Network Working Group
Internet-Draft
Intended status: Informational
Expires: 20 July 2026

F. G端nther
IBM Research Europe - Zurich
D. Stebila
University of Waterloo
S. Veitch
ETH Zurich

16 January 2026

Kemeleon Encodings
draft-irtf-cfrg-kemeleon-01

Abstract

This document specifies Kemeleon encoding algorithms for encoding ML-KEM encapsulation keys and ciphertexts as random bytestrings. Kemeleon encodings provide obfuscation of encapsulation keys and ciphertexts, relying on module LWE assumptions.

About This Document

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

The latest revision of this draft can be found at https://ssveitch.github.io/draft-kemeleon/draft-irtf-cfrg-kemeleon.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-irtf-cfrg-kemeleon/.

Source for this draft and an issue tracker can be found at https://github.com/ssveitch/draft-kemeleon.

Status of This Memo

Table of Contents

1. Introduction

ML-KEM [FIPS203] is a post-quantum key-encapsulation mechanism (KEM) recently standardized by NIST, Many applications are transitioning from classical Diffie-Hellman (DH) based solutions to constructions based on ML-KEM. The use of Elligator and related Hash-to-Curve [RFC9380] algorithms are ubiquitous in DH-based protocols where DH shares are required to be encoded as, and look indistinguishable from, random bytestrings. For example, applications using Elligator include protocols used for censorship circumvention in Tor [OBFS4], password-authenticated key exchange (PAKE) protocols [CPACE]

[OPAQUE], and private set intersection (PSI) [ECDH-PSI].

For the post-quantum transition, an analogous encoding for (ML-)KEM encapsulation keys and ciphertexts to random bytestrings is required. This document specifies such an encoding, Kemeleon, for ML-KEM encapsulation keys and ciphertexts. Kemeleon was introduced in [GSV24] for building an (post-quantum) "obfuscated" KEM whose encapsulation keys and ciphertexts are indistinguishable from random. This document specifies a version of the Kemeleon encoding that avoids any failure probability, as well as an alternate version that trades some failure probability for smaller encoding size.

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. Notation / ML-KEM Background

A KEM consists of three algorithms:

* KeyGen() -> (ek, dk): A probabilistic key generation algorithm that, with no input, generates an encapsulation key ek and a decapsulation key dk.

* Encaps(ek) -> (c, K): A probabilistic encapsulation algorithm that takes as input an encapsulation key ek, and outputs a ciphertext ct and shared secret K.

* Decaps(dk, c) -> K: A decapsulation algorithm that takes as input a decapsulation key dk and ciphertext c, and outputs a shared secret K.

The following variables and functions are adopted from [FIPS203]:

* q = 3329, n = 256

* Compress_d : x -> round((2^d/q)*x) mod 2^d (Equation 4.7)

* Decompress_d : y -> round((q/2^d)*y) (Equation 4.8)

* remaining parameters k, d_u, d_v, etc. are defined by the respective ML-KEM parameter set -- this document writes du and dv in place of d_u, d_v in pseudocode

ML-KEM.KeyGen() (Section 7.1 [FIPS203]) produces an encapsulation key, ek and a decapsulation key, dk. Encapsulation keys consist of byte-encoded vectors of coefficients in Z_q, where each coefficient is encoded in 12 bits, together with a 32-byte seed for generating the matrix A. ML-KEM.Encaps(ek) (Section 7.2 [FIPS203]) produces ciphertexts consisting of byte-encoded compressed vectors of cofficients, where each coefficient in Z_q is compressed by a certain number of bits (depending on the ML-KEM parameter set).

The following terms and notation are used throughout this document:

* a[i] denotes the ith position of a vector a of coefficients

* concat(x0, ..., xN): returns the concatenation of bytestrings.

4. Kemeleon Encoding

At a high level, the constructions in this document instantiate the following functions:

* EncodeEk(ek) -> eek is the (possibly randomized) encoding algorithm that on input an encapsulation key, outputs an obfuscated encapsulation key or an error.

* DecodeEk(eek) -> ek is the deterministic decoding algorithm that on input an obfuscated encapsulation key, outputs an encapsulation key.

* EncodeCtxt(c) -> ec is the (possibly randomized) encoding algorithm that on input a ciphertext, outputs an obfuscated ciphertext or an error.

* DecodeCtxt(ec) -> c is the deterministic decoding algorithm that on input an obfuscated ciphertext, outputs a ciphertext.

4.1. Common Functions

The following function maps a vector of length n of coefficients modulo q to a large integer. Applying the technique from [ELL2], where r is the large integer resulting from accumulating coefficients, we then choose m at random from [0,floor((2^(b+t)-r)/(q^n))], where b = ceil(n*log2(q)) and t is a security parameter, and return r + m*q^(n). Notably, the random value m need not be transmitted alongside the encoded values. This results in encoded values whose statistical distance from uniform is at most 2^-t. Notably, this statistical distance is unconditional; we hence fix t=128. This results in the encoding size increasing by t bits, i.e., 16 bytes.

```
VectorEncode(a):
   r = 0
   t = 128
   b = ceil(n*log2(q))
   for i from 1 to n:
      r += q^(i-1)*a[i]
   m <--$ [0,...,floor((2^(b+t)-r)/(q^(n)))]
   return r + m*q^n

VectorDecode(r):
   r = r % q^n
   for i from 1 to n:
      a[i] = r % q
      r = r // q
   return a
```

The following algorithm samples an uncompressed pre-image of a coefficient c at random, where u is the decompressed value of c. It must take as input values of u that are output from Decompress_d. The mapping is based on the Compress_d, Decompress_d algorithms from (Section 4.2.1 [FIPS203]).

```
SamplePreimage(d,u,c):
   if d == 10:
      if Compress_d(u + 2) == c:
         rand <--$ [-1,0,1,2]
      else if Compress_d(u - 2) == c:
         rand <--$ [-2,-1,0,1]
      else:
         rand <--$ [-1,0,1]
      return u + rand
   if d == 11:
      if Compress_d(u + 1) == c:
         rand <--$ [0,1]
      else if Compress_d(u - 1) == c:
         rand <--$ [-1,0]
      else:
         rand = 0
      return u + rand
   if d == 5:
      if u == 0:
         rand <--$ [-52,...,52]
      else if u <= 1560:
         rand <--$ [-51,...,52]
      else:
         rand <--$ [-52,...,51]
      return u + rand
   if d == 4:
      if u == 0:
         rand <--$ [-104,...,104]
      else if u <= 1456:
         rand <--$ [-103,...,104]
      else:
         rand <--$ [-104,...,103]
      return u + rand
   else:
      return err
```

4.2. Encoding Encapsulation Keys

The following algorithms encode ML-KEM encapsulation keys as random bytestrings. rho is the public seed used to generate the public matrix A [FIPS203]. This is already a random 32-byte string, so it is returned alongside the encoded value of t. t is a vector of k polynomials with n coefficients. We treat each polynomial in t as a vector of n coefficient, for which we apply VectorEncode. From this, we obtain k values that are then concatenated.

```
Kemeleon.EncodeEk(ek = (t, rho)):
   for i in range(k):
      r_i = VectorEncode(t[i])
   r = concat(r_1,...,r_k)
   return concat(r,rho)

Kemeleon.DecodeEk(eek):
   r_1,..,r_k,rho = eek # rho and each r_i is fixed length
   t = []
   for i in range(k):
      t_i = VectorDecode(r_i)
      t.append(t_i)
   return (t, rho)
```

4.3. Encoding Ciphertexts

ML-KEM ciphertexts consist of two components: c_1, a vector of k polynomials with n coefficients mod 2^du, and c_2, a polynomial with n coefficients mod 2^dv. The coefficients of these polynomials are not uniformly distributed, as a result of the compression step in encapsulation. The following encoding function decompresses and recovers a random preimage of this compression step in order to recover the uniform distribution of coefficients. Then, the same vector encoding step used for encapsulation keys can be applied.

```
Kemeleon.EncodeCtxt(c = (c_1,c_2)):
   u = Decompress_du(c_1)
   for i from 1 to k*n:
      u[i] = SamplePreimage(du,u[i],c_1[i])
   v = Decompress_dv(c_2)
   for i from 1 to n:
      v[i] = SamplePreimage(dv,v[i],c_2[i])
   for i in range(k)
      r_i = VectorEncode(u[i])
   r_(k+1) = VectorEncode(v)
   r = concat(r_0,...,r_(k+1))
   return r

Kemeleon.DecodeCtxt(r):
   r_0,...,r_(k+1) = r # each r_i is fixed length
   for i in range(k):
      u[i] = VectorDecode(r_i)
   v = VectorDecode(r_(k+1))
   c_1 = Compress_du(u)
   c_2 = Compress_dv(v)
   return (c_1,c_2)
```

4.4. Summary of Properties

Algorithm / Parameter	Output size (bytes)	Success probability
Kemeleon - ML-KEM512	ek: 814, ctxt: 1172	ek: 1.00, ctxt: 1.00
Kemeleon - ML-KEM768	ek: 1204, ctxt: 1562	ek: 1.00, ctxt: 1.00
Kemeleon - ML-KEM1024	ek: 1594, ctxt: 1953	ek: 1.00, ctxt: 1.00

Table 1: Summary of Kemeleon Properties

5. Additional Considerations for Applications

This section contains additional considerations and comments related to using Kemeleon encodings in different applications.

5.1. Smaller Outputs from Rejection Sampling

In applications willing to incur some probability of failure in encoding, a variant of the encoding algorithm that does not add the additional m value can be used. This results in smaller output sizes for public keys and ciphertexts. However, in this case it is no longer feasible to parallelize the encoding of the k polynomials; these must be treated as a single vector of k*n coefficients in order to achieve a reasonable rate of rejection. Therefore, this approach also requires arithmetic over larger integers (up to 1872B integers for ML-KEM1024). In particular, the following algorithms can be used instead of VectorEncode and VectorDecode above.

```
VectorEncode(a,k):
   r = 0
   for i from 1 to k*n:
      r += q^(i-1)*a[i]
   if msb(r) == 1:
      return err
   else:
      return r
```

```
VectorDecode(r,k):
   for i from 1 to k*n:
      a[i] = r % q
      r = r // q
   return a
```

The encoding algorithms for public keys should handle errors accordingly, returning an error if VectorEncode returns an error. For ciphertexts, the second ciphertext component need not be decompressed, and rejection sampling can be used to retain uniformity instead.

```
Kemeleon.EncodeCtxt(c = (c_1,c_2)):
   u = Decompress_du(c_1)
   for i from 1 to k*n:
      u[i] = SamplePreimage(du,u[i],c_1[i])
   r = VectorEncode(u)
   if r == err:
      return err
   for i from 1 to n:
      if c_2[1] == 0:
         return err with prob. 1/ceil(q/(2^dv))
   return concat(r,c_2)

Kemeleon.DecodeCtxt(ec):
   r,c_2 = ec # c_2 is fixed length
   u = VectorDecode(r)
   c_1 = Compress_du(u)
   return (c_1,c_2)
```

This variant of the encoding is as described in the original work [GSV24], and has the following properties.

Algorithm / Parameter	Output size (bytes)	Success probability	Additional considerations
Kemeleon - ML-KEM512	ek: 781, ctxt: 877	ek: 0.56, ctxt: 0.51	Large int (750B) arithmetic
Kemeleon - ML-KEM768	ek: 1156, ctxt: 1252	ek: 0.83, ctxt: 0.77	Large int (1150B) arithmetic
Kemeleon - ML-KEM1024	ek: 1530, ctxt: 1658	ek: 0.62, ctxt: 0.57	Large int (1500B) arithmetic

Table 2: Summary of Alternate Encoding Properties

5.2. Deterministic Encoding

The randomness used in Kemeleon ciphertext encodings MAY be derived in a deterministic manner. To do so, following a call to Encap which returns a KEM key K and a ciphertext c, the following steps can be taken:

* Using a key derivation function (KDF), derive from the key K a new key K’ and a seed for randomness rnd.

* The seed rnd can be used to generate the randomness required when encoding the ciphertext c.

* Use K’ in place of K wherever applicable in the remainder of the protocol/system.

* Upon any call to Decap, apply the same KDF to derive the new key K’, as required.

Deriving a new KEM key for use in the remainder of a system is crucial in order to ensure key separation (i.e., the implementation MUST NOT use the original key K to derive randomness and for other purposes).

The randomness used to encode an encapsulation key MAY be stored alongside the corresponding decapsulation key, if it is subsequently needed. See Section 6.2 for relevant discussion on keeping this randomness secret.

5.3. Relation to Hash-to-Curve

While the functionality of Kemeleon is similar to hash-to-curve [RFC9380] (mapping arbitrary byte strings to public keys/ciphertexts), the applications where hash-to-curve is used do not immediately follow in the KEM-based setting because having such an encapsulation key (without dk) or ciphertext (without dk or ek) does not appear to provide the same functionality, since it is not clear how to continue working with the element in the same way that can be done with an elliptic curve point.

5.4. Modifying ML-KEM Algorithms

In applications that _only_ require Kemeleon-encoded values _and_ where the underlying ML-KEM implementation can be modified, the ciphertext encoding algorithm (and ML-KEM encapsulation/decapsulation algorithms) MAY be adapted as follows for improved efficiency. In particular, the compression step in the ML-KEM encapsulation algorithm can be omitted, and therefore, the decompression step in

the Kemeleon algorithm can be omitted. In the implementation of ML-KEM, the compression step (lines 22-23 of Algorithm 14 [FIPS203]) and corresponding decompression step (lines 3-4 of Algorithm 15 [FIPS203]) can be omitted from the encapsulation/decapsulation algorithms in ML-KEM. In this case, the Kemeleon encoding algorithm for ciphertexts would omit the Decompress and SamplePreimage steps and immediately apply VectorEncode:

```
Kemeleon.EncodeCtxt(c = (c_1,c_2)):
   w = [c_1,c_2] # treat c_1,c_2 as a singular vector of (k+1)*n coefficients
   r = VectorEncode(w,k+1)
   return r
```

Decoding is adapted analogously.

```
Kemeleon.DecodeCtxt(ec):
   w = VectorDecode(r,k+1)
   c_1,c_2 = w # c_1, c_2 are fixed length
   return (c_1,c_2)
```

6. Security Considerations

This section contains additional security considerations about the Kemeleon encodings described in this document.

6.1. Computational Assumptions

In general, the obfuscation properties of the Kemeleon encodings depend on module LWE assumptions similar to those underlying the IND-CCA security of ML-KEM; see [GSV24] for the detailed security analysis of the original Kemeleon encoding. In particular, the notions of public key and ciphertext uniformity capture the indistinguishability of Kemeleon-encoded encapsulation keys and ciphertexts from random bitstrings, respectively. Both require the module LWE assumption to hold in order for Kemeleon to maintain its uniformity properties. Furthermore, distinguishing a pair of a Kemeleon-encoded encapsulation key and a Kemeleon-encoded ciphertext from uniformly random bitstrings also reduces to a module LWE assumption.

6.2. Randomness Sampling

Both encapsulation key and ciphertext encodings in the Kemeleon encoding are randomized. The randomness (or seed used to generate randomness) used in Kemeleon encodings MUST be kept secret. In particular, public randomness enables distinguishing a Kemeleon-encoded value from a random bytestring: Decoding the value in question and re-encoding it with the public randomness will yield the

original value if it was Kemeleon-encoded.

6.3. Timing Side-Channels

Beyond timing side-channel considerations for ML-KEM itself, care should be taken when using Kemeleon encodings. Algorithms required to perform large integer arithmetic may leak information via timing. Additionally, rejecting and re-generating encapsulation keys or ciphertexts may leak information about the use of Kemeleon encodings, as might the overhead of the encoding itself.

7. IANA Considerations

This document has no IANA actions.

8. References

8.1. Normative References

[ELL2] Tibouchi, M., "Elligator Squared: Uniform Points on Elliptic Curves of Prime Order as Uniform Random Strings", 2014, <https://eprint.iacr.org/2014/043>.

[FIPS203] "Module-lattice-based key-encapsulation mechanism standard", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.203, August 2024, <https://doi.org/10.6028/nist.fips.203>.

[GSV24] G端nther, F., Stebila, D., and S. Veitch, "Obfuscated Key Exchange", 2024, <https://eprint.iacr.org/2024/1086>.

[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>.

[RFC9380] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", RFC 9380, DOI 10.17487/RFC9380, August 2023, <https://www.rfc-editor.org/rfc/rfc9380>.

8.2. Informative References

[CPACE] Abdalla, M., Haase, B., and J. Hesse, "CPace, a balanced composable PAKE", Work in Progress, Internet-Draft, draft-irtf-cfrg-cpace-17, 18 December 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-cpace-17>.

[ECDH-PSI] Wang, Y., ChangWenting, Lu, Y., Hong, C., and J. Peng, "PSI based on ECDH", Work in Progress, Internet-Draft, draft-ecdh-psi-00, 21 October 2024, <https://datatracker.ietf.org/doc/html/draft-ecdh-psi-00>.

[OBFS4] "obfs4 (The obfourscator)", n.d., <https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/lyrebird/-/blob/HEAD/doc/obfs4-spec.txt>.

[OPAQUE] Bourdrez, D., Krawczyk, H., Lewi, K., and C. A. Wood, "The OPAQUE Augmented PAKE Protocol", Work in Progress, Internet-Draft, draft-irtf-cfrg-opaque-18, 21 November 2024, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque-18>.

Acknowledgments

Thanks to Michael Rosenberg, John Mattsson, and Stanislaw Jarecki for contributions to this document and helpful discussions.

Authors' Addresses

Felix G端nther
IBM Research Europe - Zurich
Email: mail@felixguenther.info

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

Shannon Veitch
ETH Zurich
Email: shannon.veitch@inf.ethz.ch