Date: 2023-03-14

Revision: 2024-12-05

FrodoKEM: Learning With Errors Key Encapsulation

Preliminary Standardization Proposal

Contents

Introduction

FrodoKEM is a family of IND-CCA2 secure key-encapsulation mechanisms (KEMs) that were designed to be conservative yet practical post-quantum constructions. The security of FrodoKEM derives from cautious parameterizations of the well-studied learning with errors (LWE) problem, which in turn has close connections to conjectured-hard problems on generic, "algebraically unstructured" lattices.

The core building block 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.

As a key encapsulation mechanism, FrodoKEM is a two-pass protocol that allows two parties to derive a shared secret. This shared secret can then be used to establish a secure communication channel using a symmetric-key algorithm such as AES.

FrodoKEM is parameterized by the pseudorandom generator (PRG) that is used for the generation of a matrix called A. This document considers two main variants, which are determined by the use of either AES128 [FIPS197] or SHAKE128 [FIPS202] for the generation of A.

In addition, FrodoKEM consists of two main variants: a "standard" variant that does not impose any restriction on the reuse of key pairs (i.e., it is suitable for applications in which a large number of ciphertexts may be encrypted to a single public key), and an "ephemeral" variant that is suitable for applications in which the number of ciphertexts produced relative to any single public key is fairly small.

Concretely, this document specifies the following FrodoKEM schemes or parameter sets targeting three security levels:

- FrodoKEM-640-⟨PRG⟩ and eFrodoKEM-640-⟨PRG⟩, which match or exceed the brute-force security of AES128.
- FrodoKEM-976-⟨PRG⟩ and eFrodoKEM-976-⟨PRG⟩, which match or exceed the brute-force security of AES192.
- FrodoKEM-1344-⟨PRG⟩ and eFrodoKEM-1344-⟨PRG⟩, which match or exceed the brute-force security of AES256.

The schemes above correspond to Levels 1, 3 and 5, respectively, as defined by NIST [NIST-CFP]. 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 schemes FrodoKEM-640-AES, FrodoKEM-976-AES, and FrodoKEM-1344-AES (and their corresponding ephemeral variants). Analogously, the second FrodoKEM variant consists of the schemes FrodoKEM-640-SHAKE, FrodoKEM-976-SHAKE, and FrodoKEM-1344-SHAKE (and their corresponding ephemeral variants).

For brevity, in some cases this document drops the PRG label in the parameter set name to refer generically to the schemes targeting a certain security level (e.g., the use of FrodoKEM-640 refers to the Level 1 parameter sets FrodoKEM-640-AES and FrodoKEM-640-SHAKE). Unless explicitly stated, those names also include the ephemeral variants.

Annex A (informative) details the various parameters defining the FrodoKEM parameter sets. Annex B (informative) contains the security estimates for the different instantiations according to the analysis based on the core-SVP hardness. Similarly, Annex C (informative) describes the security estimates according to a more refined analysis of the cost of cryptanalytic attacks, and Annex D (informative) describes the security estimates derived from the security reductions.

1 Scope

This document specifies key encapsulation mechanisms from the FrodoKEM family. In particular, it specifies:

- The process for generating key pairs;
- The process for encapsulation using a public key;
- The process for decapsulation using a secret key and ciphertext.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

3.1
ciphertext
data which has been transformed to hide its information content

3.1
IND-CPA security
provides indistinguishability under chosen-plaintext attack, in which the adversary can obtain ciphertexts for arbitrary plaintexts

3.2
IND-CCA2 security
provides indistinguishability under adaptive chosen-ciphertext attack, in which adversaries first send adaptively-chosen ciphertexts to be decrypted, and then use the results to distinguish a target ciphertext without consulting the oracle on the challenge ciphertext

4 Symbols and abbreviations

For the purposes of this document, the following symbols and abbreviations apply.

$len(a)$	bitlength of bit string a.
$a = (a_0, a_1, \dots, a_{len(a)-1})$	bit representation of bit string a, where a_i is the i-th bit of a.
$a \,\|\|\, b$	bit string a concatenated with bit string b.
$r^{(i)}$	16-bit bit string.
$(r^{(0)}, r^{(1)}, \dots, r^{(t-1)})$	sequence of t bit strings $r^{(i)}$, each 16 bits long.
$\$\rightarrow$	random sample from a distribution.
$\mathbb{Z}$	the set of integers.
$\mathbb{Z}_q$	the set of integers modulo q.
$\text{AES128}_{key}(a)$	the 128-bit AES128 [FIPS197] output under key key for a 128-bit input a.
$\text{SHAKE128}(x, y)$	the y first bits of SHAKE128 [FIPS202] output for input x.
$\text{SHAKE}(x, y)$	the y first bits of SHAKE [FIPS202] output for input x. SHAKE is either SHAKE128 or SHAKE256 depending on the parameter set (see Section 6.2).

In this document, matrices are represented in capitals with no italics (e.g., A and C). For an $n_1 \times n_2$ 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 $0 \leq i < n_1$ and $0 \leq j < n_2$. The transpose of matrix C is denoted by C^T.

5 General model for key encapsulation mechanism

A key encapsulation mechanism KEM is a tuple of algorithms $(KeyGen, Encaps, Decaps)$ along with a finite keyspace K:

- $KeyGen(\,)\ \$ \rightarrow (pk, sk)$: A probabilistic key generation algorithm that outputs a public key pk and a secret key sk.
- $Encaps(pk)\ \$ \rightarrow (c, ss)$: A probabilistic encapsulation algorithm that takes as input a public key pk, and outputs an encapsulation c and a shared secret $ss \in K$. The encapsulation is also called a ciphertext.
- $Decaps(c, sk)\ \$ \rightarrow (ss')$: A (usually deterministic) decapsulation algorithm that takes as input an encapsulation c and a secret key sk, and outputs a shared secret $ss' \in K$.

6 FrodoKEM parameters

FrodoKEM is defined by the following parameters:

$q = 2^D$	a power-of-two integer modulus with $D \leq 16$.
$n, \bar{n}$	integer matrix dimensions with $n, \bar{n} \equiv 0 \pmod 8$.
$B \leq D$	the number of bits encoded in each matrix entry.
len_{A}	the bitlength of seeds for the generation of the matrix A. This is fixed to 128 bits.
len_{sec}	the number of bits that match the bit-security level (i.e., 128 bits for Level 1, 192 bits for Level 3, and 256 bits for Level 5). 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.
len_{salt}	the bitlength of the salt value $salt$.
len_{SE}	the bitlength of the seed value $seed_{SE}$.
χ	the discrete, symmetric error distribution on $\mathbb{Z}$ used by FrodoKEM.
T_χ	a table $(T_\chi(0), T_\chi(1), \ldots, T_\chi(d))$ with $(d+1)$ positive integers based on the cumulative distribution function for χ, with $d \in \mathbb{Z}^+$.

6.1 Public matrix A

A large $n \times n$ matrix with coefficients in $\mathbb{Z}_q$ underlies FrodoKEM. This matrix, called A, is pseudorandomly generated for every generated key.

For its generation, FrodoKEM uses either AES128 (parameter sets FrodoKEM-640-AES, FrodoKEM-976-AES, and FrodoKEM-1344-AES) or SHAKE128 (parameter sets FrodoKEM-640-SHAKE, FrodoKEM-976-SHAKE, and FrodoKEM-1344-SHAKE).

6.2 Deterministic random bit generation

FrodoKEM requires the deterministic generation of random bit strings from random seed values. This is done using SHAKE. The function SHAKE is taken as either SHAKE128 (for the Level 1 parameter sets FrodoKEM-640) or SHAKE256 (for the Level 3 and 5 parameter sets FrodoKEM-976 and FrodoKEM-1344).

7 Supporting functions

The following functions are needed to implement FrodoKEM's key encapsulation routines.

7.1 Octet encoding of bit strings

A bit string $b = (b_0, b_1, \dots, b_{len(b)-1})$ is converted to an octet string (or byte array) 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 $(b_0, b_1, \dots, b_{15})$ is converted into two octets f and g (in this order) as

$$f = b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0$$

$$g = b_{15} \cdot 2^7 + b_{14} \cdot 2^6 + b_{13} \cdot 2^5 + b_{12} \cdot 2^4 + b_{11} \cdot 2^3 + b_{10} \cdot 2^2 + b_9 \cdot 2 + b_8$$

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

Note that, for octet encoding of bit strings, it is always the case that $len(b)$ is a multiple of 8 in FrodoKEM.

7.2 Matrix encoding of bit strings

The function $Encode$ encodes bit strings of length $l = B \cdot \bar{n}^2$ as $\bar{n} \times \bar{n}$ matrices with coefficients in $\mathbb{Z}_q$. Concretely, each B-bit string k in the input, reading from the least to the most significant bit, is encoded as a matrix coefficient by multiplying its integer value by $q/2^B$. This is equivalent to the operation $k \cdot 2^{D-B}$, where the result is always smaller than q for the FrodoKEM parameters (see Table A.1 —). Matrix coefficients are filled in row-by-row from the least to the most significant entry.

The corresponding decoding function $Decode$ does the reverse operation, that is, it takes as input an $\bar{n} \times \bar{n}$ matrix with entries in $\mathbb{Z}_q$ and outputs a bit string of length $l = B \cdot \bar{n}^2$. Concretely, each matrix coefficient $\mathrm{C}_{i,j}$, reading row-by-row from the least to the most significant entry, is decoded by dividing its integer value by $q/2^B$ and then rounding it to the closest integer modulo 2^B. This is equivalent to the operation $\lfloor (\mathrm{C}_{i,j} + 2^{D-B-1})/(2^{D-B}) \rfloor$. The resulting B-bit strings are concatenated in the order that coefficients were decoded (i.e., row-by-row from least to most significant coefficient).

7.3 Packing matrices modulo q

The function $Pack$ transforms an $n_1 \times n_2$ matrix C in $\mathbb{Z}_q$ to a byte array. Concretely, it extracts the least significant D bits from each matrix coefficient, reading the matrix row-by-row from least to most significant coefficient, and concatenates them to produce a bit-string, as follows:

1. For $i = 0$ to $n_1 - 1$ do
   1.1. For $j = 0$ to $n_2 - 1$ do
      1.1.1. Extract bit representation $(c_0, c_1, \dots, c_{D-1}) \leftarrow \mathrm{C}_{i,j}$
      1.1.2. For $k = 0$ to $D - 1$ do
         $b_{(i \cdot n_2 + j)D + k} \leftarrow c_{D-1-k}$
2. Output the octet encoding of $b = (b_0, b_1, \dots, b_{D \cdot n_1 n_2 - 1})$, as per Section 7.1.

For all the matrices in FrodoKEM, it holds that $len(b) = D \cdot n_1 n_2$ is always a multiple of 8 and, therefore, b fits in an exact number of bytes.

The function $Unpack$ does the reverse of this process to transform a byte array to an $n_1 \times n_2$ matrix C in $\mathbb{Z}_q$, converting the input to a bit string, and then extracting D-bit strings and storing each as matrix coefficients $\mathrm{C}_{i,j}$ for $0 \le i < n_1$ and $0 \le j < n_2$ (row-by-row from least to most significant coefficient). The procedure is as follows:

1. Convert the input octet string to a bit string $b = (b_0, b_1, \dots, b_{D \cdot n_1 n_2 - 1})$, as per Section 7.1
2. For $i = 0$ to $n_1 - 1$ do

2.1. For $j = 0$ to $n_2 - 1$ do

$$C_{i,j} \leftarrow \sum_{k=0}^{D-1} b_{(i \cdot n_2 + j)D + k} \, 2^{D-1-k} \equiv (b_{(i \cdot n_2 + j)D + D - 1}, \ldots, b_{(i \cdot n_2 + j)D + 1}, b_{(i \cdot n_2 + j)D})$$

3. Output C

7.4 Sampling from the error distribution

Given a random bit string $r = (r_0, r_1, \ldots, r_{15})$, the function *Sample* does sampling from FrodoKEM's error distribution χ via inversion sampling using a fixed distribution table $T_\chi = (T_\chi(0), T_\chi(1), \ldots, T_\chi(d))$, as follows:

1. Set $t \leftarrow \sum_{i=1}^{15} r_i 2^{i-1} \equiv (r_1, r_2, \ldots, r_{15})$
2. $e \leftarrow 0$
3. For $i = 0$ to $d - 1$ do
   3.1. (In constant time) If $t > T_\chi(i)$ then $e \leftarrow e + 1$
4. $e \leftarrow (-1)^{r_0} \cdot e$
5. Output e

The output of the algorithm is a small integer in the range $\{-d, -d+1, \ldots, -1, 0, 1, \ldots, d-1, d\}$. The tables T_χ corresponding to each of FrodoKEM's parameter sets are given in Table A.4 —.

7.5 Matrix sampling from the error distribution

The function *SampleMatrix* that samples an $n_1 \times n_2$ matrix using the function *Sample* is as follows. Given $(n_1 \times n_2)$ 16-bit random strings $r^{(i)}$ and the dimension values n_1 and n_2, this function generates an $n_1 \times n_2$ matrix (row-by-row and from least to most significant coefficient) by successively calling $n_1 \times n_2$ times the function *Sample*, as follows:

1. For $i = 0$ to $n_1 - 1$ do
   1.1. For $j = 0$ to $n_2 - 1$ do

$$E_{i,j} \leftarrow Sample(r^{(i \cdot n_2 + j)})$$

2. Output E

7.6 Pseudorandom matrix generation

The function *Gen* takes as input a $len_A = 128$-bit seed $seed_A$ and an implicit dimension n, and outputs an $n \times n$ pseudorandom matrix A, where all the coefficients are in $\mathbb{Z}_q$.

There are two versions of this function: one based on AES128 and another based on SHAKE128. These are shown below. In both cases, the matrix A is generated row-by-row from least to most significant coefficient.

7.6.1 Matrix A generation with AES128

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

1. For $i = 0$ to $n - 1$ do
   1.1. For $j = 0$ to $n - 1$ step 8 do
      1.1.1. $b \leftarrow i||j||0||0||0||0||0||0$, where each concatenated element is encoded as a 16-bit string represented in the little-endian byte order such that, e.g., $(i_0, i_1, \ldots, i_{15}) \leftarrow \sum_{k=0}^{15} i_k 2^k$, and $len(b) = 128$
      1.1.2. $C_{i,j}||C_{i,j+1}|| \ldots ||C_{i,j+7} \leftarrow \text{AES128}_{seed_A}(b)$, where each matrix coefficient $C_{i,j}$ is a 16-bit string interpreted as a nonnegative integer in the little-endian byte order, such that $C_{i,j} = \sum_{k=0}^{15} c_k 2^k$, corresponding to the bit string $(c_0, c_1, \ldots, c_{15})$ in the output of AES128.

1.1.3. For $k = 0$ to 7 do

$$\mathrm{A}_{i,j+k} \leftarrow \mathrm{C}_{i,j+k} \bmod q$$

2. Output $\mathrm{A} = (\mathrm{A}_{i,j})$

7.6.2 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 row).

1. For $i = 0$ to $n - 1$ do

   1.1. $b \leftarrow i||seed_{\mathrm{A}}$, where i is encoded as a 16-bit string represented in the little-endian byte order such that $(i_0, i_1, \ldots, i_{15}) \leftarrow \sum_{k=0}^{15} i_k 2^k$, and hence $len(b) = len_{\mathrm{A}} + 16$

   1.2. $\mathrm{C}_{i,0}||\mathrm{C}_{i,1}|| \ldots ||\mathrm{C}_{i,n-1} \leftarrow \mathrm{SHAKE128}(b, 16n)$, where each matrix coefficient $\mathrm{C}_{i,j}$ is a 16-bit string interpreted as a nonnegative integer in the little-endian byte order, such that such that $\mathrm{C}_{i,j} = \sum_{k=0}^{15} c_k 2^k$, corresponding to the bit string $(c_0, c_1, \ldots, c_{15})$ in the output of SHAKE128.

   1.3. For $j = 0$ to $n - 1$ do

   $$\mathrm{A}_{i,j} \leftarrow \mathrm{C}_{i,j} \bmod q$$

2. Output $\mathrm{A} = (\mathrm{A}_{i,j})$

8 Key encapsulation mechanism

The routines for key generation ($KeyGen$), encapsulation ($Encaps$) and decapsulation ($Decaps$) are detailed below. Public keys have bitlength $len_A + Dn\bar{n}$. Secret keys have bitlength $2 \cdot len_{sec} + len_{\mathrm{A}} + Dn\bar{n} + 16n\bar{n}$. Ciphertexts have bitlength $Dn\bar{n} + D\bar{n}^2 + len_{salt}$. Shared secrets have bitlength len_{sec}. All the resulting entries of the addition and multiplication of matrices are reduced modulo q.

8.1 Key generation

The key generation function $KeyGen$ is shown below. $KeyGen$ outputs the keypair $(pk, sk) = (seed_A||b, s||seed_A||b||\mathrm{S}^{\mathrm{T}}||pkh)$.

1. Choose uniformly random seeds $s, seed_{SE}$ and z of bitlengths len_{sec}, len_{SE} and len_A (resp.)
2. Generate pseudorandom seed $seed_A \leftarrow \mathrm{SHAKE}(z, len_A)$
3. Generate the matrix $\mathrm{A} \leftarrow Gen(seed_{\mathrm{A}})$
4. Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, \ldots, r^{(2n\bar{n}-1)}) \leftarrow \mathrm{SHAKE}(\mathrm{0x5F}||seed_{SE}, 32n\bar{n})$
5. Sample error matrix $\mathrm{S}^{\mathrm{T}} \leftarrow SampleMatrix((r^{(0)}, r^{(1)}, \ldots, r^{(n\bar{n}-1)}), \bar{n}, n)$
6. Sample error matrix $\mathrm{E} \leftarrow SampleMatrix((r^{(n\bar{n})}, r^{(n\bar{n}+1)}, \ldots, r^{(2n\bar{n}-1)}), n, \bar{n})$
7. Compute $\mathrm{B} \leftarrow \mathrm{AS} + \mathrm{E}$
8. Compute $b \leftarrow Pack(\mathrm{B})$
9. Compute $pkh \leftarrow \mathrm{SHAKE}(seed_{\mathrm{A}}||b, len_{sec})$
10. Return public key $pk \leftarrow seed_{\mathrm{A}}||b$ and secret key $sk \leftarrow s||seed_{\mathrm{A}}||b||\mathrm{S}^{\mathrm{T}}||pkh$. Here, S^{T} is encoded row-by-row from $\mathrm{S}^{\mathrm{T}}_{0,0}$ to $\mathrm{S}^{\mathrm{T}}_{\bar{n}-1,n-1}$, where each matrix coefficient $\mathrm{S}^{\mathrm{T}}_{i,j}$ is a signed integer encoded as a 16-bit string in the little-endian byte order such that $(s_0, s_1, \ldots, s_{15}) \leftarrow \mathrm{S}^{\mathrm{T}}_{i,j} = -s_{15}2^{15} + \sum_{k=0}^{14} s_k 2^k$

8.2 Encapsulation

The function $Encaps$ gets as input a public key $pk = seed_A||b$ and outputs a ciphertext $c = c_1||c_2||salt$ and a shared secret ss.

1. Choose uniformly random values u and $salt$ of bitlengths len_{sec} and len_{salt}, respectively.
2. Compute $pkh \leftarrow \text{SHAKE}(pk, len_{sec})$
3. Generate pseudorandom values $seed_{SE}||k \leftarrow \text{SHAKE}(pkh||u||salt, len_{SE} + len_{sec})$
4. Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, \dots, r^{(2\bar{n}n+\bar{n}^2-1)}) \leftarrow \text{SHAKE}(0\text{x}96||seed_{SE}, 16(2\bar{n}n + \bar{n}^2))$
5. Sample error matrix $\text{S}' \leftarrow SampleMatrix((r^{(0)}, r^{(1)}, \dots, r^{(n\bar{n}-1)}), \bar{n}, n)$
6. Sample error matrix $\text{E}' \leftarrow SampleMatrix((r^{(n\bar{n})}, r^{(n\bar{n}+1)}, \dots, r^{(2n\bar{n}-1)}), \bar{n}, n)$
7. Generate the matrix $\text{A} \leftarrow Gen(seed_{\text{A}})$
8. Compute $\text{B}' \leftarrow \text{S}'\text{A} + \text{E}'$
9. Compute $c_1 \leftarrow Pack(\text{B}')$
10. Sample error matrix $\text{E}'' \leftarrow SampleMatrix((r^{(2\bar{n}n)}, r^{(2\bar{n}n+1)}, \dots, r^{(2\bar{n}n+\bar{n}^2-1)}), \bar{n}, \bar{n})$
11. Compute $\text{B} \leftarrow Unpack(b, n, \bar{n})$
12. Compute $\text{V} \leftarrow \text{S}'\text{B} + \text{E}''$
13. Compute $\text{C} \leftarrow \text{V} + Encode(u)$
14. Compute $c_2 \leftarrow Pack(\text{C})$
15. Compute $ss \leftarrow \text{SHAKE}(c_1||c_2||salt||k, len_{sec})$
16. Return ciphertext $c \leftarrow c_1||c_2||salt$ and shared secret ss

NOTE: Implementors should evaluate the risk of releasing RNG output when the (public) salt is taken directly from a randomness source in step 1. To avoid this risk, implementations can for instance hash the RNG output prior to assigning its output to $salt$ in step 1.

8.3 Decapsulation

The function $Decaps$ takes as input a ciphertext $c = c_1||c_2||salt$ and a secret key $sk = s||seed_A||b||\text{S}^{\text{T}}||pkh$, and outputs a shared secret ss.

1. Compute $\text{B}' \leftarrow Unpack(c_1, \bar{n}, n)$
2. Compute $\text{C} \leftarrow Unpack(c_2, \bar{n}, \bar{n})$
3. Compute $\text{M} \leftarrow \text{C} - \text{B}'\text{S}$
4. Compute $u' \leftarrow Decode(\text{M})$
5. Generate pseudorandom values $seed'_{SE}||k' \leftarrow \text{SHAKE}(pkh||u'||salt, len_{SE} + len_{sec})$
6. Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, \dots, r^{(2\bar{n}n+\bar{n}^2-1)}) \leftarrow \text{SHAKE}(0\text{x}96||seed'_{SE}, 16(2\bar{n}n + \bar{n}^2))$
7. Sample error matrix $\text{S}' \leftarrow SampleMatrix((r^{(0)}, r^{(1)}, \dots, r^{(n\bar{n}-1)}), \bar{n}, n)$
8. Sample error matrix $\text{E}' \leftarrow SampleMatrix((r^{(n\bar{n})}, r^{(n\bar{n}+1)}, \dots, r^{(2n\bar{n}-1)}), \bar{n}, n)$
9. Generate the matrix $\text{A} \leftarrow Gen(seed_{\text{A}})$
10. Compute $\text{B}'' \leftarrow \text{S}'\text{A} + \text{E}'$
11. Sample error matrix $\text{E}'' \leftarrow SampleMatrix((r^{(2\bar{n}n)}, r^{(2\bar{n}n+1)}, \dots, r^{(2\bar{n}n+\bar{n}^2-1)}), \bar{n}, \bar{n})$

12. Compute B ← $Unpack(b, n, \bar{n})$
13. Compute V ← S′B + E′′
14. Compute C′ ← V + $Encode(u')$
15. (In constant time) $\bar{k} \leftarrow k'$ if B′||C = B′′||C′ else $\bar{k} \leftarrow s$
16. Compute ss ← SHAKE($c_1||c_2||salt||\bar{k}, len_{sec}$)
17. Return shared secret ss

9 Security considerations

The security of the FrodoKEM instantiations described in this document are supported by a conservative analysis of the best-known cryptanalytic attacks against the LWE problem (Section 9.1). Furthermore, the design soundness of FrodoKEM is conceptually supported by security reductions from worst-case problems (Section 9.2). An important feature of FrodoKEM is that the parameters were specifically chosen to be covered by the reductions.

9.1 Cryptanalytic attacks: the "core-SVP" hardness

The concrete security estimates for FrodoKEM are based on a conservative methodology that estimates the "core-SVP hardness" of solving the underlying LWE problem. Specifically, the FrodoKEM parameter sets contained in this document were chosen as to safeguard against the two state-of-the-art cryptanalytic attacks in lattice-based cryptography: primal and dual attacks.

Table A.7 — summarizes the estimated (classical and quantum) costs of these attacks for a single instance of the LWE problem corresponding to each FrodoKEM parameter set.

9.1.1 Refined security estimates

The core-SVP methodology counts only the first-order exponential cost of just one (quantum) shortest-vector computation on a lattice of appropriate dimension to solve the relevant LWE problem. Because it ignores lower-order terms like the significant subexponential factors in the runtime, as well as the large exponential memory requirements, it significantly underestimates the actual cost of known attacks, and allows for significant future improvement in these attacks.

A more refined analysis considers the approximated number of gates that is required to solve the LWE problem (refer to Annex C). The resulting gate counts, which are detailed in Table A.8 —, show that the FrodoKEM parameter sets comfortably match their respective target security levels with a large margin. This aligns with FrodoKEM's conservative design approach and hedges against improvements for cryptanalytic algorithms solving general lattice problems.

9.2 Security reductions

The following is a summary of the reductions supporting the security of FrodoKEM. A detailed treatment of the supporting theorems can be found in Section 5 of [FrodoKEM_spec].

1. FrodoKEM, using the concrete error distributions χ specified in Table A.3 —, is an IND-CCA-secure KEM against classical attacks in the classical random oracle model, under the assumption that FrodoPKE using a rounded Gaussian error distribution is an IND-CPA-secure public-key encryption scheme against classical attacks. This is Theorem 5.1 of [FrodoKEM_spec], and the reduction is tight.
2. FrodoKEM, using any error distribution, is an IND-CCA-secure KEM against quantum attackers in the quantum random oracle model, under the assumption that FrodoPKE using the same error distribution is an OW-CPA-secure public-key encryption scheme against quantum attackers. This is Theorem 5.8 of [FrodoKEM_spec], and the reduction is non-tight. This theorem gives support for the security of general constructions of LWE-based KEMs in the style of FrodoKEM against quantum adversaries, but it does not concretely support the bit-security of the FrodoKEM

instantiations in this document, which is why the corresponding column from Table A.9 — is omitted.

3. Changing the distribution of matrix A from a truly uniform distribution to one generated from a public random seed in a pseudorandom fashion does not affect the security of FrodoKEM or FrodoPKE, provided that the pseudorandom generator is modeled either as an ideal cipher (when using AES128) or a random oracle (when using SHAKE128). This is shown in Section 5.1.3 of [FrodoKEM_spec].
4. FrodoPKE, using any error distribution and a uniformly random A, is an IND-CPA-secure public-key encryption scheme under the assumption that the uniform-secret learning with errors decision problem is hard for the same parameters (except for a small additive loss in the number of samples), for either classical or quantum adversaries. This is a consequence of Theorems 5.9 and 5.10 of [FrodoKEM_spec], and the result is tight.
5. The uniform-secret learning with errors decision problem, using a rounded Gaussian distribution with parameter σ from Table A.3 — and an appropriate bound on the number of samples, is hard under the assumption that the worst-case bounded-distance decoding with discrete Gaussian samples problem (BDDwDGS, Definition 5.11 of [FrodoKEM_spec]) is hard for related parameters. Theorem 5.12 of [FrodoKEM_spec] gives a non-tight classical reduction against classical or quantum adversaries (in the standard model).

9.3 Decryption failures

The concrete FrodoPKE parameters induce a tiny probability of incorrect decryption (see Table A.9 —), for honestly generated keys and ciphertexts. This is because a ciphertext may decrypt to a different message than the encrypted one, if the combination of the short error matrices in the key and the ciphertext is too large. This aspect of the scheme carries over to the transformed, CCA-targeting FrodoKEM, where incorrect decryption in the underlying PKE typically causes a decryption failure.

It has long been well understood that the ability to induce incorrect decryption or decryption failure in LWE-based schemes can leak information about the secret key, up to and including full key recovery (with sufficiently many failures). In brief, this is because such failures indicate some correlation between the secret key and the encryption randomness.

In the context of chosen-ciphertext attacks on FrodoKEM, the attacker can attempt to create ciphertexts whose underlying error matrices—which are derived pseudorandomly using an attacker-chosen seed—are atypically large. Such "weak" ciphertexts have an increased probability of inducing decryption failures when submitted to a decryption oracle. The process of searching for such ciphertexts, which can be done offline (without using a decryption oracle), is known as "failure boosting."

Based on the analysis in [DGJNVV19], it has been determined that the FrodoKEM parameters FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344 suffer no loss in their claimed security (either classical or quantum) under such attacks. This is essentially because the cost of finding weak ciphertexts exceeds the benefit obtained from the corresponding increase in decryption failure probability. (For FrodoKEM-1344, failure boosting does not provide any improvement over the intrinsic failure probability of $2^{-252.5}$. This is considered to be consistent with the Level 5 requirement of 256 bits of brute-force security, because the overhead in using decryption failures to win the CCA security game exceeds 3.5 bits.)

9.4 Backdoors and all-for-the-price-of-one attacks exploiting the matrix A

If the matrix A was a fixed system parameter, there would be a risk for A to be backdoored by a malicious adversary. Moreover, a unique value A would make all the secure connections rely on a single instance of a lattice problem. Consequently, to eliminate the possibility of backdoors and to reduce the risk of all-for-the-price-of-one attacks the matrix A is generated dynamically and pseudorandomly for every generated key. Note that the many encapsulations produced under a given public key still use the same matrix A. However, the attack surface is reduced significantly by eliminating the risk of a full-scale all-for-the-price-of-one attack over all public keys.

Since the ephemeral FrodoKEM variant requires the use of a fresh matrix A after a small number of protocol executions (e.g., 2^8), the scheme instantiations eFrodoKEM-640, eFrodoKEM-976, and eFrodoKEM-1344 virtually eliminate the risk of all-for-the-price-of-one attacks exploiting A.

9.5 Security against multi-target and multi-ciphertext attacks

9.5.1 Multi-target attacks

Multi-target attacks are those which aim to break security against one of N public keys. The security analysis of FrodoKEM in [FrodoKEM_spec] does not formally cover security in the multi-target setting. However, in order to reduce the risk of batch attacks targeting multiple keys, FrodoKEM includes the hashed value of the public key pkh in the computation of the random bit strings r (Section 8.2, Line 3).

9.5.2 Multi-ciphertext attacks

Multi-ciphertext attacks are those which target a single public key, but aim to break one of N ciphertexts. Given that the message space for u is limited by the value len_{sec}, an adversary could run a multi-ciphertext attack targeting a specific public key via a brute-force search on this space of bitlength len_{sec}. This risk is mitigated by concatenating a uniformly random value $salt$ of bitlength len_{salt} to the message u (Section 8.2, Line 3) , which is chosen freshly for each encapsulation. The $salt$ value is made public as part of the ciphertext output by encapsulation. For similar reasons, the seed $seed_{SE}$ used to generate the random bit strings r needs to have a bitlength greater than len_{sec}. See the recommended sizes for len_{salt} and len_{SE} in Table A.2 —.

Multi-ciphertext attacks are only relevant when there is a possibility of producing a large number of ciphertexts per public key. Hence, these attacks do not apply to eFrodoKEM since in this case it is expected that each key pair either is generated freshly at each execution of the protocol or corresponds to a fairly small number of ciphertexts. Accordingly, the ephemeral instantiations of FrodoKEM discard the use of $salt$, and the bitlength of $seed_{SE}$ matches len_{sec} (refer to Table A.2 —).

9.6 Ephemeral FrodoKEM

The ephemeral instantiations of FrodoKEM (eFrodoKEM-640, eFrodoKEM-976, and eFrodoKEM-1344), which do not include the use of $salt$, are exclusively intended for applications that guarantee that only a small number of ciphertexts (e.g., 2^8) are produced per public key. Otherwise, the scheme's security might be degraded in the context of a multi-ciphertext attack, as explained in Section 9.5.2.

10 Implementation considerations

10.1 Reference implementations

Reference [FrodoKEM_code] provides portable and optimized implementations of FrodoKEM and eFrodoKEM in C with support for a wide range of platforms. This reference also provides reference implementations written in Python that are intended to be readable and as close as possible to a line-by-line mapping of the algorithm specification to executable code. All the implementations include functional tests and known answer tests.

10.2 Reusing A

Generating A from $seed_A$ can be a significant computational burden, but this cost can be amortized by relaxing the requirement that a fresh $seed_A$ be used for every instance of key encapsulation, e.g., by caching and reusing A for a small period of time. It has been observed that in some settings the cost of generating A may represent roughly 40% of the cost of encapsulation and decapsulation. A straightforward argument shows that the amortization above is compatible with all the relevant security reductions. But importantly, it now allows for an all-for-the-price-of-one attack against those key encapsulations that share the same A. This can be mitigated by making sure that A is cached and reused only for a small number of uses, but this needs to be done in a very careful manner.

10.3 Side-channel resistance

10.3.1 Timing attacks

One of the features of FrodoKEM is that it is easy to implement and naturally facilitates writing implementations that are compact and run in constant-time. This latter feature aids to avoid common cryptographic implementation mistakes which can lead to key-extraction based on, for instance, timing differences when executing the code.

Nonetheless, care must be taken to avoid timing leaks. In particular, as noted in Section 7.4 (Line 3.1), the comparison in the for-loop for the implementation of the error sampling must be implemented in a constant-time manner (also making sure that the sampling algorithm loops through the entire table T_χ). Likewise, during the decapsulation (Section 8.3, Line 15), both k' and s must be read and the bit string comparison of $B'||C$ and $B''||C'$ must be done in constant-time.

In FrodoKEM, matrix coefficients in $\mathbb{Z}_q$, where q is a power-of-two, can be easily reduced modulo q via masking in a constant-time manner.

10.3.2 Other side-channel attacks

Powerful attacks aimed at extracting secret data from a device through the analysis of its power consumption or electromagnetic emanation are known to be able to successfully break the security of lattice-based schemes. Protecting software and hardware implementations where such attacks are realistic is an active research area. While most work has been done so far to protect ring LWE schemes, the generic attack methods as well as the countermeasures that apply to them also do apply to LWE. Nevertheless, given FrodoKEM's simplicity relative to ring LWE (e.g., there is no use of FFT-based multiplication techniques), the attack surface is significantly smaller. Finally, the ephemeral instantiations of FrodoKEM potentially allow to reduce further the attack surface.

Annex A
(informative)

Parameters

A summary of the parameters defining the parameter sets of FrodoKEM is presented in Table A.2 —. The bitlengths of $seed_{SE}$ and $salt$ for each parameter set are presented in Table A.2 —.

Table A.1 — Parameters for (e)FrodoKEM-640, (e)FrodoKEM-976 and (e)FrodoKEM-1344.

	(e)FrodoKEM-640	(e)FrodoKEM-976	(e)FrodoKEM-1344
D	15	16	16
q	32768	65536	65536
n	640	976	1344
$\bar{n}$	8	8	8
B	2	3	4
d	12	10	6
len_A	128	128	128
len_{sec}	128	192	256
SHAKE	SHAKE128	SHAKE256	SHAKE256

Table A.2 — Sizes (in bits) of $seed_{SE}$ and $salt$.

	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
len_{SE}	256	384	512
len_{salt}	256	384	512
	eFrodoKEM-640	**eFrodoKEM-976**	**eFrodoKEM-1344**
len_{SE}	128	192	256
len_{salt}	0	0	0

The error distributions χ are detailed in Table A.3 —.

Table A.3 — Error distributions.

	σ	Probability of (in multiples of 2^{-16})												Rényi	
		0	±1	±2	±3	±4	±5	±6	±7	±8	±9	±10	±11	±12	order	divergence
$\chi_{\text{Frodo-640}}$	2.8	9288	8720	7216	5264	3384	1918	958	422	164	56	17	4	1	200	0.324×10^{-4}
$\chi_{\text{Frodo-976}}$	2.3	11278	10277	7774	4882	2545	1101	396	118	29	6	1			500	0.140×10^{-4}
$\chi_{\text{Frodo-1344}}$	1.4	18286	14320	6876	2023	364	40	2							1000	0.264×10^{-4}

The distribution tables $T_\chi = (T_\chi(0), T_\chi(1), \ldots, T_\chi(d))$ that are used for sampling by each parameter set are detailed in Table A.4 —.

Table A.4 — The distribution table entries for sampling for each parameter set.

Table entries	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
$T_\chi(0)$	4643	5638	9142
$T_\chi(1)$	13363	15915	23462
$T_\chi(2)$	20579	23689	30338
$T_\chi(3)$	25843	28571	32361
$T_\chi(4)$	29227	31116	32725
$T_\chi(5)$	31145	32217	32765
$T_\chi(6)$	32103	32613	32767
$T_\chi(7)$	32525	32731	
$T_\chi(8)$	32689	32760	
$T_\chi(9)$	32745	32766	
$T_\chi(10)$	32762	32767	
$T_\chi(11)$	32766		
$T_\chi(12)$	32767		

The sizes (in bytes) of inputs and outputs for each instantiation are detailed in Table A.5 — and Table A.6 —.

Table A.5 — Size (in bytes) of inputs and outputs of FrodoKEM.

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

Table A.6 — Size (in bytes) of inputs and outputs of the ephemeral instantiations of FrodoKEM.

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

Annex B
(informative)

FrodoKEM security estimates: core-SVP hardness

The costs of the primal and dual attacks for our suggested parameters are given in Table A.7 —. The costs are listed for a single instance of the LWE problem.

Table A.7 — Primal and dual attacks on a single instance of an LWE problem. Attack costs are given as the base-2 logarithm.

	Attack	Classical	Quantum	Plausible
Frodo-640	Primal	150.8	137.6	109.6
	Dual	149.6	136.5	108.7
Frodo-976	Primal	216.0	196.7	156.0
	Dual	214.5	195.4	154.9
Frodo-1344	Primal	281.6	256.3	202.6
	Dual	279.8	254.7	201.4

Annex C
(informative)

Refined security analysis

The refined analysis of the security of the FrodoKEM parameter sets follows the methodology in the Round-3 specification document of the CRYSTALS-Kyber key encapsulation mechanism [Kyber-spec].

The scripts for these refined estimates are provided in a git branch of the leaky-LWE-estimator of [DDGR20], and lead to the estimates given in Table A.8 —. Refer to [Kyber-spec] for the details of this analysis. For the classical hardness of the LWE problem at Levels 1 and 2, it is estimated in [Kyber-spec] that the true cost is no more than 16 bits away from this estimate, in either direction.

A similarly refined count of quantum gates seems to be essentially irrelevant: the work of [AGPS20] concluded that obtaining a quantum speed-up for sieving is rather tenuous, while the quantum security target for each level is significantly lower than the classical target.

Table A.8 — Refined estimates for the LWE hardness.

	$\log_2$(gates)	$\log_2$(memory in bits)
Frodo-640	175.1	110.4
Frodo-976	240.0	155.8
Frodo-1344	305.4	202.1

Annex D
(informative)

Security estimates for FrodoKEM derived from reductions

Table A.9 — details the security claims for FrodoKEM and its components, resulting from a series of security reductions. These claims are assumed to be weaker than the ones supported by analyses based on concrete cryptanalytic attacks (Annex B and Annex C).

The columns LWE security C, Q and P respectively denote security, in bits, for classical, quantum, and plausible attacks on $2\bar{n}$ instances of the normal-form (decisional) LWE problem with Gaussian error distribution as estimated by the methodology of Section 5.1 in [FrodoKEM_spec]. The column IND-CCA security C denotes IND-CCA security, in bits, for classical random oracle model attacks according to Theorem 5.1 in [FrodoKEM_spec].

Table A.9 — Security estimates derived from reductions.

	Failure rate	LWE security			IND-CCA security
		C	Q	P	C
Frodo-640	$2^{-138.7}$	145	132	104	141
Frodo-976	$2^{-199.6}$	210	191	150	206
Frodo-1344	$2^{-252.5}$	275	250	197	268

Bibliography

[AGPS20] M. R. Albrecht, V. Gheorghiu, E. W. Postlethwaite, and J. M. Schanck. Estimating quantum speedups for lattice sieves. ASIACRYPT'20, 2020. https://eprint.iacr.org/2019/1161.

[DDGR20] D. Dachman-Soled, L. Ducas, H. Gong, and M. Rossi. LWE with side information: Attacks and concrete security estimation. CRYPTO'20, 2020. https://eprint.iacr.org/2020/292.pdf. The leaky-LWE-estimator is available at: https://github.com/lducas/leaky-LWE-Estimator/tree/NIST-round3.

[DGJNVV19] J.-P. D'Anvers, Q. Guo, T. Johansson, A. Nilsson, F. Vercauteren, and I. Verbauwhede. Decryption failure attacks on IND-CCA secure lattice-based schemes. PKC 2019, 2019.

[Kyber_spec] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation, version 3.02, August 04, 2021. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[FIPS197] National Institute of Standards and Technology. Advanced Encryption Standard (AES). (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 197, 2001. https://doi.org/10.6028/NIST.FIPS.197.

[FIPS202] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 202, 2015. http://dx.doi.org/10.6028/NIST.FIPS.202.

[FrodoKEM_code] E. Alkim, J. Bos, L. Ducas, K. Easterbrook, B. LaMacchia, P. Longa, I. Mironov, M. Naehrig, V. Nikolaenko, C. Peikert, A. Raghunathan, and D. Stebila. FrodoKEM: Reference implementations in C and Python. https://github.com/microsoft/PQCrypto-LWEKE.

[FrodoKEM_spec] E. Alkim, J. Bos, L. Ducas, K. Easterbrook, B. LaMacchia, P. Longa, I. Mironov, M. Naehrig, V. Nikolaenko, C. Peikert, A. Raghunathan, and D. Stebila. FrodoKEM: Algorithm specifications and supporting documentation. https://frodokem.org/.

[NIST-CFP] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

Revision history

March 14, 2023

- Initial public release of this document.

December 5, 2024

- Several minor corrections and improvements throughout the document.
- Corrected typos in Table A.4 — for parameter FrodoKEM-1344: entry $T_\chi(5)$ was corrected to 32765, and missing entry $T_\chi(6) = 32767$ was added.