $e'_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
5. $\mathbf{a} \leftarrow \mathsf{FFT}(a)$.
6. $\mathbf{s} \leftarrow \mathsf{FFT}(s)$.
7. $\mathbf{e}' \leftarrow \mathsf{FFT}(e')$.
8. $\mathbf{b} \leftarrow (\mathbf{a} \otimes \mathbf{s}) \oplus \mathbf{e}'$, with pointwise multiplication and addition $\pmod q$.
9. $\mathfrak{sk} \leftarrow (\mathbf{s}, \mathbf{a}, \mathbf{b})$.
10. $\mathfrak{pk} \leftarrow (\mathbf{a}, \mathbf{b})$.
11. Return $(\mathfrak{pk}, \mathfrak{sk})$

Observe that s is a polynomial with small (Gaussian) coefficients, a is a random polynomial (with large, truly random coefficients), and $\mathbf{b}$ is the Fourier transform of $b = a \cdot s + e'$ where e' is also a polynomial with small (Gaussian) coefficients.

An alternative form of key generation which produces a "compressed" public/secret key is as follows:

1. $\mathsf{XOF} \leftarrow \mathsf{KMAC}(\mathbf{seed}_0, 0x01, 0)$
2. $\mathbf{seed}_1 \leftarrow \mathsf{KMAC}^{[384]}(\mathbf{seed}_0)$.
3. $\mathsf{XOF}' \leftarrow \mathsf{KMAC}(\mathbf{seed}_1, 0x01, 0)$
4. $a = (a_0, \ldots, a_{N-1}) \underset{\mathsf{XOF}'}{\leftarrow} \mathbb{F}_q^N$. Interpret a as an element of R_q.
5. For $i = 0$ to $N-1$ do $s_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
6. For $i = 0$ to $N-1$ do $e'_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
7. $\mathbf{a} \leftarrow \mathsf{FFT}(a)$.
8. $\mathbf{s} \leftarrow \mathsf{FFT}(s)$.
9. $\mathbf{e}' \leftarrow \mathsf{FFT}(e')$.
10. $\mathbf{b} \leftarrow (\mathbf{a} \otimes \mathbf{s}) \oplus \mathbf{e}'$, with pointwise multiplication and addition $\pmod q$.
11. $\mathfrak{sk} \leftarrow (\mathbf{seed}_0, \mathbf{seed}_1, \mathbf{b})$.
12. $\mathfrak{pk} \leftarrow (\mathbf{seed}_1, \mathbf{b})$.
13. Return $(\mathfrak{pk}, \mathfrak{sk})$

Then to "uncompress" the public key one computes:

1. $\mathsf{XOF}' \leftarrow \mathsf{KMAC}(\mathbf{seed}_1, 0x01, 0)$
2. $a = (a_0, \ldots, a_{N-1}) \underset{\mathsf{XOF}'}{\leftarrow} \mathbb{F}_q^N$
3. $\mathbf{a} \leftarrow \mathsf{FFT}(a)$.

A similar procedure can be performed to uncompress the components of the private key.

Note that, either in the regular or compressed case, the public key $\mathfrak{pk}$ is implicitly contained in the private key $\mathfrak{sk}$. We will exploit this property without further comment in the remainder of this chapter.

Enc-CPA-Sub($\mathbf{m}, \mathfrak{pk}, \mathsf{XOF}$):

We now define an algorithm Enc-CPA-Sub that we will use as a subroutine in our all of our encryption schemes and KEMs. The encryption mechanism takes as input the public key $\mathfrak{pk} = (\mathbf{a}, \mathbf{b})$, a message $\mathbf{m} \in \{0,1\}^{|\mathbf{m}|}$, where $|\mathbf{m}| < N$, and an already initialised XOF XOF. We assume that $|\mathbf{m}|$ is a multiple of eight, i.e. we transmit whole bytes only, thus we think of $\mathbf{m}$ as a byte vector of length $|\mathbf{m}|/8$. The encryption algorithm will return $\perp$ if the current XOF state does not produce a valid tuple of randomness.

1. $\ell = |\mathbf{m}|$.
2. If $\ell > N$ then return $\perp$.
3. $\mu \leftarrow \mathsf{BV\text{-}2\text{-}RE}(\mathbf{m})$,
4. For $i = 0$ to $N-1$ do $v_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
5. For $i = 0$ to $N-1$ do $e_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
6. For $i = 0$ to $N-1$ do $d_i \leftarrow \mathsf{GenerateGaussianNoise}_{\mathsf{XOF}}(\sigma)$.
7. $\mathbf{v} \leftarrow \mathsf{FFT}(v)$, $\mathbf{e} \leftarrow \mathsf{FFT}(e)$.
8. $x \leftarrow d + \Delta_q \cdot \mu \pmod q$.
9. $s \leftarrow \mathsf{FFT}^{-1}(\mathbf{b} \otimes \mathbf{v})$.
10. $t \leftarrow s + x$.
11. $c_0 \leftarrow \mathsf{Trunc}(t, \ell)$.
12. $\mathbf{c}_1 \leftarrow (\mathbf{a} \otimes \mathbf{v}) \oplus \mathbf{e}$, with pointwise multiplication and addition $\pmod q$.

Note that c_0 is the ring element $b \cdot v + d + \Delta_q \cdot \mu$ truncated to ℓ coefficients, whilst $\mathbf{c}_1$ is the Fourier transform of the element $a \cdot v + e$. We implicitly assume that the internal representation of c_0 holds the number of coefficients ℓ; see Section 2.8 for details of on the wire encodings of c_0 and $\mathbf{c}_1$. Also note that the size of a ciphertext (in bits) is equal to approximately

$$(N + \ell) \cdot \log_2 q \approx (N + |\mathbf{m}|) \cdot \log_2 q.$$

We are now ready to define an IND-CPA public key encryption scheme. It uses the same KeyGen algorithm as above, and an encryption algorithm Enc-CPA and a decryption algorithm Dec-CPA that are defined immediately below.

Enc-CPA($\mathbf{m}, \mathfrak{pk}, \mathbf{r}$):

The encryption algorithm again takes as input the public key $\mathfrak{pk} = (\mathbf{a}, \mathbf{b})$, a message $\mathbf{m} \in \{0,1\}^{|\mathbf{m}|}$, where $|\mathbf{m}| < N$, but this time it also takes as input a string $\mathbf{r}$ containing at least 256-bits of entropy (although we recommend $\mathbf{r}$ contains at least 384 bits of entropy).

1. If $|\mathbf{r}| < 256$ or $|\mathbf{m}| > N$ then return $\perp$.
2. $\mathsf{XOF} \leftarrow \mathsf{KMAC}(\mathbf{r}, 0x02, 0)$.
3. $\mathbf{c} \leftarrow$ Enc-CPA-Sub($\mathbf{m}, \mathfrak{pk}, \mathsf{XOF}$)
4. Return $\mathbf{c}$.

Dec-CPA($\mathbf{c}, \mathfrak{sk}$):

On input a ciphertext $\mathbf{c} = (c_0, \mathbf{c}_1)$ and a secret key $\mathfrak{sk} = (\mathbf{s}, \mathbf{a}, \mathbf{b})$, decryption is performed as follows.

1. Define ℓ to be the length of c_0, i.e. the number of field elements used to represent c_0.
2. If $\ell \neq 0 \pmod 8$ then return $\perp$.
3. $v \leftarrow \mathsf{FFT}^{-1}(\mathbf{s} \otimes \mathbf{c}_1)$.
4. $t \leftarrow \mathsf{Trunc}(v, \ell)$.
5. $f \leftarrow c_0 - t$.
6. Convert f into centered-representation. That is, let $f = (f_0, \ldots, f_{\ell-1})$ where each $f_i \in \mathbb{Z}_q$. If $0 \leq f_i \leq \frac{q-1}{2}$, then leave it unchanged. Else if $\frac{q}{2} < f_i \leq q-1$, then set $f_i \leftarrow f_i - q$ (over the integers).
7. $\mu \leftarrow \left| \left\lfloor \frac{2}{q} f \right\rceil \right|$ (i.e., round the scaled coefficients of f to the nearest integer and take the absolute value; the resulting coefficents will be 0 or 1).
8. $\mathbf{m} \leftarrow \mathsf{RE\text{-}2\text{-}BV}(\mu)$.
9. Return $\mathbf{m}$.

2.5 IND-CCA Public-Key Encryption

The scheme above is not secure under chosen-ciphertext attacks. In this section, we show how to achieve IND-CCA security by utilizing the highly efficient (first) transform of Fujisaki and Okamoto [FO99]. The key generation and basic primitives all remain the same; the modification is only with respect to encryption and decryption. If the original encryption scheme ($\mathsf{KeyGen}, \mathsf{Enc\text{-}CPA}, \mathsf{Dec\text{-}CPA}$) can encrypt messages of $\lfloor N/8 \rfloor$ bytes in length, this IND-CCA scheme encrypts messages of $\lfloor N/8 \rfloor - 32$ bytes in length; again we assume messages are a whole number of bytes long. We select the constant 32 here so as to enable the encryption of as large a message as possible, whilst achieving the desired security levels.

In what follows we let $\mathbf{r} \leftarrow \mathbf{r} + 1$ denote the operation of converting the bit string to an integer in a little-endian manner, adding one to the value, and then converting back to a bit string of the same length as the input string.

Enc-CCA($\mathbf{m}, \mathfrak{pk}, \mathbf{r}$):

1. If $|\mathbf{r}| \neq 256$ or $|\mathbf{m}| \geq N - 256$ then return $\perp$.
2. $\mu \leftarrow \mathbf{m} \| \mathbf{r}$.
3. $\mathsf{XOF} \leftarrow \mathsf{KMAC}(\mu, 0x03, 0)$.
4. $\mathbf{c} \leftarrow \mathsf{Enc\text{-}CPA\text{-}Sub}(\mu, \mathfrak{pk}, \mathsf{XOF})$.
5. Return $\mathbf{c}$.

Dec-CCA($\mathbf{c}, \mathfrak{sk}$):

1. $\mu \leftarrow \mathsf{Dec\text{-}CPA}(\mathbf{c}, \mathfrak{sk})$.
2. If $|\mu| < 256$ then return $\perp$.
3. $\mathsf{XOF} \leftarrow \mathsf{KMAC}(\mu, 0x03, 0)$.
4. $\mathbf{c}' \leftarrow \mathsf{Enc\text{-}CPA\text{-}Sub}(\mu, \mathfrak{pk}, \mathsf{XOF})$.

5. If $\mathbf{c} \neq \mathbf{c}'$ then return $\perp$.
6. $\mathbf{m}\|\mathbf{r} \leftarrow \mu$, where $\mathbf{r}$ is 256 bits long.
7. Return $\mathbf{m}$.

Note for this IND-CCA encryption algorithm the bit-size of a ciphertext is equal to approximately

$$(N + |\mathbf{m}| + 256) \cdot \log_2 q$$

and messages are limited to being at most $N - 256$ bits in size.

2.6 IND-CPA Key Encapsulation Mechanism

In this section, and following the NIST call, we present an IND-CPA KEM. This is useful in contexts where an *ephemeral* public key is generated and then used to transmit a symmetric key to another party, with the ephemeral public key being signed by a long term static key. The idea is that the KEM should only be used to transport a single symmetric key.

For key encapsulation in the post-quantum setting, we aim to transmit a key with at least $\ell \geq 256$ bits of entropy, where ℓ is divisible by eight and $\ell \leq N$. To do this we require sufficient entropy to perform the encapsulation; we recommend at least 384 bits of entropy for this. Thus, in total we assume an input entropy pool of at least $\ell + 384 \geq 256 + 384 = 640 = 8 \cdot 80$ bits.

The key generation algorithm KeyGen is the same as in that of our basic CPA encryption scheme, detailed in Section 2.4. The encapsulation algorithm Encap-CPA and decapsulation algorithm Decap-CPA are defined immediately below.

Encap-CPA$(\ell, \mathfrak{pk}, \mathbf{r})$**:**

This algorithm takes as input a public key $\mathfrak{pk}$, a bit length ℓ and an input string of random bits $\mathbf{r}$ such that $|\mathbf{r}| \geq \ell + 384$. The procedure outputs an encapsulation $\mathbf{c} = (c_0, \mathbf{c}_1)$ and the key $\mathbf{k}$, of length ℓ bits, it encapsulates. Again we expect, with high probability, that the while loop in the pseudo-code is only executed once.

1. If $|\mathbf{r}| < \ell + 384$ or $\ell > N$ or $\ell < 256$ then return $\perp$.
2. Write $\mathbf{r}$ as $\mathbf{t}\|\mathbf{k}$ where $|\mathbf{k}| = \ell$.
3. XOF $\leftarrow$ KMAC$(\mathbf{t}, 0x04, 0)$.
4. $\mathbf{c} \leftarrow$ Enc-CPA-Sub$(\mathbf{k}, \mathfrak{pk}, \text{XOF})$.
5. Return $(\mathbf{c}, \mathbf{k})$.

Decap-CPA$(\mathbf{c}, \mathfrak{sk})$**:**

This algorithm takes as input a secret key $\mathfrak{sk}$ and an encapsulation $\mathbf{c} = (c_0, \mathbf{c}_1)$, and outputs the key $\mathbf{k}$ it encapsulates. It simply runs the decryption algorithm of our IND-CPA public-key encryption scheme.

1. $\mathbf{k} \leftarrow$ Dec-CPA$(\mathbf{c}, \mathfrak{sk})$.
2. Output $\mathbf{k}$.

Note that for this IND-CPA KEM, the size of an encapsulation is equal to approximately

$$(N + \ell) \cdot \log_2 q$$

bits.

2.7 IND-CCA Key Encapsulation Mechanism

In this section, we present an IND-CCA KEM. We adopt the method of Dent [Den03, Theorem 5], which builds an IND-CCA secure KEM from an IND-CPA secure encryption scheme. We essentially reuse the encryption scheme from Section 2.4 with algorithms (KeyGen, Enc-CPA, Dec-CPA), but we replace Enc-CPA with an algorithm derived from Enc-CPA-Sub. The "hashing of random coins" approach that Dent's transform relies on is reflected in our construction by using a random value $\mathbf{r}$ as an input to an XOF to generate the randomness consumed by Enc-CPA-Sub, and by also treating $\mathbf{r}$ as the message input to Enc-CPA-Sub. Our security analysis will treat the XOF as a random oracle.

The key generation algorithm KeyGen of our IND-CCA KEM is the same as in that detailed in Section 2.4. The encapsulation algorithm Encap-CCA and decapsulation algorithm Decap-CCA are defined immediately below.

Encap-CCA$(\ell, \mathfrak{pk}, \mathbf{s})$**:**

This algorithm takes as input a public key $\mathfrak{pk}$, a bit length ℓ (divisible by eight), and a string of random bits $\mathbf{r}$ such that $256 \leq |\mathbf{r}| \leq N$. The output is an encapsulation $\mathbf{c} = (c_0, \mathbf{c}_1)$ and the key $\mathbf{k} \in \{0,1\}^{\ell}$ it encapsulates.

1. If $|\mathbf{r}| < 384$ or $|\mathbf{r}| > N$ then return $\perp$.
2. XOF $\leftarrow$ KMAC$(\mathbf{r}, 0x05, 0)$.
3. $\mathbf{c} \leftarrow$ Enc-CPA-Sub$(\mathbf{r}, \mathfrak{pk}, \text{XOF})$.
4. $\mathbf{k} \leftarrow \text{KDF}^{[\ell]}(\mathbf{r})$.
5. Return $(\mathbf{c} = (c_0, \mathbf{c}_1), \mathbf{k})$.

Decap-CCA$(\ell, \mathbf{c}, \mathfrak{sk})$**:**

This algorithm takes as input a secret key key $\mathfrak{sk}$ and an encapsulation $\mathbf{c} = (c_0, \mathbf{c}_1)$, and outputs the key $\mathbf{k}$ it encapsulates, or an error symbol $\perp$.

1. $\mathbf{r} \leftarrow$ Dec-CPA$(\mathbf{c}, \mathfrak{sk})$.
2. If $|\mathbf{r}| < 384$ then return $\perp$.
3. XOF $\leftarrow$ KMAC$(\mathbf{r}, 0x05, 0)$.
4. $\mathbf{c}' \leftarrow$ Enc-CPA-Sub$(\mathbf{r}, \mathfrak{pk}, \text{XOF})$.
5. If $\mathbf{c} \neq \mathbf{c}'$ then return $\perp$.
6. $\mathbf{k} \leftarrow \text{KDF}^{[\ell]}(\mathbf{r})$.

Note for this IND-CCA KEM, the size of a ciphertext is equal to approximately

$$(N + |\mathbf{r}|) \cdot \log_2 q$$

bits.

2.8 Encoding of Ciphertexts and Public Keys

In this section we detail how data items are encoded as strings of bytes. We first encode the parameter set as a single byte for ease of reference, using the following table:

Scheme	N	q	pCode
LIMA-2p	512	18433	0
LIMA-2p	1024	40961	1
LIMA-2p	2048	40961	2
LIMA-sp	1018	12521473	3
LIMA-sp	1306	48181249	4
LIMA-sp	1822	44802049	5
LIMA-sp	2062	16900097	6

We encode integers, such as length fields and elements of $\mathbb{F}_q$, given by

$$a = a_0 + 256 \cdot a_1 + \cdots + a_t \cdot 256^t$$

as the sequence of bytes

$$a_t \| \ldots \| a_1 \| \ldots \| a_0.$$

Thus, the number 12345 gets encoded as the byte string $0x30 \| 0x39$. We write this byte string as $\mathsf{Val}(a)$. In many situations the value of t is implicit, i.e. $t = \lceil \log_{256} q \rceil$. If we want to pad the representation (to the left), to exactly t bytes then we write $\mathsf{Val}_t(a)$.

To encode a public key $\mathfrak{pk} = (\mathbf{a}, \mathbf{b})$ where

$$\mathbf{a} = a_0 + a_1 \cdot X + \cdots + a_{N-1} \cdot X^{N-1} \text{ and } \mathbf{b} = a_0 + a_1 \cdot X + \cdots + a_{N-1} \cdot X^{N-1},$$

as a byte string, which we express as $B \leftarrow \mathsf{Encode}(\mathfrak{pk})$, we use the byte string

$$B = \mathsf{pCode} \| \mathsf{Val}(a_0) \| \mathsf{Val}(a_1) \| \ldots \| \mathsf{Val}(a_{N-1}) \| \mathsf{Val}(b_0) \| \mathsf{Val}(b_1) \| \ldots \| \mathsf{Val}(b_{N-1}),$$

where we take $a_i, b_i \in [0, q-1)$. The inverse operation is denoted by $\mathsf{Decode}(B)$. Note that the value of pCode allows us to infer the value N and the byte length of the integers q

As the IND-CCA versions of our public-key encryption scheme and KEM require the public key for decryption, we store the public-key with the secret key, in one data item, we encode a secret key $\mathfrak{sk} = (\mathbf{s}, \mathbf{a}, \mathbf{b})$ in a similar way by writing $B \leftarrow \mathsf{Encode}(\mathfrak{sk})$ where:

$$B = \mathsf{pCode} \| \mathsf{Val}(a_0) \| \mathsf{Val}(a_1) \| \ldots \| \mathsf{Val}(a_{N-1}) \| \mathsf{Val}(b_0) \| \mathsf{Val}(b_1) \| \ldots \| \mathsf{Val}(b_{N-1}) \| \mathsf{Val}(s_0) \| \mathsf{Val}(s_1) \| \ldots \| \mathsf{Val}(s_{N-1}).$$

Note that the first bytes in such a byte string B correspond to the encoding of a public key, so one can extract the public key by calling $\mathfrak{pk} \leftarrow \mathsf{Decode}(B)$, using a suitable overloading of the function Decode.

In the case of our compressed public keys we replace the byte string

$$\mathsf{Val}(a_0) \| \mathsf{Val}(a_1) \| \ldots \| \mathsf{Val}(a_{N-1})$$

in the above representations by the byte string representing the 384-bits of $\mathbf{p}$.

To encode a ciphertext $\mathbf{c} = (c_0, \mathbf{c}_1)$ where

$$c_0 = c_0 + c_1 \cdot X + \cdots + c_{\ell-1} \cdot X^{\ell-1}$$

and

$$\mathbf{c}_1 = (c'_0, c'_1, \ldots, c'_{N-1}),$$

we write $B \leftarrow \mathsf{Encode}(\mathbf{c})$ where:

$$B = \mathsf{pCode} \| \mathsf{Val}_2(\ell) \| \mathsf{Val}(c_0) \| \mathsf{Val}(c_1) \| \ldots \| \mathsf{Val}(c_{\ell-1}) \| \mathsf{Val}(c'_0) \| \mathsf{Val}(c'_1) \| \ldots \| \mathsf{Val}(c'_{N-1})$$

The inverse operation is denoted by $\mathbf{c} \leftarrow \mathsf{Decode}(B)$. Notice that the ciphertext also contains a reference to the parameter set to which it is related.

Chapter 3

Known Answer Test Values

The KAT subdirectory contains a number of files containing Known Answer Test values. The names are *not the same* as the ones NIST test files produce. This is because we would get a clash in names as we have two encryption and two encapsulation algorithms (one CPA and one CCA in each case). Thus we have renamed the files so that our naming is more consistent with what the contents represent.

We overview these here:

- The first two files give test data on intermediate routines, in particular the XOF and FFT algorithms.
- The second set of files gives the input and output data from the encryption and encapsulation programs.

XOF-KAT.txt	This file specifies KAT values for the XOF functionalities. Given an input string and a diversifier it gives the first 32 bytes of output of the associated XOF object. From this 32 bytes of output it also gives the finite field elements and Gaussian samples that would be generated from such an output string. This file thereby enables testing of the XOF output, and how the XOF output is processed into random elements consumed by our schemes.
FFT-KAT.txt	For each ring used in LIMA this gives the output of applying the FFT algorithm to the all ones vector. Thus this file enables testing of FFT algorithms.

lima_2p_512_EncCPA-KAT.txt lima_2p_1024_EncCPA-KAT.txt lima_2p_2048_EncCPA-KAT.txt lima_sp_1018_EncCPA-KAT.txt lima_sp_1306_EncCPA-KAT.txt lima_sp_1822_EncCPA-KAT.txt lima_sp_2062_EncCPA-KAT.txt	For each of the seven parameter sets we give input and output of key generation, encryption and decryption for Enc-CPA. For key generation we give the associated input seed, and then the output public key and secret key (using the encoding methods in Section 2.8). For encryption we give a message, seed and then the associated ciphertext. This is then decrypted to give the original message back.
lima_2p_512_EncCCA-KAT.txt lima_2p_1024_EncCCA-KAT.txt lima_2p_2048_EncCCA-KAT.txt lima_sp_1018_EncCCA-KAT.txt lima_sp_1306_EncCCA-KAT.txt lima_sp_1822_EncCCA-KAT.txt lima_sp_2062_EncCCA-KAT.txt	For each of the seven parameter sets we give input and output of key generation, encryption and decryption for Enc-CCA. For key generation we give the associated input seed, and then the output public key and secret key (using the encoding methods in Section 2.8); these are the same keys are earlier as we use the same initial seed to generate them. For encryption we give a message, seed and then the associated ciphertext. This is then decrypted to give the original message back.
lima_2p_512_EncapCPA-KAT.txt lima_2p_1024_EncapCPA-KAT.txt lima_2p_2048_EncapCPA-KAT.txt lima_sp_1018_EncapCPA-KAT.txt lima_sp_1306_EncapCPA-KAT.txt lima_sp_1822_EncapCPA-KAT.txt lima_sp_2062_EncapCPA-KAT.txt	For each of the seven parameter sets we give input and output of key generation, encryption and decryption for Encap-CPA. For key generation we give the associated input seed, and then the output public key and secret key (using the encoding methods in Section 2.8); these are the same keys are earlier as we use the same initial seed to generate them. For encapsulation we give a seed and then the associated ciphertext, and the key which it encapsulates. This is then decapsulated to check the same key is produced.
lima_2p_512_EncapCCA-KAT.txt lima_2p_1024_EncapCCA-KAT.txt lima_2p_2048_EncapCCA-KAT.txt lima_sp_1018_EncapCCA-KAT.txt lima_sp_1306_EncapCCA-KAT.txt lima_sp_1822_EncapCCA-KAT.txt lima_sp_2062_EncapCCA-KAT.txt	For each of the seven parameter sets we give input and output of key generation, encryption and decryption for Encap-CCA. For key generation we give the associated input seed, and then the output public key and secret key (using the encoding methods in Section 2.8); these are the same keys are earlier as we use the same initial seed to generate them. For encapsulation we give a seed and then the associated ciphertext, and the key which it encapsulates. This is then decapsulated to check the same key is produced.

Chapter 4

Security Analysis

In this chapter, we formally analyse the security of the schemes, i.e. we present provable security results. We also discuss the probability of decryption errors, i.e. correctness. Since correctness is simpler, we start there. In the next chapter, we will then discuss known attacks on the schemes and how we make use of these attacks to derive the parameter sets presented in Chapter 1.

4.1 Correctness

4.1.1 Correctness of the CPA-encryption scheme

We first look at correctness of the CPA-encryption scheme; namely whether the decryption algorithm Dec-CPA returns the same message that was input to the encryption algorithm Enc-CPA. Note that f in this routine is the truncation of the following element to degree at most ℓ:

$$\begin{aligned} c_0 - s \cdot c_1 &= (b \cdot v + d + \Delta_q \cdot \mu) - s \cdot (a \cdot v + e) \\ &= ((a \cdot s + e') \cdot v + d + \Delta_q \cdot \mu) - s \cdot (a \cdot v \cdot e) \\ &= (e' \cdot v + d + \Delta_q \cdot \mu) - s \cdot e \\ &= (e' \cdot v + d - s \cdot e) + \Delta_q \cdot \mu, \end{aligned}$$

which is equal to "small" plus $\Delta_q \cdot \mu$ since all of e', v, d, s, e are polynomials with small Gaussian-like coefficients. Thus, when multiplying by $\frac{2}{q}$, these all disappear and the only value that remains after rounding is $\frac{2}{q} \cdot \Delta_q \cdot \mu = \mu$. Observe that decryption works as long as $e' \cdot v + d - s \cdot e$ remains small in infinity norm. To ensure correctness we want to obtain a bound B on each coefficient E_k of the expression

$$d + v \cdot e' - s \cdot e,$$

that holds with high probability and then pick $q > 4 \cdot B$, so that decryption will be correct with high probability.

LIMA-2p: Writing v_i for the coefficients of v, e'_i for the coefficients of e' and so on, we find that the k-th coefficient of this term is equal to

$$E_k = d_k + \sum_{\substack{0 \le i,j < N, \\ i+j=k}} (v_i \cdot e'_j + e_i \cdot s_j) - \sum_{\substack{0 \le i,j < N, \\ i+j=N+k}} (v_i \cdot e'_j + e_i \cdot s_j).$$

LIMA-sp: The expression for E_k is now more complicated, due to the more complex form of reduction

```
from error import Scheme, RandomVariable

class Lima(Scheme):
    def __init__(self, n=1024, q=133121, eta=20, d_t=None, prec=None, sp=False):
        params = {"n": n, "q": q, "eta": eta, "d_t": d_t}
        Scheme.__init__(self, prec=prec, amplify=256, **params)
        self.sp = sp
        if sp:
            self._ell = 2*n
        else:
            self._ell = n

    @cached_method
    def _D(self):
        params = self.params
        prec = self.prec
        e_ = v = d = s = e = RandomVariable.CenteredBinomial(params["eta"], prec=prec)
        ell = self._ell

        D = RandomVariable.sum(e_ * v, ell)
        D += d
        if self.params["d_t"]:
            D += RandomVariable.RoundingError(params["q"], 2**params["d_t"])
        D -= RandomVariable.sum(s * e, ell)
        return D

Lima(n=512,  q=18433, sp=False, eta=20, d_t=False)()
Lima(n=1024, q=40961, sp=False, eta=20, d_t=False)()
Lima(n=2048, q=40961, sp=False, eta=20, d_t=False)()

Lima(n=1018, q=12521473, sp=True, eta=20, d_t=False)()
Lima(n=1306, q=48181249, sp=True, eta=20, d_t=False)()
Lima(n=1822, q=44802049, sp=True, eta=20, d_t=False)()
Lima(n=2062, q=16900097, sp=True, eta=20, d_t=False)()
```

Figure 4.1: Probability of decryption failure computation

modulo $\Phi_p(X)$. We have

$$E_k = d_k + \left(\sum_{i+j=k} s_i \cdot e_j + e'_i \cdot v_j\right) - \left(\sum_{i+j=N} s_i \cdot e_j + e'_i \cdot v_j\right) + \left(\sum_{i+j=N+1+k} s_i \cdot e_j + e'_i \cdot v_j\right).$$

Now, to compute B, we insist that the probability of failure of decryption be bounded by 2^{-128} and then numerically compute distribution convolutions to obtain B. This computation is accomplished using the code available at `https://bitbucket.org/malb/lwe-decryption-failure/` which is an adaptation of a script accompanying [BDK+17], available at `https://github.com/pq-crystals/kyber/blob/master/scripts/proba_util.py` We obtain the probabilities of failure to decrypt given in Table 4.1. It was obtained calling the code given in Figure 4.1.

4.1.2 Correctness of Dec-CCA

Correctness of the CCA encryption scheme follows analogously to the correctness of the CPA encryption scheme.

4.1.3 Correctness of Decap-CPA

Correctness of the CPA KEM scheme follows analogously to the correctness of the CPA encryption scheme.

n	q	scheme	$\log_2 \Pr[\text{fail}]$
512	18433	**LIMA-2p**	-134
1024	40961	**LIMA-2p**	-319
2048	40961	**LIMA-2p**	-175
1018	12521473	**LIMA-sp**	≤ -320
1306	48181249	**LIMA-sp**	≤ -320
1822	44802049	**LIMA-sp**	≤ -320
2062	16900097	**LIMA-sp**	≤ -320

Table 4.1: Probability of decryption failure

4.1.4 Correctness of Decap-CCA

Correctness of the CCA KEM scheme follows analogously to the correctness of the CPA encryption scheme.

4.2 Security of Gaussian Sampling

Formally, the usual worst-case to average-case security reductions for LWE will not apply to our scheme as we are not using a true rounded Gaussian as in [Reg05, LPR10] but only an approximation. However, we are well outside the range for the worst-case to average-case analysis to hold in any case, since $\sigma < \sqrt{n}$. However, there is no known attack which exploits these differences of distributions in LWE encryption resp. encapsulation. Thus we expect that our approximation of a Discrete Gaussian will pose no security problem in practice. Indeed, we could have selected errors from a suitably chosen uniform distribution, but we use the Gaussian to enable a finer grained correctness analysis above and to provide some design validation, having based the general design on a closely related problem which does have worst-case to average-case security.

Another concern might be the fact that in the safe-prime variant we do not define errors in the canonical embedding but in the polynomial embedding. An often claimed advantage of power-of-two cyclotomics is that choosing small errors in the polynomial embedding is equivalent to selecting small errors in the canonical embedding (where the actual Ring-LWE problem lies). However, for safe-prime cyclotomics the two embeddings are quite close geometrically (for example the ring constant defined in [DPSZ12] is very close to one for prime cyclotomics). Thus, we contend that this difference can be ignored, much like the difference between our approximate Gaussians and true Gaussians mentioned above.

4.3 Security Reductions

We are now ready to examine the formal security assurances of our two public-key encryption schemes and two key encapsulation mechanisms. Throughout this section, to aid the reader, we assume the public key and second ciphertext components are not located in the FFT domain. This is purely a notational convenience to aid exposition; converting between the standard and FFT domains is a keyless operation.

4.3.1 Hard Problems

We recall the Ring-LWE problem:

Definition 1 (Ring-LWE)**.** *Consider the following experiment: a challenger picks $s \in R_q$ and a bit $\beta \in \{0,1\}$ uniformly at random. The adversary is given an oracle which on empty input returns a pair $(a,b) \in R_q^2$, where if $\beta = 0$ the two elements are chosen uniformly at random, and if $\beta = 1$ the value a is chosen uniformly at random and b is selected such that $b = a \cdot s + e$ where e is selected according to a Gaussian distribution in the canonical embedding. The adversary's goal is to output a guess as to the bit β.*

In our situation, we slightly modify the above Ring-LWE problem. In particular, the secret values s are selected from χ_σ^N. It is well known that we may pick s from a "small" distribution without affecting security, see for example [MR09, ACPS09]. We therefore define the LIMA-LWE problem as follows.

Definition 2 (LIMA Ring-LWE)**.** *Let χ_σ denote the distribution defined earlier. Consider the following experiment: a challenger picks $s \in \chi_\sigma^N \subset R_q$ and a bit $\beta \in \{0,1\}$. The adversary $\mathcal{A}$ is given an oracle which on empty input returns a pair $(a,b) \in R_q^2$, where if $\beta = 0$ the two elements are chosen uniformly at random, and if $\beta = 1$ the value a is chosen uniformly at random and b is selected such that $b = a \cdot s + e$ where $e \in \chi_\sigma^N \subset R_q$. At the end of the experiment the adversary outputs its guess β' as to the hidden bit β. For an adversary which makes n_Q calls to its oracle and running in time t, we define*

$$\mathsf{Adv}^{\mathsf{LWE}}(\mathcal{A}, n_Q, t) = 2 \cdot \Big| \Pr[\beta = \beta'] - \frac{1}{2} \Big|.$$

We conjecture that $\mathsf{Adv}^{\mathsf{LWE}}(\mathcal{A}, n_Q, t)$ is negligible for all adversaries and all LIMA parameter sets. The only known "attack" on this problem is essentially via lattice reduction.

Conjecture 1. *For suitable choices of σ, N and q depending on the security parameter λ, $\epsilon = \mathsf{Adv}^{\mathsf{LWE}}(A, n_Q, t)$ is a negligible function in the security parameter λ. In particular, for all adversaries running in time t we have $t/\epsilon^2 \geq 2^\lambda$.*

We note that in the conjecture above we normalize the running time by success probability as $1/\epsilon^2$ — instead of the more customary $1/\epsilon$ — because we are considering a decision problem. In Section 5.1, we estimate the expected probabilities of an adversary solving this problem in a given time period via lattice reduction.

4.3.2 Security Reduction for the Basic Encryption Scheme

The IND-CPA security of our basic encryption scheme (KeyGen, Enc-CPA, Dec-CPA) is established in the following theorem, whose proof is given in the Appendix of the full version [AOP+17b] of [AOP+17a].

Theorem 1. *If the LWE/LWE problem is hard, then the scheme* (KeyGen, Enc-CPA, Dec-CPA) *is IND-CPA secure in the random oracle model. In particular, if there is an adversary $\mathcal{A}$ against the IND-CPA security of* (KeyGen, Enc-CPA, Dec-CPA) *in the random oracle model, then there are adversaries $\mathcal{B}$ and $\mathcal{D}$ such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}(\mathcal{A}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{LWE}}(\mathcal{B}, 1, t) + 2 \cdot \mathsf{Adv}^{\mathsf{LWE}}(\mathcal{D}, t).$$

4.3.3 Security Reduction for our IND-CCA Secure PKE scheme

Our construction of an IND-CCA secure encryption scheme uses the Fujisaki-Okamoto transform [FO99] applied to our basic scheme. Before we can apply this transform, we first need to establish its γ-uniformity.

Definition 3 (γ-Uniformity)**.** *Consider an IND-CPA encryption scheme given by the tuple of algorithms* (KeyGen, Enc-CPA, Dec-CPA) *with* Enc-CPA $: \mathcal{M} \times \mathcal{R} \longrightarrow \mathcal{C}$ *being the encryption function mapping messages and randomness to ciphertexts. Such a scheme is said to be γ uniform if for all public keys $\mathfrak{pk}$ output by* KeyGen*, all $m \in \mathcal{M}$ and all $c \in \mathcal{C}$ we have $\gamma(\mathfrak{pk}, m, c) \leq \gamma$, where*

$$\gamma(\mathfrak{pk}, m, c) = \Pr[r \in \mathcal{R} : c = \mathsf{Enc\text{-}CPA}(m, \mathfrak{pk}, r)].$$

The lemma below establishes that Ring-LWE-based encryption has low γ-uniformity. The proof can be found in [AOP+17a].

Lemma 1. *Let* (KeyGen, Enc-CPA, Dec-CPA) *with parameters N, χ_σ, q be the basic PKE scheme described in Section 2.4 and let σ such that $\Pr[X = x \mid X \leftarrow_r \chi_\sigma] \leq 1/2$ for any x. Then this scheme is γ-uniform with $\gamma \leq 2^{-N}$.*

Note that in our construction the condition $\forall x, \Pr[X = x \mid X \leftarrow_r \chi_\sigma] \leq 1/2$ is always satisfied by picking $\sigma > 1$. Also note that if we truncate c_0 to ℓ components then the above bound becomes $2^{-(N+\ell)}$ by considering d truncated to ℓ components directly as being sampled from χ_σ^ℓ. Applying the main result (Theorem 3) of Fujisaki and Okamoto [FO99], we obtain the following:[1] The extra $q_H \cdot 2^{-\mathsf{error}}$ term comes

[1] Using $k = N$ and $k_0 = 256$ in Theorem 3 of [FO99].

from the probability that a ciphertext which should decrypt incorrectly in the actual scheme is found to not decrypt incorrectly in the simulation. To add this extra term the proof in [FO99] needs to be modified as the knowledge extractor may return a decrypted message, when the actual scheme returns a decryption error. As H (the XOF used in encryption) is modelled as a random oracle the only way the adversary can accomplish, and hence distinguish the real world from the simulation, would be for the adversary to query the random oralce until they found "bad" random coins which would result in an invalid decryption. Note, even then it is hard for the adversary to actually distinguish what is "bad" random coins from "good" ones. The probability random coins output by the XOF being bad (for a given message m) is $2^{-\mathsf{error}}$ as we already established.

Theorem 2. *Suppose that* (KeyGen, Enc-CPA, Dec-CPA) *is* (t', ϵ') *IND-CPA secure and* γ*-uniform. For any* q_H, q_D*, the scheme* (KeyGen, Enc-CCA, Dec-CCA)*, derived from* (KeyGen, Enc-CPA, Dec-CPA) *as in Section 2.5, is* (t, ϵ) *IND-CCA secure for any adversary making at most* q_H *queries to* XOF *(modelled as a random oracle) and at most* q_D *queries to the decryption oracle, where*

$$t = t' - q_H \cdot (T_{\mathsf{Enc}} + v \cdot N),$$
$$\epsilon = \epsilon' \cdot (1-\gamma)^{-q_D} + q_H \cdot \left(2^{-255} + \cdot 2^{-\mathsf{error}}\right).$$

Here T_{Enc} *is the running time of the encryption function and* v *is a constant.*

4.3.4 Security Reduction for our IND-CPA KEM

Theorem 3. *If the LIMA-LWE problem is hard then the KEM* (KeyGen, Encap-CPA, Decap-CPA) *is IND-CPA in the random oracle model.*

Proof. Given an adversay $\mathcal{A}$ against the IND-CPA security of (KeyGen, Encap-CPA, Decap-CPA), we can trivially construct an adversary $\mathcal{B}$ against the IND-CPA security of (KeyGen, Enc-CPA, Dec-CPA) such that

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}(\mathcal{B}).$$

The result follows immediately. □

4.3.5 Security Reduction for our IND-CCA KEM

As remarked earlier, our IND-CCA KEM construction is obtained by applying the construction of Dent [Den03, Table 5]. This builds an IND-CCA secure KEM from a OW-CPA secure PKE scheme. By Theorem 1, we know that our encryption scheme is IND-CPA secure. It also has large message space. It follows that it is OW-CPA secure. Directly applying the generic result [Den03, Theorem 5], after modifying the proof as in the case of the FO-transform above to take into account the possibility of decryption failures, we would obtain the following security theorem for our IND-CCA KEM:

Theorem 4. *Suppose there is an adversary* $\mathcal{A}$ *which breaks the IND-CCA security of* (KeyGen, Encap-CCA, Decap-CCA) *in the random oracle model, with advantage* ϵ*, running in time* t*, making at most* q_D *decapsulation queries,* q_H *queries to the random oracle implementing the XOF function and* q_K *queries to the random oracle implementing the KDF. Then there is an adversary* $\mathcal{B}$ *breaking the OW-CPA security of the underlying encryption scheme* (KeyGen, Enc-CPA, Dec-CPA) *running in time essentially* t*, with advantage* ϵ'' *such that*

$$\varepsilon \leq (q_D + q_H + q_K) \cdot \varepsilon' + q_D \cdot \left(\frac{1}{2^{\ell'}} + \gamma\right) + q_H \cdot 2^{-\mathsf{error}}.$$

where $\ell' \geq 384$ *is the bit-size of* $\mathbf{r}$ *in our construction, and* 2^{error} *corresponds to the probability of a valid ciphertext decrypting incorrectly.*

The problem with this result is that it does not give a very tight reduction. We thus presented a new tight proof of our construction, in [AOP+17a], which is *not generic*, i.e. we make explicit use of the Ring-LWE based construction of the underlying encryption scheme.[2] This proof did not take into account the decryption error probability; but this can be easily fixed (much as in the fix to the FO transform outlined above) since the same argument can be applied to obtain the extra distinguishing of $q_H \cdot 2^{-\mathsf{error}}$ term between game G_0 and G_1. However, essentially the same scheme has also been analysed in [HHK17] via a generic reduction (as opposed to a specific reduction) to obtain a similar result in the presence of decryption errors. We present the modified theorem statement from [AOP+17a], with a similar statement being obtained in [HHK17].

Theorem 5. *If the LIMA-LWE problem is hard then* $(\mathsf{KeyGen}, \mathsf{Encap\text{-}CCA}, \mathsf{Decap\text{-}CCA})$ *is an IND-CCA secure KEM In the random oracle model. In particular if* $\mathcal{A}$ *is an adversary against the IND-CCA security of* $(\mathsf{KeyGen}, \mathsf{Encap\text{-}CCA}, \mathsf{Decap\text{-}CCA})$ *running in time* t*, making at most* q_H *queries to the random oracle implementing the XOF function and at most* q_K *queries to the random oracle implementing the KDF, then there are adversaries* $\mathcal{B}$ *and* $\mathcal{D}$ *such that*

$$\epsilon \leq 2 \cdot \left(\epsilon' + \epsilon'' + \frac{q_H + q_K}{2^{\ell'}} + q_D \cdot \gamma + q_H \cdot 2^{-\mathsf{error}} \right),$$

where $\epsilon = \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}(\mathcal{A}, t)$*,* $\epsilon' = \mathsf{Adv}^{\mathsf{LWE}}(\mathcal{B}, 1, t)$*,* $\epsilon'' = \mathsf{Adv}^{\mathsf{LWE}^\dagger}(\mathcal{D}, t)$*, and* $\ell' \geq 384$ *is the bit-size of* $\mathbf{r}$ *in our construction.*

[2] Note that the roles of ℓ and ℓ' are reversed here compared to the original presentation in [AOP+17a].

Chapter 5

Known Attacks

Due to our security reduction results the main *mathematical* attacks against our schemes will those on the underlying LWE problem. All current best known attacks against the LWE problem are based on lattice reduction. Thus, we spend the first part of this section discussing lattice reduction. We then go on to discuss how lattice reduction can be applied to attack the LWE problem and hence our scheme. Using estimates from this analysis, we are able to estimate the security offered by our selected parameter sets. We then go on to consider other potential attack strategies, including *implementation* attacks.

5.1 Lattice Reduction

Lattice reduction algorithms have been studied for many years in [LLL82, Sch87, GN08, HPS11, CN11, MW16]. From a theoretical perspective, one of the best lattice reduction algorithm is the slide reduction algorithm from [GN08]. Alternatively, we may call the BKZ algorithm [Sch87] and its variants [HPS11, CN11], performing best in practice. The BKZ algorithm is parameterized by a block size β and consists of repeated calls to an oracle solving the Shortest Vector Problem (SVP) in dimension β combined with calls to the famous LLL algorithm. After performing BKZ-β reduction on a random lattice the first vector in the transformed lattice basis will have norm $\delta_0^m \cdot \det(\Lambda)^{1/m}$ where $\det(\Lambda)$ is the determinant of the lattice under consideration, m its dimension and the root-Hermite factor δ_0 is a constant based on the block size parameter β. Increasing the parameter β leads to a smaller δ_0 but also leads to an increase in run-time; the run-time grows at least exponentially in β.

In estimating costs for running an attack via BKZ we have a number of different costs which we need to estimate.

- An attack strategy has a probability of success (see for example the ϵ for the dual attack on LWE mentioned later) depending on the block size β. It is often more efficient to repeatedly perform lattice reduction with a block size $\beta' < \beta$ which leads to a low probability of success for solving LWE than to target a high success probability directly using a bigger block size β. Thus, an entire attack strategy may need to be repeated a number of times to amplify the chance of success.

- Each call to BKZ will itself make a certain number of calls to an internal SVP oracle. The precise number depends on how one optimizes BKZ (e.g. using early abort).

- Each SVP oracle will itself require a certain amount of time to execute. It is at this point that potential quantum speed-ups could come to fruition.

In what follows, we make the following assumptions about the BKZ algorithm and lattice reduction in general.

Block Size. To establish the required block size β we solve

$$\log \delta_0 = \log \left(\frac{\beta}{2\pi e} (\pi\beta)^{\frac{1}{\beta}} \right) \cdot \frac{1}{2(\beta - 1)}$$

for β, see the PhD Thesis of Yuanmi Chen [Che13] for a justification of this.

Cost of SVP. Several algorithms can be used to realize the SVP oracle inside BKZ. Asymptotically, the fastest known algorithms are sieving algorithms. The fastest, known *classical* algorithm runs in time $2^{0.292\,\beta+o(\beta)}$ [BDGL16]. The fastest, known *quantum* algorithm runs in time $2^{0.265\,\beta+o(\beta)}$ [Laa15]. We note that this algorithm applies Grover's algorithm assuming an optimal speed-up. In [ADPS16] a third category – "paranoid" – was introduced, which gives a plausible lower bound for currently known techniques as $2^{0.2075\,\beta+o(\beta)}$, but we do not make use of this bound here. Experiments in [BDGL16] suggest $o(\beta) \approx 16$. All times are expressed in elementary operations mod q. For estimating time on a classical computer, we assume that such an operation can be performed using 1 clock cycle on a modern 64-bit CPU. To obtain a rough binary gate count, these numbers would need to be multiplied by $\log_2 q$. However, this will affect the final result only marginally. For the quantum estimates, the basic unit is a Grover iteration which certainly is much more expensive than 1 cycle on a classical computer.

Calls to SVP. The BKZ algorithm proceeds by repeatedly calling an oracle for computing a shortest vector on a smaller lattice of dimension β. In each "tour", n such calls are made and the algorithm is typically terminated once it stops making sufficient progress in reducing the basis. Experimentally, it has been established that only the first few tours make significant progress [Che13], so we can assume that one BKZ call costs as much as $4n$ calls to the SVP oracle. However, it seems plausible that the cost of these calls can be amortized across different calls, which is why [ADPS16] assumes the cost of BKZ to be the same as *one* SVP oracle call.

BKZ Cost. In summary, we assume a call to BKZ-β costs $2^{0.292\,\beta+16}$ operations classically and $2^{0.265\,\beta+16}$ operations quantumly. The additive constant 16 accounts for the $o(\beta)$ overhead of sieving and the fact that the SVP oracle needs to be called several times in BKZ.

Calls to BKZ. To pick parameters, we normalize running times to a fixed success probability. That is, all our expected costs are for an adversary winning with probability 51%. However, as mentioned above, it is often more efficient to run some algorithm many times with parameters that have a low probability of success instead of running the same algorithm under parameter choices which ensure a high probability of success. Thus, in general, we would expect several calls to BKZ to be required: $\approx 1/\varepsilon$ for search problems resp. $\approx 1/\varepsilon^2$ for decision problems.

5.2 Solving LWE

Having discussed lattice reduction, we now show how this can be applied to solve the LWE problem. Since, according to current knowledge, there is no difference between the attacks on Ring-LWE and standard LWE, to simplify the presentation we look at standard LWE.

Standard LWE is the problem of determining given $(A, \vec{c}) \in \mathbb{Z}_q^{m \times N} \times \mathbb{Z}_q^m$ whether $\vec{c} = A \cdot \vec{s} + \vec{e}$ or whether $\vec{c}$ is chosen uniformly at random. In what follows, $\alpha = \sqrt{2\pi}\sigma/q$. Here, we use the matrix form of writing LWE, where m is the number of samples made available to an attacker, N is the dimension of the secret, and q the modulus as before. In what follows, we will assume the attacker has access to as many samples as it requires, which is optimistic for the attacker, i.e. it gets to choose m.

We may pursue one of the two following strategies for solving LWE, called the primal and dual strategies. There are other possible strategies but we will point out that they are not competitive.

Primal. Find some $\vec{s}'$ such that $\|\vec{w} - \vec{c}\|$ with $\vec{w} = \vec{A} \cdot \vec{s}'$ is minimized, under the guarantee that $\vec{w}$ is not too far from $\vec{c}$. This is known as the Bounded Distance Decoding problem (BDD). To solve BDD, we may embed the BDD instance into a unique SVP (uSVP) instance and apply lattice reduction to solve it. To predict the required root-Hermite factor, [ADPS16] gives

$$\sqrt{\beta}\sigma \leq \delta_0^{2\beta-d} \cdot \mathrm{Vol}(L)^{1/d},$$

where $d = m + 1$ is the dimension of the lattice L. For the primal attack, only one BKZ call suffices, i.e. the attack either succeeds with high probability or it fails. To solve BDD, we may also perform lattice reduction followed by lattice point enumeration [LP11, LN13]. However, since, according to our estimates, this approach does not improve performance considerably compared to the uSVP approach, we do not consider it further here.

Dual. Find a short $\vec{y}$ in the integral row span of A. This problem is known as the Short Integer Solution problem (SIS). Given such a $\vec{y}$, we can then compute $\langle \vec{y}, \vec{c} \rangle$. On the one hand, if $\vec{c} = A \cdot \vec{s} + \vec{e}$, then $\langle \vec{y}, \vec{c} \rangle = \langle \vec{y} \cdot A, \vec{s} \rangle + \langle \vec{y}, \vec{e} \rangle \equiv \langle \vec{y}, \vec{e} \rangle \pmod{q}$. If $\vec{y}$ is short then $\langle \vec{y}, \vec{e} \rangle$ is also short. On the other hand, if $\vec{c}$ is uniformly random, so is $\langle \vec{y}, \vec{c} \rangle$, cf., for example, [LP11]. Following [APS15], the required log root-Hermite factor is

$$\log \delta_0 = \frac{\log^2 \left(\alpha \left(\sqrt{\ln(\frac{1}{\epsilon})}/\pi \right)^{-1} \right)}{4\,N \log q},$$

where ϵ is the advantage of distinguishing $\langle \vec{c}, \vec{e} \rangle$ from uniform mod q.

Note that the dual attack solves the decision version of LWE. Thus, applying the Chernoff bound to amplify an advantage ϵ to a constant advantage, we need to perform $\approx 1/\epsilon^2$ experiments and pick by majority vote. However, for the dual attack, too, we are going to assume that *one* BKZ call is sufficient regardless of ϵ. This BKZ-β is then followed by $\approx 1/\epsilon^2$ calls to LLL. This assumption is justified heuristically in that we can rerandomise an already reduced basis followed by some light lattice reduction such as LLL to achieve a different basis which is almost as reduced as the input [Alb17]. In contrast to [Alb17], however, we are going to assume that the LLL reduced bases have the same quality as the initial BKZ-β reduced basis, which is optimistic for the attacker. As a consequence, we assume that amplifying success probability is relatively cheap compared to previous work.

Other Methods. The dual strategy can also be realized using variants of the BKW algorithm [GJS15, KF15]. However, for the parameter choices considered here, these algorithms are not competitive with lattice-reduction based algorithms.

Another potential vector of attack is to combine combinatorial methods with lattice reduction leading to the "hybrid attack". Recently, this attack was revisited in the quantum setting [GvVW17] and the authors found that it offers not advantage over the attacks considered here if (a) the secret is not unusually short and (b) quantum sieving is assumed to realise the SVP oracle inside BKZ. Both conditions are satisfied in our security analysis.

Finally, Arora and Ge proposed an asymptotically efficient algorithm for solving LWE [AG11], which was later improved in [ACF+15]. However, these algorithms involve large constants in the exponent, ruling them out for parameters typically considered in cryptography such as here.

5.3 Parameter Analysis

We now analyze the parameters of our scheme to determine the estimated classical and quantum hardness, given the various options for lattice based attacks presented above. In selecting our parameters, we implemented the following strategy (which is implemented in Sage code in Appendix A). We first recall that the choice on q is restricted in the following ways:

LIMA-2p $q \equiv 1 \pmod{2 \cdot N}$.

LIMA-sp $q \equiv 1 \pmod{2^e \cdot p}$, where $e = \lceil \log_2(2 \cdot p) \rceil$ and $p = N + 1$.

Secondly, note that for a given dimension N, there is a minimal q of the right form such that

$$q > 4 \cdot B, \tag{5.1}$$

where the bound B comes from Section 4.1. Thus, given N and a type, we pick the smallest q satisfying Equation 5.1 subject to the correctness constraints. We always use the same σ. Hence, given N, we can derive the remaining parameters.

Then, given N, q, σ we estimate the cost of solving an LWE instance with such parameters as outlined above, i.e. applying the primal or the dual attack and considering different classical or quantum implementations for the SVP oracle required by the BKZ algorithm. Using this strategy and calling the code in the appendix, we can estimate the hardness of the parameter sets. In Table 5.1 we present for each parameter set the best classical and quantum attack given our model above. For more precise details on these estimates we give all the data in Table 5.2.

			LIMA-2p			
N	q	$\lceil (N \cdot \lceil \log_2 q \rceil)/8 \rceil$	classical	strategy	quantum	strategy
512	18433	960	133.3	dual	124.4	dual
1024	133121	2048	260.1	dual	237.6	dual
2048	184321	4096	569.0	primal	517.9	primal
			LIMA-sp			
$N = 2\,p'$	q	$\lceil (N \cdot \lceil \log_2 q \rceil)/8 \rceil$	classical	strategy	quantum	strategy
1018	12521473	3054	151.8	primal	139.2	primal
1306	48181249	4245	183.3	primal	167.8	primal
1822	44802049	5922	271.5	primal	247.9	primal
2062	16900097	6444	329.6	dual	303.5	dual

Table 5.1: Parameter choices for LIMA-2p and LIMA-sp: N is the ring dimension, q the modulus, $\lceil (N \cdot \lceil \log_2 q \rceil)/8 \rceil$ is the size of one ring element in bytes. We also list the best known running times ($\log_2$ of number of elementary operations) for attacking the underlying LWE problem in the classical and in the quantum setting with success probability at least 51%. The strategy columns state which of the strategies in Section 5.2 is most efficient and hence listed here.

5.4 Mapping to NIST Security Levels

NIST asks for security levels to be expressed in relation to the cost of classically and quantumly breaking AES and SHA3. We reproduce NIST's estimates for these tasks in Table 5.3. In particular, note that NIST parameterizes its quantum predictions by MAXDEPTH: the maximum depth a quantum circuit is expected to support.

Looking at Table 5.3, we observe that quantum computers will not provide any advantage if MAXDEPTH $< 2^{27}$ ($md < 27$ in our notation), if we furthermore assume that a single quantum gate costs at least as much as a classical gate. Thus, in what follows, we will assume MAXDEPTH is at least 2^{27}.

In contrast, our quantum estimates assume perfect Grover speed-up, i.e. we potentially assume a more powerful quantum adversary than NIST depending on the value of MAXDEPTH. Thus, to map our cost estimates to NIST, we say that a LIMA set of parameters offers security comparable to AES-x resp. SHA3-x whenever our log-cost estimates is bigger than NIST's quantum log-cost estimate minus 27 resp. NIST's classical log-cost estimate. Thus, we obtain Table 5.4.

We highlight that, technically, our $n = 512$ estimate does not reach the cost of 143. We consider it sufficiently close regardless to claim AES-128 equivalent security.

strategy	adversary	$\log_2$ cost	δ_0	dim	β	repeat
Type 1, $n = 512$, $q = 18433$						
primal	classical	136.0	1.003908	1118	411	$2^{0.0}$
dual	classical	133.3	1.003980	1125	401	$2^{92.4}$
primal	quantum	124.9	1.003908	1118	411	$2^{0.0}$
dual	quantum	124.5	1.003944	1130	406	$2^{85.4}$
Type 1, $n = 1024$, $q = 40961$						
primal	classical	264.8	1.002307	2147	852	$2^{0.0}$
dual	classical	260.1	1.002339	2157	836	$2^{189.4}$
primal	quantum	241.8	1.002307	2147	852	$2^{0.0}$
dual	quantum	237.6	1.002339	2157	836	$2^{189.4}$
Type 1, $n = 2048$, $q = 40961$						
primal	classical	569.0	1.001246	4103	1894	$2^{0.0}$
dual	classical	570.2	1.001244	4183	1898	$2^{361.8}$
primal	quantum	517.9	1.001246	4103	1894	$2^{0.0}$
dual	quantum	519.0	1.001244	4183	1898	$2^{361.8}$
Type 2, $n = 1018$, $q = 12521473$						
primal	classical	151.8	1.003584	2080	465	$2^{0.0}$
dual	classical	151.8	1.003589	2155	465	$2^{89.4}$
primal	quantum	139.2	1.003584	2080	465	$2^{0.0}$
dual	quantum	140.4	1.003610	2149	461	$2^{97.8}$
Type 2, $n = 1306$, $q = 48181249$						
primal	classical	183.3	1.003091	2713	573	$2^{0.0}$
dual	classical	183.3	1.003091	2736	573	$2^{109.8}$
primal	quantum	167.8	1.003091	2713	573	$2^{0.0}$
dual	quantum	167.8	1.003091	2736	573	$2^{109.8}$
Type 2, $n = 1822$, $q = 44802049$						
primal	classical	271.5	1.002260	3734	875	$2^{0.0}$
dual	classical	272.4	1.002254	3776	878	$2^{162.4}$
primal	quantum	247.9	1.002260	3734	875	$2^{0.0}$
dual	quantum	248.7	1.002254	3776	878	$2^{161.8}$
Type 2, $n = 2062$, $q = 16900097$						
primal	classical	336.3	1.001904	4190	1097	$2^{0.0}$
dual	classical	329.6	1.001935	4213	1074	$2^{277.4}$
primal	quantum	306.7	1.001904	4190	1097	$2^{0.0}$
dual	quantum	303.5	1.001920	4229	1085	$2^{244.4}$

Table 5.2: Cost of primal and dual attacks for best known quantum [Laa15] and classical [BDGL16] adversaries. The column "dim" holds the dimension of the lattice considered. The column "repeat" gives the number of times such a reduction would have to be repeated to amplify the success probability to $> 50\%$, depending on whether we are solving a search or decision problem. This cost is already taken into account in the "$\log_2$ cost" given. The primal attack is predicted to always succeed with probability close to one.

Algorithm	Quantum Gates	Classical Gates
AES 128	$170 - \log_2 \text{MAXDEPTH}$	143
SHA3-256	—	146
AES 192	$233 - \log_2 \text{MAXDEPTH}$	207
SHA3-384	—	210
AES 256	$298 - \log_2 \text{MAXDEPTH}$	272
SHA3-512	—	274

Table 5.3: NIST Cost Estimates for AES and SHA3.

n	q	NIST Level	Comment
		LIMA-2p	
512	18433	AES-128	classical/quantum estimates ≈ 128
1024	40961	SHA-384	classical/quantum estimates > 210
2048	40961	SHA-512	classical/quantum estimates > 274
		LIMA-sp	
1018	12521473	AES-128	classical/quantum estimates ≈ 143
1306	48181249	SHA-256	classical/quantum estimates > 146
1822	44802049	AES-192	classical/quantum estimates > 207
2062	16900097	SHA-512	classical/quantum estimates > 274

Table 5.4: NIST Mapping

5.5 (Quantum) Algebraic Attacks

A potential concern for Ring-LWE based schemes are attacks exploiting the additional structure implied by the ring setting. That is, while the best known attacks against Ring-LWE work by treating it as a normal LWE instance, there could potentially exist additional structural weaknesses.

For example, polynomial-time quantum attacks finding short generators of principal ideals in cyclotomic rings were introduced in [CGS14, CDPR16]. This result was then extended in [CDW17] to short elements in general ideal lattices. While the latter will only recover vectors which are sub-exponentally longer than a shortest vector, this improves on what can be achieved classically, i.e. exponential approximation factors [LLL82]. On the other hand, we note that these approximation factors are too big to apply to most Ring-LWE schemes. Furthermore, it is currently not clear if these results to extend to Ring-LWE, i.e. it is possible that Ring-LWE is strictly harder than Ideal-SVP.

Another line of attack was first sketched in [GS02] and recently revived in [ABD16, CJL16b]. In this attack on NTRU with very large moduli, an attacker exploits the presence of subfields to map the challenge instance to a smaller dimensional instance. Soon after, it was shown that the presence of subfields is not required [KF17]. On the other hand, [BBdV+17] shows that in some rings subfields can be exploited to find shorter vectors. However, it is not clear how to extend these attacks to Ring-LWE and to the rings considered here. Thus, at present, they do not seem to pose a threat to Ring-LWE based public-key encryption.

5.6 Implementation Attacks

Whilst we use constant time methods for sampling from an approximate Gaussian, we do have an obvious timing side-channel in that our encryption algorithms use a form of rejection sampling of the underlying random coins. We contend that such a non-constant time implementation does not cause a security concern. Firstly, this occurs in the public-key encryption and key encapsulation operations only, and so involves no

side-channel on the secret key. Secondly, any message-dependent leakage (for example in our IND CCA encryption scheme) is masked by first applying SHA-3 to the message and randomness, before deciding whether to reject. Thirdly, and probably most importantly, the probability of rejection is so small that the probability that it occurs can actually be ignored in practice.

Chapter 6

Seed-Sizes, Bandwidth, Performance, Advantages and Disadvantages

Here we examine various aspects of our constructions.

6.1 Seed Size Discussion

In general, and in the test routines attached to this submission, we use 48 bytes for the seed sizes for all routines (key generation and public-key encryption/key encapsulation), except for the IND-CCA encryption algorithm, where we use only 32 bytes of random seed, and the IND-CPA encapsulation algorithm, where we use 80 bytes of random seed. These are our recommendations for use with the LIMA system. The reason for the restriction in the case of IND-CCA encryption is that we want to maximize the possible message size which could be encrypted in the IND-CCA case. We believe 32 bytes of randomness to be sufficient for most purposes, but err on the side of caution in recommending 48 bytes in most instances. For the case of IND-CPA encapsulation we use 80 bytes of randomness so as to enable a simple implementation to encapsulate a 32 byte key. If 80 bytes of randomness is considered too much then expansion can be done via the XOF. Table 6.1 summarises our choices.

Operation	Recommended Seed Size (Bytes)
IND-CPA Encryption	
KeyGen	48
Enc-CPA	48
IND-CCA Encryption	
KeyGen	48
Enc-CCA	32
IND-CPA Encapsulation	
KeyGen	48
Encap-CPA	80
IND-CCA Encapsulation	
KeyGen	48
Encap-CCA	48

Table 6.1: Recommended Seed Sizes

In the NIST subdirectories we present test programs using the NIST test harness to produce KAT vectors for the various algorithms and parameters. For the case of key encapsulation we always encapsulate a 32 byte key, and for the case of encryption we encrypt message lengths up to the maximum supported by the parameter set/primitive combination.

6.2 Bandwidth

A major concern in relation to lattice based schemes is the consumption in bandwidth of public keys, secret keys and ciphertexts. In what follows we assume that finite field elements are held in an uncompressed byte format, i.e. an element in $\mathbb{F}_q$ takes $\lceil \log_{256} q \rceil$ bytes to represent it, and not (the possibly lower) $\lceil \log_2 q \rceil$ bits.

In the Table 6.2 we present various sizes (in bytes) for the different parameter sets when stored as byte strings using our Encode routines. For encryption operations we assume first a message of 32-bytes (256 bits), and then a message of the maximum possible length enabled by the scheme. For key encapsulation operations we assume a key of length 32-bytes (256 bits). In all cases we assume the seed lengths given in the previous subsection. We present two sizes for the public key size, one the standard size (which is the one used in the Reference and Optimized implementations), and one for the compressed representation given in the text above. A similar compression can be obtained for the secret key, as also alluded to above. Further bandwidth savings are possible by compressing the representation of the finite field elements, using (say) Huffman coding. We then obtain the (approximate) sizes (again in whole bytes) in Table 6.3, giving savings of between 20 and 30 percent. Here we just give the public and secret key sizes, with similar savings for the other measures.

	LIMA-2p	**LIMA-2p**	**LIMA-2p**	**LIMA-sp**	**LIMA-sp**	**LIMA-sp**	**LIMA-sp**
N	512	1024	2048	1018	1306	1822	2062
q	18433	40961	40961	12521473	48181249	44802049	16900097
$b = \lceil \log_{256} q \rceil$	2	2	2	3	4	4	4
$\lvert \mathfrak{sk} \rvert = 3 \cdot N \cdot b + 1$	3073	6145	12289	9163	15673	21865	24745
$\lvert \mathfrak{pk} \rvert = 2 \cdot N \cdot b + 1$	2049	4097	8193	6109	10449	14577	16497
$\lvert \mathfrak{pk} \rvert = N \cdot b + 385$	1409	2433	4481	3439	5609	7673	8633
$\lvert c_{\mathsf{Enc\text{-}CPA}} \rvert = (N + 256) \cdot b + 3$	1539	2563	4611	3825	6251	8315	9275
$\lvert c_{\mathsf{Enc\text{-}CCA}} \rvert = (N + 512) \cdot b + 3$	2051	3075	5123	4593	7275	9339	10299
$\lvert c_{\mathsf{Enc\text{-}CPA}} \rvert = 2 \cdot N \cdot b + 3$	2051	4099	8195	6111	10451	14579	16499
$\lvert c_{\mathsf{Enc\text{-}CCA}} \rvert = 2 \cdot N \cdot b + 3$	2051	4099	8195	6111	10451	14579	16499
$\lvert c_{\mathsf{Encap\text{-}CPA}} \rvert = (N + 256) \cdot b + 3$	1539	2563	4611	3825	6251	8315	9275
$\lvert c_{\mathsf{Encap\text{-}CCA}} \rvert = (N + 384) \cdot b + 3$	1795	2819	4867	4209	6763	8827	9787

Table 6.2: Bandwidth Estimates for Various Parameters

	LIMA-2p	**LIMA-2p**	**LIMA-2p**	**LIMA-sp**	**LIMA-sp**	**LIMA-sp**	**LIMA-sp**
N	512	1024	2048	1018	1306	1822	2062
q	18433	40961	40961	12521473	48181249	44802049	16900097
$b' = \lceil \log_2 q \rceil$	14	15	15	24	26	26	25
$\lvert \mathfrak{sk} \rvert \approx \lceil 3 \cdot N \cdot b'/8 \rceil$	2688	5760	11520	9162	12734	17765	19332
$\lvert \mathfrak{pk} \rvert \approx \lceil 2 \cdot N \cdot b'/8 \rceil$	1792	3840	7680	6108	8489	11843	12888

Table 6.3: Bandwidth Estimates Under Huffman Encoding

6.3 Performance

We give some performance numbers in milliseconds and cycle counts for our seven different parameter sets in Tables 6.4–6.10. For these timings we utilize the seed sizes defined above, and give times for encrypting (resp. encapsulating) a 32 byte message (resp. secret key). The underlying Keccak routines are compiled using the `asmX86-64` target for the Keccak library, and where run no an Intel i7-7700 CPU running at 4.20 GhZ and with a core of 8192 KB. All performance numbers were computed using the *Optimized* variant supplied in our submission. This is an ANSI-C compliant implementation.

The seed for each routine is generated by called `dev/urandom` and then using this "as is"; since we pass the output to an XOF, we deemed it unnecessary to carry our further processing via the NIST DRBG (as used in the NIST test harness).

Note that on some processors the encryption/encapsulation time is sometimes much slower than the associated decryption/decapsulation time. Upon investigation we found that this is because on this test system the time to access `dev/urandom` is an order of magnitude different compared to other systems. Since encryption requires entropy (although only a small amount), this equates to making encryption slower than decryption. This is despite our IND-CCA decryption algorithms involving a re-encryption step, albeit with a known random seed.

Operation	Time	Cycle Count
Key Generation	0.098117 ms	412088
CPA Encryption	0.062333 ms	261796
CPA Decryption	0.021327 ms	89570
CCA Encryption	0.062224 ms	261339
CCA Decryption	0.079494 ms	333870
CPA Encapsulation	0.067396 ms	283059
CPA Decapsulation	0.020634 ms	86659
CCA Encapsulation	0.063978 ms	268704
CCA Decapsulation	0.081038 ms	340354

Table 6.4: Timings for **LIMA-2p-512**

Operation	Time	Cycle Count
Key Generation	0.189556 ms	796127
CPA Encryption	0.121375 ms	509773
CPA Decryption	0.038555 ms	161928
CCA Encryption	0.125735 ms	528083
CCA Decryption	0.153524 ms	644792
CPA Encapsulation	0.126111 ms	529662
CPA Decapsulation	0.039151 ms	164431
CCA Encapsulation	0.122650 ms	515127
CCA Decapsulation	0.151050 ms	634402

Table 6.5: Timings for **LIMA-2p-1024**

The encryption and decryption algorithms could be further optimized to run the FFT in parallel, using AVX2 and AVX-512 instructions. We have not implemented these optimizations since recent research[1] shows that this results in severe performance penalties on real systems. In particular, when using AVX2 and

[1] See `https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/`.

Operation	Time	Cycle Count
Key Generation	0.391186 ms	1642968
CPA Encryption	0.247980 ms	1041503
CPA Decryption	0.075480 ms	317011
CCA Encryption	0.241985 ms	1016324
CCA Decryption	0.303821 ms	1276036
CPA Encapsulation	0.253470 ms	1064562
CPA Decapsulation	0.077322 ms	324749
CCA Encapsulation	0.248788 ms	1044900
CCA Decapsulation	0.302238 ms	1269382

Table 6.6: Timings for **LIMA-2p-2048**

Operation	Time	Cycle Count
Key Generation	0.381527 ms	1602397
CPA Encryption	0.317866 ms	1335025
CPA Decryption	0.103435 ms	434421
CCA Encryption	0.306117 ms	1285681
CCA Decryption	0.407695 ms	1712301
CPA Encapsulation	0.313137 ms	1315159
CPA Decapsulation	0.101092 ms	424585
CCA Encapsulation	0.315350 ms	1324455
CCA Decapsulation	0.398488 ms	1673629

Table 6.7: Timings for **LIMA-sp-1018**

Operation	Time	Cycle Count
Key Generation	0.699387 ms	2937394
CPA Encryption	0.588364 ms	2471102
CPA Decryption	0.189588 ms	796258
CCA Encryption	0.599666 ms	2518571
CCA Decryption	0.777387 ms	3264991
CPA Encapsulation	0.579267 ms	2432891
CPA Decapsulation	0.192002 ms	806397
CCA Encapsulation	0.583512 ms	2450728
CCA Decapsulation	0.759146 ms	3188381

Table 6.8: Timings for **LIMA-sp-1306**

AVX-512 instructions, the clock speed is reduced due to power issues. If a piece of code only uses AVX2 and AVX-512 instructions, then the overall speedup that can be achieved may be very good because the high parallelization offsets the lower clock speed. However, if the code uses additional, non-AVX2/AVX-512, instructions, then those instructions run at the lower clock speed and the overall performance is impaired. This means that using AVX2 and AVX-512 instructions for a task like encryption, which is typically run in conjunction with arbitrary other tasks, may be problematic.

Operation	Time	Cycle Count
Key Generation	0.772168 ms	3243073
CPA Encryption	0.619998 ms	2603963
CPA Decryption	0.199730 ms	838859
CCA Encryption	0.618675 ms	2598404
CCA Decryption	0.816476 ms	3429160
CPA Encapsulation	0.634172 ms	2663493
CPA Decapsulation	0.200243 ms	841008
CCA Encapsulation	0.627362 ms	2634891
CCA Decapsulation	0.812481 ms	3412379

Table 6.9: Timings for **LIMA-sp-1822**

Operation	Time	Cycle Count
Key Generation	1.358148 ms	5704158
CPA Encryption	1.180363 ms	4957465
CPA Decryption	0.385487 ms	1619027
CCA Encryption	1.156492 ms	4857209
CCA Decryption	1.516876 ms	6370810
CPA Encapsulation	1.157531 ms	4861575
CPA Decapsulation	0.392261 ms	1647477
CCA Encapsulation	1.164657 ms	4891505
CCA Decapsulation	1.529845 ms	6425276

Table 6.10: Timings for **LIMA-sp-2062**

6.4 Advantages and Disadvantages

We end with a summary of the known advantages and disadvantages of our LIMA proposal.

6.4.1 Advantages

- We provide the choice of LIMA-2p and LIMA-sp parameter sets, depending on confidence in subfield tower protections.
- We provide the Enc-CCA scheme for public-key encryption of short messages.
- The Encap-CCA scheme, when combined with a DEM such as AES-GCM, gives public-key encryption for long messages.
- We use the well-studied underlying primitive of Ring-LWE. a relatively boring, but safe, pedigree/design choice.
- We employ Gaussian-like distributions to provide design validation against worst-case/average-case results.
- All the schemes have tight security proofs.

6.4.2 Disadvantages

- Our Reference and Optimized implementations are not currently constant time. But operations dependent on secret data (keys and messages) could be made constant time with little cost in performance.

Bibliography

[ABD16] Martin R. Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 153–178. Springer, Heidelberg, August 2016.

[ACF+15] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Algebraic algorithms for LWE problems. *ACM Comm. Computer Algebra*, 49(2):62, 2015.

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 595–618. Springer, Heidelberg, August 2009.

[AD17] Martin R. Albrecht and Amit Deo. Large modulus ring-LWE >= module-LWE. Cryptology ePrint Archive, Report 2017/612, 2017. http://eprint.iacr.org/2017/612.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 327–343. USENIX Association, 2016.

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part I*, volume 6755 of *LNCS*, pages 403–415. Springer, Heidelberg, July 2011.

[Alb17] Martin R. Albrecht. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 103–129. Springer, Heidelberg, May 2017.

[AOP+17a] Martin R. Albrecht, Emmanuela Orsini, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. Tightly secure ring-lwe based key encapsulation with short ciphertexts. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 29–46. Springer, 2017.

[AOP+17b] Martin R. Albrecht, Emmanuela Orsini, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. Tightly secure ring-lwe based key encapsulation with short ciphertexts. *IACR Cryptology ePrint Archive*, 2017:354, 2017.

[APS15] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[BBdV+17] Jens Bauch, Daniel J. Bernstein, Henry de Valence, Tanja Lange, and Christine van Vredendaal. Short generators without quantum computers: The case of multiquadratics. In Coron and Nielsen [CN17], pages 27–59.

[BCLvV16] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime. Cryptology ePrint Archive, Report 2016/461, 2016. http://eprint.iacr.org/2016/461.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, *27th SODA*, pages 10–24. ACM-SIAM, January 2016.

[BDK+17] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634, 2017. http://eprint.iacr.org/2017/634.

[BG14] Shi Bai and Steven D. Galbraith. Lattice decoding attacks on binary LWE. In Willy Susilo and Yi Mu, editors, *ACISP 14*, volume 8544 of *LNCS*, pages 322–337. Springer, Heidelberg, July 2014.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *TOCT*, 6(3):13:1–13:36, 2014.

[CDPR16] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 559–585. Springer, Heidelberg, May 2016.

[CDW17] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-SVP. In Coron and Nielsen [CN17], pages 324–348.

[CGS14] Peter Campbell, Michael Groves, and Dan Shepherd. Soliloquy: A cautionary tale. In *ETSI 2nd Quantum-Safe Crypto Workshop*, pages 1–9, 2014.

[Che13] Yuanmi Chen. *Réduction de réseau et sécurité concrète du chiffrement complétement homomorphe*. PhD thesis, Paris 7, 2013.

[CJL16a] Jung Hee Cheon, Jinhyuck Jeong, and Changmin Lee. An algorithm for ntru problems and cryptanalysis of the ggh multilinear map without a low-level encoding of zero. *LMS Journal of Computation and Mathematics*, 19(A):255–266, 2016.

[CJL16b] Jung Hee Cheon, Jinhyuck Jeong, and Changmin Lee. An algorithm for NTRU problems and cryptanalysis of the GGH multilinear map without a low level encoding of zero. Cryptology ePrint Archive, Report 2016/139, 2016. http://eprint.iacr.org/2016/139.

[CMV+15] Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray C. C. Cheung, Derek Pao, and Ingrid Verbauwhede. High-speed polynomial multiplication architecture for Ring-LWE and SHE cryptosystems. *IEEE Trans. on Circuits and Systems*, 62-I(1):157–166, 2015.

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 1–20. Springer, Heidelberg, December 2011.

[CN17] Jean-Sébastien Coron and Jesper Buus Nielsen, editors. *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*. Springer, Heidelberg, May 2017.

[Den03] Alexander W. Dent. A designer's guide to KEMs. In Kenneth G. Paterson, editor, *9th IMA International Conference on Cryptography and Coding*, volume 2898 of *LNCS*, pages 133–151. Springer, Heidelberg, December 2003.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. How to enhance the security of public-key encryption at minimum cost. In Hideki Imai and Yuliang Zheng, editors, *PKC'99*, volume 1560 of *LNCS*, pages 53–68. Springer, Heidelberg, March 1999.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/2012/144.

[GH15] Tim Güneysu and Helena Handschuh, editors. *CHES 2015*, volume 9293 of *LNCS*. Springer, Heidelberg, September 2015.

[GJS15] Qian Guo, Thomas Johansson, and Paul Stankovski. Coded-BKW: Solving LWE using lattice codes. In Gennaro and Robshaw [GR15], pages 23–42.

[GN08] Nicolas Gama and Phong Q. Nguyen. Finding short lattice vectors within Mordell's inequality. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 207–216. ACM Press, May 2008.

[GR15] Rosario Gennaro and Matthew J. B. Robshaw, editors. *CRYPTO 2015, Part I*, volume 9215 of *LNCS*. Springer, Heidelberg, August 2015.

[GS02] Craig Gentry and Michael Szydlo. Cryptanalysis of the revised NTRU signature scheme. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 299–320. Springer, Heidelberg, April / May 2002.

[GvVW17] Florian Göpfert, Christine van Vredendaal, and Thomas Wunderer. A hybrid lattice basis reduction and quantum search attack on LWE. Cryptology ePrint Archive, Report 2017/221, 2017. http://eprint.iacr.org/2017/221.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *TCC 2017, Part I*, LNCS, pages 341–371. Springer, Heidelberg, March 2017.

[HPS11] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Analyzing blockwise lattice algorithms using dynamical systems. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 447–464. Springer, Heidelberg, August 2011.

[KF15] Paul Kirchner and Pierre-Alain Fouque. An improved BKW algorithm for LWE with applications to cryptography and lattices. In Gennaro and Robshaw [GR15], pages 43–62.

[KF17] Paul Kirchner and Pierre-Alain Fouque. Revisiting lattice attacks on overstretched NTRU parameters. In Coron and Nielsen [CN17], pages 3–26.

[Laa15] Thijs Laarhoven. *Search problems in cryptography: From fingerprinting to lattice sieving*. PhD thesis, Eindhoven University of Technology, 2015.

[LLL82] Arjen K. Lenstra, Hendrik W. Lenstra, Jr., and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, December 1982.

[LN13] Mingjie Liu and Phong Q. Nguyen. Solving BDD by enumeration: An update. In Ed Dawson, editor, *CT-RSA 2013*, volume 7779 of *LNCS*, pages 293–309. Springer, Heidelberg, February / March 2013.

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

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

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 35–54. Springer, Heidelberg, May 2013.

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

[LSR+15] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. Efficient ring-LWE encryption on 8-bit AVR processors. In Güneysu and Handschuh [GH15], pages 663–682.

[MR09] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 147–191, Berlin, Heidelberg, New York, 2009. Springer, Heidelberg.

[MW16] Daniele Micciancio and Michael Walter. Practical, predictable lattice basis reduction. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 820–849. Springer, Heidelberg, May 2016.

[NIS16] NIST National Institute for Standards and Technology. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash, 2016. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf.

[Pei14] Chris Peikert. Lattice cryptography for the internet. Cryptology ePrint Archive, Report 2014/070, 2014. http://eprint.iacr.org/2014/070.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-LWE implementation. In Güneysu and Handschuh [GH15], pages 683–702.

[RVM+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-LWE cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 371–391. Springer, Heidelberg, September 2014.

[Sch87] Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53:201–224, 1987.

Appendix A

Parameter Search Code

We used the script in this section in conjunction with the LWE estimator from [APS15] to estimate the cost of solving LWE instances with our parameter choices.

In contrast to [Alb17], in our estimates we conservatively assume that producing many short bases from a short basis does not increase the norm of the output vectors. Thus, we apply the following patch to commit `6f2534014b7c4a4798ebe2df829c45bf01e47cbc` of `https://bitbucket.org/malb/lwe-estimator`:

```
--- estimator.orig.py   2017-06-23 14:55:33.229724436 +0100
+++ estimator.py        2017-06-23 14:55:27.281917030 +0100
@@ -2147,7 +2147,7 @@
     delta_0 = best["delta_0"]

     if use_lll:
-        scale = 2
+        scale = 1  # TODO conservative compared to upstream
     else:
         scale = 1
```

Our parameter search is then implemented as:

```
# -*- coding: utf-8 -*-

import estimator as est
from sage.all import RR, log, sqrt, pi, get_verbose, ZZ
from functools import partial
from collections import OrderedDict


sigma = sqrt((20)/2)


def ADPS16(beta, d, B=None, mode="classical"):
    u"""
     Runtime estimation from [USENIX:ADPS16]_ but with additive constant 16.

     :param beta: block size
     :param n: LWE dimension `n > 0`
     :param B: bit-size of entries
     """

    if mode not in ("classical", "quantum", "paranoid"):
        raise ValueError("Mode '%s' not understood"%mode)

    c = {"classical": 0.2920,
         "quantum":   0.2650,  # paper writes 0.262 but this isn't right, see above
         "paranoid":  0.2075}

    c = c[mode]

    return ZZ(2)**RR(c*beta + 16)


cost_models = OrderedDict([(partial(ADPS16, mode="classical"), "classical"),
                           (partial(ADPS16, mode="quantum"),   "quantum")])
```

```
def _compose(strategy, n, alpha, q, reduction_cost_model, *args, **kwds):
    """
    Run ‘strategy‘ using ‘reduction_cost_model‘.

    :param strategy: dual, primal, etc.
    :param n: LWE dimension
    :param alpha: noise rate
    :param q: modulus
    :param reduction_cost_model: cost model for BKZ

    """
    if get_verbose() >= 1:
        print "%20s"%cost_models[reduction_cost_model],

    r = strategy(n=n, alpha=alpha, q=q, *args,
                 reduction_cost_model=reduction_cost_model,
                 success_probability=0.51, **kwds)

    if get_verbose() >= 1:
        print r
    return r


dual   = partial(_compose,
                 strategy=est.dual_scale,
                 secret_distribution=True,
                 use_lll=True)

primal = partial(_compose, strategy=est.primal_usvp,
                 secret_distribution=True)


def complete_parameters(N, q, type=1):
    """

    Extend dimension ‘‘N‘‘ to complete parameter set.

    :param N: dimension
    :param type: 1 or 2

    """
    alpha = RR(sqrt(2*pi)*sigma/q)
    return N, alpha, q


def security_level(n, alpha, q, reduction_cost_model):
    cost_1 = primal(n=n, alpha=alpha, q=q, reduction_cost_model=reduction_cost_model)
    cost_2 =   dual(n=n, alpha=alpha, q=q, reduction_cost_model=reduction_cost_model)

    best = sorted([cost_1, cost_2], key=lambda x: x["red"])[0]

    return best, (cost_1, cost_2)


def print_candidate_security(n, q, type=1):
    n, alpha, q = complete_parameters(n, q, type=type)
    _, costs_c = security_level(n, alpha, q, reduction_cost_model=cost_models.keys()[0])
    _, costs_q = security_level(n, alpha, q, reduction_cost_model=cost_models.keys()[1])

    fmt = "%6s & %10s & %5.1f & %8.6f & %4d & %3d & $2^{%5.1f}$\\\\"

    print "\\multicolumn{7}{c}{Type %d, $n=%d$, $q=%d$}\\\\"%(type, n, q)
    print "\\midrule"

    for i, c_ in enumerate(costs_c):
        if i == 1:
            attack = "dual"
        elif i == 0:
            attack = "primal"
        else:
            continue

        r_ = log(c_.data.get("repeat", 1), 2)
        print(fmt%(attack, "classical", log(c_.values()[0], 2), c_["delta_0"], c_["d"], c_["beta"], r_))
```

```
    for i, c_ in enumerate(costs_q):
        if i == 1:
            attack = "dual"
        elif i == 0:
            attack = "primal"
        else:
            continue

        r_ = log(c_.data.get("repeat", 1), 2)
        print(fmt%(attack, "quantum", log(c_.values()[0], 2), c_["delta_0"], c_["d"], c_["beta"], r_))

    print "\\midrule"


def print_all_candidate_securities():
    """
    """
    print_candidate_security(512,  18433, 1)
    print_candidate_security(1024, 40961, 1)
    print_candidate_security(2048, 40961, 1)

    print_candidate_security(1018, 12521473, 2)
    print_candidate_security(1306, 48181249, 2)
    print_candidate_security(1822, 44802049, 2)
    print_candidate_security(2062, 16900097, 2)
```

In particular, we used the following calls to produce Tables 5.1 and 5.2.

```
sage: attach("security_estimates.py")
sage: _ = all_candidates(up_to=320, type=1)
sage: _ = all_candidates(up_to=320, type=2)
sage: print_all_candidate_securities()
```