ETSI TR 104 239-2 V1.1.1 (2026-06)

TECHNICAL REPORT

Cyber Security (CYBER);
Quantum-Safe Cryptography (QSC);
Secure Implementation Guidance for
Key Encapsulation Mechanisms and
Digital Signature Schemes;
Part 2: ML-KEM

Reference
DTR/CYBER-QSC-0027-2

Keywords
cyber security, key exchange,
quantum safe cryptography

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI IPR online database.

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Technical Report (TR) has been produced by ETSI Technical Committee Cyber Security (CYBER).

The present document is part 2 of a multi-part deliverable. Full details of the entire series can be found in part 1 [i.3].

The present document provides implementation guidance for the Module-Lattice-based Key Encapsulation Mechanism (ML-KEM).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document provides developers with guidance to aid the secure implementation of the quantum-safe key encapsulation mechanism ML-KEM as specified by NIST FIPS 203 [i.5]. This highlights some potential ML-KEM implementation hazards; identifies some side-channel and fault attack issues specific to ML-KEM; considers some testing and formal verification aspects relevant to ML-KEM; and briefly discusses the secure usage of ML-KEM.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding, but are not required for conformance to the present document.

[i.1] ETSI TS 103 744 (V1.2.2): "CYBER; Quantum-Safe Cryptography (QSC); Quantum-safe Hybrid Key Establishment".

[i.2] ETSI TR 103 966: "CYBER Security (CYBER); Quantum-Safe Cryptography (QSC); Deployment Considerations for Hybrid Schemes".

[i.3] ETSI TR 104 239-1: "Cyber Security (CYBER); Quantum-Safe Cryptography (QSC); Secure Implementation Guidance for Key Encapsulation Mechanisms and Digital Signature Schemes; Part 1: General".

[i.4] IETF RFC 9935: "Internet X.509 Public Key Infrastructure - Algorithm Identifiers for the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)".

[i.5] NIST FIPS 203: "Module-Lattice-Based Key-Encapsulation Mechanism Standard".

[i.6] NIST SP 800-133 Rev. 2: "Recommendation for Cryptographic Key Generation".

[i.7] NIST SP 800-227: "Recommendations for Key-Encapsulation Mechanisms".

[i.8] NIST: "Automated Cryptographic Validation Test System".

[i.9] NIST: "Implementation Guidance for FIPS 140-3 and the Cryptographic Module Validation Program".

[i.10] C. Aguilar Melchor et al.: "Hamming Quasi-Cyclic (HQC): Fourth round version". NIST Post-Quantum Cryptography Standardization Process.

[i.11] J.B. Almeida et al.: "Formally verifying Kyber Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt". CRYPTO 2024.

[i.12] R. Avanzi et al.: "CRYSTALS-Kyber: Algorithm specification and supporting documentation (version 3.02)". NIST Post-Quantum Cryptography Standardization Process.

[i.13] D.J. Bernstein et al.: "KyberSlash: Exploiting secret-dependent division timings in Kyber implementations". CHES 2025.

[i.14] J. Bos et al.: "CRYSTALS-Kyber: A CCA-secure module-lattice-based KEM". Euro S&P 2018.

[i.15] J. Bos et al.: "Masking Kyber: First- and higher-order implementations". CHES 2021.

[i.16] BSI TR-02102-1: "Cryptographic mechanisms: Recommendations and key lengths", January 2026.

[i.17] Community Cryptography Specification Project: "Project Wycheproof".

[i.18] C. Cremers, A. Dax and N. Medinger: "Keeping up with the KEMs: Stronger security notions for KEMs and automated analysis of KEM-based protocols". CCS 2024.

[i.19] Cybersecurity & Infrastructure Security Agency, US Government: "Secure-by-Design - Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software".

[i.20] Department of Science, Innovation and Technology, UK Government: "Software Security Code of Practice".

[i.21] B. Gierlichs et al.: "Mutual information analysis". CHES 2008.

[i.22] C. Glénaz et al.: "Finding bugs in implementations of HQC, the fifth post-quantum standard".

[i.23] Q. Guo, T. Johansson and A. Nilsson: "A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM". CRYPTO 2020.

[i.24] J. Hermelink, P. Pessl and T. Pöppelmann: "Fault-enabled chosen-ciphertext attacks on Kyber". INDOCRYPT 2021.

[i.25] K. Hövelmanns and M. Kudinov: "Treating dishonest ciphertexts in post-quantum KEMs - Explicit vs. implicit rejection in the FO transform". PQCrypto 2025.

[i.26] M.J. Kannwischer, P. Pessl and R. Primas: "Single-trace attacks on Keccak". CHES 2020.

[i.27] E. Karatsiolis et al.: "Public Key Linting for ML-KEM and ML-DSA". ACNS 2025.

[i.28] S. Nkotto: "Template and CPA side channel attacks on the Kyber/ML-KEM pair-pointwise multiplication". IACR ePrint Archive 2025/1577.

[i.29] Post-Quantum Cryptography Alliance: "mlkem-native".

[i.30] P. Ravi et al.: "Number 'not used' once - Practical fault attack on pqm4 implementations of NIST candidates". COSADE 2019.

[i.31] P. Ravi et al.: "On configurable SCA countermeasures against single trace attacks for the NTT". SPACE 2020.

[i.32] S. Schmieg: "Unbindable Kemmy Schmidt: ML-KEM is neither MAL-BIND-K-CT nor MAL-BIND-K-PK". IACR ePrint Archive 2024/523.

[i.33] Symbolic Software: "Crucible".

[i.34] T. Tosun, E. Oswald and E. Savaş: "Non-profiled higher-order side-channel attacks against lattice-based post-quantum cryptography". IACR Communications in Cryptology, 2025.

3 Definition of terms, symbols and abbreviations

3.1 Terms

Void.

3.2 Symbols

For the purposes of the present document, the following symbols apply:

$a[i:j]$	Sub-string a_i ... a_{j-1} of the byte string $a := a_0 \ldots a_{n-1}$
(a, b)	Ordered pair of values a and b
$a \mid\mid b$	Concatenation of the byte strings a and b
$a = b$	Return true if the values a and b are equal and false otherwise
$a \leftarrow b$	Assign the value b to the variable a

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AES	Advanced Encryption Standard
CBD	Centred Binomial Distribution
CVE	Common Vulnerabilities and Exposures
DPA	Differential Power Analysis
ECDH	Elliptic Curve Diffie-Hellman
FIPS	Federal Information Processing Standard
FO	Fujisaki-Okamoto
HMAC	Hash-based Message Authentication Code
HQC	Hamming Quasi-Cyclic
IND-CCA	Indistinguishability against Chosen Ciphertext Attacks
IND-CPA	Indistinguishability against Chosen Plaintext Attacks
KDF	Key Derivation Function
KEM	Key Encapsulation Mechanism
LWE	Learning With Errors
ML-DSA	Module-Lattice-based Digital Signature Algorithm
ML-KEM	Module-Lattice-based Key Encapsulation Mechanism
NIST	National Institute of Standards and Technology
NTT	Number Theoretic Transform
PKE	Public Key Encryption
PKE	Public-Key Encryption
PQC	Post-Quantum Cryptography
RBG	Random Bit Generator
SHA	Secure Hash Algorithm
SHAKE	Secure Hash Algorithm Keccak
SLH-DSA	Stateless Hash-based Digital Signature Algorithm
SP	Special Publication
TLS	Transport Layer Security

4 Introduction

The Module-Lattice-based Key Encapsulation Mechanism (ML-KEM) is a post-quantum Key Encapsulation Mechanism (KEM) selected for standardization following the multi-year NIST Post-Quantum Cryptography (PQC) Standardization Process. It is specified in NIST FIPS 203 [i.5] and support for ML-KEM is being integrated into internet protocols such as TLS.

NOTE: ML-KEM is based on the CRYSTALS-Kyber submission to the NIST PQC Process [i.12], but changes made during the development of NIST FIPS 203 [i.5] mean that it is not interoperable with Kyber.

ML-KEM received a significant amount of analysis during the NIST PQC Process, but its security also depends on the way in which it is implemented and used. The present document builds on the general guidance in ETSI TR 104 239-1 [i.3] to provide developers with some specific guidance to aid the secure implementation of ML-KEM:

- It describes the ML-KEM interfaces and different options for private key formats (clause 5).

- It extends the general implementation considerations from ETSI TR 104 239-1 [i.3] with ML-KEM specific aspects such as input validation, consistency checks, and error handling (clause 6).
- It discusses different side-channel considerations for ML-KEM key generation, encapsulation and decapsulation (clause 7).
- It touches on testing and formal validation for ML-KEM implementations (clause 8).

The present document also gives some background on ML-KEM security properties (annex B) and some considerations on the use of ML-KEM in hybrid schemes (annex C).

As with ETSI TR 104 239-1 [i.3], the guidance is primarily aimed at secure implementations in software, although some of the recommendations will also be relevant to hardware implementations. Developers implementing ML-KEM should also ensure that they follow best practice for developing secure systems in general [i.19] and [i.20].

5 ML-KEM interfaces

5.1 Parameters

NIST FIPS 203 [i.5] specifies three approved parameter sets for ML-KEM:

- **ML-KEM-512** targets security category 1, which is equivalent to key recovery for AES-128;
- **ML-KEM-768** targets security category 3, which is equivalent to key recovery for AES-192; and
- **ML-KEM-1024** targets security category 5, which is equivalent to key recovery for AES-256.

ML-KEM-768 is recommended as the default parameter set. ML-KEM-512 can be used in applications that require smaller public keys or ciphertexts. ML-KEM-1024 can be used in applications that require a higher level of security.

Table 1 lists the sizes of the ML-KEM private decapsulation keys, public encapsulation keys and ciphertexts.

Table 1: ML-KEM private key, public key, ciphertext and shared secret lengths (in bytes)

Parameter set	Private decapsulation key	Public encapsulation key	Ciphertext	Shared secret
ML-KEM-512	1 632	800	768	32
ML-KEM-768	2 400	1 184	1 088	32
ML-KEM-1024	3 168	1 568	1 568	32

5.2 Key generation

5.2.1 Full key generation

The specification of ML-KEM key generation in NIST FIPS 203 [i.5], algorithm 19 samples a pair of 32-byte values and deterministically generates the key pair from those values via an internal key generation function (see table 2).

Table 2: ML-KEM key generation interface

ML-KEM.KeyGen()		
Input:	None	
Output:	Public encapsulation key $\boldsymbol{pk}$ and private decapsulation key $\boldsymbol{sk}$, or Error	
1.	$d \leftarrow$ RBG(32)	
2.	$z \leftarrow$ RBG(32)	
3.	if d = NULL or z = NULL then	
4.	return Error	(RBG failure)
5.	end if	
6.	$(pk, sk) \leftarrow$ ML-KEM.KeyGen_internal(d, z)	(Generate key pair from private seed)
7.	return (pk, sk)	

5.2.2 Key formats

The ML-KEM public encapsulation key pk is a byte array that encodes a pair of values $(\hat{t}, \rho)$.

The ML-KEM private decapsulation key sk is a byte array that encodes a 4-tuple of values $(\hat{s}, pk, h, z)$.

NOTE 1: The private decapsulation key sk contains a copy of the public encapsulation key pk as it is needed for the re-encryption step of decapsulation (see clause 6.2.1) and it would not otherwise be possible to re-derive pk from sk.

NOTE 2: The private decapsulation key sk separately contains the SHA3-256 hash value h of the public encapsulation key pk. This is needed when deriving the shared secret and the randomness used in the re-encryption step of decapsulation. Although the hash can be computed from the copy of pk contained in sk, including h avoids the cost of the hash computation.

NOTE 3: If an adversary can change the value of h contained in the private decapsulation key, then they could potentially construct two different ciphertexts that decapsulate to the same shared secret for different private keys [i.32]. To avoid issues with protocols that assume a binding between the ciphertext and shared secret, private key validation includes a check that $h = \text{SHA3-256}(pk)$ (see clauses 6.1.2 and B.2.2).

NOTE 4: The private decapsulation key sk includes a value z that is used to derive a pseudo-random shared secret from the ciphertext when the re-encryption check fails during decapsulation (see clause 6.2.1).

NOTE 5: If an adversary can change the value of z contained in the private decapsulation key, then they could potentially construct a single ciphertext that decapsulates to the same shared secret for two private keys corresponding to different public keys [i.32]. This binding failure cannot be mitigated through private key validation (see clause B.2.2).

5.2.3 Seed format key generation

ML-KEM private decapsulation keys are significantly larger than the public encapsulation keys or ciphertexts. Large private keys could be problematic for applications where secure key storage is limited or where there are many private keys.

Further, ML-KEM private decapsulation keys require private key validation before use to prevent certain binding attacks (see clauses 6.1.2 and B.2.2).

As a consequence of this, NIST FIPS 203 [i.5], section 3.3 explicitly allows a 64-byte private seed to be stored and used in place of the full private decapsulation key, provided that it is given the same level of protection as the private key.

The change to key generation is minimal (see table 3).

Table 3: ML-KEM seed format key generation interface

ML-KEM.SeedKeyGen()		
Input:	None	
Output:	Public encapsulation key $\boldsymbol{pk}$ and private seed $\boldsymbol{seed}$, or Error	
1.	$d \leftarrow$ RBG(32)	
2.	$z \leftarrow$ RBG(32)	
3.	if d = NULL or z = NULL then	
4.	return Error	(RBG failure)
5.	end if	
6.	$(pk, \_) \leftarrow$ ML-KEM.KeyGen_internal(d, z)	(Expand public key from private seed)
7.	$seed \leftarrow (d \parallel z)$	
8.	return $(pk, seed)$	

It is possible to recompute the private decapsulation key from the private seed via private key expansion (see table 4).

Table 4: ML-KEM private key expansion

ML-KEM.ExpandSK($seed$)		
Input:	Private seed $\boldsymbol{seed}$	
Output:	Private key $\boldsymbol{sk}$	
1.	$d \leftarrow seed[0 : 32]$	
2.	$z \leftarrow seed[32 : 64]$	
3.	$(\_, sk) \leftarrow$ ML-KEM.KeyGen_internal(d, z)	(Expand private key from private seed)
4.	return sk	

However, it is generally not possible to recover the private seed from the private decapsulation key. Some protocols therefore allow the use of the private decapsulation key, the private seed, or both [i.4].

5.3 Encapsulation

The specification of ML-KEM encapsulation in NIST FIPS 203 [i.5], algorithm 20 samples a 32-byte value m and deterministically generates a shared secret and a ciphertext that encapsulate the shared secret from m and the public encapsulation key via an internal encapsulation function (see table 5).

Table 5: ML-KEM encapsulation interface

ML-KEM.Encaps(pk)		
Input:	Public encapsulation key $\boldsymbol{pk}$.	
Output:	Shared secret $\boldsymbol{K}$ and ciphertext $\boldsymbol{c}$, or Error	
1.	$m \leftarrow$ RBG(32)	
2.	if m = NULL then	
3.	return Error	(RBG failure)
4.	end if	
5.	$(K, c) \leftarrow$ ML-KEM.Encaps_internal(pk, m)	(Encapsulate using random seed)
6.	return (K, c)	

5.4 Decapsulation

The specification of ML-KEM decapsulation in NIST FIPS 203 [i.5], algorithm 21 simply calls an internal decapsulation function (see table 6).

Table 6: ML-KEM decapsulation interface

ML-KEM.Decaps(sk, c)	
Input:	Private decapsulation key $\boldsymbol{sk}$ and ciphertext $\boldsymbol{c}$.
Output:	Shared secret $\boldsymbol{K}$.
1.	$K \leftarrow$ ML-KEM.Decaps_internal(sk, c)
2.	return K

When a private seed is used in place of the private decapsulation key, the only change to the decapsulation routine is that it needs to expand the private seed to recover the private key before calling the internal decapsulation function (see table 7).

Table 7: ML-KEM seed format decapsulation interface

ML-KEM.Decaps($seed$, c)	
Input:	Private seed $\boldsymbol{seed}$ and ciphertext $\boldsymbol{c}$.
Output:	Shared secret $\boldsymbol{K}$.
1.	$sk \leftarrow$ ML-KEM.ExpandSK($seed$)
2.	$K \leftarrow$ ML-KEM.Decaps_internal(sk, c)
3.	return K

6 General implementation considerations

6.1 Perform input validation

6.1.1 Encapsulation inputs

NIST FIPS 203 [i.5] requires validation of the public encapsulation key before it can be used in ML-KEM encapsulation (see section 7.2). Validation of the public encapsulation key involves checking that it is a byte array of the expected length and that the encoding of $\hat{t}$ corresponds to a sequence of integers in the expected range.

EXAMPLE 1: The hash of the public key is used directly in the derivation of the shared secret (see table 8). If the public key has not been encoded correctly, then the shared secret returned by encapsulation will also be incorrect.

NOTE 1: These checks do not guarantee that the public encapsulation key is valid output from ML-KEM key generation. Full validation of the public key is not possible without access to the private seed used to generate it.

Validation of the public encapsulation key is not necessarily required every time it is used. There are some cases where input validation might not be needed (see [i.7], section 3.2).

EXAMPLE 2: If a public encapsulation key has previously been validated and it has subsequently been stored in a way that prevents modification, then it does not need to be validated again.

EXAMPLE 3: If a public encapsulation key has been provided by a trusted third party with assurances that it is valid and it has subsequently been stored in a way that prevents modification, then it does not need to be validated.

NOTE 2: If the implementation supports different ML-KEM parameter sets, then checking the length of the public encapsulation key can prevent the key from being used with the wrong parameter set.

6.1.2 Decapsulation inputs

NIST FIPS 203 [i.5] requires validation of the private decapsulation key and ciphertext before they can be used in ML-KEM decapsulation (see section 7.3). Validation of the private decapsulation key involves checking that it is a byte array of the expected length and that $h = \text{SHA3-256}(pk)$ for the hash value h and copy of the public encapsulation key pk encoded in the private key.

EXAMPLE 1: The hash value h is used in decapsulation to bind the shared secret to the public encapsulation key. Failure to check that $h = \text{SHA3-256}(pk)$ can break this binding (see clauses 5.2.2 and B.2.2).

NOTE 1: These checks do not guarantee that the private decapsulation key is valid output from ML-KEM key generation. Full validation of the private key is not possible without access to the private seed used to generate it.

NOTE 2: Additional checks can be performed on the private decapsulation key; for example, validating the copy of the public encapsulation key contained in the private key. However, these checks are not required by NIST FIPS 203 [i.5]. Any issues would likely be identified by a pairwise consistency test, provided that the public encapsulation key is also accessible (see clause 6.2.2).

Validation of the private decapsulation key is not necessarily required every time it is used. There are some cases where input validation might not be needed (see [i.7], section 3.2).

EXAMPLE 2: An ephemeral private decapsulation key generated by the implementation and used immediately does not need to be validated.

EXAMPLE 3: A static private decapsulation key generated by the implementation and subsequently stored in a way that prevents modification does not need to be validated.

NOTE 3: If the implementation supports different ML-KEM parameter sets, then checking the length of the private decapsulation key can prevent the key from being used with the wrong parameter set.

EXAMPLE 4: The private decapsulation key contains a copy of the corresponding public encapsulation key, pk, its public hash value, h, and a private value z used for implicit rejection (see clause 6.2.1). If the private key was used with a smaller parameter set than intended, an implementation could take z from part of the public data in the key and implicit rejection would lead to predictable shared secrets.

Validation of the ciphertext only involves checking that it is a byte array of the expected length. This is required every time the ciphertext is used in ML-KEM decapsulation (see [i.5], section 7.3). The ciphertext will generally have been received by the implementation from another source to be decapsulated, and hence the examples above do not apply.

EXAMPLE 5: An implementation that accepted a longer-than-expected ciphertext and ignored the extra bytes instead of returning an error could potentially allow an adversary to take a valid ciphertext and create a second ciphertext that decapsulates to the same shared secret. This violates the IND-CCA security of ML-KEM (see clause B.1.2)

EXAMPLE 6: An implementation that accepted a shorter-than-expected ciphertext and attempted to perform decapsulation using this ciphertext instead of returning an error could potentially leak information from memory.

NOTE 4: ML-KEM decapsulation involves a re-encryption check. When implemented correctly, any ciphertext that passes this check is guaranteed to be valid output from ML-KEM encapsulation (see clause 6.2.1).

When a private seed is used in place of the private decapsulation key, this also needs validation before it can be used in ML-KEM decapsulation. The only possible check is that it is a byte array of length 64 bytes.

6.1.3 Internal algorithms

The specification of ML-KEM [i.5] does not require input validation for any of the internal ML-KEM algorithms. However, it is good practice for an implementation to perform input validation on all functions, especially if these functions are exposed to the user, for example for testing purposes.

EXAMPLE 1: ML-KEM.KeyGen_internal ([i.5], algorithm 16) deterministically generates a key pair from two 32-byte values, d and z. Allowing key generation to use shorter byte arrays for d could potentially lead to predictable key pairs.

EXAMPLE 2: ML-KEM.Encaps_internal ([i.5], algorithm 17) deterministically generates a shared secret and ciphertext from the public encapsulation key pk and a 32-byte value m. Allowing encapsulation to use shorter byte arrays for m could potentially lead to predictable shared secrets (see, for example, CVE-2023-1732).

NOTE: General recommendations for input validation can be found in clause 6.2 of ETSI TR 104 239-1 [i.3].

6.2 Perform output validation

6.2.1 Ciphertext re-encryption check

The internal ML-KEM encapsulation function ([i.5], algorithm 17) derives the shared secret from a 32-byte value m and encrypts m using the underlying public-key encryption scheme, K-PKE (see table 8).

Table 8: ML-KEM internal encapsulation function

ML-KEM.Encaps_internal(pk, m)		
Input:	Public encapsulation key $\boldsymbol{pk}$ and 32-byte value $\boldsymbol{m}$	
Output:	Shared secret $\boldsymbol{K}$ and ciphertext $\boldsymbol{c}$	
1.	$(K, r) \leftarrow$ SHA3-512(m || SHA3-256(pk))	
2.	$c \leftarrow$ K-PKE.Encrypt(pk, m, r)	(Encrypt m using randomness r)
3.	return (K, c)	

K-PKE decryption is not guaranteed to return the correct ciphertext and decryption failures leak information about the private decryption key. The ML-KEM parameter sets have been chosen to ensure that the probability of a random decryption failure is negligible. However, it is straightforward for an adversary to perform a K-PKE key recovery attack using malformed ciphertexts.

Consequently, after decrypting the ciphertext to produce a putative plaintext m', the internal ML-KEM decapsulation function ([i.5], algorithm 18) re-encrypts m' and checks that this yields the original ciphertext. If this check fails, decapsulation returns a pseudo-random shared secret K'' that does not depend on the K-PKE private decryption key sk' (see table 9).

NOTE 1: This is an implicit rejection variant of the Fujisaki-Okamoto (FO) transform (see clause B.1.2).

Table 9: ML-KEM internal decapsulation function

ML-KEM.Decaps_internal(sk, c)		
Input:	Private decapsulation key $\boldsymbol{sk}$ and ciphertext $\boldsymbol{c}$	
Output:	Shared secret $\boldsymbol{K}$	
1.	$(sk', pk', h, z) \leftarrow sk$	
2.	$m' \leftarrow$ K-PKE.Decrypt(sk', c)	
3.	$(K', r') \leftarrow$ SHA3-512(m' || h)	
4.	$\overline{K} \leftarrow$ SHAKE256(z || c, 256)	
5.	$c' \leftarrow$ K-PKE.Encrypt(pk', m', r')	(Re-encrypt m' using randomness r')
6.	if $c \neq c'$ then	
7.	$K' \leftarrow \overline{K}$	(Implicit rejection)
8.	end if	
9.	return K'	

An implementation of ML-KEM decapsulation that omitted the re-encryption check, or failed to implement it correctly, could potentially allow a key recovery attack via decapsulation failures.

EXAMPLE: The NIST Round 4 reference implementation of HQC [i.10] contained two flaws which together meant that the re-encryption check during decapsulation would aways pass, even for malformed ciphertexts (see CVE-2024-54137 [i.22]).

NOTE 2: It is enough for an adversary to be able to detect when a failure of the re-encryption check was caused by a K-PKE decryption failure; for example, via side-channel information (see clause 7.4).

6.2.2 Pairwise consistency test

NIST FIPS 203 [i.5] describes optional key pair validation (see section 7.1). Validation of the key pair involves validating the public encapsulation key (see clause 6.1.1), validating the private decapsulation key (see clause 6.1.2), and performing a pair-wise consistency test (see table 10).

Table 10: ML-KEM pairwise consistency test

ML-KEM.PairwiseConsistency(pk, sk)		
Input:	Public encapsulation key $\boldsymbol{pk}$ and private decapsulation key $\boldsymbol{sk}$	
Output:	Boolean, or Error	
1.	$m \leftarrow$ RBG(32)	
2.	if m = NULL then	
3.	return Error	(RBG failure)
4.	end if	
5.	$(K, c) \leftarrow$ ML-KEM.Encaps_internal(pk,m)	
6.	$K' \leftarrow$ ML-KEM.Decaps_internal(sk, c)	
7.	return $K = K'$	

Validation of the key pair is recommended by NIST FIPS 203 [i.5] when it has been provided by a third party that is not trusted or has not given assurances that the key pair is valid.

NOTE 1: These checks do not guarantee that the key pair is valid output from ML-KEM key generation. Full validation of the key pair is not possible without access to the private seed used to generate it.

NOTE 2: Additional checks can be performed on the key pair; for example, checking that the copy of the public key contained in the private decapsulation key matches the public encapsulation key in the key pair. However, these are not required by NIST FIPS 203 [i.5], and any issues would likely be identified by the pairwise consistency test.

NIST's implementation guidance for FIPS 140-3 [i.9], section 10.3.A requires that cryptographic modules perform a pairwise consistency test before a module-generated ML-KEM key can be exported or used for the first time.

NOTE 3: As the pairwise consistency test involves the internal ML-KEM encapsulation and decapsulation functions, it is important to validate both the public encapsulation key and private decapsulation key before performing this check.

NOTE 4: The implementation guidance only requires that a cryptographic module performs the pairwise consistency test for ML-KEM key pairs generated by that module. It is not required for imported key pairs that were generated outside the module.

When a private seed is used in place of the private decapsulation key, it is possible to perform full validation of the key pair by checking that the public encapsulation key can be recomputed from the private seed via private key expansion (see clause 5.2.2).

NOTE 5: If an implementation accepts a private key format that includes both the private decapsulation key and the private seed, it is important to check that the private decapsulation key can also be recomputed from the private seed.

6.2.3 Self testing

For high-assurance applications, it is often recommended that implementations perform a self-test before an algorithm is used for the first time after start-up. Self-tests typically involve checking that the algorithm returns the expected output when provided known inputs. They are intended to detect transient failures caused by hardware faults or implementation corruption, including tampering by an adversary.

EXAMPLE 1: A self-test for ML-KEM key generation could include calling ML-KEM.KeyGen_internal with the inputs $d = 0x00 \ldots 00$ and $z = 0x00 \ldots 00$, then checking that algorithm returns the expected public encapsulation key and private decapsulation key for the parameter set.

EXAMPLE 2: A self-test for ML-KEM encapsulation could include calling ML-KEM.Encaps_internal with a test public encapsulation key pk and the input $m = 0x00 \ldots 00$, then checking that the algorithm returns the expected shared secret and ciphertext for the parameter set.

NOTE 1: Self-tests are distinct from the implementation testing described in clause 8.1.

NIST's implementation guidance for FIPS 140-3 [i.9], section 10.3.A requires that cryptographic modules include self-tests for ML-KEM encapsulation, decapsulation and key generation when the relevant algorithms are implemented.

NOTE 2: The implementation guidance requires that the decapsulation self-tests include both valid and invalid ciphertexts.

NOTE 3: The implementation guidance only requires that the key generation self-test checks that the algorithm returns the expected private decapsulation key as this includes a copy of the public encapsulation key.

NOTE 4: If an implementation supports multiple ML-KEM parameter sets, then the implementation guidance only requires that self-tests are performed for one of the parameter sets.

6.2.4 Public key linting

More detailed public encapsulation key and certificate checks have been proposed in [i.27]. In addition to the public key length and format checks from clause 6.1.1, these include checks on the length and distribution of the seed ρ contained in the public key and checks on the key usage extensions and algorithm identifiers when the public key is provided in a certificate.

6.3 Prevent leakage of intermediate values

6.3.1 Zeroise intermediate values after use

It is generally recommended that implementations zeroise intermediate values before the key generation, encapsulation or decapsulation algorithms return the output.

EXAMPLE 1: Recovering the input m to ML-KEM.Encaps_internal (see table 5) would allow an adversary to recover the shared secret directly.

EXAMPLE 2: Recovering the input r to K-PKE.Encrypt (see table 8) would allow an adversary to recover the shared secret from the ciphertext.

EXAMPLE 3: Recovering the output m' from K-PKE.Decrypt (see table 9) would allow an adversary to identify K-PKE decryption failures and potentially recover the private key.

NIST SP 800-227 [i.7], section 3.2 requires that intermediate values are zeroised with two exceptions:

- Private seeds used in key generation (see clause 5.2.2); and
- Intermediate values derived entirely from public data.

NOTE 1: Private seeds should be stored with the same level of protection as the private decapsulation key.

NOTE 2: Intermediate values derived entirely from public data should be stored with the same level of protection as the public encapsulation key.

EXAMPLE 4: The ML-KEM public encapsulation key includes a 32-byte value ρ which is used to derive a matrix $\hat{A}$ deterministically via the SampleNTT function ([i.5], algorithm 7). The matrix $\hat{A}$ is computed during encapsulation and decapsulation. Storing $\hat{A}$ between encapsulation or decapsulation calls is permitted by NIST SP 800-227 [i.7] since it is entirely derived from public data (see also NIST FIPS 203 [i.5], section 3.3).

NOTE 3: When static key pairs are used, it can be more efficient to store $\hat{A}$ rather than re-computing it each time.

Although NIST FIPS 203 [i.5] describes specific ML-KEM key pair and ciphertext formats for interoperability, NIST SP 800-227 [i.7], section 3.2 allows implementations to store these internally using alternative formats.

EXAMPLE 5: The ML-KEM private decapsulation key includes the byte array encoding of a value $\hat{s}$ which needs to be decoded during decapsulation via the ByteDecode_{12} function ([i.5], algorithm 6). Storing $\hat{s}$ as part of the private key rather than the byte array is permitted by NIST SP 800-227 [i.7] since it is an alternative format for the private key.

6.3.2 Limit external access to internal functions

ML-KEM decapsulation includes a re-encryption check to prevent decryption failures in the underlying public-key encryption scheme, K-PKE, from leaking information about the private key (see clause 6.2.1). If an adversary could access K-PKE decryption directly with the same private key, this would bypass the re-encryption check.

NOTE 1: NIST FIPS 203 [i.5], section 3.3 forbids K-PKE from being used as a standalone cryptographic scheme.

Access to the internal functions ML-KEM.KeyGen_internal, ML-KEM.Encaps_internal and ML-KEM.Decaps_internal can be necessary for testing purposes. However, direct access bypasses input validation which can lead to unpredictable behaviour and potentially violates ML-KEM security guarantees (see clause 6.1).

NOTE 2: NIST FIPS 203 [i.5], section 3.3 recommends that external access to internal ML-KEM functions is not allowed for operational purposes.

6.4 Handle errors gracefully

6.4.1 Specified errors

NIST FIPS 203 [i.5] for requires that ML-KEM key generation and encapsulation return an explicit error when a call to the random bit generator fails (see tables 2 and 5). If an implementation does not check that the call to the random bit generator was successful, or if it ignores any failures, then this can result in predictable key pairs and shared secrets (see, for example, CVE-2023-1732).

An input validation failure prior to encapsulation (see clause 6.1.1) or decapsulation (see clause 6.1.2) will also return an explicit error. A pairwise consistency test (see clause 6.2.2) can return an error caused by a failure of the test itself or of the random bit generation. NIST FIPS 203 [i.5] does not allow implementations to use public encapsulation keys, private decapsulation keys, or ciphertexts that have not been successfully validated.

NOTE: NIST's implementation guidance for FIPS 140-3 [i.9], section 10.3.D requires that higher-level cryptographic modules maintain an error log accessible by authorized users.

6.4.2 Unspecified errors

SampleNTT [i.5], algorithm 7 takes a 34-byte seed for SHAKE128 as input and uses rejection sampling to generate a sequence of integers modulo q. This is called during ML-KEM key generation, encapsulation and decapsulation to expand the public value ρ. The pseudo-code in NIST FIPS 203 [i.5] implements SampleNTT as a while loop that requires a variable number of iterations (see table 11).

Table 11: SampleNTT function (fragment)

SampleNTT(B)	
Input:	34-byte value $\boldsymbol{B}$
Output:	256-long array $\hat{\boldsymbol{a}}$ of integers modulo $\boldsymbol{q}$
1.	$ctx \leftarrow$ SHAKE128.Init()
2.	$ctx \leftarrow$ SHAKE128.Absorb(ctx, B)
3.	$j \leftarrow 0$
4.	while $j < 256$ do
5.	$ctx \leftarrow$ SHAKE128.Squeeze(ctx, 3)
⋮	
16.	end while
17.	return $\hat{\boldsymbol{a}}$

Although NIST FIPS 203 [i.5] recommends that implementations do not limit the number of iterations, annex B of [i.5] gives a bound for which SampleNTT will succeed with overwhelming probability. If there is a fault or a mistake in the implementation, then the number of iterations could exceed this bound. In this case, a bounded-loop version of SampleNTT would return an error that needs to be checked and handled by the calling functions.

Other explicit errors can occur depending on any additional validation or consistency checks included in an ML-KEM implementation; for example, checking that a memory allocation has succeeded.

NOTE: General recommendations for error handling can be found in clause 6.5 of ETSI TR 104 239-1 [i.3]. In particular, it is important that implementations that indicate errors through a status value do not also return the output that might have caused the error since this could be inadvertently used elsewhere.

6.4.3 Implicit rejection

A failure of the re-encryption check in ML-KEM decapsulation will not produce an explicit error. Instead, ML-KEM uses implicit rejection; that is, a failed decapsulation will return a pseudo-random shared secret that is intentionally unrelated to the shared secret produced by encapsulation (see table 9).

The ML-KEM parameter sets have been chosen so that the probability of a random decapsulation failure is negligible, but failures due to malformed ciphertexts can happen if the ciphertext has been garbled during transmission or if it has been modified by an adversary as part of a decapsulation failure attack. Protocols and applications integrating ML-KEM will need to consider techniques such as key confirmation to manage the possibility of decapsulation failures gracefully.

NOTE: It is important that implementations do not to leak information about the cause of a decapsulation failure; for example, via timing or power side channels (see clause 7.4).

6.5 Randomness

ML-KEM key generation and encapsulation both need a source of random bits. Weakly generated random bits can lead to predictable key pairs or predictable shared secrets. Reuse of random values in key generation can violate ML-KEM binding properties (see clauses 5.2.2 and B.2.2). Reuse of random values in encapsulation can compromise shared secrets.

NIST FIPS 203 [i.5], section 3.3 requires that implementations use an approved random bit generator with an appropriate level of security for the parameter set.

7 Side-channel and fault attack considerations

7.1 Introduction

The present clause describes the timing and power analysis side-channel and fault attack considerations that apply to implementations of the three major ML-KEM functions: key generation, encapsulation and decapsulation. Many of these considerations will not be relevant to all deployment scenarios. See the general guidance ETSI TR 104 239-1 [i.3] for detailed discussion of the different attacks and the threat models under which mitigations may be needed.

This guidance focuses on protections for attacks targeting values that are essential to the cryptographic security of ML-KEM, such as the secret seed, expanded secret key and ephemeral shared secrets. Values that are part of the public key, such as the matrix seed ρ and vector $\hat{t}$ are assumed to be known to an adversary.

7.2 Key generation

7.2.1 Overview

As discussed in clause 5.2, ML-KEM.KeyGen_internal takes two 32-byte seeds as input and returns a key pair consisting of the public encapsulation key and corresponding private decapsulation key.

The two 32-byte seeds, d and z, perform different roles within the overall ML-KEM algorithm:

- d is used to seed the key generation algorithm for the K-PKE scheme that is transformed into a KEM via the FO transform.
- z is used in the implicit rejection step of the FO transform to return a pseudo-random shared key if the ciphertext comparison fails.

Leaking of d allows an adversary to learn the decapsulation key, completely compromising the cryptographic security of the scheme. Leaking of z degrades the FO transform from implicit rejection to explicit rejection. Avoiding this is preferred, but recent research [i.25] indicates explicit rejection achieves a similar level of cryptographic security to implicit rejection.

The expansion process is fully determined by the private seed d, which implies that the relevant side-channel attacks are single trace attacks. At first glance, the key generation algorithm is called only once per public-private key pair, which implies a limited scope for side-channel or fault attacks.

However, NIST FIPS 203 [i.5] permits the storage of keys in semi-expanded format (see clause 5.2.2) or in seed format (see clause 5.2.3). Before use in decapsulation, a key stored in the seed format needs to be expanded via the private key expansion process outlined in table 3, requiring a call to ML-KEM.KeyGen_internal. It may therefore be possible for an adversary to trigger repeated calls to the private key expansion process for static ML-KEM keys. This could be used to reduce the noise in a single trace attack by aggregating many traces performing identical calculations. As described in clause 7.1.4 of the general guidance ETSI TR 104 239-1 [i.3], an adversary's ability to access and "profile" the device under attack can significantly reduce their trace requirements, possibly to a single trace. Triggering repeated calls to key generation may additionally have serious implications for fault attacks

If side-channel or fault attacks are considered a viable threat for a particular system, it may be preferable to store ML-KEM keys in semi-expanded format over seed format. This may also be a consideration when it comes to running checks such as the pair wise consistency check outlined in clause 6.2.2. This consistency check could detect malicious modification of the stored key pair, but it also provides an additional opportunity for an attacker to exercise potentially sensitive code.

7.2.2 Timing analysis considerations

As discussed in the general guidance, ETSI TR 104 239-1 [i.3], it is necessary to avoid secret-dependent branching, variable time operations and secret-dependent memory addressing in cryptographic code. Cryptographic code should be isochronous, which means it avoids secret-dependent branching or addressing and the application of variable time instructions to secret-dependent data. The pseudo-code published in NIST FIPS 203 [i.5] for ML-KEM.KeyGen, ML-KEM.KeyGen_internal, K-PKE.KeyGen and their subroutines does not contain secret-dependent branching or secret-dependent memory addressing. It is necessary to carefully evaluate proposed optimizations, such as the introduction of look-up tables to improve performance, to ensure they do not introduce secret-dependence.

Fundamental operations will need to be assessed on a per-architecture basis. Division and modular reduction operations are particularly likely to be non-constant time.

EXAMPLE 1: A secret-dependent timing variation introduced by the division operator occurred in many early implementations of Kyber, the post-quantum scheme on which ML-KEM is based [i.13].

Where constant-time fundamental operations do not exist, implementations could consider replacing them with functionally equivalent algorithms designed to be constant-time. Commonly used examples for modular reduction and multiplication are Barrett reduction and Montgomery multiplication respectively. The precise functional equivalence can be targeted to known properties of data at particular points in the algorithm to allow for optimizations.

EXAMPLE 2: For modular reduction, when values are known to lie in $[0, 2q)$, where q is the size of the underlying finite field, constant-time conditional subtraction can suffice.

7.2.3 Power analysis considerations

As discussed in clause 7.2.1, the deterministic nature of ML-KEM.KeyGen_internal implies that only simple power analysis attacks apply. However, the power of these attacks can be significant, particularly under threat models where the adversary has "profiling" access to the device.

EXAMPLE 1: Kannwischer et al [i.26] describe a single trace attack after a profiling period on the SHA3 hash function used in ML-KEM key generation, and elsewhere, to expand sensitive values.

Simple power analysis typically recovers the Hamming weight of secret values. The entropy loss depends on the distribution of values.

EXAMPLE 2: An AES key byte distributed uniformly in the range 0-255 has an entropy of 8 bits. Learning its Hamming weight leads to an average Shannon entropy loss of 2,5 bits [i.21].

EXAMPLE 3: Coefficients of the ML-KEM secret polynomials s and e are centrally binomially distributed in the range $\pm\eta_1$ where $\eta_1 \in \{2, 3\}$. If $\eta_1 = 2$, each coefficient provides 2,03 bits of entropy. Negative values are represented as unsigned integers mod q, which means there is almost a unique Hamming weight per value. Learning the Hamming weight leads to an average Shannon entropy loss of 1,81 bits.

Example 3 demonstrates that in worst case, where an adversary can perfectly learn the Hamming weights of an ML-KEM secret polynomial, a significant security loss occurs. Once generated, and before further processing, the secret polynomials are transformed into the NTT domain, where the coefficients are more uniformly distributed, and so learning Hamming weights of values is less useful to an adversary. Each coefficient in the NTT representation depends on many coefficients in the polynomial representation and vice versa. This makes it harder for an adversary to guess a small part of the secret in the polynomial representation and confirm their guess by correlating with later values being processed in the NTT representation, but the NTT calculation itself can be used for this guess-and-check attack [i.31].

A commonly employed mitigation for defending against simple power analysis attack is applying random permutation to loops. This randomization is done on a per-call basis and prevents an adversary from correlating learned Hamming weights with the correct coefficient index. The permutations should be considered secret values, but since they are refreshed on a per-call basis, this is unlikely to introduce exploitable secret-dependent memory addressing (see clause 7.2.2).

EXAMPLE 4: In the K-PKE.KeyGen routine in NIST FIPS 203 [i.5], algorithm 13, loop randomization could be employed within the SamplePolyCBD [i.5], algorithm 8) and NTT ([i.5], algorithm 9) functions. It could also be used in the ByteEncode function ([i.5], algorithm 5), though this operates on values in the NTT domain so the security loss from learning Hamming weights is lower.

EXAMPLE 5: The matrix multiplication calculation of $\hat{A} \circ \hat{s}$ provides an adversary with several observations of each secret coefficient being multiplied with public values. This could be viewed as providing several traces in a Differential Power Analysis (DPA) attack. Randomizing the loop order would prevent an adversary from labelling each "trace" with the correct public value to carry out guess-and-check correlations with small parts of the secret value.

7.2.4 Fault attack considerations

An adversary faulting the ML-KEM key generation process will be attempting to learn or set the long-term secret key. A clear target is the generation and expansion of the random seed in ML-KEM.KeyGen ([i.5], algorithm 19) and K-PKE.KeyGen ([i.5], algorithm 13). ML-KEM.KeyGen already includes an explicit check that the random returned value is not null. Faulting the expansion in K-PKE.KeyGen will only be useful if the key is stored in semi-expanded format, or if the adversary can consistently fault the key expansion process. As discussed in clause 6.5, NIST FIPS 203 [i.5], section 3.3 requires that implementations use an approved random bit generator. Where fault attacks are part of the threat model, this random bit generator will require additional protections that are out of scope of the present document.

If an ML-KEM key is stored in the seed format, an adversary may be able to repeatedly trigger the key expansion process as part of decapsulation. In this scenario, faulting the expansion may leak information about the secret key. The information leaked will depend on the output available to the adversary: it is assumed that they do not see the expanded secret key, but they may see the public value $\hat{t} = \hat{A} \circ \hat{s} + \hat{e}$ or just the output from a decapsulation process.

EXAMPLE 1: An adversary sees the public key returned by the key expansion function in K-PKE.KeyGen. They trigger a fault to reset the value of N, the random number generator iteration counter, between the generation of the secret values s and e, causing e to be equal to s. They see a faulted value of $\hat{t}$, and can apply the inverse NTT to find $t = (A + I) \circ s$ and then invert $(A + I)$ to recover s [i.30].

EXAMPLE 2: An adversary sees only the return from the decapsulation process. They can fault the key expansion process to skip the assignment instruction in step 5 of SamplePolyCBD ([i.5], algorithm 8) for a given index i, during the expansion of either s or e. They present an honestly generated ciphertext: if the decapsulation process returns the correct shared secret, the adversary has learned that unfaulted value of $s[i]$ or $e[i]$ is 0. This will determine approximately $3/8$ of the coefficients of the secret values, leading to a significantly reduced dimension for a lattice basis recovery attack.

Each of these examples are illustrative only: enumerating all possible faults and designing individual protections is extremely expensive. As discussed in the general guidance, ETSI TR 104 239-1 [i.3], a simpler mitigation is to repeat steps at risk of faulting and compare the outputs, only proceeding if output agrees. This requires an adversary to accurately induce more than one fault per iteration, which is generally considered extremely difficult.

Reducing an adversary's ability to target a particular instruction by introducing random delays or loop shuffling can limit the effectiveness of fault attacks, as well as the simple power analysis attacks described in clause 7.2.3. Tampering controls can provide an additional level of assurance.

7.3 Encapsulation

7.3.1 Overview

As discussed in clause 5.3, ML-KEM.Encaps ([i.5], algorithm 20) takes a checked encapsulation key and returns a 32-byte shared secret key and a ciphertext. The ciphertext is returned to the owner of the encapsulation key, who can then use the corresponding decapsulation key to recover the shared secret. The vulnerable information is the short-term shared secret. No long-term secret information can be deduced from side-channels when observing only the encapsulation process, since it only uses the public encapsulation key.

The encapsulation process is seeded by a secret 32-byte random value, m, which is expanded deterministically to produce the necessary pseudorandom values for the encapsulation process. NIST FIPS 203 [i.5], section 3.3 specifies that a fresh value of m is used for every iteration of ML-KEM.Encaps, and that m is generated from an approved RBG. If correctly implemented according to NIST FIPS 203 [i.5], this implies that only single trace attacks can be applied to ML-KEM.Encaps, limiting the applicability of side-channel attacks.

NOTE: The K-PKE.Encrypt ([i.5], algorithm 14) subroutine is also called during the decapsulation process as part of the FO transform, where its output is compared with the received ciphertext. Differences between the two values can leak information about the long-term decapsulation key, and so a side-channel protected version of K-PKE.Encrypt may be required in ML-KEM.Decaps ([i.5], algorithm 21). This is discussed further in clause 7.4.

7.3.2 Timing analysis considerations

As discussed in the general guidance, ETSI TR 104 239-1 [i.3], it is necessary to avoid secret-dependent branching, variable time operations and secret-dependent memory addressing in cryptographic code. The pseudo-code published in NIST FIPS 203 [i.5] for ML-KEM.Encaps, ML-KEM.Encaps_internal, K-PKE.Encrypt and their subroutines does not contain secret-dependent branching or secret-dependent memory addressing.

Fundamental operations will need to be assessed on a per-architecture basis. Division and modular reduction operations are particularly likely to be non-constant time.

7.3.3 Power analysis considerations

Similarly to the key general process, the deterministic nature of the ML-KEM.Encaps_internal ([i.5], algorithm 17) algorithm implies that only simple power analysis attacks apply here.

NOTE: DPA attacks do not apply to K-PKE.Encrypt ([i.5], algorithm 14) when called as part of the encapsulation algorithm, but additional protections may be required when it is used as part of the FO transform during decapsulation. See clause 7.4.3 for these considerations.

The analysis is similar to that discussed in clause 7.2.3 for ML-KEM.KeyGen. The coefficients of the secret polynomials y, e_1 and e_2 are restricted to a small number of Hamming weight values, which may be susceptible to recovery via simple power analysis. Once transformed into the NTT domain, secret values are more uniformly distributed, lessening the information derived from their Hamming weights.

As suggested in clause 7.2.3, loop shuffling can be employed to mitigate simple power analysis attacks.

EXAMPLE 1: In the K-PKE.Encrypt routine ([i.5], algorithm 14), loop randomization could be employed within the SamplePolyCBD ([i.5], algorithm 8), NTT ([i.5], algorithm 9) and NTT^{-1} ([i.5], algorithm 10) functions. It can also be applied in the ByteEncode ([i.5], algorithm 5), ByteDecode ([i.5], algorithm 6), Compress ([i.5], equation 4.7) and Decompress ([i.5], equation 4.8) functions, but these operate on values in the NTT domain or uniformly random bytes, so the information derived from Hamming weights is lower.

EXAMPLE 2: The matrix multiplication calculation of $\hat{A} \circ \hat{y}$ provides an adversary with the same opportunity as the calculation of $\hat{A} \circ \hat{s}$ in the K-PKE.KeyGen algorithm ([i.5], algorithm 13). Randomizing the multiplication loop order mitigates the DPA-style attack.

7.3.4 Fault attack considerations

Note that an adversary can produce valid ciphertexts for any secret seed m they desire by honestly following the encapsulation process. Therefore, fault attacks on a device running the ML-KEM encapsulation process are only useful if the adversary expects to either passively monitor or actively interfere with subsequent communications with the device. They can do this either by learning or manipulating the device's value of the random seed m or the shared secret, K, used for later communications.

Fault attacks on the encapsulation process are therefore likely to focus on the random generation and expansion process, which occur in step 1 of ML-KEM.Encaps ([i.5], algorithm 20) and step 1 of ML-KEM.Encaps_internal ([i.5], algorithm 17).

NOTE: Fault attacks do not apply to K-PKE.Encrypt ([i.5], algorithm 14) when called as part of the encapsulation algorithm, but additional protections may be required when it is used as part of the FO transform during decapsulation. See clause 7.4.4 for these considerations.

ML-KEM.Encaps already includes an explicit check that the value returned by the random bit generator is not null. As discussed in clause 6.5, NIST FIPS 203 [i.5], section 3.3 requires that implementations use an approved random bit generator. Where fault attacks are part of the threat model, this random bit generator will require additional protections that are out of scope of the present document.

Faulting the expansion of K is only useful if the adversary is actively communicating with the device. This is because the legitimate owner of the decapsulation key will not calculate the same value of K from the returned ciphertext. In this scenario, knowledge of the ML-KEM decapsulation key is implicitly being used to authenticate a user of the device. Additional authentication mechanisms could be considered as part of the overall protocol design or alternatively fault resistant implementations of the expansion function could be used.

7.4 Decapsulation

7.4.1 Overview

The ML-KEM decapsulation process takes a decapsulation key and a ciphertext and returns a shared secret key. The underlying K-PKE public-key encryption scheme is vulnerable to chosen ciphertext attacks, where an adversary can craft malicious ciphertexts and observe the output to determine the long-term decryption key. The FO transform has been used to convert the IND-CPA secure PKE into an IND-CCA secure KEM (see clause B.1).

The FO transform performs a re-encryption of the decrypted value to confirm that the input ciphertext was honestly generated. If a malformed ciphertext is detected, a pseudo-random shared secret is returned instead.

A small variation in an honest ciphertext will usually not lead to a decryption failure - i.e. the incorrect value of m being recovered - but is still detectable because the ciphertext is deterministically generated from m and the encapsulation key. That is, for a given encapsulation key and random seed m, there is exactly one valid ciphertext.

Larger modifications to a ciphertext will lead to a greater likelihood of a decryption failure, implying a different value of m. This will cause the re-encryption process to produce a significantly different ciphertext. An adversary able to distinguish between small and large differences in the re-encrypted ciphertext can use this as a decryption failure oracle and recover the long-term decapsulation key. This presents a significant challenge in the context of side-channel attacks, because the re-encrypted ciphertext and all derived values require side-channel protections until validity is confirmed at the very end of the decapsulation algorithm.

Protocol level mitigations could include limiting the number of times a decapsulation key is used. The probability of an honestly generated ciphertext failing the FO transform check is cryptographically insignificant, so receipt of a ciphertext that fails this check could be treated in a similar manner to incorrect password guesses.

7.4.2 Timing analysis considerations

As discussed in the general guidance, ETSI TR 104 239-1 [i.3], cryptographic code should aim to avoid secret-dependent branching, variable time operations and secret-dependent memory addressing.

The comparison of the input ciphertext and recalculated ciphertext on step 9 of ML-KEM.Decaps_internal ([i.5], algorithm 18) is a secret-dependent branch and steps 9-10 should be implemented in a constant time fashion, otherwise the security of the scheme is reduced from implicit to explicit rejection. Calculating $\overline{K}$ only when needed - i.e. moving step 7 within the `if` statement - could appear to be a minor efficiency optimization, but should not be done for the same reason.

The comparison itself should also be done in a constant time fashion: the ability to distinguish between large and small differences in the two values breaks the IND-CCA security of the scheme completely [i.23] (see clause B.2.2).

The pseudo-code published in NIST FIPS 203 [i.5] for ML-KEM.Decaps, ML-KEM.Decaps_internal, K-PKE.Decrypt and their subroutines does not contain secret-dependent memory addressing.

Fundamental operations will need to be assessed on a per-architecture basis. Division and modular reduction operations are particularly likely to be non-constant time.

7.4.3 Power analysis considerations

ML-KEM decapsulation keys can be stored in seed format and re-expanded via a call to ML-KEM.KeyGen_internal ([i.5], algorithm 16). The single trace attacks that may arise from an adversary being able to repeatedly trigger the private key expansion process for static ML-KEM keys are outlined in clause 7.2. A similar risk can arise from repeated calls to ByteDecode on step 5 of K-PKE.Decrypt ([i.5], algorithm 15): it may be preferable to store and use $\hat{s}$ directly.

Power analysis attacks present a significant danger to ML-KEM decapsulation implementations in scenarios where an adversary is able to supply chosen ciphertext values. As discussed above, access to a decryption failure oracle compromises the cryptographic security of the scheme completely and so protections need to be applied to the entirety of the process before the FO transform check is completes. In addition, DPA can be applied to the calculations that involve the secret polynomial s and these risks apply whether or not the adversary can choose the ciphertext values.

In the case where an adversary knows but does not control the value of the ciphertext, the hash function J in step 7 of ML-KEM.Decaps_internal ([i.5], algorithm 18) is susceptible to DPA attacks that can reveal the secret value z. This reduces the security of the FO transform from implicit to explicit rejection.

Implementations needing to mitigate power analysis risks will require protected implementations of the K-PKE.Encrypt ([i.5], algorithm 14) and K-PKE.Decrypt ([i.5], algorithm 15) subroutines as well as the final ciphertext comparison. Most of the component functions will require masking, as can be seen in the diagram in section 3 of [i.15]. This will incur significant overheads in the running time and randomness requirements of the algorithm. Careful evaluation of proposed countermeasures will be required, including experimental measurement that leakage has been mitigated on the devices where the protected implementation is expected to run. Research in this area is ongoing [i.28] and [i.34].

7.4.4 Fault attack considerations

The most obvious target for a fault attack in the ML-KEM decapsulation process is the comparison of the input ciphertext with the recalculated ciphertext in step 9 of ML-KEM.Decaps_internal ([i.5], algorithm 18). An adversary able to skip this check gains direct access to the decryption failure oracle described in clause 7.4.1.

More sophisticated fault attacks could "correct" the output from the re-encryption step to ensure that a modified ciphertext passes the FO transform check [i.24].

Enumerating all possible faults and designing individual protections is extremely expensive. As discussed in the general guidance, ETSI TR 104 239-1 [i.3], a simpler mitigation is to repeat steps at risk of faulting and compare the outputs, only proceeding if output agrees. This requires an adversary to accurately induce more than one fault per iteration, which is generally considered extremely difficult.

8 Testing and formal verification considerations

8.1 Testing considerations

The specification of ML-KEM in NIST FIPS 203 [i.5] provides pseudocode for nineteen different algorithms which together define ML-KEM (see annex A). Each of these algorithms can be validated using known-answer tests, but to allow for implementation optimizations or side-channel protections NIST does not require implementations to follow the algorithm structure exactly as written in the standard [i.7]. Instead, NIST only provide test vectors for each of the three main algorithms: ML-KEM.KeyGen_internal, ML-KEM.Encaps_internal, and ML-KEM.Decaps_internal [i.8].

It is important to be careful when exposing internal algorithms for testing purposes. For example, the K-PKE internal algorithms do not provide IND-CCA security before the FO transform is applied (see clause B.1), so these should not be exposed to users except for testing purposes. Additionally, choosing to store private keys in the seed format as discussed in clause 5.2.2 will require an additional external algorithm (see table 3) to convert the seed into a private key that can be used in the signing algorithm. It is important to test any such algorithm.

It is not possible or practical to write an exhaustive list of all the required tests in the present document, since the choice of tests will depend on the threat model and use case of the implementation. However, in addition to the test vectors available from NIST [i.8], supplementary known-answer tests can be written and used to test or fuzz an implementation (see [i.3], clause 8.2). For example, additional tests could be written for each of the general implementation considerations outlined in clause 6. It is important that tests are provided for all code paths, including edge cases and exceptions. Additional "unofficial" test vectors are available from a variety of sources such as Project Wycheproof [i.17] and Crucible [i.33].

8.2 Formal verification considerations

Formal verification can provide more assurance than testing alone that an implementation of ML-KEM is secure [i.3]. However, formal verification is not binary, instead it is possible to choose the extent to which the implementation is formally verified. This choice needs to be made based on a range of factors, such as the threat model, efficiency requirements, and development resources available for the implementation.

EXAMPLE 1: The Post-Quantum Cryptography Alliance have released mlkem-native [i.29] which is an open-source implementation of ML-KEM that is formally verified. The performance-critical components are written in optimized assembly code which is proven functionally correct using HOL Light. The other components are written in C, and these are not proven functionally correct but are proven to be type-safe and memory-safe using the C Bounded Model Checker. The assembly code includes critical functions such as the Number Theoretic Transform, base multiplication, and rejection sampling. The proofs of functional correctness are still reported to take hours to run, even on powerful machines. On the other hand, the type-safety and memory-safety checks for the code written in C are automated and run quickly as part of the continuous integration testing that occurs every time the code is built.

EXAMPLE 2: Another example used EasyCrypt to write a formal specification of ML-KEM and wrote a corresponding implementation in Jasmin that has been proven functionally correct with respect to the specification [i.11]. This approach has been a multi-year effort by a significant number of researchers. The proof is stronger than the previous example because it proves functional correctness of the entire algorithm rather than just performance-critical components, but it is more expensive to write code this way and the resulting implementation is more fragile to implementation changes. This example also proves that the code is isochronous, which is beneficial for defending against side-channel timing analysis attacks (see clause 7).

These examples are both at the upper end of what is achievable for a formally verified implementation of ML-KEM. They both rely on use of programming languages and formal verification tools that will require upskilling for most developers to use reliably. For organizations with less resources and expertise it may not be practical to achieve this level of assurance, and instead rigorous testing should be prioritized as the minimum baseline.

Annex A:
NIST FIPS 203 algorithms

A.1 Key encapsulation (external)

- **ML-KEM.KeyGen (algorithm 19).** Returns a KEM key pair.
- **ML-KEM.Encaps (algorithm 20).** Takes a KEM public key as input. Returns a ciphertext and a shared secret encapsulated by the ciphertext.
- **ML-KEM.Decaps (algorithm 21).** Takes a KEM private key and ciphertext as input. Returns the shared secret.

A.2 De-randomized key encapsulation (internal only)

- **ML-KEM.KeyGen_internal (algorithm 16).** De-randomized key generation called by ML-KEM.KeyGen. Takes a pair of seeds as input. Returns a KEM key pair.
- **ML-KEM.Encaps_internal (algorithm 17).** De-randomized encapsulation routine called by ML-KEM.Encaps. Takes a KEM public key and random value as input. Returns a ciphertext and shared secret encapsulated by the ciphertext.
- **ML-KEM.Decaps_internal (algorithm 18).** Internal decapsulation routine called by ML-KEM.Decaps. Takes a KEM private key and ciphertext as input. Returns the shared secret.

A.3 Public-key encryption (internal only)

- **K-PKE.KeyGen (algorithm 13).** De-randomized key generation for the PKE scheme underlying ML-KEM. Called by ML-KEM.KeyGen_internal. Takes a random seed as input. Returns a PKE key pair.
- **K-PKE.Encrypt (algorithm 14).** De-randomized encryption routine for the PKE scheme underlying ML-KEM. Called by ML-KEM.Encaps_internal. Takes a PKE public key, message and random value as input. Returns a ciphertext.
- **K-PKE.Decrypt (algorithm 15).** Decryption routine for the PKE scheme underlying ML-KEM. Called by ML-KEM.Decaps_internal. Takes a PKE private key and ciphertext as input. Returns the message.

A.4 Sampling

- **SampleNTT (algorithm 7).** Takes a random value as input. Returns a pseudo-random element in the NTT domain.
- **SamplePolyCBD (algorithm 8).** Takes a parameter and random value as input. Returns a polynomial whose coefficients are sampled from the centred binomial distribution with the corresponding parameter.

A.5 Arithmetic

- **NTT (algorithm 9).** Performs the Number Theoretic Transform. Takes a polynomial as input. Returns the corresponding element in the NTT domain.
- **NTT^{-1} (algorithm 10).** Performs the inverse Number Theoretic Transform. Takes an element in the NTT domain as input. Returns the corresponding the polynomial.

- **MultiplyNTTs (algorithm 11).** Performs multiplication in the NTT domain. Takes a pair of elements in the NTT domain as input. Returns their product.
- **BaseCaseMultiply (algorithm 12).** Base case multiplication routine called by MultiplyNTTs. Takes two degree 1 polynomials and a root of unity modulo q as input. Returns their product modulo the corresponding quadratic.

A.6 Encoding

- **BitsToBytes (algorithm 3).** Takes a bit array as input. Returns the corresponding byte array. The length of the bit array is assumed to be a multiple of 8 so no padding is required.
- **BytesToBits (algorithm 4).** Takes a byte array as input. Returns the corresponding bit array,
- **ByteEncode (algorithm 5).** Takes a parameter d and an array of d-bit integers as input. Returns their encoding as a byte array.
- **ByteDecode (algorithm 6).** Takes a parameter d byte array as input. Returns the decoded array of d-bit integers.

NOTE 1: If $d = 12$, then ByteDecode reduces the integers modulo q before they are returned. This means that $\text{ByteEncode}(12, \text{ByteDecode}(12, B))$ will only return B if all integers b in B are in the range $0 \leq b < q$.

- **Compress (equation 4.7).** Takes a parameter d and an integer modulo q as input. Returns the compressed integer modulo 2^d.
- **Decompress (equation 4.8).** Takes a parameter d and an integer modulo 2^d as input. Returns an uncompressed integer modulo q.

NOTE 2: If $d < 12$, then $\text{Compress}(d, \text{Decompress}(d, x))$ is guaranteed to return x for all integers x modulo 2^d.

Annex B:
ML-KEM security properties

B.1 Indistinguishability

B.1.1 IND-CPA security

The main security notion for the public-key encryption scheme K-PKE underlying ML-KEM is Indistinguishability under Chosen-Plaintext Attack (IND-CPA). This is informally defined as follows:

- **IND-CPA:** Given a public encryption key pk, a message m, and a pair of ciphertexts c_1 and c_2, the adversary cannot determine whether c_1 or c_2 is the encryption of m using pk.

IND-CPA security roughly corresponds to a passive adversary who can only observe ciphertexts sent from an honest sender to the recipient. IND-CPA secure public-key encryption schemes can generally only use ephemeral key pairs safely; long-term reuse of keys can be dangerous if an adversary is able to choose or modify ciphertexts.

NOTE 1: NIST SP 800-227 [i.7] requires that ephemeral key pairs are used once and then destroyed.

When implemented correctly, the IND-CPA security of K-PKE depends on the hardness of the Module-LWE problem corresponding to the parameter set [i.12]. However, implementation vulnerabilities can still cause IND-CPA failures.

EXAMPLE: An implementation of K-PKE key generation that does not use a secure random bit generator can lead to predictable key pairs.

K-PKE decryption failures leak information about the private decryption key, so it is not generally safe to use [i.12].

NOTE 2: NIST FIPS 203 [i.5] does not permit K-PKE to be used as a standalone cryptographic scheme.

B.1.2 IND-CCA security

The main security notion for ML-KEM is Indistinguishability under Chosen-Ciphertext Attack (IND-CCA). This is informally defined as follows:

- **IND-CCA:** Given a public encapsulation key pk, a shared secret K, and ciphertext c, the adversary cannot determine whether K was encapsulated by c or was chosen randomly, even when they are given access to a decapsulation oracle that returns the session key encapsulated by any other ciphertext $c' \neq c$.

NOTE 1: This version of Indistinguishability under Chosen-Ciphertext Attack is usually referred to as IND-CCA2 security in the academic literature. A more restricted version, IND-CCA1, only allows the adversary to query the decapsulation oracle before they have been given the challenge ciphertext c.

IND-CCA security roughly corresponds to an active adversary who can construct malformed ciphertexts and observe the recipient's responses in order to learn information about the shared secret. IND-CCA key encapsulation mechanisms can generally reuse key pairs safely provided that the keys are appropriately protected when not in use.

NOTE 2: NIST SP 800-227 [i.7] requires that devices securely store long-term private decapsulation keys.

ML-KEM uses an implicit rejection variant of the Fujisaki-Okamoto transform to convert K-PKE into a KEM. When implemented correctly, ML-KEM will be IND-CCA secure provided that the underlying public-key encryption scheme K-PKE remains IND-CPA secure [i.12]. However, implementation vulnerabilities can still cause IND-CCA failures.

EXAMPLE 1: A small modification c' of a valid K-PKE ciphertext c will decrypt to the same message as c with reasonable probability. ML-KEM decapsulation includes a re-encryption check that should reject c' as an invalid ciphertext (see table 9). An implementation that does not perform this check correctly can return the same shared secrets for c' and c. This is an IND-CCA failure.

NOTE 3: If the ML-KEM implementation never rejects invalid ciphertexts, then an adversary will be able to identify K-PKE decryption failures and use this information to recover the private decapsulation key [i.12].

Implementation vulnerabilities in ML-KEM decapsulation can also allow gradual key recovery attacks even if they do not directly lead to an IND-CCA failure.

EXAMPLE 2: Side-channel vulnerabilities can reveal to an adversary when a failure of the re-encryption check was caused by a K-PKE decryption failure (see clause 7.4).

B.2 Binding

B.2.1 Re-encapsulation attacks

Although IND-CCA is the main security property that KEMs are expected to satisfy, there are protocols where this is not sufficient to meet the security goals of the protocol [i.18]. The motivating example is the KEM-based key exchange from [i.14] which incorporates static key pairs for each party to provide implicit authentication (see figure B.1).

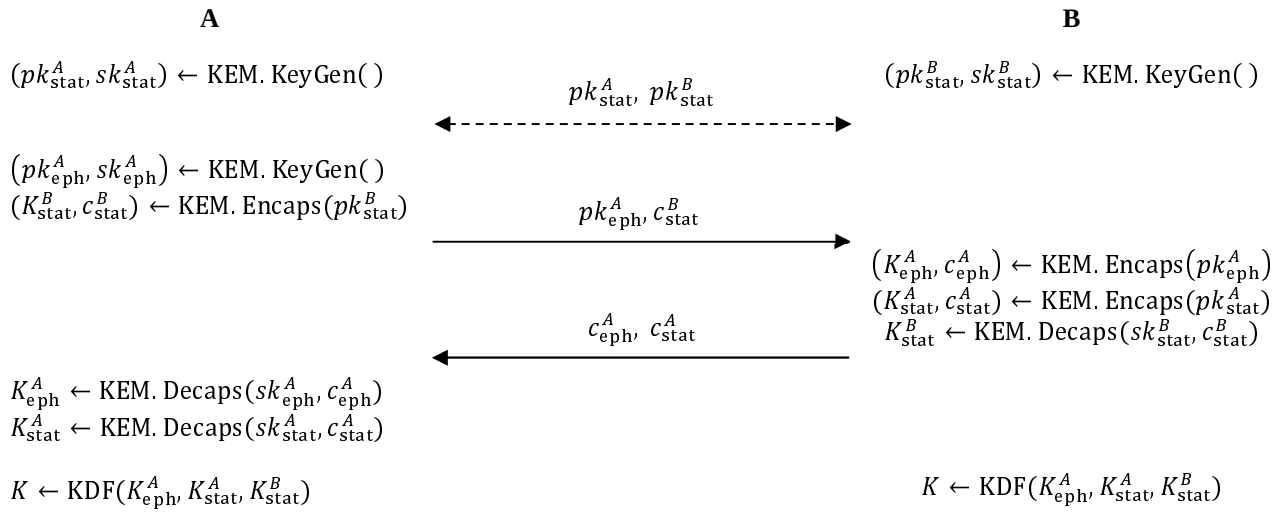

Figure B.1: Implicitly authenticated key exchange

If for a third party E with a different static key pair $(pk^E_{\text{stat}}, sk^E_{\text{stat}})$ an adversary can find a ciphertext c^E_{stat} such that decapsulation $K^E_{\text{stat}} \leftarrow$KEM.Decaps$(sk^E_{\text{stat}}, c^E_{\text{stat}})$ gives the same shared secret $K^E_{\text{stat}} = K^B_{\text{stat}}$, then this breaks the implicit authentication. The adversary can replace B by E in the exchange without A being aware of the switch (see figure B.2).

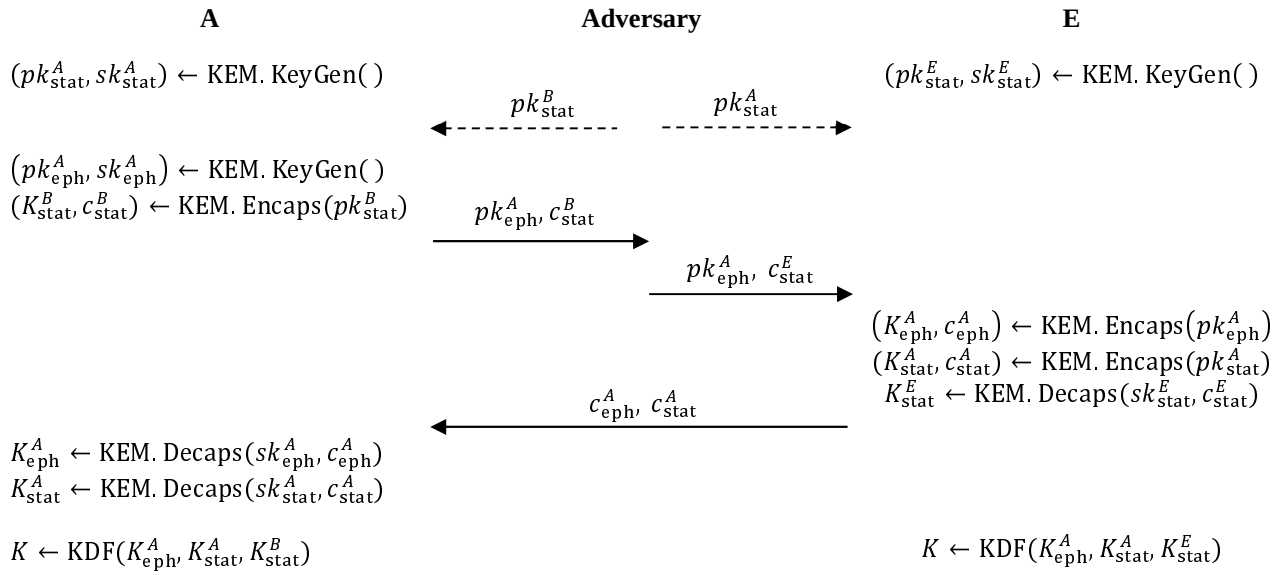

Figure B.2: Re-encapsulation attack

This implicit authentication failure can be prevented by requiring that the KEM binds the shared secret to the public encapsulation key; that is, requiring that it satisfies the BIND-K-PK property described in clause B.2.2.

NOTE: The re-encapsulation attack could also be prevented at the protocol level by including the static public encapsulation keys from both parties in the final key derivation.

B.2.2 Binding properties

Cremers et al [i.18] propose four shared-secret binding properties:

- **BIND-K-CT:** Given two public encapsulation keys pk_1 and pk_2, which need not be distinct, the adversary cannot construct two distinct ciphertexts $c_1 \neq c_2$ that will decapsulate to the same shared secret K with the corresponding private decapsulation keys sk_1 and sk_2.
- **BIND-K-PK:** Given two distinct public encapsulation keys $pk_1 \neq pk_2$, the adversary cannot construct two ciphertexts c_1 and c_2, which need not be distinct, that will decapsulate to the same shared secret K with the corresponding private decapsulation keys sk_1 and sk_2.
- **BIND-K,CT-PK:** Given two distinct public encapsulation keys $pk_1 \neq pk_2$, the adversary cannot construct a single ciphertext c that will decapsulate to the same shared secret K with the corresponding private decapsulation keys sk_1 and sk_2.
- **BIND-K,PK-CT:** Given a single public encapsulation key pk, the adversary cannot construct two distinct ciphertexts $c_1 \neq c_2$ that will decapsulate to the same shared secret K with the corresponding private decapsulation key sk.

NOTE 1: The two additional binding properties from [i.18], BIND-CT-K and BIND-CT-PK, aim to bind the ciphertext. These properties can never be satisfied by key encapsulation mechanisms such as ML-KEM that use implicit rejection since decapsulation still returns a shared secret for invalid ciphertexts.

Each of these binding properties can involve one of three types of adversary:

- **HON:** The key pairs are generated honestly, and the adversary has access to oracles that will decapsulate any ciphertext with the relevant private decapsulation keys.
- **LEAK:** The key pairs are generated honestly, and the adversary has access to the relevant private decapsulation keys.
- **MAL:** The adversary can choose the key pairs, which do not need to be generated honestly.

NOTE 2: The binding properties are modified slightly for MAL adversaries. For full details, see [i.18].

When implemented correctly, ML-KEM will be BIND-K-CT, BIND-K-PK, BIND-K,CT-PK and BIND-K,PK-CT secure against HON and LEAK adversaries. However, implementation vulnerabilities can still cause binding failures.

EXAMPLE 1: If the implementation of ML-KEM decapsulation does not perform the re-encryption check correctly, this can allow two different ciphertexts to decapsulate to the same shared secret (see clause B.1.2). This is a BIND-K-CT and BIND-K,PK-CT failure.

ML-KEM can never be BIND-K-PK or BIND-K,CT-PK secure against MAL adversaries [i.32].

EXAMPLE 2: If a ciphertext fails the re-encryption check, ML-KEM decapsulation returns a shared secret derived from the ciphertext and an implicit rejection value z contained in the private decapsulation key (see table 9). An adversary can choose two distinct private decapsulation keys that share the same implicit rejection value z. Any invalid ciphertext will then return the same shared secret for both private keys.

ML-KEM can also fail to be BIND-K-CT secure against MAL adversaries [i.32].

EXAMPLE 3: If a ciphertext passes the re-encryption check, decapsulation returns a shared secret derived from the output m from K-PKE.Decrypt and a hash value h contained in the private key (see table 9). An adversary can choose two private ML-KEM decapsulation keys that contain distinct K-PKE decryption keys but share the same hash value h. Any pair of valid ciphertexts that decrypt to the same m for the different K-PKE private keys will return the same shared secret for both ML-KEM private keys.

NOTE 3: Validating the private ML-KEM decapsulation key can prevent BIND-K-CT failures (see clause 6.1.2).

Annex C:
Hybrid ML-KEM

C.1 Deployment considerations

It is often recommended that ML-KEM is deployed in a hybrid scheme together with a traditional key exchange such as ECDH in order to mitigate potential ML-KEM implementation vulnerabilities (see, for example, [i.16]).

While a well-designed hybrid scheme will preserve ML-KEM's security properties when implemented correctly, some hybrid constructions might not.

EXAMPLE: NIST SP 800-227 [i.7] lists three mechanisms from NIST SP 800-133 [i.6] as approved methods for combining shared secrets in a hybrid scheme: simple concatenation, XOR, and key extraction using HMAC. Simple concatenation and XOR will not provide IND-CCA security, even if both component algorithms are IND-CCA secure.

NOTE: When implemented correctly, the CatKDF hybrid key establishment scheme from ETSI TS 103 744 [i.1] will preserve IND-CCA security.

For further discussion of hybrid deployment considerations, see ETSI TR 103 966 [i.2].

C.2 Implementation considerations

It is important to realize that deploying ML-KEM in a hybrid scheme is not an alternative to implementing it securely. Vulnerabilities in the ML-KEM implementation can still impact the security provided by a well-designed hybrid scheme.

EXAMPLE 1: A side-channel vulnerability in an ML-KEM implementation that allows an adversary to recover the ML-KEM private key will still allow the adversary to recover the ML-KEM component of the hybrid private key when the vulnerable ML-KEM implementation is used in a hybrid scheme.

NOTE: If the ML-KEM component of the hybrid scheme is compromised, then the hybrid is no longer secure against quantum adversaries and cannot be used to protect against the "harvest now, decrypt later" threat.

Similarly, hybrid schemes should still follow the general implementation guidance from ETSI TR 104 239-1 [i.3]:

- Check the format of input hybrid keys and ciphertexts and preform any relevant cryptographic checks on the traditional and ML-KEM components before they are passed to the corresponding component algorithms.
- Use expected formats for output hybrid keys and ciphertexts and perform any relevant consistency checks on the traditional and ML-KEM components before returning the output.
- Zeroise inputs and outputs for the traditional and ML-KEM component algorithms after use and limit direct access to the component algorithms with the relevant components of the hybrid private key.
- Handle errors from the traditional or ML-KEM component algorithms gracefully and avoid leaking sensitive information through errors where necessary.
- Ensure that implementations of the traditional and ML-KEM component algorithms use a secure random bit generator with good source of entropy.

EXAMPLE 2: An implementation of a hybrid scheme that fails to handle input lengths correctly when parsing hybrid keys or ciphertexts can leak information from memory (see, for example, CVE-2024-37305).

EXAMPLE 3: An implementation of a hybrid scheme that fails to check that the ECDH shared secret is non-zero could inadvertently output a shared secret that only depends on the contribution from ML-KEM.

EXAMPLE 4: An implementation of a hybrid scheme that does not handle errors raised by the ML-KEM implementation correctly could inadvertently output a shared secret that only depends on the contribution from the traditional component.

Annex D:
Change history

Date	Version	Information about changes
November 2025	V0.0.1	Initial document. Content added to clauses 5, 6.1, 7 and 8.
November 2025	V0.0.2	Clause 7 reference updates. Minor editorial updates elsewhere.
February 2025	V0.0.3	Clause 6.1 updates. Content added to clauses 6.2 and 6.4.
March 2025	V0.0.4	Final draft.

History

Version	Date	Status
V1.1.1	June 2026	Publication