Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 20 September 2026

D. Connolly
Oracle

19 March 2026

Security Considerations for ML-DSA
draft-connolly-cfrg-ml-dsa-security-considerations-02

Abstract

NIST standardized ML-DSA as FIPS 204 in August 2024. This document discusses how to use ML-DSA 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://dconnolly.github.io/draft-connolly-cfrg-ml-dsa-security-considerations/draft-connolly-cfrg-ml-dsa-security-considerations.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-connolly-cfrg-ml-dsa-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/dconnolly/draft-connolly-cfrg-ml-dsa-security-considerations.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Digital signatures are a standardized class of cryptographic scheme that can be used in protocols to detect unauthorized modifications to data and to authenticate the identity of the signer.

Post-quantum (PQ) cryptographic algorithms are based on problems that are considered to be resistant to attacks that are efficient on a cryptographically-relevant quantum computer (CRQC), a quantum computer powerful (QC) enough to break schemes based on traditional cryptographic assumptions such as factoring, finite field or elliptic curve Diffie-Hellman (DH). While it is not believed that a CRQC exists at the time of this writing, there remains the possibility that an adversary could forge signatures on critical messages, or that long-lived public verifying keys will need to remain trustworthy well into the era when CRQCs may exist.

Unlike the case for key agreement and encryption, where an adversary can record ciphertexts now and decrypt them later when a CRQC becomes available ("harvest now, decrypt later"), the threat model for digital signatures is different. An adversary with a CRQC could forge signatures to impersonate a signer whose public verifying key is still trusted. This is a particular concern for:

* Long-lived code signing keys
* Certificate authority root keys and the time needed to establish new ones
* Firmware signing keys
* Digital contract signing keys
* Any public verifying key that will be in use while an active CRQC is a threat

Because of this threat, NIST has published FIPS 204 [FIPS204], which standardizes ML-DSA (Module-Lattice-Based Digital Signature Algorithm), a digital signature scheme that is considered resistant to quantum attacks. ML-DSA is based on structured lattices, specifically reducing to the Module Learning with Errors problem, and is derived from the CRYSTALS-Dilithium scheme.

ML-DSA is a digital signature scheme, where a signer generates a key pair consisting of a private signing key and a public verifying key. The signer uses the signing key to produce a signature on a message, and anyone with the verifying key can verify that the signature is valid for that message. An adversary without the signing key cannot produce a valid signature, even if they have access to a CRQC (see Section 3).

2. Using ML-DSA

As a digital signature scheme, ML-DSA is comprised of three algorithms:

2.1. Key Generation

The first step for the signer is to generate a key pair.

In FIPS 204, the key generation function is ML-DSA.KeyGen() (see section 6.1 of [FIPS204]). It internally calls the random number generator for a 32-byte seed (xi, 両 in FIPS 204) and produces both a public verifying key pkand a private signing key sk.

FIPS 204 supports two signing key formats: the 32-byte seed used as input to key generation serves as a compact signing key format with significant security advantages; the full expanded signing key can be deterministically regenerated from this seed at any time. Using the seed as the signing key format makes it impossible to create a malformed key: the key generation algorithm enforces the correct mathematical distributions for all derived key components [SCHMIEG25]. Further, the whole value of the seed contributes equally to the derived key material; changing a single bit of the seed changes the values of the expanded keypair in its entirety, making it impossible to independently choose parts of the signing key.

The expanded signing key components (rho, K, tr, s1, s2, t0), may be cached in memory for the duration of signing operations to avoid re-running key generation on each signature (rho is ρ in FIPS 204). This cached expanded key material requires the same protections as the seed signing key and benefits from being securely deleted when no longer needed.

The public verifying key can be freely published; verifiers will need it to verify signatures. However, the signing key material needs to be kept secret and protected from modification.

ML-DSA key generation is very fast. The signing key components are sampled from a uniform distribution, making key generation straightforward and not a significant computational burden.

2.2. Signing

The second step is for the signer to produce a signature on a message.

To do this, the signer would perform what FIPS 204 calls ML-DSA.Sign(sk, M, ctx) (see section 6.2 of [FIPS204]). This takes as input the signing key sk, a message M, and an optional context string ctx (up to 255 bytes), and produces a signature sigma (σ in FIPS 204).

Internally, the signing process uses rejection sampling: the raw signing procedure is not always successful on the first attempt, and is repeated until a valid signature is produced. This is normal behavior and not an indication of error, resulting from the construction of ML-DSA from Fiat-Shamir with Aborts [Lyubashevsky09]. On average, signing requires a small number of iterations. However, this does mean that signing time has some variance.

2.2.1. Hedged vs. Deterministic Signing

ML-DSA supports two modes of signing:

* *Hedged (randomized) signing* (the default): Fresh randomness is incorporated into the signing process alongside the signing key material and the message. This is the mode specified by ML-DSA.Sign().

* *Deterministic signing*: The randomness input (rnd) is set to all zeros, making the signature a deterministic function of the signing key and message. This is specified by ML-DSA.Sign() with rnd set to 32 zero bytes.

There is no reason to prefer deterministic signing over hedged signing; hedged signing is the safer default in all environments, and is essential where fault injection or side-channel attacks are a concern. See Section 3.2.1 for details.

The hedged variant degrades gracefully: if the random number generator provides no entropy (all zeros), the hedged variant produces the same output as the deterministic variant.

2.2.2. Context Strings

ML-DSA supports an optional context string (ctx) of up to 255 bytes. The context string is incorporated into the signed message and provides domain separation between different applications or uses of the same key pair.

If a context string is used during signing, the same context string is required during verification. Defining a fixed context string for a given protocol's use case prevents cross-protocol attacks.

2.3. Verification

The third step is for the verifier to check a signature against a message and public verifying key.

To perform this step, the verifier would perform what FIPS 204 calls ML-DSA.Verify(pk, M, sigma, ctx) (see section 6.3 of [FIPS204]). This takes the verifying key pk, the message M, the signature sigma, and the optional context string ctx, and returns whether the signature is valid as a boolean value.

Verification is deterministic and does not require access to any secret key material. It is also computationally straightforward and does not involve rejection sampling unlike signing.

The verifier needs to ensure that the public verifying key pk being used is authentic— that is, it genuinely belongs to the claimed signer. ML-DSA like all signature schemes provides no authentication of the verifying key itself; this is the responsibility of the protocol (e.g., via a certificate or other trust mechanism) or is out of band/scope.

2.4. Parameter Sets

FIPS 204 specifies three parameter sets: ML-DSA-44, ML-DSA-65, and ML-DSA-87. The names refer to the module ranks (k, l) used in each parameter set. It is assumed that the signer and verifier both know which parameter set is in use (either by negotiation or by having one selection fixed in a protocol).

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

	vk size	sk size	expanded sk size	sig size	NIST Level (~as strong as)
ML-DSA-44	1312	32	2560	2420	2 (~SHA(3)-256)
ML-DSA-65	1952	32	4032	3309	3 (~AES-192)
ML-DSA-87	2592	32	4896	4627	5 (~AES-256)

Table 1: vk = public verifying key, sk = signing key (seed form), expanded sk = signing key components cached for signing, sig = signature, all lengths in bytes

All three parameter sets store the signing key as a 32-byte seed. The expanded signing key components can be deterministically regenerated from this seed and cached in memory as needed for signing operations.

3. Security Considerations

3.1. Digital Signature Security Considerations

This section pertains to digital signature schemes in general, including ML-DSA.

A digital signature scheme requires a high-quality source of entropy during key pair generation. If an adversary can recover the random bits used during key generation, they can recover the signing key. ML-DSA additionally requires randomness during signing (in hedged mode); if an adversary can recover these random bits, they may be able to recover the signing key. The random bytes need to be generated securely [RFC4086].

Standard cryptographic analysis assumes that the adversary has access only to the public verifying key, messages, and signatures. Depending on the deployment scenario, the adversary may have access to various side channels, such as the amount of time taken during the signing process, or possibly the power consumption or electromagnetic emissions of the signing device. The implementor will need to assess this possibility and possibly use an implementation that is resistant to such leakage.

The signer needs to keep the signing key secret and protected from modification. Zeroizing the signing key when the signer has no further need of it prevents later compromise.

A digital signature scheme (including ML-DSA) does not authenticate the verifying key. The verifier needs some means of trusting that the public verifying key belongs to the claimed signer. This is typically accomplished through a Public Key Infrastructure (PKI) such as X.509 certificates, or through other trust mechanisms.

3.2. ML-DSA Security Considerations

This section pertains specifically to ML-DSA, and may not be true of digital signature schemes in general.

The fundamental security property of ML-DSA is that someone with the public verifying key and access to signatures cannot forge a signature on a new message, and this is true even if the adversary has access to a CRQC. ML-DSA is EUF-CMA (Existentially Unforgeable under Chosen Message Attack) secure; that is, it remains secure (infeasible to forge verifibale signatures under the public verifying key) even if an adversary can request signatures on arbitrary messages of their choosing. The adversary still cannot produce a valid signature on any message that was not previously signed.

ML-DSA requires that a source of randomness with security strength greater than or equal to the security strength of the ML-DSA parameter set be used during ML-DSA.KeyGen() and during ML-DSA.Sign() (in the default hedged mode). The cryptographic library that implements ML-DSA may access this source of randomness internally. A fresh string of random bytes is needed for every invocation of key generation and signing.

The signer needs to keep their signing key both secret and protected from modification. Modification of the signing key could result in signatures that leak information about the key material.

It is secure to use a single key pair for signing many messages. That is, the signer may generate a key pair once and use it to sign many messages over the lifetime of the key pair. ML-DSA does not degrade in security with the number of signatures produced, under the standard security model.

3.2.1. Signing and Fault Resistance

ML-DSA’s signing process involves computing a nonce-like commitment value (y) as part of each signing attempt. In deterministic mode, this value is a deterministic function of the signing key and the message.

Without mitigation, this creates a vulnerability to fault injection attacks: if an attacker can cause a fault during the signing process and obtain both a correct and a faulted signature on the same message, they can potentially recover the signing key. This is because the deterministic mode will produce the same intermediate value y when signing the same message twice, and comparing correct and faulted outputs can reveal the signing key [KPLG24].

The default hedged (randomized) signing mode mitigates this by incorporating fresh randomness into the computation of y, so that repeated signing of the same message produces different intermediate values. This prevents the nonce-reuse scenario that fault attacks exploit.

Fault injection attacks against ML-DSA are an active area of research [KosXag25]. Notably, the fault attack surface extends to operations such as public parameter generation that are not sensitive to side-channel attacks and therefore might be left unprotected in some implementations.

The deterministic variant of ML-DSA is vulnerable to fault injection attacks, making it unsuitable for platforms where such attacks are a concern. Even in environments where fault injection is not considered a practical threat, hedged signing provides an additional layer of protection.

Section 3.6.1 of [FIPS204] notes that while the signing randomness rnd should ideally be generated by an approved RBG, other methods for generating fresh random values may be used. Because the primary purpose of rnd is to mitigate side-channel and fault attacks on deterministic signatures, even a weak or non-NIST-approved source of randomness is preferable to the fully deterministic variant. Implementations on constrained platforms that lack access to a full-strength RBG should still use whatever randomness source is available for rnd rather than falling back to deterministic signing.

3.2.2. Rejection Sampling and Signing Time

ML-DSA’s signing algorithm uses rejection sampling: candidate signatures are generated and checked against bounds, and the process repeats until a valid signature is found. This means:

* Signing time is variable. On average, only a small number of iterations are needed, but in the worst case, more iterations may be required.

* The number of iterations is not secret-dependent in the default hedged mode (it depends on the random commitment value). Care is warranted to avoid introducing timing side channels in other parts of the signing process.

* Imposing a fixed upper bound on the number of iterations that causes signing to fail is unnecessary and harmful. The probability of requiring a very large number of iterations is negligible (see Section 3.2.6.2).

3.2.3. Signing Key Format

FIPS 204 permits two representations of the ML-DSA signing key:

* *Seed signing key format (32 bytes)*: The random seed used as input to ML-DSA.KeyGen_internal(). The full expanded signing key can be deterministically regenerated from this seed at any time. NIST considers a ML-DSA.KeyGen_internal() seed to be an acceptable alternative format for a signing key, including for generation in one cryptographic module and import/export to another [NIST-PQC-FAQ]. The seed signing key format inherently prevents malformed keys, since the key generation algorithm ensures that all derived values satisfy the required mathematical properties. Every bit of the seed contributes equally to the derived key material, making it impossible to independently choose parts of the signing key [SCHMIEG25].

* *Expanded signing key components*: The full output of ML-DSA.KeyGen(), containing (rho, K, tr, s1, s2, t0). This format avoids the need to re-run key generation before each signing operation. The expanded form is useful as an in-memory cache for performance during signing operations, while the seed signing key format serves as the canonical stored representation. The expanded format admits the possibility of malformed keys if the components are modified or constructed outside of the key generation algorithm.

Implementations that store only the seed and regenerate (or cache) the expanded key as needed are inherently protected against malformed key attacks. Implementations that accept or store expanded signing keys benefit from validating that the key components are well-formed before use.

When both formats are present, the expanded key needs to be the output of running ML-DSA.KeyGen_internal() with the corresponding seed. Inconsistencies between the two representations could lead to undefined behavior [SCHMIEG25].

3.2.4. External Mu (μ)

ML-DSA's signing algorithm (Algorithm 7 of [FIPS204]) computes a fixed-size (64-byte) message representative mu (μ in FIPS 204) as the first step, derived from the hash of the public verifiying key tr and the message M, before any private signing key material is involved. All subsequent signing operations use only mu, not the original message. This structure means that mu can be pre-computed in a separate cryptographic module from the one that holds the signing key, and NIST has explicitly confirmed this is permitted by FIPS 204 [NIST-PQC-ExtMu].

This "external mu" approach solves the use cases that HashML-DSA (Section 3.2.5) was intended to address:

* *Large messages*: The module computing mu can stream arbitrarily large messages through SHAKE256. The signing module only ever receives the fixed-size 64-byte mu, regardless of message size.

* *HSM and remote signing*: An application computes mu from the public verifying key and the message, then sends only mu to a signing module such as an HSM. The signing module generates its own rnd internally and produces the signature.

* *No verification ambiguity*: The resulting signatures are standard ML-DSA signatures. Verifiers do not need to know whether mu was computed locally or externally.

The module computing mu needs a validated implementation of SHAKE256 to be FIPS-compliant. The signing module needs to implement the ML-DSA signing algorithm from mu onward, including its own random number generation for rnd [NIST-PQC-ExtMu].

3.2.5. HashML-DSA

FIPS 204 defines HashML-DSA, a variant that signs a hash of the message rather than the message directly, intended for cases where the full message cannot be transmitted to the signer. However, HashML-DSA conflates a protocol-level decision with the cryptographic primitive, introducing several problems [SCHMIEG24]:

* *Verification ambiguity*: A HashML-DSA signature differs from an ML-DSA signature on the same message. The verifier needs to know which variant was used, which requires protocol-level resolution anyway, defeating the purpose of building this into the primitive.

* *Algorithm confusion risk*: HashML-DSA requires hash algorithm identifiers and parameters to be transmitted alongside the signature. If these are carried through untrusted channels, this introduces algorithm confusion risks similar to those seen with JSON Web Tokens.

The external mu approach described in Section 3.2.4 solves the same use cases without these drawbacks, and NIST has explicitly blessed it for use with FIPS-validated modules [NIST-PQC-ExtMu].

3.2.6. 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 section is here to discuss them, and show why they are not practical issues. If you have not heard of them, you may ignore this.

3.2.6.1. ML-DSA operations not being constant time

During key generation and verification, the public seed rho is expanded to form the matrix A, and this involves rejection sampling of SHAKE256 output to achieve coefficient values that are uniformly distributed. This means that a rote implementation will perform a variable number of SHAKE256 calls, and expansion is not constant time.

However, the public seed rho is part of the public verifying key and is therefore publicly known. The timing variation during matrix expansion does not leak any information about the signing key.

Signing also involves rejection sampling to generate the commitment vector y and to check signature bounds. In the default hedged mode, these values depend on fresh randomness and do not leak signing key information through timing. In deterministic mode, the timing could in theory leak information about the deterministic nonce, though this is a much less practical concern than the fault injection attacks described in Section 3.2.1.

3.2.6.2. Signing loop bounds and probability of failure

ML-DSA’s signing algorithm (Algorithm 7 of [FIPS204]) uses a rejection sampling loop as part of the Fiat-Shamir with Aborts construction [Lyubashevsky09]: each iteration computes a candidate signature, and checks whether it satisfies several norm bounds. If any check fails, the candidate is discarded and the loop repeats with fresh randomness and an incremented counter.

It may appear that this loop could run for a very long time, or that implementations need to guard against the possibility of the loop failing to terminate: in practice, this is not a concern. The expected number of iterations is small: 4.25 for ML-DSA-44, 5.1 for ML-DSA-65, and 3.85 for ML-DSA-87 (Table 1 of [FIPS204]). Each iteration independently succeeds with probability at least 1/5.1 (the worst case across all parameter sets), so the probability of requiring more than n iterations is at most ((5.1 - 1) / 5.1)^n where n, which decreases exponentially. Appendix C of [FIPS204] shows that the probability of needing more than 814 iterations is less than 2^-256 for all parameter sets — that is, it likely will not happen before the heat death of the universe. A great majority of signing operations complete in fewer than 10 iterations.

To remain FIPS 204 compliant, implementations should not (equivalent to 'SHOULD NOT') impose a fixed upper bound on the number of signing loop iterations. If an implementation does impose a limit, FIPS 204 requires that it shall not (equiv. to 'MUST NOT') be lower than 814, and that the signing algorithm shall (equiv. to 'MUST') return an error, produce no other output, and destroy the candidate results of the unsuccessful signing attempts (Appendix C of [FIPS204]). An artificially low iteration limit is more likely to cause problems than it is to prevent them: it could cause signing to fail for valid keys under normal operation, and in deterministic mode could introduce a detectable failure mode that leaks information about the signing key.

4. IANA Considerations

This document has no IANA actions.

5. References

5.1. Normative References

[FIPS204] "Module-Lattice-Based Digital Signature Standard", NIST FIPS 204, August 2024, <https://doi.org/10.6028/NIST.FIPS.204>.

5.2. Informative References

[EBATS] Bernstein, D. J. and T. Lange, "eBATS: ECRYPT Benchmarking of Asymmetric Systems", 2024, <https://bench.cr.yp.to/results-sign/amd64-hertz.html>.

[KosXag25] Kosuge, H. and K. Xagawa, "The Security of ML-DSA against Fault-Injection Attacks", 2025, <https://eprint.iacr.org/2025/904>.

[KPLG24] Krahmer, E., Pessl, P., Land, G., and T. Guneysu, "Correction Fault Attacks on Randomized CRYSTALS-Dilithium", 2024, <https://eprint.iacr.org/2024/138>.

[Lyubashevsky09] Lyubashevsky, V., "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures", DOI 10.1007/978-3-642-10366-7_35, 2009, <https://doi.org/10.1007/978-3-642-10366-7_35>.

[NIST-PQC-ExtMu] National Institute of Standards and Technology (NIST), "FAQ - FIPS 204 - Computing mu", March 2025, <https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/faq/fips204-sec6-03192025.pdf>.

[NIST-PQC-FAQ] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography FAQs", 2025, <https://csrc.nist.gov/Projects/post-quantum-cryptography/faqs#Rdc7>.

[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>.

[SCHMIEG24] Schmieg, S., "HashML-DSA considered harmful", November 2024, <https://keymaterial.net/2024/11/05/hashml-dsa-considered-harmful/>.

[SCHMIEG25] Schmieg, S., "How not to format a private key", February 2025, <https://keymaterial.net/2025/02/19/how-not-to-format-a-private-key/>.

Acknowledgments

TODO acknowledge.

Author’s Address

Deirdre Connolly
Oracle
Email: durumcrustulum@gmail.com