

Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: 13 October 2025

F. G端nther  
IBM Research Europe - Zurich  
D. Stebila  
University of Waterloo  
S. Veitch  
ETH Zurich  
11 April 2025

Kameleon Encodings  
draft-irtf-cfrg-kameleon-00

## Abstract

This document specifies Kameleon encoding algorithms for encoding ML-KEM encapsulation keys and ciphertexts as random bytestrings. Kameleon encodings provide obfuscation of encapsulation keys and ciphertexts, relying on module LWE assumptions. This document specifies a number of variants of these encodings, with differing failure rates, output sizes, and performance profiles.

## 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-kameleon/draft-irtf-cfrg-kameleon.html>. Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-irtf-cfrg-kameleon/>.

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

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 13 October 2025.

## Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

## Table of Contents

1. Introduction . . . . .	2
2. Conventions and Definitions . . . . .	3
3. Notation / ML-KEM Background . . . . .	3
4. Kemeleon Encoding . . . . .	4
4.1. Common Functions . . . . .	4
4.2. Encoding Encapsulation Keys . . . . .	6
4.3. Encoding Ciphertexts . . . . .	7
4.4. Non-Rejection Sampling Variant . . . . .	8
4.5. Faster Arithmetic Variant . . . . .	9
4.6. Summary of Encodings . . . . .	10
5. Additional Considerations for Applications . . . . .	10
5.1. Deterministic Encoding . . . . .	10
5.2. Relation to Hash-to-Curve . . . . .	11
5.3. Modifying ML-KEM Algorithms . . . . .	11
6. Security Considerations . . . . .	12
6.1. Computational Assumptions . . . . .	12
6.2. Randomness Sampling . . . . .	12
6.3. Timing Side-Channels . . . . .	13
7. IANA Considerations . . . . .	13
8. References . . . . .	13
8.1. Normative References . . . . .	13
8.2. Informative References . . . . .	13
Authors' Addresses . . . . .	14

## 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. Beyond the original construction, this document additionally specifies variants that avoid the encoding failing or the use of large integer computations, or allow for a deterministic encoding. Aside from these variants, it is notable that the Kemeleon encodings of encapsulation keys results in smaller representations than in the original ML-KEM specification.

## 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((2d/q)*x) mod 2d` (Equation 4.7)
- \* `Decompress_d : y -> round((q/2d)*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  $\mathbb{Z}_q$ , 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 coefficients, where each coefficient in  $\mathbb{Z}_q$  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:

- \*  $msb(x)$  refers to the most significant bit of the value  $x$
- \*  $a[i]$  denotes the  $i$ th position of a vector  $a$  of coefficients
- \*  $concat(x_0, \dots, x_N)$ : returns the concatenation of bytestrings.

#### 4. Kemeleon Encoding

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

- \*  $EncodePk(ek) \rightarrow eek$  is the (possibly randomized) encoding algorithm that on input an encapsulation key, outputs an obfuscated encapsulation key or an error.
- \*  $DecodePk(eek) \rightarrow ek$  is the deterministic decoding algorithm that on input an obfuscated encapsulation key, outputs an encapsulation key.
- \*  $EncodeCtxt(c) \rightarrow ec$  is the (possibly randomized) encoding algorithm that on input a ciphertext, outputs an obfuscated ciphertext or an error.
- \*  $DecodeCtxt(ec) \rightarrow c$  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  $k \cdot n$  coefficients modulo  $q$  to a large integer, rejecting if the most significant bit of the integer is 1.

```
VectorEncode(a):  
    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):  
    for i from 1 to k*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  $c$ . 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:
      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 c == 0:
      rand <--$ [-52,...,52]
    else:
      rand <--$ [-51,...,52]
    return u + rand
  if d == 4:
    if c == 0:
      rand <--$ [-104,...,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, but in the following pseudocode  $t$  is treated as a vector of  $k*n$  coefficients.

```

Kemeleon.EncodePk(ek = (t, rho)):
  r = VectorEncode(t)
  if r == err:
    return err
  else:
    return concat(r,rho)

```

```

Kemeleon.DecodePk(eek):
    r,rho = eek # rho is fixed length
    t = VectorDecode(r)
    return (t, rho)

```

While the encapsulation key encoding algorithm has some failure probability, the decoding algorithm can never fail. Otherwise, a failure in decoding would provide a distinguishing factor between Kemeleon-encoded values and random bitstrings.

#### 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 is applied. For the second ciphertext component, rejection sampling is performed to retain uniformity, rather than decompressing.

```

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[i] == 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)

```

#### 4.4. Non-Rejection Sampling Variant

Applying a technique from [ELL2] (Section 3.4), the original Kemeleon construction can be adapted to avoid rejection sampling. This results in larger output sizes, but the encoding algorithm never fails. Applying the technique from [ELL2], where  $r$  is the encoded vector before rejection occurs in `VectorEncode`, we then choose  $m$  at random from  $[0, \text{floor}((2^{(b+t)} - r) / (q^{(k*n)}))]$ , where  $b = \log_2(q^{(k*n)})$  and  $t$  is a security parameter, and return  $r + m * q^{(k*n)}$ . This variant results in encoded values whose statistical distance from uniform is at most  $2^{-t}$ . This results in an increased output size of  $t$  bits, where  $t$  is the security parameter. For example, with  $t=128$ , this increases the output size by 16 bytes.

For encapsulation key encodings, one can immediately replace `VectorEncode` and `VectorDecode` calls with calls to the following algorithms.

```
VectorEncodeNR(a):
  r = 0
  t = sec_param # e.g. t = 128, 256, ...
  b = log_2(q^(k*n))
  for i from 1 to k*n:
    r += q^(i-1)*a[i]
  m <--$ [0,...,floor((2^(b+t)-r)/(q^(k*n)))]
  return r + m*q^(k*n)

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

Notably, the random value  $m$  need not be transmitted alongside the encoded values.

For ciphertext encodings, one must also avoid rejection sampling based on coefficients of the second component of the ciphertext. Therefore, the new ciphertext encoding must decompress and `VectorEncodeNR` the second component of the ciphertext. This more significantly increases the size of the encoded ciphertext.

```

Kemeleon.EncodeCtxtNR(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])
  w = [u,v] # treat u,v as a singular vector of (k+1)*n coefficients
  r = VectorEncodeNR(w) # this call should use k+1 rather than k when accumulating to a
large integer
  return r

Kemeleon.DecodeCtxtNR(r):
  w = VectorDecodeNR(r)
  u,v = w # u, v are fixed length
  c_1 = Compress_du(u)
  c_2 = Compress_dv(v)
  return (c_1,c_2)

```

#### 4.5. Faster Arithmetic Variant

OPEN ISSUE: Is the faster variant of interest? If so, the following can be extended with a complete description.

Observing that  $q = 3329 = 13 \cdot 2^8 + 1$ , a variant of Kemeleon with faster integer arithmetic can be specified. First, the encoding rejects any polynomial with a coefficient equal to  $q-1 = 3328$ . This ensures that all arithmetic can be computed with values modulo  $q-1 = 13 \cdot 2^8$ . Then, note that rather than accumulating values to a large integer mod  $q^{(k \cdot n)}$ , it is only required to accumulate values to an integer mod  $13^{(k \cdot n)}$ , while keeping track of the 8 lower order bits of each coefficient. The output size of the encoding does not change, but this results in an increased rejection rate.

In particular, Table 1 gives success probabilities for encapsulation key and ciphertext encodings:

Parameter	ek success probability	ctxt success probability
ML-KEM-512	0.49	0.45
ML-KEM-768	0.29	0.25
ML-KEM-1024	0.53	0.47

Table 1: Success probabilities for faster Kemeleon encoding

#### 4.6. Summary of Encodings

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
KemeleonNR - ML-KEM512	ek: 797, ctxt: 1140	ek: 1.00, ctxt: 1.00	Large int (1123B) arithmetic
KemeleonNR - ML-KEM768	ek: 1172, ctxt: 1514	ek: 1.00, ctxt: 1.00	Large int (1498B) arithmetic
KemeleonNR - ML-KEM1024	ek: 1546, ctxt: 1889	ek: 1.00, ctxt: 1.00	Large int (1872B) arithmetic
KemeleonFT - ML-KEM512	ek: 781, ctxt: 877	ek: 0.49, ctxt: 0.45	Smaller int (235B) arithmetic
KemeleonFT - ML-KEM768	ek: 1156, ctxt: 1252	ek: 0.29, ctxt: 0.25	Smaller int (355B) arithmetic
KemeleonFT - ML-KEM1024	ek: 1530, ctxt: 1658	ek: 0.53, ctxt: 0.47	Smaller int (475B) arithmetic

Table 2: Summary of Kemeleon Variants, NR = No Reject, FT = Faster

#### 5. Additional Considerations for Applications

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

##### 5.1. 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  $\text{rnd}$ .
- \* The seed  $\text{rnd}$  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.

## 5.2. 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  $\text{dk}$ ) or ciphertext (without  $\text{dk}$  or  $\text{ek}$ ) 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.

## 5.3. 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) # this call should use k+1 rather than k when accumulating to a large integer  
    return r
```

Decoding is adapted analogously.

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

Naturally, these algorithms can use `VectorEncodeNR`, `VectorDecodeNR`, if the non-rejecting variant is desirable.

## 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 original 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, in particular those with a non-zero failure probability. 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. Additionally, the algorithms required to perform big integer arithmetic may leak information via timing.

## 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-13, 14 October 2024, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-cpace-13>>.
- [ECDH-PSI] wangyuchen, Chang, 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-transport/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>>.

## Authors' Addresses

Felix G<sup>端</sup>nther  
IBM Research Europe - Zurich  
Email: [mail@felixguenther.info](mailto:mail@felixguenther.info)

Douglas Stebila  
University of Waterloo  
Email: [dstebila@uwaterloo.ca](mailto:dstebila@uwaterloo.ca)

Shannon Veitch  
ETH Zurich  
Email: [shannon.veitch@inf.ethz.ch](mailto:shannon.veitch@inf.ethz.ch)