CFRG
Internet-Draft
Intended status: Informational
Expires: 18 September 2025

P. Longa
Microsoft
J. W. Bos
NXP Semiconductors
S. Ehlen
Federal Office for Information Security (BSI)
D. Stebila
University of Waterloo

17 March 2025

FrodoKEM: key encapsulation from learning with errors
draft-longa-cfrg-frodokem-00

Abstract

This internet draft specifies FrodoKEM, an IND-CCA2 secure Key Encapsulation Mechanism (KEM).

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-longa-cfrg-frodokem/.

Source for this draft and an issue tracker can be found at github.com/dstebila/frodokem-internet-draft.

Status of This Memo

Table of Contents

1. Introduction

FrodoKEM [Frodo17] is a conservative yet practical post-quantum key encapsulation mechanism (KEM) whose security derives from cautious parameterizations of the well-studied learning with errors problem, which in turn has close connections to conjectured-hard problems on generic, "algebraically unstructured" lattices.

As a key encapsulation mechanism, FrodoKEM is a three-tuple of algorithms (_KeyGen_, _Encapsulate_, _Decapsulate_):

* _KeyGen_ takes no inputs, requires randomness, and outputs a private key and a public key;

* _Encapsulate_ takes as input a public key, requires randomness, and outputs a ciphertext and a shared secret;

* _Decapsulate_ takes as input a ciphertext and a private key, and outputs a shared secret.

These algorithms are assembled as a two-pass protocol that allows two parties, A and B, to derive a shared secret in an interactive fashion. Assume that party A is responsible for the _KeyGen_ and _Decapsulate_ operations, and that party B is responsible for _Encapsulate_. In the first pass, after receiving or retrieving party A’s public key, party B produces a ciphertext with the _Encapsulate_ operation. In the second pass, party A uses its secret key and the ciphertext to execute the _Decapsulate_ operation. The shared secret produced by this protocol can then be used to establish a secure communication channel using a symmetric-key algorithm.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Overview

The core of FrodoKEM is a public-key encryption scheme called FrodoPKE, whose IND-CPA security is tightly related to the hardness of a corresponding learning with errors problem. Here we briefly recall the scientific lineage of these systems. See the surveys [Mic10], [Reg10] and [Pei16] for further details. The seminal works of Ajtai [Ajt96] (published in 1996) and Ajtai窶冱 work [AD97] (published in 1997) gave the first cryptographic constructions whose security properties followed from the conjectured worst-case hardness of various problems on point lattices in R^n. In subsequent years, these works were substantially refined and improved. Notably, in work published in 2005, Regev [Reg09] defined the learning with errors (LWE) problem, proved the hardness of (certain parameterizations of) LWE assuming the hardness of various worst-case lattice problems against quantum algorithms, and defined a public-key encryption scheme whose IND-CPA security is tightly related to the hardness of LWE.

Later on, in work published in 2011, Lindner and Peikert [LP11] gave a more efficient LWE-based public-key encryption scheme that uses a square public matrix A of dimension n with integer coefficients modulo q, instead of an oblong rectangular one.

The FrodoPKE scheme described in this document is an instantiation and implementation of the Lindner窶撤eikert scheme [LP11] with some modifications, such as: pseudorandom generation of the public matrix A from a small seed, more balanced key and ciphertext sizes, and new LWE parameters.

3.1. Chosen-Ciphertext Security

FrodoKEM achieves IND-CCA security by way of a transformation of the IND-CPA-secure FrodoPKE. In work published in 1999, Fujisaki and Okamoto [FO99] gave a generic transform from an IND-CPA PKE to an IND-CCA PKE, in the random-oracle model. At a high level, the Fujisaki窶徹kamoto (FO) transform derives encryption coins pseudorandomly, and decryption regenerates these coins to re-encrypt and check that the ciphertext is well-formed. In 2016, Targhi and Unruh [TU16] gave a modification of the Fujisaki窶徹kamoto transform that achieves IND-CCA security in the quantum random-oracle model (QROM) by adding an extra hash. In 2017, Hofheinz, Hovelmanns, and Kiltz [HHK17] gave several variants of the Fujisaki窶徹kamoto and Targhi窶填nruh transforms that in particular convert an IND-CPA-secure PKE into an IND-CCA-secure KEM, and analyzed them in both the classical and quantum random-oracle models, even for PKEs with non-zero decryption error. Jiang et al. [JZC_18] show how to prove security of one of these variant FO transforms in the QROM without requiring the extra hash from Targhi窶填nruh. FrodoKEM is constructed from FrodoPKE using a slight variant of Jiang et al.’s transform that includes additional values in hash computations to reduce the risk of multi-target attacks.

4. Notation

We describe the symbols and abbreviations used throughout this document.

* Z represents the set of integers and Z_q represents the set of integers modulo q.

* 蛿x is the rounding of x to the nearest integer. If x = y + 1/2 for some y in Z, then x = y + 1.

* r^(i) is a 16-bit bit string.

* (r^(0), r^(1), ..., r^(t-1)) is a sequence of t 16-bit bit strings r^(i).

* AES128(k, a) denotes the 128-bit AES128 output under key k for a 128-bit input a.

* SHAKE128(x, y) and SHAKE256(x, y) denote the y first bits of SHAKE128 and SHAKE256 (resp.) output for input x.

* Matrices are represented in capitals with no italics (e.g., A and C). For an n1 * n2 matrix C, its (i,j)th coefficient (i.e., the entry in the ith row and jth column) is denoted by C[i,j], where 0 <= i < n1 and 0 <= j < n2. The transpose of matrix C is denoted by C^T.

AES128 and SHAKE are specified in [FIPS197] and [FIPS202], respectively.

5. Parameters

The FrodoKEM parameters are implicit inputs to the FrodoKEM algorithms defined in the next sections. A FrodoKEM parameter set specifies the following:

* A positive integer D <= 16 that defines the modulus parameter q = 2^D.

* Positive integers n, nHat specifying matrix dimensions. It holds that n, nHat ≡ 0 mod 8.

* A positive integer B <= D specifying the number of bits encoded in each matrix entry.

* A positive integer lenA specifying the bitlength of seeds for the generation of the matrix A.

* A positive integer lensec specifying the number of bits that match the bit-security level. Valid values are 128, 192 and 256. This is used to determine the bitlength of seeds (not associated to the matrix A), of hash value outputs and of values associated to the generation of the shared secrets.

* A positive integer lenSE specifying the bitlength of the seed value seedSE.

* A positive integer lensalt specifying the bitlength of the value salt.

* A discrete, symmetric error distribution X on Z with support given by S_X = {−d, −d+1, ..., −1, 0, 1, ..., d−1, d} for a small integer d.

* A table T_X = (T_X(0), T_X(1), ..., T_X(d)) with (d+1) positive integers based on the cumulative distribution function for X.

The values for these parameters corresponding to each parameter set are given in Section 9.1.

6. Supporting Functions

6.1. Octet Encoding of Bit Strings

This document follows the little-endian formatting for octet encoding of bit strings.

A bit string b = (b[0], b[1], ..., b[|b|-1]) is converted to an octet string by taking bits from left to right, packing those from the least significant bit of each octet to the most significant bit, and moving to the next octet when each octet fills up. For example, the 16-bit bit string (b[0], b[1], ..., b[15]) is converted into two octets f and g (in this order) as

```
f = b[7] * 2^7 + b[6] * 2^6 + b[5] * 2^5 + b[4] * 2^4 + b[3] * 2^3 + ...
    b[2] * 2^2 + b[1] * 2 + b[0]
g = b[15] * 2^7 + b[14] * 2^6 + b[13] * 2^5 + b[12] * 2^4 + ...
    b[11] * 2^3 + b[10] * 2^2 + b[9] * 2 + b[8]
```

The conversion from octet string to bit string is the reverse of this process.

For FrodoKEM, it is always the case that |b| is a multiple of 8 when performing octet encoding of bit strings.

6.2. Matrix Encoding of Bit Strings

We define how bit strings are encoded as mod-q integer matrices.

From the FrodoKEM parameters one has that 2^B <= q. The encoding function ec() encodes an integer 0 <= val < 2^B as an element in Z_q by multiplying it by q/2^B = 2^(D-B):

ec(val) = val * q / 2^B.

Using this function, the function Encode(b) encodes a given bit string b = (b[0], ..., b[l-1]) of length l = B * nHat^2 as an nHat * nHat matrix C with coefficients C[i,j] in Z_q by applying ec() to B-bit sub-strings sequentially and filling the matrix row by row entry-wise. The function Encode(b) is defined as follows.

```
for i = 0 to nHat - 1 do
   for j = 0 to nHat - 1 do
      val = 0
      for k = 0 to B - 1 do
         val = val + b[(i * nHat + j)B + k] * 2^k
      end for
      C[i,j] = val * q / 2^B
   end for
end for

return C
```

The corresponding decoding function Decode(C) decodes an nHat * nHat matrix C into a bit string of length l = B * nHat^2. It extracts B bits from each entry by applying the function dc():

dc(c) = c * 2^B / q mod 2^B.

That is, the Z_q-entry is interpreted as an integer, then divided by q/2^B and rounded. This amounts to rounding to the B most significant bits of each entry. With these definitions, it is the case that dc(ec(val)) = val for all 0 <= val < 2^B. The function Decode(C) is defined as follows.

```
for i = 0 to nHat - 1 do
    for j = 0 to nHat - 1 do
        c =  C[i,j] * 2^B / q  mod 2^B
        Set c = c[0] * 2^0 + c[1] * 2^1 + ... + c[B-1] * 2^{B-1}
        for k = 0 to B - 1 do
            b[(i * nHat + j)B + k] = c[k]
        end for
    end for
end for

return (b[0], ..., b[l-1])
```

6.3. Packing Matrices Modulo q

We define packing and unpacking functions to transform matrices with entries in Z_q to bit strings and vice versa.

The function Pack packs an n1 * n2 matrix C with entries C[i,j] in Z_q to an octet string by concatenating the D-bit matrix coefficients. The function Pack(C) is defined as follows.

```
for i = 0 to n1 - 1 do
    for j = 0 to n2 - 1 do
        Set C[i,j] = c[0] * 2^0 + c[1] * 2^1 + ... + c[D-1] * 2^{D-1}
        for k = 0 to D - 1 do
            b[(i * n2 + j)D + k] = c[D-1-k]
        end for
    end for
end for

return the octet string corresponding to the bit string
b = (b[0], b[1], ..., b[D * n1 * n2 - 1]), as per Section 6.1.
```

The function Unpack does the reverse of this process to transform an octet string o to an n1 * n2 matrix C with entries C[i,j] in Z_q, converting the input to a bit string, and then extracting D-bit strings and storing each as matrix coefficients C[i,j] for 0 <= i < n1 and 0 <= j < n2 (row-by-row from C[0,0] to C[n1-1, n2-1]). The function Unpack(o, n1, n2) is defined as follows:

Convert the input octet string o to a bit string
b = (b[0], b[1], ..., b[D * n1 * n2 - 1]), as per Section 6.1.

```
for i = 0 to n1 - 1 do
    for j = 0 to n2 - 1 do
        C[i,j] = 0
        for k = 0 to D - 1 do
            C[i,j] = C[i,j] + b[(i * n2 + j)D + k] * 2^(D-1-k)
        end for
    end for
end for

return C
```

6.4. Sampling from the Error Distribution

The error distribution X used in FrodoKEM is a discrete, symmetric distribution on Z, centered at zero and with small support, which approximates a rounded continuous Gaussian distribution.

The support of X is S_X = {−d, −d+1, ..., −1, 0, 1, ..., d−1, d} for a positive integer d specified by the FrodoKEM parameter set. The probabilities X(z) = X(−z) for z in S_X are given by a discrete probability density function, which is described by a table

T_X = (T_X(0), T_X(1), ..., T_X(d))

of d+1 positive integers related to the cumulative distribution function.

Given a random 16-bit string r = (r[0], r[1], ..., r[15]), the function Sample(r) returns a sample e from FrodoKEM’s error distribution X via inversion sampling using a table T_X, as follows (note that T_X(d) is never accessed):

```
t = r[1] * 2^0 + r[2] * 2^1 + ... + r[15] * 2^14
e = 0

for i = 0 to d - 1 do
    if t > T_X(i) then
        e = e + 1
    end if
end for

e = (-1)^(r[0]) * e

return e
```

The output of the algorithm is a small integer in the range {-d, -d+1, ..., -1, 0, 1, ..., d-1, d}. The tables T_X corresponding to each of FrodoKEM’ s parameter sets are given in Table 5.

We emphasize that it is important to perform this sampling in constant time to avoid exposing timing side-channels, which is why the for-loop of the algorithm does a complete loop through the entire table T_X. Similarly, the comparison in the if-loop needs to be implemented in a constant-time manner.

6.5. Matrix Sampling from the Error Distribution

We define the function SampleMatrix which samples an n1 * n2 matrix using the function Sample.

Given (n1 * n2) 16-bit random strings r^(i) and the dimension values n1 and n2, SampleMatrix((r^(0), ..., r^(n1 * n2 - 1)), n1, n2) generates an n1 * n2 matrix E row-by-row from E[0,0] to E[n1-1,n2-1] by successively calling the function Sample n1 * n2 times, as follows:

```
for i = 0 to n1 - 1 do
    for j = 0 to n2 - 1 do
        E[i,j] = Sample(r^(i * n2 + j))
    end for
end for

return E
```

6.6. Pseudorandom Matrix Generation

The function Gen takes as input a seed, seedA, of length lenA=128 bits and an implicit dimension n that is fixed per parameter set, and outputs an n * n pseudorandom matrix A, where all the coefficients are in Z_q. There are two options for instantiating Gen: one based on AES128 and another based on SHAKE128. In both cases, the matrix A is generated row-by-row from A[0,0] to A[n-1,n-1].

6.6.1. Matrix A Generation with AES128

The algorithm for the case using AES128 is shown below. Each call to AES128 generates 8 coefficients.

```
for i = 0 to n - 1 do
    for j = 0 to n - 1 step 8 do
        b = i || j || 0 || 0 || 0 || 0 || 0 || 0
        # Each concatenated element is encoded as a 16-bit string
        # represented in little-endian byte order, such that:
        # (i[0], i[1], ..., i[15]) ≡ i[0] * 2^0 + ... + i[15] * 2^15
        # and |b| = 128

        C[i,j] || C[i,j+1] || ... || C[i,j+7] = AES128(seedA, b)
        # Each matrix coefficient C[i,j] is a 16-bit string
        # interpreted as a non-negative integer in little-endian
        # byte order:
        # C[i,j] = c[0] * 2^0 + c[1] * 2^1 + ... + c[15] * 2^15
        # corresponding to the bit string (c[0], c[1], ..., c[15])

        for k = 0 to 7 do
            A[i,j+k] = C[i,j+k] mod q
        end for
    end for
end for

return A
```

6.6.2. Matrix A Generation with SHAKE128

The algorithm for the case using SHAKE128 is shown below. Each call to SHAKE128 generates n coefficients (i.e., a full matrix row).

```
for i = 0 to n - 1 do
    b = i || seedA
    # Element i is encoded as a 16-bit string
    # represented in little-endian byte order, such that:
    # (i[0], i[1], ..., i[15]) ≡ i[0] * 2^0 + ... + i[15] * 2^15
    # and |b| = lenA + 16

    C[i,0] || C[i,1] || ... || C[i,n-1] = SHAKE128(b, 16 * n)
    # Each matrix coefficient C[i,j] is a 16-bit string interpreted
    # as a non-negative integer in little-endian byte order:
    # C[i,j] = c[0] * 2^0 + c[1] * 2^1 + ... + c[15] * 2^15
    # corresponding to the bit string (c[0], c[1], ..., c[15])

    for j = 0 to n - 1 do
        A[i,j] = C[i,j] mod q
    end for
end for

return A
```

7. FrodoKEM

7.1. Key Generation

The key generation algorithm accepts no input, requires randomness, and outputs the keypair (pk, sk) = (seedA || b, s || seedA || b || S^T || pkh).

```
Choose uniformly random seed s of bitlength lensec
Choose uniformly random seed seedSE of bitlength lenSE
Choose uniformly random seed z of bitlength lenA
# Generate pseudorandom seed:
seedA = SHAKE(z, lenA)
# Generate the matrix A:
A = Gen(seedA)
# Generate pseudorandom bit string:
r = SHAKE(0x5F || seedSE, 32 * n * nHat)
# Sample matrix S transposed:
S^T = SampleMatrix((r^(0), r^(1), ..., r^(n * nHat - 1)), nHat, n)
# Sample error matrix E:
E = SampleMatrix((r^(n * nHat), r^(n * nHat + 1), ...,
                  r^(2 * n * nHat - 1)), n, nHat)
B = A * S + E
b = Pack(B)
pkh = SHAKE(seedA || b, lensec)
pk = (seedA || b)
sk = (s || seedA || b || S^T || pkh)

return pk, sk  # Return public key and secret key
```

Here, the matrix ST = S^T is encoded row-by-row from ST[0,0] to ST[nHat-1,n-1], where each matrix coefficient ST[i,j] is a signed integer encoded as a 16-bit string (s[0], s[1], ..., s[15]) in the little-endian byte order, i.e.

ST[i,j] = -s[15] * 2^15 + (s[0] + s[1] * 2 + s[2] * 2^2 + ... + s[14] * 2^14).

7.2. Encapsulation

The encapsulation algorithm takes as input a public key pk = (seedA || b), requires randomness, and outputs a ciphertext c = (c1 || c2 || salt) and a shared secret ss.

```
Choose uniformly random value u of bitlength lensec
Choose uniformly random value salt of bitlength lensalt
pkh = SHAKE(pk, lensec)
# Generate pseudorandom values:
seedSE || k = SHAKE(pkh || u || salt, lenSE + lensec)
# Generate pseudorandom bit string:
r = SHAKE(0x96 || seedSE, 16 * (2 * nHat * n + nHat^2))
# Sample matrices S' and E':
S' = SampleMatrix((r^(0), r^(1), ..., r^(nHat * n - 1)), nHat, n)
E' = SampleMatrix((r^(nHat * n), r^(nHat * n + 1), ...,
                   r^(2 * nHat * n - 1)), nHat, n)
# Generate the matrix A:
A = Gen(seedA)
B' = S' * A + E'
c1 = Pack(B')
# Sample error matrix E":
E" = SampleMatrix((r^(2 * nHat * n), r^(2 * nHat * n + 1), ...,
                   r^(2 * nHat * n + nHat^2 - 1)), nHat, nHat)
B = Unpack(b, n, nHat)
V = S' * B + E"
C = V + Encode(u)
c2 = Pack(C)
ss = SHAKE(c1 || c2 || salt || k, lensec)

return (c1 || c2 || salt), ss  # Return ciphertext and shared secret
```

7.3. Decapsulation

The decapsulation algorithm takes as input a ciphertext c = (c1 || c2 || salt) and a secret key sk = (s || seedA || b || S^T || pkh), and outputs a shared secret ss.

```
B′ = Unpack(c1, nHat, n)
C = Unpack(c2, nHat, nHat)
M = C - B′ * S
u′ = Decode(M)
# Generate pseudorandom values:
seedSE′ || k′ = SHAKE(pkh || u′ || salt, lenSE + lensec)
# Generate pseudorandom bit string:
r = SHAKE(0x96 || seedSE′, 16 * (2 * nHat * n + nHat^2))
# Sample matrices S′ and E′:
S′ = SampleMatrix((r^(0), r^(1), ..., r^(nHat * n - 1)), nHat, n)
E′ = SampleMatrix((r^(nHat * n), r^(nHat * n + 1), ...,
                   r^(2 * nHat * n - 1)), nHat, n)
# Generate the matrix A:
A = Gen(seedA)
B" = S′ * A + E′
# Sample error matrix E":
E" = SampleMatrix((r^(2 * nHat * n), r^(2 * nHat * n + 1), ...,
                   r^(2 * nHat * n + nHat^2 - 1)), nHat, nHat)
B = Unpack(b, n, nHat)
V = S′ * B + E"
C′ = V + Encode(u′)
kHat = k′ if B′ == B" and C == C′ else kHat = s
ss = SHAKE(c1 || c2 || salt || kHat, lensec)

return ss  # Return shared secret ss
```

8. FrodoKEM Variants

FrodoKEM is parameterized by the pseudorandom generator (PRG) that is used for the generation of the matrix A. As explained in Section 6.6 there are two options for PRG: AES128 and SHAKE128.

In addition, FrodoKEM consists of two main variants: a "standard" variant that does not impose any restriction on the reuse of key pairs, and an "ephemeral" variant that is intended for applications in which the number of ciphertexts produced relative to any single public key is small. Concretely, the use of standard FrodoKEM is recommended for applications in which the number of ciphertexts produced for a single public key is expected to be equal or greater than 2^8. Ephemeral FrodoKEM MUST be used for applications in which that same figure is expected to be smaller than 2^8.

In contrast to ephemeral FrodoKEM, standard FrodoKEM incorporates some changes to address certain multi-ciphertext attacks [Annex]. Specifically, standard FrodoKEM doubles the length of the seedSE value and incorporates a public random salt value into encapsulation (see Table 2).

9. Parameter Sets

This document specifies the following twelve parameter sets:

1. FrodoKEM-640-PRG and eFrodoKEM-640-PRG, which match or exceed the brute-force security of AES128.

2. FrodoKEM-976-PRG and eFrodoKEM-976-PRG, which match or exceed the brute-force security of AES192.

3. FrodoKEM-1344-PRG and eFrodoKEM-1344-PRG, which match or exceed the brute-force security of AES256.

The label "eFrodoKEM" corresponds to the ephemeral variants. The options for PRG are AES or SHAKE, when either AES128 or SHAKE128 (respectively) is used for the generation of the matrix A. Thus, the first FrodoKEM variant consists of the parameter sets FrodoKEM-640-AES, FrodoKEM-976-AES and FrodoKEM-1344-AES (and their corresponding ephemeral variants). The second FrodoKEM variant consists of the parameter sets FrodoKEM-640-SHAKE, FrodoKEM-976-SHAKE and FrodoKEM-1344-SHAKE (and their corresponding ephemeral variants).

9.1. Summary of Parameters

The parameter values characterizing the FrodoKEM parameter sets are listed below.

Name	(e)FrodoKEM-640	(e)FrodoKEM-976	(e)FrodoKEM-1344	Description
D	15	16	16	Bitlength of q
q	32768	65536	65536	Power-of-two integer modulus
n	640	976	1344	Integer matrix dimension
nHat	8	8	8	Integer matrix dimension
B	2	3	4	Number of bits encoded per matrix entry
d	12	10	6	Integer defining the support of X
lenA	128	128	128	Bitlength of seeds for generation of matrix A
lensec	128	192	256	Number of bits matching the bit-security level
SHAKE	SHAKE128	SHAKE256	SHAKE256	SHAKE variant used for hashing

Table 1: Parameters for FrodoKEM.

Name	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344	Description
lenSE	256	384	512	Bitlength of seedSE in FrodoKEM
lensalt	256	384	512	Bitlength of salt in FrodoKEM

Table 2: Additional parameters for FrodoKEM variant.

Name	eFrodoKEM-640	eFrodoKEM-976	eFrodoKEM-1344	Description
lenSE	128	192	256	Bitlength of seedSE in eFrodoKEM
lensalt	0	0	0	No salt in eFrodoKEM

Table 3: Additional parameters for eFrodoKEM variant.

Name	X_Frodo-640	X_Frodo-976	X_Frodo-1344
sigma	2.8	2.3	1.4
0	9288	11278	18286
+-1	8720	10277	14320
+-2	7216	7774	6876
+-3	5264	4882	2023
+-4	3384	2545	364
+-5	1918	1101	40
+-6	958	396	2
+-7	422	118	
+-8	164	29	
+-9	56	6	
+-10	17	1	
+-11	4		
+-12	1		
order	200	500	1000
divergence	0.324 x 10^-4	0.140 x 10^-4	0.264 x 10^-4

Table 4: Error distributions. Probabilities are shown for each integer value from 0 up to +-12. The last two rows correspond to Renyi's order and divergence.

Table entries	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
T_X(0)	4,643	5,638	9,142
T_X(1)	13,363	15,915	23,462
T_X(2)	20,579	23,689	30,338
T_X(3)	25,843	28,571	32,361
T_X(4)	29,227	31,116	32,725
T_X(5)	31,145	32,217	32,765
T_X(6)	32,103	32,613	32,767
T_X(7)	32,525	32,731	
T_X(8)	32,689	32,760	
T_X(9)	32,745	32,766	
T_X(10)	32,762	32,767	
T_X(11)	32,766		
T_X(12)	32,767		

Table 5: The distribution table entries T_X(i), for 0 <= i <= d, for sampling.

Scheme	secret key sk	public key pk	ciphertext ct	shared secret ss
FrodoKEM-640	19,888	9,616	9,752	16
eFrodoKEM-640	19,888	9,616	9,720	16
FrodoKEM-976	31,296	15,632	15,792	24
eFrodoKEM-976	31,296	15,632	15,744	24
FrodoKEM-1344	43,088	21,520	21,696	32
eFrodoKEM-1344	43,088	21,520	21,632	32

Table 6: Sizes (in bits) of inputs and outputs.

10. Security Considerations

FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344 are designed to be post-quantum IND-CCA2 secure KEMs at the security levels of AES-128, AES-192 and AES-256, respectively.

Users are recommended to use the highest possible security level that a given application allows. In particular, the designers of FrodoKEM recommend to use either FrodoKEM-976 or FrodoKEM-1344 for most applications, and limit the use of FrodoKEM-640 to applications that require short-term security.

Lattice-based cryptographic schemes such as FrodoKEM are still relatively young. Therefore, it is recommended to use FrodoKEM in combination with a classical scheme (e.g., based on elliptic curves) while our confidence in the security of lattice schemes increases over time.

11. IANA Considerations

This document has no IANA actions.

12. References

12.1. Normative References

[FIPS197] (NIST), N. I. of S. and T., "Advanced Encryption Standard (AES), FIPS 197", November 2001, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf>.

[FIPS202] (NIST), N. I. of S. and T., "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, FIPS 202", August 2015, <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

12.2. Informative References

[AD97] Ajtai, M. and C. Dwork, "A public-key cryptosystem with worst-case/average-case equivalence", 29th Annual ACM Symposium on Theory of Computing, pages 284293 , 1997.

[Ajt96] Ajtai, M., "Generating hard instances of lattice problems (extended abstract)", 28th Annual ACM Symposium on Theory of Computing, pages 99108 , 1996.

[Annex] Alkim, E., Bos, J. W., Ducas, L., Longa, P., Mironov, I., Naehrig, M., Nikolaenko, V., Peikert, C., Raghunathan, A., and D. Stebila, "Annex on FrodoKEM updates", April 2023, <https://frodokem.org/files/FrodoKEM-annex-20230418.pdf>.

[FO99] Fujisaki, E. and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes", Advances in Cryptology CRYPTO’ 99, volume 1666 of Lecture Notes in Computer Science, pages 537554 , 1999.

[Frodo17] Alkim, E., Bos, J. W., Ducas, L., Longa, P., Mironov, I., Naehrig, M., Nikolaenko, V., Peikert, C., Raghunathan, A., and D. Stebila, "FrodoKEM: Learning With Errors Key Encapsulation", 2017, <https://frodokem.org>.

[HHK17] Hofheinz, D., Hovelmanns, K., and E. Kiltz, "A modular analysis of the Fujisaki-Okamoto transformation", 15th Theory of Cryptography Conference (TCC 2017), Part I, volume 10677 of Lecture Notes in Computer Science, pages 341371 , 2017.

[JZC_18] Jiang, H., Zhang, Z., Chen, L., Wang, H., and Z. Ma, "IND-CCA-secure key encapsulation mechanism in the quantum random oracle model, revisited", Advances in Cryptology CRYPTO 2018, Part III, volume 10993 of Lecture Notes in Computer Science, pages 96125 , 2018.

[LP11] Lindner, R. and C. Peikert, "Better key sizes (and attacks) for LWE-based encryption", Topics in Cryptology CT-RSA 2011, volume 6558 of Lecture Notes in Computer Science, pages 319339 , 2011.

[Mic10] Micciancio, D., "Cryptographic functions from worst-case complexity assumptions", Information Security and Cryptography, pages 427452 , 2010.

[Pei16] Peikert, C., "A decade of lattice cryptography", Foundations and Trends in Theoretical Computer Science, 10(4):283424. , 2016.

[Reg09] Regev, O., "On lattices, learning with errors, random linear codes, and cryptography", Journal of the ACM, 56(6):34, 2009. Preliminary version in STOC 2005 , 2009.

[Reg10] Regev, O., "The learning with errors problem (invited survey)", IEEE Conference on Computational Complexity, pages 191204 , 2010.

[TU16] Targhi, E. E. and D. Unruh, "Post-quantum security of the Fujisaki-Okamoto and OAEP transforms", 14th Theory of Cryptography Conference (TCC 2016-B), Part II, volume 9986 of Lecture Notes in Computer Science, pages 192216 , 2016.

Authors’ Addresses

Patrick Longa
Microsoft
Email: plonga@microsoft.com

Joppe W. Bos
NXP Semiconductors

Email: joppe.bos@nxp.com

Stephan Ehlen
Federal Office for Information Security (BSI)
Email: stephan.ehlen@bsi.bund.de

Douglas Stebila
University of Waterloo
Email: dstebila@uwaterloo.ca