Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 14 November 2026

S. Fluhrer
Cisco Systems
Q. Dang
NIST
J. Preu Mattsson
Ericsson
K. Milner
Individual
D. Shiu
Arqit Quantum Inc

13 May 2026

ML-KEM Security Considerations
draft-sfluhrer-cfrg-ml-kem-security-considerations-05

Abstract

NIST standardized ML-KEM as FIPS 203 in August 2024. This document discusses how to use ML-KEM within protocols - that is, what problem it solves, and what you need to do to use it securely.

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://sfluhrer.github.io/ml-kem-security-considerations/draft-sfluhrer-cfrg-ml-kem-security-considerations.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-sfluhrer-cfrg-ml-kem-security-considerations/.

Discussion of this document takes place on the Crypto Forum Research Group mailing list (mailto:cfrg@irtf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cfrg/. Subscribe at https://www.ietf.org/mailman/listinfo/cfrg/.

Source for this draft and an issue tracker can be found at https://github.com/sfluhrer/ml-kem-security-considerations.

Status of This Memo

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

Copyright Notice

Table of Contents

1. Introduction

A Cryptographically Relevant Quantum Computer (CRQC) is a large and reliable Quantum Computer that can break protocols which rely on the traditional RSA, DH, or ECDH methods of securely exchanging keys. Even though it is not believed that, at the time of this writing, there exists a CRQC, there still remains the possibility that an adversary may record the protocol exchange, and then later (when they have access to a CRQC) go ahead and read the traffic.

Because of this threat, NIST has published FIPS 203 [FIPS203], which standardizes a method for allowing two systems to securely exchange keying material and which is not vulnerable to a CRQC. This method is based on module lattices, and is called ML-KEM.

ML-KEM is a Key Encapsulation Mechanism (KEM), which can be used to generate a shared secret key between two parties. A KEM is a public key mechanism where one side (Alice) can generate a public/private key pair, and send the public key to the other side (Bob). Bob then can use it to generate both a ciphertext and a shared secret key. Bob then sends the ciphertext to Alice, who uses her private key to generate the shared secret key. The idea is that someone in the middle, listening into the exchanged public keys and ciphertexts will not be able to recover the shared secret key that Alice and Bob learn. Hence, Alice and Bob can use their shared secret key to establish secure symmetric communication. For ML-KEM, this shared secret is always 32 bytes, and is indistinguishable from random by an adversary (that is, it could be used directly as a symmetric key).

One common misunderstanding of the term KEM is the expectation that Bob freely chooses the shared secret, and encrypts that when sending to Alice. While there do exist KEMs where this is true, this is not true for ML-KEM. In ML-KEM, randomness from both sides is used to contribute to the shared secret. That is, ML-KEM internally generates the shared secret in a way that Bob cannot select the value. Now, Bob can generate a number of ciphertext/shared secret pairs, and select the shared secret that he prefers, but he cannot freely choose it or make the secrets across two different ML-KEM exchanges be equal.

A KEM (such as ML-KEM) sounds like it may be a drop-in replacement for Diffie-Hellman (and in some scenarios, it can be). However this is not always the case. In Diffie-Hellman, the parties exchange two public keys, whereas in a KEM, the ciphertext is necessarily a function of Alice’s public key, and thus can only be useful with that specific public key. Additionally, a KEM differs from Diffie-Hellman which is asynchronous and non-interactive. In particular, for an ’ephemeral-ephemeral’ key establishment, an encapsulator cannot pre-

emptively initiate a key establishment, but requires an encapsulation key. Nor can participants compute parts of the key establishment in parallel as is the case with Diffie-Hellman. As long as the application can handle larger public keys and ciphertexts, a KEM is a drop-in replacement for 'ephemeral-ephemeral' key exchange in protocols like TLS [RFC8446], SSH [RFC4253], Wireguard [WIRE], and EDHOC [RFC9528] as well as 'static-ephemeral' key exchange in protocols like ECIES/HPKE [RFC9180], that is, in cases where Alice has a long term public key, and Bob can use that long term public key to establish communication. A KEM is not a drop-in replacement in applications such as the Diffie-Hellman ratchet in Signal [SIGNAL], implicit 'ephemeral-static' DH authentication in Noise [NOISE], WireGuard [WIRE], and EDHOC [RFC9528], and 'static-static' configurations in CMS [RFC6278] and Group OSCORE [I-D.ietf-core-oscore-groupcomm], where both sides have long-term public keys.

ML-KEM can also be used to perform public key encryption, that is, where a sender encrypts a message with a public key, and only the holder of the private key can decrypt the message. To use ML-KEM for this task, it is recommended that you use it within the Hybrid Public Key Encryption framework [RFC9180] to perform the operations. You can use [I-D.draft-ietf-hpke-pq], which defines three ML-KEM parameter sets that have been proposed for HPKE.

2. Using ML-KEM

To use ML-KEM, there are three steps involved:

2.1. ML-KEM Key Generation

The first step for Alice is to generate a public and private keypair.

In FIPS 203, the key generation function is ML-KEM.KeyGen() (see section 7.1 of [FIPS203]). It internally calls the random number generator for a seed and produces both a public key (known as an encapsulation key in FIPS 203) and a private key (known as a decapsulation key). The seed can be securely stored, but must be treated with the same safeguards as the private key. The seed format allows fast reconstruction of the expanded key pair format, and elides the need for format checks of the expanded key formats. Other intermediate data besides the matrix A_hat must be securely deleted. A_hat may be saved for repeated Decapsulation operation(s) with the same decapsulation key.

The public key can be freely published (and Bob will need it for his part of the process); this step may be performed simply by transmitting the key to Bob. However, the private key (in either format) must be kept secret.

It is essential that the public key is generated correctly when the initial key generation is performed. Lattice public keys consist of a lattice and a secret hidden by an error term; if additional error can be introduced into the public key generation stage, then the success of decapsulation can reveal enough of the secret that successive queries determine the private key. Notably, this means a public key can be ’poisoned’ such that a future adversary can recover the private key even though it will appear correct in normal usage.

To try to prevent such errors before a keypair is used, FIPS requires that an approved implementation perform a Pair-wise Consistency Test (PCT) on each freshly generated keypair: the implementation performs an encapsulation followed by a decapsulation against the new keypair, and verifies that both sides derive the same shared secret. The purpose of the test is to catch key generation errors that would result in a non-functional or weakened keypair - whether from a software bug, a hardware fault, or deliberate fault injection - before the keypair is exported or used for any real exchange. The PCT will reliably detect a keypair that is non-functional, but it cannot rule out the more subtle ’poisoned’ keys described above, which decapsulate honestly generated ciphertexts correctly while still leaking information through decapsulation failures on adversarially chosen ciphertexts.

2.2. ML-KEM Encapsulation

The second step is for Bob to generate a ciphertext and a shared secret key.

To perform this step, Bob would first run the Encapsulation Key Check on Alice’s public key as outlined at the beginning of section 7.2 of [FIPS203]. If that test passes, then Bob would perform what FIPS 203 terms as ML-KEM.Encaps() (see section 7.2 of [FIPS203]). This step takes the validated public key, internally calls the random number generator for a seed, and produces both a ciphertext and a 32-byte shared secret key. Intermediate data other than the ciphertext, shared secret key and the matrix A_hat (and the "matrix data" internal to ML-KEM, which can be deduced from the public key) must be securely deleted. The matrix A_hat may be saved and reused for later encapsulation operations with the same encapsulation key.

The ciphertext can be transmitted back to Alice; if the exchange is successful, the 32-byte shared secret key will be the key shared with Alice.

It may be that some libraries combine the validation and the encapsulation step; implementations should determine whether the library they are using does. For static public keys, the Encapsulation Key Check only needs to be performed once.

2.3. ML-KEM Decapsulation

The third and final step is for Alice to take the ciphertext and generate the shared secret key.

To perform this step, Alice would first run the input validation steps at the beginning of section 7.3 of [FIPS203]: a ciphertext type check on Bob’s ciphertext, and a decapsulation key type check and decapsulation key hash check on her own private key. If those tests pass, then Alice would perform what FIPS 203 terms as ML-KEM.Decaps() (see section 7.3 of [FIPS203]). This step takes the ciphertext from Bob and the private key that was previously generated by Alice, and produces a 32-byte shared secret key. It also repeats some or all of the encapsulation process to ensure that the ciphertext was created strictly according to the specification, with invalid ciphertexts generating an unrelated 32 byte value that gives no information. Although not necessary for the correctness of the key establishment, this step should not be skipped as a maliciously generated ciphertext could induce decapsulation failures that can allow an attacker to deduce the private key with a sufficient number of exchanges. Intermediate data other than the shared secret key and the matrix A_hat must be securely deleted. The matrix A_hat may be saved for later Decapsulation operations with the same decapsulation key.

If the exchange is successful, the 32-byte key generated on both sides will be the same.

It may be that some libraries combine the validation and the decapsulation step; implementations should determine whether the library they are using does. For a static private key, the decapsulation key type and hash checks only need to be performed once; the ciphertext type check is required for every incoming ciphertext.

2.4. ML-KEM Parameter Sets

FIPS 203 specifies three parameter sets; ML-KEM-512, ML-KEM-768 and ML-KEM-1024. It is assumed that Alice and Bob both know which parameter set they use (either by negotiation or by having one selection fixed in the protocol).

Table 1 shows the sizes of the cryptographic material of ML-KEM for each parameter set, as well as their relative cryptographic strength:

	pk size	sk size	ct size	ss size	as strong as
ML-KEM-512	800	1632	768	32	AES-128
ML-KEM-768	1184	2400	1088	32	AES-192
ML-KEM-1024	1568	3168	1568	32	AES-256

Table 1: pk = public key, sk = private key, expanded form, ct = ciphertext, ss = shared key, all lengths in bytes

Table 2 shows an example of ML-KEM performance of each parameter set on one specific platform:

	key generation	encapsulation	decapsulation
ML-KEM-512	244000	153000	202000
ML-KEM-768	142000	103000	134000
ML-KEM-1024	109000	77000	99000

Table 2: Single-core performance in operations per second (higher is better) on AMD Ryzen 7 7700

Data sourced from [EBACS]

As can be seen from Table 1 and Table 2, ML-KEM has significantly larger public keys and ciphertexts than ECDH but very good performance.

3. KEM Security Considerations

This section pertains to KEM (Key Encapsulation Mechanisms) in general, including ML-KEM.

A KEM requires high-quality source of entropy during both the keypair generation and ciphertext generation steps. If an adversary can recover the random bits used in either of these processes, they can recover the shared secret. If an adversary can recover the random bits used during key generation, they can also recover the secret key.

Standard cryptographical analysis assumes that the adversary has access only to the exchanged messages. Depending on the deployment scenario, the adversary may have access to various side channels, such as the amount of time taken during the cryptographical computations, or possibly the power used or the electrical noise generated. The implementor will need to assess this possibility, and possibly use an implementation that is resistant to such leakage.

Alice needs to keep her private key secret. It is recommended that they zeroize the private key when they will have no further need of it, that is, when they know they never need to decapsulate any further ciphertexts with it.

A KEM (including ML-KEM) provides no authentication of either communicating party. If an adversary could replace either the public key or the ciphertext with its own, it would generate a shared key with Alice or Bob. Hence, it is important that the protocol that uses a KEM lets Bob be able to verify that the public key he obtains came from Alice and lets Alice verify that the ciphertext came from Bob (that is, an entity that Alice is willing to communicate with). Such verification can be performed by cryptographic methods such as a digital signature or a MAC to verify integrity of the protocol exchange.

4. ML-KEM Security Considerations

This section pertains specifically to ML-KEM, and may not be true of KEMs in general.

The fundamental security property of ML-KEM is that someone listening to the exchanges (and thus obtains both the public key and the ciphertext) cannot reconstruct the shared secret key, and this is true even if the adversary has access to a CRQC. ML-KEM is IND-CCA2 secure; that is, it remains secure even if an adversary is able to submit arbitrary ciphertexts against a fixed public key and observe the resulting shared key. Submitting invalid ciphertexts to ML-

KEM.Decaps() does not help the attacker obtain information about the decryption key of the PKE-Decrypt function inside the ML-KEM.Decaps(). Substituting the public key Alice sends Bob by another public key chosen by the attacker will not help the attacker get any information about Alice's private key, it would just make Alice and Bob not have a same shared secret key. However, if it is possible to substitute the stored copy of the public key on both sides (Alice's copy, which is bound into her decapsulation key, and Bob's copy, which he encapsulates against), an attacker can introduce a malicious public key where the same private key can be used for decapsulation, but the probability of decryption failure is marginally higher. This is the same 'poisoning' attack described in the Key Generation section above, but performed against the stored public key after generation rather than during generation itself; the consequence is the same in either case. As decryption failures can leak information about the secret decapsulation key, it is important that Alice keeps a secure copy of the public key as part of her secret key. For practical purposes, IND-CCA2 means that ML-KEM is secure to use with static public keys.

ML-KEM requires that a source of random bits with security strength greater than or equal to the security strength of the ML-KEM parameter set be used when generating the keypair and ciphertext during ML-KEM.KeyGen() and ML-KEM.Encaps() respectively. The cryptographic library that implements ML-KEM may access this source of randomness internally. A fresh string of bytes must be used for every sampling of random bytes in key generation and encapsulation. The random bytes should be generated securely [RFC4086].

Alice must keep her private key secret (both private and secure from modification). A copy of the public key and its hash are included in the private key and must be protected from modification.

If the ciphertext that Alice receives from Bob is tampered with (either by small modification or by replacing it with an entirely different ciphertext), the shared secret key that Alice derives will be uncorrelated with the shared secret key that Bob obtains. An attacker will not be able to determine any information about the correct shared secret key or Alice's private key, even if the attacker obtains Alice's modified shared secret key which is the output of the ML-KEM.Decaps() function taking the modified ciphertext as input.

It is secure to reuse a public key multiple times. That is, instead of Alice generating a fresh public and private keypair for each exchange, Alice may generate a public key once, and then publish that public key, and use it for multiple incoming ciphertexts, generating multiple shared secret keys. While this is safe, it is recommended

that if the protocol already has Alice send Bob her unauthenticated public key, they should generate a fresh keypair each time (and zeroize the private key immediately after ML-KEM.Decaps()) to obtain Perfect Forward Secrecy. Generally key generation of ML-KEM is very fast (see Table 2). Hence, if Alice generates a fresh ML-KEM key each time, then even if Alice's system is subverted (either by a hacker or a legal warrant), the previous communications remain secure (because Alice no longer has the information needed to recover the shared secret keys).

Alice and Bob must perform the input validation steps in [FIPS203]: Bob must perform the Encapsulation Key Check on Alice's public key, and Alice must perform the ciphertext type check on Bob's ciphertext and the decapsulation key type and hash checks on her own private key. The cryptographic libraries that Alice and Bob use may automatically perform such checks; they should each verify that is the case.

The shared secret key for all three parameter sets, ML-KEM-512, ML-KEM-768 and ML-KEM-1024 is 32 bytes which are indistinguishable from 32-byte pseudorandom byte-strings of 128, 192 and 256 bits of strengths respectively. As such, the 32-byte string is suitable for both directly as a symmetric key (for use by a symmetric cipher such as AES or a MAC), and also as input into a Key Derivation Function. This is in contrast to a Diffie-Hellman (or ECDH) operation, where the output is distinguishable from random.

If the adversary has control over the ML-KEM private key, it has been shown that the adversary can cause a 'misbinding' between the shared key and either the ciphertext or the public key. That is, by generating an impossible private key (a key that cannot occur with the standard ML-KEM key generation process), the adversary can create public keys for which different ciphertexts or public keys may result in the same shared secret (these security notions are called MAL-BIND-K-CT and MAL-BIND-K-PK in the cryptographical literature [CDM23] [KEMMY24]). This is not a threat to normal uses of ML-KEM as a key exchange or a public key encryption method. If ML-KEM is used as an authentication method where the shared key is used for authentication (and adversary control of the private key is possible), it may be advisable if the protocol also authenticates the public key and ciphertext as well.

4.1. Issues that are likely not a concern

This section contains issues that you may have heard of, but are quite unlikely to be a concern in your use case. This is here to discuss them, and show why they are not practical issues. Readers who have not encountered these issues can safely skip this section.

4.1.1. Decapsulation failure

With ML-KEM, there is a tiny probability of decapsulation failure. That is, even if Alice and Bob perform their roles honestly and the public key and ciphertext are transmitted correctly, there is a tiny probability that Alice and Bob will not derive the same shared key. However, even though that is a theoretical possibility, practically speaking this will never happen. For all three parameter sets, the probability is so low that most likely an actual decapsulation failure because of this will never be seen for any ML-KEM exchange anywhere (not only for your protocol, but over all protocols that use ML-KEM). Hence, the advice we give is to ignore the possibility.

4.1.2. ML-KEM public key expansion not being constant time

During the ML-KEM key generation and the encapsulation process, the public seed is expanded, and this involves rejection sampling of XOF (extendable-output function) output to achieve coefficient values that are uniformly distributed. This means that a straight-forward implementation will perform a variable number of XOF calls to generate this output to find values that are in range. Converting this into a constant time operation is expensive enough that it is rarely done. One consequence of this is that public key expansion is not constant time and this timing sidechannel can be used to provide information about the seed (and the expanded key, which is a function of the seed) in cases where they are not publicly known. On the other hand, this public seed is in the public key, which is almost always publicly known, and so this timing variation does not leak any information about any secret data. One exception is in some methods that implement Password Authenticated Key Exchange with ML-KEM, where the public key may be encrypted with the password. In this rather narrow use case, this variable timing needs to be taken into account.

5. IANA Considerations

This document has no IANA actions.

6. References

6.1. Normative References

[FIPS203] "Module-Lattice-Based Key-Encapsulation Mechanism Standard", NIST FIPS 203, August 2024, <https://doi.org/10.6028/NIST.FIPS.203>.

6.2. Informative References

[CDM23] Cremers, C., Dax, A., and N. Medinger, "Keeping Up with the KEMs: Stronger Security Notions for KEMs and automated analysis of KEM-based protocols", 2023, <https://eprint.iacr.org/2023/1933.pdf>.

[EBACS] "eBACS: ECRYPT Benchmarking of Cryptographic Systems", n.d., <https://bench.cr.yp.to/results-kem/amd64-hertz.html>.

[I-D.draft-ietf-hpke-pq] Barnes, R. and D. Connolly, "Post-Quantum and Post-Quantum/Traditional Hybrid Algorithms for HPKE", Work in Progress, Internet-Draft, draft-ietf-hpke-pq-04, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-hpke-pq-04>.

[I-D.ietf-core-oscore-groupcomm] Tiloca, M., Selander, G., Palombini, F., Mattsson, J. P., and R. Hglund, "Group Object Security for Constrained RESTful Environments (Group OSCORE)", Work in Progress, Internet-Draft, draft-ietf-core-oscore-groupcomm-28, 23 December 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-core-oscore-groupcomm-28>.

[KEMMY24] Schmieg, S., "Unbindable Kemmy Schmidt: ML-KEM is neither MAL-BIND-K-CT nor MAL-BIND-K-PK", 2024, <https://eprint.iacr.org/2024/523.pdf>.

[NOISE] "Noise Protocol Framework", n.d., <http://www.noiseprotocol.org/>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/rfc/rfc4086>.

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <https://www.rfc-editor.org/rfc/rfc4253>.

[RFC6278] Herzog, J. and R. Khazan, "Use of Static-Static Elliptic Curve Diffie-Hellman Key Agreement in Cryptographic Message Syntax", RFC 6278, DOI 10.17487/RFC6278, June 2011, <https://www.rfc-editor.org/rfc/rfc6278>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

[RFC9528] Selander, G., Preu Mattsson, J., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", RFC 9528, DOI 10.17487/RFC9528, March 2024, <https://www.rfc-editor.org/rfc/rfc9528>.

[SIGNAL] "The Double Ratchet Algorithm", November 2011, <https://signal.org/docs/specifications/doubleratchet/>.

[WIRE] "WireGuard", n.d., <https://www.wireguard.com/>.

Acknowledgments

The authors would like to thank Rebecca Guthrie and Thom Wiggers for their valuable comments and feedback.

Authors' Addresses

Scott Fluhrer
Cisco Systems
Email: sfluhrer@cisco.com

Quynh Dang
National Institute of Standards and Technology
Email: Quynh.Dang@nist.gov

John Preu Mattsson
Ericsson
Email: john.mattsson@ericsson.com

Kevin Milner
Individual
Email: kamilner@kamilner.ca

Daniel Shiu
Arqit Quantum Inc
Email: daniel.shiu@arqit.uk