On the Secrecy of the Encapsulation Coin in ML-KEM

M. G. Tehrani* W. J. Buchanan† M. Lemoudden‡

Abstract

ML-KEM (FIPS 203) draws a fresh 32-byte coin at each encapsulation. The shared secret is a deterministic function of the public key and this coin, so a known coin is a recovered key. This is elementary. We ask instead how well the coin's secrecy is protected in practice, and we answer by experiment. On six unmodified libraries (OpenSSL 3.5, wolfSSL 5.9, AWS-LC, Go 1.26, Bouncy Castle 1.83, and CIRCL), and a from-scratch reference, the coin-recovery is reachable in every one; what differs is the guard, from a test-walled package in Go to an ordinary production call in wolfSSL. A second path needs no injection function at all: substituting the generator at build time makes the ordinary encapsulation predictable, while the public re-seed interface correctly refuses to. Outside the validated FIPS-140-3 configuration that most deployments do not yet use, the coin's secrecy rests on convention, not construction. The predictability this permits is externally invisible and parameter-controlled, of the class shown once before in `Dual_EC_DRBG`. We claim no backdoor; we claim only that the door is reachable, and say so while it is still being closed.

1 Introduction

The security of ML-KEM rests on a small amount of secret randomness: a 32-byte encapsulation coin m drawn at each encapsulation, and two 32-byte coins d, z drawn at key generation. That the scheme collapses if such a coin is known is not in dispute and not, by itself, interesting; it is an immediate consequence of the standard's algorithms being deterministic once the coin is fixed. The standard accordingly instructs implementers to draw these coins from an approved cryptographic generator. Yet every conforming implementation must *also* provide a way to supply a fixed coin, because the Known-Answer Tests required for validation are defined in exactly those terms. The coin is thus both the object that must remain secret and an input the implementation is obliged to be able to set.

This tension motivates our question:

> *Is the randomness of ML-KEM a parameterisable point of failure? That is, do paths exist (whether through direct coin-injection functions, or through control of the generator itself) by which the behaviour of encapsulation can be made fully predictable, without that manipulation being detectable from outside?*

We answer in the affirmative, and make the answer precise. Our contributions:

- **A leakage theorem (§2).** We state, with a proof from the FIPS 203 internal algorithms, what the leakage of each coin yields: m breaks the corresponding session, d exposes accepted

*Blockpass ID Lab, Edinburgh Napier University.
†Blockpass ID Lab, Edinburgh Napier University. Corresponding author: `w.buchanan@napier.ac.uk`.
‡Blockpass ID Lab, Edinburgh Napier University.

decapsulations, and z exposes the rejection branch, with full offline decapsulation requiring d and z together. We note in particular that the shared secret K is fixed by (ek, m) before the ciphertext exists, so recovery from m needs the public key and the coin, never the ciphertext.

- **A six-library reachability survey (§3, Table 1).** We run the m-recovery attack end-to-end on six unmodified libraries and a reference implementation. All expose a reachable injection path; the libraries differ markedly in how that path is guarded, ranging from a test-only package (Go) through an exported "experimental" API (AWS-LC) to an ordinary production call (wolfSSL). The protection of the coin is, in these general-purpose builds, a matter of *convention rather than construction*.

- **The generator path, and its safe counterpart (§3).** We show that predictability is also reachable with no injection function at all, by substituting the generator the ordinary encapsulation call draws from at build time; and we show that the matching public re-seed interface correctly refuses to do this. The danger therefore lies not in normal use but in control of the build or distribution chain. This is the same kind of externally invisible, parameter-controlled predictability seen historically in `Dual_EC_DRBG` [2, 3].

- **Where the doors come from (§4).** We document, from NIST's own standardisation record [8], that these deterministic entry points were added so that ML-KEM could be tested for validation. The risk we raise is not that they exist, but that the line between "present for testing" and "reachable in shipped code" is kept only by convention in the general-purpose builds most systems use today.

We are deliberate about what we do *not* claim. We do not assert that any standard generator, or ML-KEM itself, contains a deliberate backdoor; we have no such evidence and make no such claim. We claim only that a mechanism for externally invisible, parameter-controlled predictability exists and is reachable in deployed code, and that its existence deserves to be stated plainly while validation of FIPS 203 modules (under which the direct hooks become unavailable) is still in progress across the industry.

We follow FIPS 203 [10]. Write $\mathbb{B} = \{0,1\}^8$ for one byte and $\mathbb{B}_\ell$ for ℓ bytes. The scheme fixes three hash maps built from Keccak: H = SHA3-256, G = SHA3-512, the implicit-rejection map J = SHAKE256, and the extendable output function used for the matrix and noise. An *encapsulation key* (public) is denoted ek; a *decapsulation key* (private) is dk. We use the FIPS 203 names Encaps, Decaps, KeyGen and their internal forms throughout, and refer to the standard's own algorithm numbers (e.g. Encaps_internal is Algorithm 17 of FIPS 203 [10]) rather than reprinting them.

Three 32-byte values are sampled uniformly at random in the scheme:

$$d \xleftarrow{\$} \mathbb{B}_{32}, \qquad z \xleftarrow{\$} \mathbb{B}_{32} \qquad \text{(both in } \mathsf{KeyGen}\text{)}, \qquad m \xleftarrow{\$} \mathbb{B}_{32} \qquad \text{(in } \mathsf{Encaps}\text{)}.$$

Everything else (ek, dk, the ciphertext c, the shared secret K) is a deterministic function of these three coins and the public parameters. This is the whole of what follows.

2 A leakage theorem

We fix the target. An adversary *breaks* ML-KEM if, given ek and a ciphertext c, it outputs the shared secret that $\mathsf{Decaps}(dk, c)$ returns. The following theorem records what each of the three coins yields. It is elementary, a direct reading of the deterministic FIPS 203 internal maps, and we

claim no novelty for it; we state it carefully because the empirical sections rest on its first part, and because the three coins are not interchangeable.

Theorem 1 (Leakage of the internal randomness)**.** *Fix a well-formed encapsulation key ek, with matching decapsulation key dk generated from seeds (d, z).*

(i) *If the encapsulation coin m of* $\mathsf{Encaps}(ek; m)$ *is known to $\mathcal{A}$, then $\mathcal{A}$ recomputes the ciphertext c and shared secret K with no interaction. For the matching dk, decapsulation of c returns this same K, except with the ML-KEM decapsulation-failure probability δ.*

(ii) *If the key seed d is known to $\mathcal{A}$, then $\mathcal{A}$ recomputes the K-PKE decryption key and hence the decapsulation output for every ciphertext that the re-encryption check* accepts*. For ciphertexts on the rejection branch, knowledge of z is required in addition.*

(iii) *If the rejection seed z is known to $\mathcal{A}$, then $\mathcal{A}$ computes the decapsulation output $\mathrm{J}(z \,|\, c)$ for every ciphertext on the rejection branch. Knowledge of z alone does not determine the output on accepted ciphertexts.*

Proof. (i). Encapsulation (FIPS 203 Algorithm 17 [10]) computes

$$(K, r) \;=\; \mathrm{G}\big(m \,|\, \mathrm{H}(ek)\big), \qquad c \;=\; \mathsf{K\text{-}PKE.Encrypt}(ek, m, r),$$

where G = SHA3-512 and H = SHA3-256 are the two Keccak hashes of FIPS 203, K-PKE.Encrypt is the underlying public-key encryption (Algorithm 14), and $|$ denotes byte-string concatenation. Both are functions of (ek, m) alone, so $\mathcal{A}$ evaluates them and obtains (c, K) outright; the recovery of K by $\mathcal{A}$ is exact. The honest receiver runs decapsulation (Algorithm 18): it computes $m' = \mathsf{K\text{-}PKE.Decrypt}(dk_{\mathrm{PKE}}, c)$, re-derives $(K', r') = \mathrm{G}(m' \,|\, \mathrm{H}(ek))$, and returns K' when the re-encryption matches c. Whenever $m' = m$ one has $K' = K$. The two parties therefore agree except on the measure-δ set of decapsulation failures, where $\delta \le 2^{-138}$ for the three parameter sets [10]; the adversary's own knowledge of K carries no such caveat.

(ii). K-PKE.KeyGen (Algorithm 13) expands d deterministically into the secret vector and the private key $dk_{\mathrm{PKE}} = \mathrm{ByteEncode}_{12}(\hat{s})$. Holding d, $\mathcal{A}$ recomputes dk_{PKE}, runs K-PKE.Decrypt on any c to recover m', re-derives (K', r'), and re-encrypts. On the accept branch this returns the decapsulated key. The reject branch returns $\mathrm{J}(z \,|\, c)$, which d does not determine; hence d alone covers exactly the accepted ciphertexts.

(iii). On the reject branch, decapsulation returns $\bar{K} = \mathrm{J}(z \,|\, c)$, with J = SHAKE256, a function of (z, c). Knowing z, $\mathcal{A}$ computes $\bar{K}$ for any c that takes that branch. An honestly generated c passes re-encryption and never reaches the branch, so z alone says nothing about accepted ciphertexts. □

Remark 1. The three leakages are not equivalent. The coin m is per-encapsulation: its disclosure breaks one session. The seed d is long-term key material: its disclosure exposes every *accepted* decapsulation under ek. The seed z is long-term rejection material: its disclosure exposes only the rejection branch. Full offline decapsulation of arbitrary ciphertexts requires enough to compute both branches, that is, d together with z (equivalently, the stored seed pair, which FIPS 203 says must be guarded as a decapsulation key [10]). The empirical part of this paper concerns case (i): the per-session coin m, which is the one every surveyed library exposes.

Remark 2. K is the first half of $\mathrm{G}(m \,|\, \mathrm{H}(ek))$ and is fixed *before* c exists. In particular K is a function of (m, ek) and is independent of the ciphertext c. A derivation of the form $K = \mathrm{SHAKE256}(m \,|\, c)$ does not occur in FIPS 203 and is impossible by the ordering of FIPS 203 Algorithm 17; we note it because the error is easy to make and changes nothing in case (i) except the formula one computes.

Remark 3. Case (i) is the zero-query, chosen-coin setting. It is adjacent to, but distinct from, the *reuse* attacks of Okada and Wang [11], which recover Bob's randomness in Kyber and SABER from a handful of decryption queries under a meta-PKE model. There the coin is secret but reused; here it is simply known. The two results bound the same quantity, the cost of any failure of the encapsulation coin, from opposite ends.

3 A concrete realisation of Theorem 1

FIPS 203, §6.2, sanctions one way to supply the encapsulation coin externally: the test parameter `OSSL_KEM_PARAM_IKME` ("m"), exposed by OpenSSL since version 3.5 [14]. Setting it makes `EVP_PKEY_encapsulate` consume a caller-supplied m rather than drawing one from the library generator. We stress the negative companion fact: the default path obtains m from `RAND_bytes_ex` on the library context's DRBG, which *no* `openssl.cnf` setting redirects; configuration alone cannot make encapsulation deterministic. The coin can leak only through the program interface or through observation of the running process, which is exactly the threat Theorem 1 formalises.

Procedure. We generate an honest ML-KEM-768 key pair, fix a 32-byte file m, and encapsulate twice against the public key with m supplied through the IKME parameter; then we decapsulate the first ciphertext with the private key. The adversary's view is (ek, m); the honest party's view is (dk, c).

```
# fixed coin m, deterministic encapsulation, honest decapsulation
./encap_ikme  pk.pem m.bin ct1.bin ss_enc1.bin     # adversary path (run 1)
./encap_ikme  pk.pem m.bin ct2.bin ss_enc2.bin     # adversary path (run 2)
openssl pkeyutl -decap -inkey sk.pem -in ct1.bin -secret ss_dec.bin
```

We carried out the same procedure in a second, independent library, wolfSSL 5.9.1 [15], whose with-randomness entry point `wc_MlKemKey_EncapsulateWithRandom` accepts the coin directly. Unlike OpenSSL's IKME, this is a *documented production* function, exposed for external-entropy (e.g. quantum) integration rather than gated as test-only; the injection surface is therefore wider.

Observed. In both libraries the two encapsulations agreed, and the secret computed from (ek, m) equalled the secret recovered from (dk, c). The wolfSSL run, with the fixed coin $m =$ `4b4e...2121`, produced

$$K \;=\; \texttt{8217d697668fa9fdd6145bb43866b335d550730a78ffd8a64e1cf36a1ab9d201},$$

identical on the encapsulation and decapsulation sides:

check	OpenSSL 3.5 (IKME)	wolfSSL 5.9.1 (EncapsulateWithRandom)
$c_1 = c_2$	yes	yes
$K_{\mathrm{enc},1} = K_{\mathrm{enc},2}$	yes	yes
$K_{\mathrm{enc}} = K_{\mathrm{dec}}$	yes	yes

The first two equalities show the coin was consumed; the third is Theorem 1 in a single line. Consistently with Remark 2, K is fixed by (ek, m): re-deriving it requires the public key and the coin, not the ciphertext. We ran the same experiment in four further implementations: AWS-LC[1] (via its exported `EVP_PKEY_encapsulate_deterministic`), Go 1.26 (via the test-only `mlkemtest` package), Bouncy Castle 1.83 [13] (via `internalGenerateEncapsulated`), and Cloudflare's CIRCL [4] (via

the public `EncapsulateTo` and `EncapsulateDeterministically`); all produced $K_{\text{enc}} = K_{\text{dec}}$ (§5). Together with a from-scratch FIPS 203 reference in Python, seven independent implementations agree on the same lemma, which indicates the effect is a property of the standard, not of any one codebase.

The injection surface varies by library. The mechanism by which the coin can be supplied externally, and whether that mechanism survives FIPS 140-3 validation, differs across implementations. Table 1 summarises the current state.

Library	Coin-injection mechanism	Class	FIPS-203 status
OpenSSL 3.5	`OSSL_KEM_PARAM_IKME`	direct, test-only	module in process
wolfSSL 5.9	`EncapsulateWithRandom`	direct, production	in development
AWS-LC	`EVP_PKEY_encapsulate_deterministic`	direct, exported-experimental	in CMVP process
Go 1.26	`mlkemtest` package only	direct, test-walled	module in process
Bouncy Castle 1.83	`internalGenerateEncapsulated`	direct, "internal" method	not validated
CIRCL (Cloudflare)	`EncapsulateTo`, `Encapsulate Deterministically`	direct, public production API	not validated

Table 1: Reachability of the encapsulation coin across ML-KEM libraries, and how each claim here is supported. The mechanism class ranges over: a parameter documented test-only (OpenSSL); a plain production function (wolfSSL); an exported function placed in an `experimental/` header and named `_deterministic` (AWS-LC); a function reachable only from a separate package explicitly labelled test-only (Go); and a method named `internal` taking the coin bytes directly (Bouncy Castle); and a public production method on the standard interface (CIRCL). All six rows were executed end-to-end on one machine (fixed m twice, then decapsulation with the private key, all giving $K_{\text{enc}} = K_{\text{dec}}$). wolfSSL and CIRCL expose the coin through ordinary production calls; the others guard, label, or wall it off, yet in each the coin is reachable. In every *validated* FIPS-140-3 configuration the direct hooks are unavailable, since supplying caller-chosen bytes as the approved RNG output is not an approved service; the variation above is in the general-purpose builds, which is what most deployments use while validation of FIPS 203 modules is still in progress.

Scope. The constructions use no modified library code and no disabled safeguard; each uses a documented entry point in the role it was built for. They say nothing about ML-KEM run with a sound, secret coin, and nothing about the validated FIPS-140-3 configuration, where the direct hooks are unavailable. They say only what Theorem 1 says: a known coin is a recovered key. The point of Table 1 is orthogonal and sharper: the coin's secrecy is, in several deployed libraries, a matter of convention rather than of construction.

The generator path: predictability without any injection function. The table concerns *direct* coin injection. A second, independent path is to leave the ordinary encapsulation call untouched and instead control the generator it draws from. We tested this in wolfSSL, whose normal `wc_MlKemKey_Encapsulate` draws the coin via `wc_RNG_GenerateBlock` from a supplied `WC_RNG`. Two sub-cases separate cleanly and instructively:

- *Public re-seed (safe).* Re-seeding the DRBG through the public `wc_RNG_DRBG_Reseed` with a fixed value does *not* make the output predictable: two runs under the same fixed seed

produced different shared secrets. The interface mixes the caller's bytes with system entropy rather than replacing it, which is the correct DRBG behaviour.

- *Build-time generator substitution (predictable).* wolfSSL's documented `CUSTOM_RAND_GENERATE_BLOCK` build hook replaces the generator source outright. Compiling the library's `random.c` with this macro pointed at a deterministic counter, the *ordinary* `Encapsulate` (no injection function) became fully reproducible: a party knowing only the generator parameter recomputed the victim's exact shared secret, while the key is indistinguishable from a sound one externally.

The contrast is the finding. Predictability here is not reachable through normal *use* of the public API, but it is reachable by whoever controls the library *build or distribution*, the supply-chain position. This is the same class of externally invisible, parameter-controlled predictability exhibited historically by `Dual_EC_DRBG`; we claim the existence of the mechanism and its reachability, not that any standard DRBG is itself compromised.

4 Why the functions exist, and the risk that they persist

It is worth being precise about where the deterministic entry points of Table 1 come from, because the history is documented and the analogy we draw is a careful one.

They were created for standardisation testing. The injection functions are not accidental. In the public comment period for FIPS 203, reviewers observed that the core ML-KEM algorithms are hard to test precisely because they are non-deterministic. NIST's response, recorded in the Federal Register notice issuing the standard [8], was to split each function in two: "a deterministic function that accepts randomness as input (to enable testing) and an externally callable function that generates the needed randomness." This is exactly the shape we found in every library: Encaps_internal (Algorithm 17 of FIPS 203 [10]) takes the coin as an argument so that the Known-Answer Tests of the validation programs can supply a fixed value and check a fixed output. The doors exist by design, and for a legitimate reason.

The standards bodies know they must not leak into production. The testing programs are themselves aware that these internals are not for callers. In a 2025 NIST KEM workshop, the validation community noted plans to add a test of the internal K-PKE encryption routine while being "fully aware this algorithm is not allowed to be used directly by callers" [7]. The intended boundary is explicit: the deterministic interface is a test artifact, the randomised interface is the product.

The risk is that the boundary is forgotten. Our concern is not that the functions exist, but that the line between "present for validation" and "reachable in shipped code" is maintained only by convention in the general-purpose builds most deployments currently use. Table 1 shows the line being drawn in four different places, and in one library (wolfSSL) not drawn at all. The cautionary precedent is `Dual_EC_DRBG`, which entered the SP 800-90 standard through the normal process and remained a recommended generator for roughly seven years before being withdrawn [2, 12]. Notably, the researchers who first demonstrated its weakness did *not* claim an intentional backdoor; they showed only that the design *permitted* one for whoever knew a secret parameter relation [12]. Our claim about ML-KEM is of the same restrained form: a documented, legitimate testing mechanism creates a parameter-controlled, externally invisible path to predictable keys, and the only thing

standing between that path and production deployment is developer discipline plus eventual FIPS-140-3 validation, neither of which is yet universal. The door was opened for a good reason; the danger is that it is left open after the reason has passed.

5 Reproducibility

All code, build scripts, and step-by-step guides are in a public repository:

`https://github.com/owlmt/mlkempaper/`

The repository has one folder per experiment (the six libraries, the Python reference, and the generator experiment), each with its own guide and the exact commands used. Every experiment was run on Ubuntu 24.04 with the library versions named in Table 1. A run is successful when the program prints `YES` on all three comparison lines: the two encapsulations under a fixed coin agree, and decapsulation with the private key recovers the same shared secret.

We note our process openly: the code, the per-experiment documentation, the repository's commit and push steps, and the cross-checking of each experiment's output were carried out with the assistance of an AI coding assistant, and then verified by hand against the library sources and the FIPS 203 algorithm definitions.

Future directions. Two extensions follow naturally. First, the construction this paper concerns is not unique to one protocol: the concatenation of an ML-KEM shared secret with a traditional secret recurs in TLS 1.3, in SSH (`draft-ietf-sshm-mlkem-hybrid-kex`)[6], and in IKEv2 (`draft-ietf-ipsecme-ikev2-mlkem`)[5], each of which derives session keys from a shared secret that a known coin would make predictable at the layer beneath. We examine the protocol-level consequences, and the differing exposure of pure versus hybrid key establishment, separately. Second, the reachability question we ask of ML-KEM applies equally to ML-DSA (FIPS 204)[9], whose signing procedure likewise admits a deterministic, externally supplied randomness input introduced for known-answer testing; whether deployed libraries guard that input by convention or by construction, and what a controlled value there permits, is a parallel evaluation we intend to carry out.

6 Conclusion

The shared secret of ML-KEM is fixed by the public key and a single 32-byte coin, so a known coin is a recovered key; this is elementary, and not our point. Our point is empirical: across six independent, unmodified libraries and a reference implementation, a path to fix that coin is reachable in every one, and the only barrier between that path and a predictable production key is convention, developer discipline, and a FIPS-140-3 validation that most current deployments have not yet completed. We make no claim of a backdoor in ML-KEM or in any generator; we claim only that the mechanism for externally invisible, parameter-controlled predictability exists, is reachable today, and is of the same class history has already shown us once in `Dual_EC_DRBG`, and that it deserves to be named while the doors are still being closed.

References

[1] Amazon Web Services. AWS-LC: A general-purpose cryptographic library. Software, 2025. ML-KEM via `EVP_PKEY_encapsulate_deterministic` in `include/openssl/experimental/kem_deterministic_api.h`.

[2] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A Rev. 1, National Institute of Standards and Technology, Gaithersburg, MD, June 2015.

[3] Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen. Dual EC: A standardized back door. Cryptology ePrint Archive, Paper 2015/767, 2015.

[4] Armando Faz-Hernández and Kris Kwiatkowski. *CIRCL: An Advanced Cryptographic Library*. Cloudflare, 2019. Version 1.6.3, accessed 2026. `https://github.com/cloudflare/circl`.

[5] Panos Kampanakis. Post-quantum key exchange with ML-KEM in the Internet Key Exchange Protocol Version 2 (IKEv2). Internet-Draft draft-ietf-ipsecme-ikev2-mlkem-05, March 2026. Work in Progress.

[6] Panos Kampanakis, Douglas Stebila, and Torben Hansen. PQ/T hybrid key exchange with ML-KEM in SSH. Internet-Draft draft-ietf-sshm-mlkem-hybrid-kex-10, February 2026. Work in Progress.

[7] Stephan Müller. A Glimpse of the Future: NIST's KEM Workshop. atsec information security blog, March 2025. Published March 13, 2025.

[8] National Institute of Standards and Technology. Announcing Issuance of Federal Information Processing Standards (FIPS) FIPS 203, Module-Lattice-Based Key-Encapsulation Mechanism Standard, FIPS 204, Module-Lattice-Based Digital Signature Standard, and FIPS 205, Stateless Hash-Based Digital Signature Standard. Federal Register, August 2024. 89 FR 66052–66057, Document No. 2024-17956, published August 14, 2024.

[9] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Federal Information Processing Standards Publication FIPS 204, U.S. Department of Commerce, August 2024.

[10] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication NIST FIPS 203, National Institute of Standards and Technology, Gaithersburg, MD, August 2024.

[11] Satoshi Okada and Yuntao Wang. Recovery Attack on Bob's Reused Randomness in CRYSTALS-KYBER and SABER. In Qiong Huang and Yu Yu, editors, *Provable and Practical Security*, volume 13059 of *Lecture Notes in Computer Science*, pages 155–173, Cham, 2021. Springer.

[12] Bruce Schneier. Did NSA put a secret backdoor in new encryption standard? Wired, November 2007. Describes the Shumow–Ferguson result that Dual_EC_DRBG's curve constants permit, but do not prove intentional, a parameter-controlled backdoor.

[13] The Legion of the Bouncy Castle Inc. Bouncy Castle Cryptography APIs for Java, version 1.83. Software, 2025. ML-KEM via `MLKEMGenerator.internalGenerateEncapsulated` in package `org.bouncycastle.pqc.crypto.mlkem`.

[14] The OpenSSL Project Authors. *EVP_KEM-ML-KEM*. OpenSSL, 2025. OpenSSL 3.5 manual page; documents the `ikme` parameter `OSSL_KEM_PARAM_IKME`.

[15] wolfSSL Inc. wolfSSL Embedded SSL/TLS Library, version 5.9. Software, 2025. ML-KEM via `wc_MlKemKey_EncapsulateWithRandom` in `wolfcrypt/src/wc_mlkem.c`.