Public Key Linting for ML-KEM and ML-DSA

Evangelos Karatsiolis[1], Franziskus Kiefer[2], Juliane Krämer[3], Mirjam Loiero[1], Christian Tobias[1], and Maximiliane Weishäupl[3]

[1] MTG AG, Darmstadt, Germany
[2] Cryspen, Berlin, Germany
[3] University of Regensburg, Regensburg, Germany

Abstract. With the advancing standardization of post-quantum cryptographic schemes, the need for preparing the IT security infrastructure for integrating post-quantum schemes increases. The focus of this work is a specific part of the IT security infrastructure, namely public key infrastructures. For public certification authorities, it is crucial to guarantee the quality of public keys certified by them. To this end, linting is deployed, which describes the process of analyzing the content of a certificate with respect to predefined rules, the so-called lints.
In this work, we initiate the study of lints for post-quantum cryptography. As a starting point, we choose lattice-based schemes and analyze the public keys of the NIST standards ML-KEM and ML-DSA. We base our analyses on the NIST FIPS standards and IETF documents. We formally describe the identified lints and classify them with respect to the property of the public key that the lint checks. We implement the lints for a common X.509 certificate linter and provide an open-source tool.

Keywords: ML-KEM · ML-DSA · PQC · X.509 · Linting

1 Introduction

A public key infrastructure (PKI) is an important building block to secure our current IT systems and services. In a PKI, a public certification authority (CA) issues certificates to entities to enable protocols like TLS (to secure the Internet) or S/MIME (to secure email communication). The operations of such CAs are crucial for the security and proper functioning of those protocols as well as to establish trust among users. Several policies and requirements about those operations exist and failing to comply with them would lead to an incident. A list of such incidents can be found in the CA/Incident Dashboard[4]. A significant number of these incidents is related to the content of the certificate and could be avoided, if proper mechanisms like certificate linting would always be deployed.

Linting is the process of analyzing the content of a certificate and checking whether it adheres to certain rules: The CA creates the certificate and then

* Work of Maximiliane Weishäupl was funded by the German Federal Ministry of Research, Technology and Space (BMFTR) under the project Quant-ID (16KISQ111).

[4] `https://wiki.mozilla.org/CA/Incident_Dashboard`

initiates the linting process. During linting, the certificate is checked for each rule that is implemented by the underlying linting library. If the check for one rule fails, the linting fails. Then, the CA discards the certificate and notifies the key owner who requested the certificate. Thus, only certificates without lint errors are sent to a certificate transparency (CT) log server and are finally issued. This helps CAs avoid misissuance and report incidents. To avoid errors, linting checks should be applied at an early stage. Ballot SC-75 of CA/Browser Forum[5] renders pre-sign linting mandatory from March 2025. The CA/Browser Forum also states that validating the key material presented by the key owner is a responsibility of the CA and specifies that certain checks must be performed to identify weak keys. Currently, RSA and elliptic curve keys as well as certain vulnerabilities are covered by this requirement. Linting is an appropriate mechanism to implement these checks and comply with the CA/Browser Forum requirements.

Linting is well understood for classical cryptographic schemes like RSA or elliptic-curve cryptography. For post-quantum cryptographic algorithms, however, linting has not been studied yet, although with the introduction of the standards ML-KEM, ML-DSA, and SLH-DSA,[6] the wait for standardized post-quantum cryptography (PQC) is over and the demand for using PQC in practice will increase. Hence, our current IT infrastructure has to be prepared for integrating post-quantum schemes. In this work, we contribute to this endeavor by initiating the research on linting for post-quantum cryptography. As a starting point, we analyze the public key material of lattice-based schemes. We analyze the public keys of the standards ML-KEM [18] and ML-DSA [17], which are based on the algorithms CRYSTALS-KYBER [21] and CRYSTALS-DILITHIUM [14], respectively.[7] We focus on ML-KEM and ML-DSA as a starting point as these are the "primary algorithms to be implemented for most use cases"[8] according to NIST and because they are new to current PKIs. Thus, we provide CAs with a method and a tool to check the public keys of these novel schemes and increase assurance about the quality of the certificates they issue.

Note that next to the public key material, which is the focus of this paper, there are multiple other factors that can compromise the security of the schemes. For the lattice problem ring-LWE, for instance, it was shown that a wrong choice of the underlying ring can result in weak instances [6, 9]. Furthermore, LWE with unusually small secrets or binary errors can be attacked [1, 2]. In this work, however, only potential impairments of the public key are of interest.

[5] https://cabforum.org/2024/06/27/ballot-sc-75-pre-sign-linting/

[6] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/s_Wez9FanHw/m/ipiYjLSdBAAJ

[7] Handling of bad key material has also already been discussed in NIST's pqc-forum, albeit with a focus on private keys: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/9nVfHKtid-k?pli=1

[8] https://www.nist.gov/news-events/news/2022/07/pqc-standardization-process-announcing-four-candidates-be-standardized-plus

Related Work. Several certificate linters already exist. They contain lints for algorithms from classical cryptography and target checking compliance rather than cryptographic aspects. In this work, we present results for two prominent lattice-based schemes, i.e., focus on PQC algorithms and particularly their cryptographic aspects. We observe that these new algorithms require linting for new cases, such as randomness distributions that have been less of an issue before. Therefore, a direct comparison is only possible to a limited degree.

A widely used linter is ZLint [12, 24]. ZLint lints both certificates and revocation lists. It mainly focuses on checking compliance with various specifications about the contents of certificates and revocation lists. Examples are checks about the encoding of fields and extensions, correctness of data, or the absence or presence of certain data. Also, a few lints that check the cryptographic properties of certain schemes exist, e.g., a lint that checks whether an RSA modulus can be factored using Fermat's factorization method[9]. Another certificate linter is pkilint [8], which also lints some PKI protocol messages like online status validation of certificates, i.e., OCSP responses [19]. pkilint is currently under active development. During the writing of this work it added support for linting PQC-certificates, similar to the interoprability lints discussed in Section 4.1. Lastly, two other certificate linters exist, certlint [5] and x509lint [13], which are not being developed actively.

JZLint [16] is a port of Zlint in the programming language Java. It contains most of the lints of ZLint and adds further lints. Together with ZLint, it can be used to perform differential testing for both libraries but also the underlying programming languages and the libraries they use. JZLint also lints other PKI objects like OCSP responses. JZLint is the target software platform where we have integrated our PQC linting work.

Project Wycheproof [23] tests crypto libraries against known attacks. It is a collection of unit tests that check for expected behavior of cryptographic algorithms. The tests covered in Wycheproof comprise (amongst others) known-answer tests, timing attacks, wrong exceptions, and default values (as, e.g., weak or deprecated algorithms). Wycheproof differs from linting considerably: While Wycheproof focuses on cryptographic libraries and the implementation of cryptographic algorithms, certificate linting checks static data objects like public key certificates. Testing of private keys is not covered in Wycheproof, but may be done by linting. Also, linting checks single objects (e.g., a single public key) whereas Wycheproof can be used to create a set of objects of the same type and check for patterns. The correct encoding of public keys (unreduced values, non-invertible matrices used, etc.) is checked in both scenarios. At the time of writing, Wycheproof works on the integration of post-quantum algorithms.

[9] `https://github.com/zmap/zlint/blob/master/v3/lints/community/lint_rsa_fermat_factorization.go`

Contribution. We initiate[10] the study on PQC linting by identifying lints for ML-KEM and ML-DSA. These algorithms are in the early stages of deployment in the Internet and mechanisms to evaluate the quality of their public keys increase assurance for their use. Based on existing formal descriptions of lints, we provide refined formal descriptions. We introduce a classification of lints that depends on the property of the public key that the lint checks. Moreover, we identify several lints that cannot be checked by the CA, but must be performed at the application side. We call such lints application lints and explain why these are different from the lints checked by a CA. An overview of our ML-KEM and ML-DSA lints can be found in Table 1 and Table 2, respectively. Note that the structural similarities of ML-DSA and ML-KEM public keys also transfer to their respective lints.

We provide an open source implementation of the ML-KEM lints. Additionally, we implement a linting tool in Java which incorporates these implementations in order to lint certificates employing PQC algorithms. This tool can be used by entities issuing certificates, for example public CAs. The implementation is provided under `https://github.com/MTG-AG/jzlint` and `https://github.com/cryspen/libcrux/`.

Note that we see our work only as a first step: Our results can be used for ML-KEM and ML-DSA only. Both for other lattice-based schemes and other post-quantum schemes in general, each scheme has to be analyzed thoroughly to identify useful lints. In particular, the code-based KEM HQC, which was recently selected for standardization[11] by NIST, requires a separate analysis. In this work, we showcase that linting of lattice-based algorithms is often more complex than for classical cryptography, and we expect the same to be true for other post-quantum schemes. Moreover, currently different approaches are discussed for the migration from classical to post-quantum cryptography. It is very likely that composite, i.e., hybrid algorithms will be used, consisting of a classical and a post-quantum algorithm. This adds complexity to the cryptographic algorithms and the public key infrastructure used. Hence, for hybrid algorithms more complex lints will have to be identified.

Organization. This work is organized as follows: In Section 2, we describe the used notation and provide background on ML-KEM and ML-DSA. In Section 3, we explain our methodology, which comprises the formal description of lints, our strategy for finding lints, and their implementation. In Section 4, we present the lints for ML-KEM and in Section 5 the ones for ML-DSA. We provide the formal description of all our lints in Appendix A.

[10] Concurrent to this research, the certificate linter pkilint added support for X.509 certificates using the PQC schemes ML-KEM, ML-DSA, and SLH-DSA.

[11] `https://www.nist.gov/news-events/news/2025/03/nist-selects-hqc-fifth-algorithm-post-quantum-encryption`

2 Background

In this section, we provide necessary background information on ML-KEM. We start with describing the notation in Section 2.1 and then present ML-KEM in Section 2.2.

2.1 Notation

Throughout this work, we write $\mathbb{B} := \{0,1\}^8$ for the set of bytes and we use $||$ to denote the concatenation of bit and byte strings. For a bitstring $b \in \{0,1\}^n$ and $i \leq j \leq n$, the notation $b[i,j]$ will be used for $b[i]||b[i+1]||\ldots||b[j]$ where $b[k]$ denotes the k-th bit of b (note that indexing starts at 1). By $[[\cdot]]$ we denote the boolean predicate, that evaluates the expression inside the brackets. We denote the modulus by q and write $\mathbb{Z}$ for the ring of integers and $\mathbb{Z}_q$ for the finite field with q elements. Following the notation from FIPS 203 and 204 [17, 18], the symbol '$\circ$' will be used to denote the multiplication in $\mathbb{Z}_q^{256}$ while '$\cdot$' is used in all other cases. For a positive integer n, we consider the polynomial rings $R = \mathbb{Z}[X]/(X^n+1)$ and $R_q = \mathbb{Z}_q[X]/(X^n+1)$. For a matrix A and a vector b we denote by $A[i,j]$ and $b[i]$ the entry at the position (i,j) and i, respectively. The public key matrix from ML-KEM will be denoted by $\hat{A}$, which is a square matrix of dimension k. Lastly note that we will use H and G as notation for hash functions and XOF for extendable-output functions.

2.2 ML-KEM

ML-KEM is a NIST standard for key-encapsulation mechanisms (KEM) using module lattices. It is based on the KEM CRYSTALS-Kyber, which was submitted to NIST's PQC standardization process and selected for standardization.

In the following section, we provide the algorithms from the standard FIPS 203 [18] that are relevant for our work and short high-level descriptions of them. In the standard, key generation, en- and decapsulation are divided into external and internal components. The former generate randomness, check whether the randomness generation was successful, and finally call their internal counterparts, which then perform the actual steps of the procedures. Since the public key is relevant solely for the internal components, we will describe only those in detail. Further, we focus particularly on the key generation and the encapsulation as we are linting the public key. Next to this, we describe the subalgorithms used to encode, decode, and expand the public key, as we give lints for the public key in all of its forms. Lastly, we give an overview on which lints concern which components of the public key in Algorithm 2; the corresponding lint identifiers are listed on the very right of each affected line in the pseudocode.[12] Following NIST's notation, we call ML-KEM public keys also encapsulation keys.

[12] Note that this does not include interoperability and generic lints as those cannot be assigned to certain components of a public key.

Algorithm 1 ML-KEM.KeyGen_internal(d, z)

Uses randomness to generate an encapsulation key and a corresponding decapsulation key.
Input randomness $d, z \in \mathbb{B}^{32}$
Output encapsulation key $ek \in \mathbb{B}^{384k+32}$, decapsulation key $dk \in \mathbb{B}^{768k+96}$
1: $(ek_{\text{PKE}}, dk_{\text{PKE}}) \leftarrow$ K-PKE.KeyGen(d)
2: $ek \leftarrow ek_{\text{PKE}}$
3: $dk \leftarrow (dk_{\text{PKE}} \| ek \| H(ek) \| z)$
4: **return** (ek, dk)

On a high level, ML-KEM is constructed in two steps: Firstly, a public key encryption scheme K-PKE is described and secondly, this scheme is turned into a KEM via an application of the FO-transform. This structure is also reflected in the algorithms for ML-KEM, which call the corresponding algorithms of the underlying PKE.

In Algorithm 1 the internal ML-KEM key generation is described and in Algorithm 2 the one of K-PKE is depicted. Essentially, an ML-KEM encapsulation key is obtained by generating an LWE sample $\hat{t} = \hat{A} \circ \hat{s} + \hat{e}$ for a random matrix $\hat{A}$ and s, e sampled from a centered binomial distribution. More precisely, the ML-KEM encapsulation key consists of an encoded version of $\hat{t}$ together with a seed ρ that can be used to generate the matrix $\hat{A}$. Note that the NTT is used to switch between representations of the elements as polynomials and vectors over $\mathbb{Z}_q$ (following the notation of the standard, we denote $\hat{f} = \text{NTT}(f)$).

In Algorithm 3 the internal ML-KEM encapsulation is described and in Algorithm 4 the encryption of the underlying PKE is given. The encapsulation returns a key K and a ciphertext. The former is computed as the hash of a randomly chosen message m and the encapsulation key, while the latter is obtained using the encryption of K-PKE. During this encryption, the matrix $\hat{A}$ and vector $\hat{t}$ are retrieved from the encapsulation key and used to compute another noisy linear equation incorporating the message m (during decapsulation, the vector $\hat{s}$ that is contained in the ML-KEM decapsulation key can be used to recover m and then in turn K).

In the key generation and the encapsulation, the encapsulation key is encoded and decoded, respectively. We describe the corresponding procedures in Algorithm 5 and 6. Further, the algorithm used to generate the matrix $\hat{A}$ from the seed ρ is given in Algorithm 7. Note that, for the rest of the paper, we will use the descriptors *encoded*, *extracted*, and *expanded* to refer to the encapsulation key after encoding, decoding, and expansion (i.e., the generation of A out of ρ). Lastly, the most important parameters are $q = 3329 = 2^8 \cdot 13 + 1$, $n = 256$, and $k \in \{2, 3, 4\}$.

[13] The functions Init(), Absorb(), and Squeeze() relate to the SHAKE128 API. Details can be found in [18].

Algorithm 2 K-PKE.KeyGen(d)

Uses randomness to generate an encryption key and a corresponding decryption key.
Input: randomness $d \in \mathbb{B}^{32}$
Output: encryption key $ek_{\text{PKE}} \in \mathbb{B}^{384k+32}$, decryption key $dk_{\text{PKE}} \in \mathbb{B}^{384k}$

$(\rho, \sigma) \leftarrow G(d||k)$ ▷ DIM_02, DIS_01, DIS_02, DIS_03
$N \leftarrow 0$
for $(i \leftarrow 0; i < k; i{+}{+})$ **do** ▷ DIM_03
  for $(j \leftarrow 0; j < k; j{+}{+})$ **do**
    $\hat{A}[i, j] \leftarrow \text{SampleNTT}(\rho||j||i)$ ▷ DOM_01, DIS_04, DIS_05, DIS_06
  end for
end for
for $(i \leftarrow 0; i < k; i{+}{+})$ **do**
  $s[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
  $N \leftarrow N + 1$
end for
for $(i \leftarrow 0; i < k; i{+}{+})$ **do**
  $e[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
  $N \leftarrow N + 1$
end for
$\hat{s} \leftarrow \text{NTT}(s)$
$\hat{e} \leftarrow \text{NTT}(e)$
$\hat{t} \leftarrow \hat{A} \circ \hat{s} + \hat{e}$ ▷ DIM_04, DOM_02
$ek_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{t}) \| \rho$ ▷ DIM_01
$dk_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{s})$
return $(ek_{\text{PKE}}, dk_{\text{PKE}})$

2.3 ML-DSA

ML-DSA is a NIST standard for digital signatures using module-lattices. It is based on the digital signature scheme CRYSTALS-Dilithium, which was submitted to NIST's PQC standardization process and selected for standardization.

As for ML-KEM, we give the algorithms from the standard FIPS 204 [17] that are most relevant for linting ML-DSA public keys, i.e., the key generation and verification as well as the subalgorithms for encoding, decoding, and expansion. We focus again on the internal components,[14] as those are the ones for which the public key is relevant. The external components are built analogously to the ML-KEM ones, except for the fact that there is an additional check (length of the context string) and that the message is formatted before it is passed on to the corresponding internal component. We provide an overview on which lints concern which components of the public key in Algorithm 8 (the lint identifiers are listed on the very right in the pseudocode).[15]

[14] For ML-DSA, also the official tests from the Cryptographic Algorithm Validation Program (CAVP) only test internal APIs (see for example `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/s_Wez9FanHw/m/hEBZ4psWBgAJ?pli=1`).

[15] As for ML-KEM, this does not include interoperability and generic lints as those cannot be assigned to certain components of a public key.

Algorithm 3 ML-KEM.Encaps_internal(ek, m)

Uses the encapsulation key and randomness to generate a shared key and a ciphertext.
Input: encapsulation key $ek \in \mathbb{B}^{384k+32}$, randomness $m \in \mathbb{B}^{32}$
Output: key $K \in \mathbb{B}^{32}$, ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$
1: $(K, r) \leftarrow G(m \parallel H(ek))$
2: $c \leftarrow$ K-PKE.Encrypt(ek, m, r)
3: **return** (K, c)

In Algorithm 8 the ML-DSA key generation is described. Similar to ML-KEM, a public key is obtained by generating an LWE sample $t = As_1 + s_2$ for a random matrix A and two random short vectors s_1 and s_2 (note that rejection sampling is used to guarantee shortness of s_1 and s_2). The vector t is then further compressed for performance reasons, resulting in t_1. The final public key consists of an encoding of t_1 and a seed ρ that can be used to generate the matrix A. In particular note, that for ML-KEM the encoding is not applied to the matrix seed, while for ML-DSA it is. Another difference lies in the fact that for ML-KEM the vector $\hat{t}$ is included in the public key in NTT-representation, while for ML-DSA it is not.

The ML-DSA verification is described in Algorithm 9. During verification, the matrix A and vector t_1 are recomputed from the public key and used to reconstruct the signer's commitment. The key step here is the computation $w'_{\text{approx}} = Az - ct_1 \cdot 2^d$, where z and c are contained in and derived from the signature, respectively, and $t_1 \cdot 2^d \approx t$. During the final verification check, it is, among other conditions, checked whether the recomputed commitment matches the one that is part of the signature.

The encoding and decoding algorithm that are applied in key generation and encapsulation, are described in Algorithm 10 and Algorithm 11. Furthermore the algorithm used to expand the matrix A from the seed ρ is given in Algorithm 12. As for ML-KEM, we will use the descriptors *encoded*, *extracted*, and *expanded* to refer to the public key after encoding, decoding, and expansion (i.e., after application of ExpandA[16]). Lastly, the most important parameters are $q = 8380417 = 2^{23} - 2^{13} + 1$, $n = 256$, and $(k, l) \in \{(4, 4), (6, 5), (8, 7)\}$.

3 Methodology

In this section, we present the methodology of our work. In Section 3.1, we explain our strategy for finding lints. In Section 3.2, we describe the formal description of lints that we use in this work, introduce the concepts of lint classes and application lints, and comment on the completeness of the lints and lint classes studied in this work. In Section 3.3, we provide details of the implementation.

[16] Note the uncompressed version t of t_1 does not need to be explicitly computed for signing and verification, hence we refer to (A, t_1) as the expanded public key.

Algorithm 4 K-PKE.Encrypt(ek_{PKE}, m, r)

Uses the encryption key to encrypt a plaintext message using the randomness r.
Input: encryption key $ek_{\text{PKE}} \in \mathbb{B}^{384k+32}$, message $m \in \mathbb{B}^{32}$, randomness $r \in \mathbb{B}^{32}$
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$

$N \leftarrow 0$
$\hat{t} \leftarrow \text{ByteDecode}_{12}(ek_{\text{PKE}}[0 : 384k])$
$\rho \leftarrow ek_{\text{PKE}}[384k : 384k + 32]$
for $(i \leftarrow 0; i < k; i{+}{+})$ **do**
    for $(j \leftarrow 0; j < k; j{+}{+})$ **do**
        $\hat{A}[i, j] \leftarrow \text{SampleNTT}(\rho || j || i)$
    end for
end for
for $(i \leftarrow 0; i < k; i{+}{+})$ **do**
    $y[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(r, N))$
    $N \leftarrow N + 1$
end for
for $i \leftarrow 0$ to $k - 1$ **do**
    $e_1[i] \leftarrow \text{SamplePolyCBD}_{\eta_2}(\text{PRF}_{\eta_2}(r, N))$
    $N \leftarrow N + 1$
end for
$e_2 \leftarrow \text{SamplePolyCBD}_{\eta_2}(\text{PRF}_{\eta_2}(r, N))$
$\hat{y} \leftarrow \text{NTT}(y)$
$u \leftarrow \text{NTT}^{-1}(\hat{A}^\top \circ \hat{y}) + e_1$
$\mu \leftarrow \text{Decompress}_1(\text{ByteDecode}_1(m))$
$v \leftarrow \text{NTT}^{-1}(\hat{t}^\top \circ \hat{y}) + e_2 + \mu$
$c_1 \leftarrow \text{ByteEncode}_u(\text{Compress}_{d_u}(u))$
$c_2 \leftarrow \text{ByteEncode}_v(\text{Compress}_{d_v}(v))$
return $c \leftarrow (c_1 \parallel c_2)$

3.1 Finding Lints

As a strategy for finding lints we consider the public key in all of its occurring forms, each time checking whether all rules are fulfilled and if the properties of an honestly generated key are given. On a first level, we consider the certificate, then secondly the actual public key—in encoded form, after decoding, and lastly fully expanded. We pay special attention to the algorithms that have the public key as input to assess the consequences of a malformed public key.

Next to this, we take into account the input validation checks described in FIPS 203 and existing ML-KEM test vectors discussed in the pqc-forum.[17]

3.2 Formal Description of Lints

Lint Names. Every lint has a unique name, which is necessary to both identify it and offer a mechanism to reference it. This is helpful to keep track of the

[17] `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/aCAX-2QrUFw/m/rLoIz6duCwAJ`

Algorithm 5 $\text{ByteEncode}_d(F)$

Encodes an array of d-bit integers into a byte array, for $1 \leq d \leq 12$.
Input: integer array $F \in \mathbb{Z}_m^{256}$, where $m = 2^d$ if $d < 12$ and $m = q$ if $d = 12$
Output: byte array $B \in \mathbb{B}^{32d}$
for $(i \leftarrow 0; i < 256; i{+}{+})$ **do**
    $a \leftarrow F[i]$
    for $(j \leftarrow 0; j < d; j{+}{+})$ **do**
        $b[i \cdot d + j] \leftarrow a \mod 2$
        $a \leftarrow (a - b[i \cdot d + j])/2$
    end for
end for
$B \leftarrow \text{BitsToBytes}(b)$
return B

Algorithm 6 $\text{ByteDecode}_d(B)$

Decodes a byte array into an array of d-bit integers, for $1 \leq d \leq 12$.
Input: byte array $B \in \mathbb{B}^{32d}$
Output: integer array $F \in \mathbb{Z}_m^{256}$, where $m = 2^d$ if $d < 12$ and $m = q$ if $d = 12$
$b \leftarrow \text{BytesToBits}(B)$
for $(i \leftarrow 0; i < 256; i{+}{+})$ **do**
    $F[i] \leftarrow \sum_{j=0}^{d-1} b[i \cdot d + j] \cdot 2^j \mod m$
end for
return F

lints that have been executed and the result of the linting. The name of the lint consists of a label specifying the severity of a lint failure and a short description of what this lint checks. For our lints the label is always e_ to identify an error, because failing one of our lints always results in an error.

The formal description used in this work follows the notation of zlint which is extensively used by public CAs. On the one hand it is a practical representation of lints and on the other hand—since it is widely used—it facilitates integration of our post-quantum lints into existing processes. The formal description of all our lints is shown in Appendix A.

In addition to its name, each lint has an ID that consists of three parts: first the algorithm is referenced, e.g., ML_DSA. This is followed by the abbreviation of the lint's classification—details on the lint classes can be found in the next paragraph. The last part of the ID is a number, starting at 01 for each combination of scheme and class and incremented by 1 for each additional lint. Additionally to the ID, a citation is always present to point to an external document, usually a specification, from which the lint requirements have been derived and where more context about the lint can be found. Lastly, each lint has a description to briefly describe the requirement on the certificate or public key that is checked by the lint. In this description also a short background on the check is given.

Algorithm 7 SampleNTT(B)

On input a 32-byte seed and two indices, it outputs a pseudorandom element of $\mathbb{Z}_q^{256}$.
Input: byte array $B \in \mathbb{B}^{34}$ ▷ a 32-byte seed along with two indices
Output: array $\hat{a} \in \mathbb{Z}_q^{256}$

$ctx \leftarrow$ XOF.Init()[13]
$ctx \leftarrow$ XOF.Absorb(ctx,B)
j$\leftarrow$0
while $j < 256$ **do**
    $(ctx, C) \leftarrow$ XOF.Squeeze($ctx, 3$)
    $d_1 \leftarrow C[0] + 256 \cdot (C[1] \mod 16)$
    $d_2 \leftarrow \lfloor C[1]/16 \rfloor + 16 \cdot C[2]$
    if $d_1 < q$ **then**
        $\hat{a}[j] \leftarrow d_1$
        $j \leftarrow j + 1$
    end if
    if $d_2 < q$ and $j < 256$ **then**
        $\hat{a}[j] \leftarrow d_2$
        $j \leftarrow j + 1$
    end if
end while
return $\hat{a}$

Lint Classes. We assign each lint to one of five classes. These are: interoperability (INTER), dimension (DIM), domain (DOM), distribution (DIS), and generic (GEN). The class name tells what kind of property of the key the lint checks: The interoperability lints focus on certificate properties which assist applications in properly communicating among each other. The dimension lints test the size of the public key (in encoded form as well as after extraction and expansion), while the domain lints check whether the type of the objects is correct. Examples for this are if the public key matrix has the right number of rows and columns and entries from the correct space. For some of the objects distribution properties are claimed, which are verified by distribution lints. Lastly, lints that can be defined for any scheme are referred to as generic.

Application Lints. While studying lints for ML-KEM and ML-DSA, we realized that not all identified lints can be verified by a CA during the linting process. To explain these lints, we first recapitulate how the certificate request to a public CA and the linting process proceed, as is shown in Figure 1. In the certificate use part of Figure 1, the use of the certificate by anyone is depicted. We call this the application side since an application, like an email client or a browser, extracts the public key from the certificate in order to, e.g., verify a digital signature. In the case of ML-KEM and ML-DSA, apart from extracting the public key from the certificate, the application must perform further operations, like expanding the key. At this particular time, the CA has neither access to the public key, nor can it influence the cryptographic library used by

Algorithm 8 ML-DSA.KeyGen_internal(ξ)

Generates a public-private key pair from a seed.

Input: $\xi \in \mathbb{B}^{32}$

Output: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$, private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen}(2\eta)+dk)}$

▷ DIM_01

1: $\mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \ni (\rho, \rho', K) \leftarrow H(\xi||\text{IntegerToBytes}(k,1)||\text{IntegerToBytes}(l,1), 128)$

▷ DIM_02, DOM_01, DIS_01, DIS_02

2: $\hat{A} \leftarrow \text{ExpandA}(\rho)$

▷ DIM_04, DIS_04, DIS_05, DIS_06

3: $(s_1, s_2) \leftarrow \text{ExpandS}(\rho')$

4: $t \leftarrow \text{NTT}^{-1}(\hat{A} \circ \text{NTT}(s_1)) + s_2$

5: $(t_1, t_0) \leftarrow \text{Power2Round}(t)$ ▷ DIM_03, DOM_02

6: $pk \leftarrow \text{pkEncode}(\rho, t_1)$

7: $tr \leftarrow H(pk, 64)$

8: $sk \leftarrow \text{skEncode}(\rho, K, tr, s_1, s_2, t_0)$

9: **return** (pk, sk)

the application. Therefore, some lints must be performed at the application side (if at all), where the public key is used. We call such a lint an *application lint*. For example, only after expanding the key, the ML_KEM_DIM_03 lint, which checks whether the ML-KEM public key matrix has the correct dimensions, can be performed, cf. Section 4.2. Application lints should also be covered by known answer tests on the cryptographic implementation and should therefore only fail in exceptional cases. They may fail when, e.g., (i) unpacked keys are used, i.e., the keys are never serialized to the format standardized by NIST[18], or (ii) the implementation used by the application is flawed. In the latter case, failure of the application lint does not necessarily imply that the original serialized key used is faulty. This should not be interpreted as a recommendation against using unpacked or seed-only keys. However, it raises a counter point to a popular opinion that using seed-only keys is preferable. While seed-only keys have many advantages, faulty implementations may expand seeds to weak keys. In our implementation, which targets CAs, we do not explicitly implement such application lints. Table 1 indicates which lints are application lints.

Completeness of Lints and Lint Classes. While all of our lints can be sorted into the described classes, we do not exclude the possibility of further lints arising that require the introduction of new classes. Regarding the existing classes, we consider the dimension and domain check lints to be complete, as we give the corresponding checks for all states (encoded, extracted, expanded) of the public key. The same is not true for the distribution lints, which are easily

[18] For ML-KEM, many implementations will use such unpacked keys due to performance reasons. The key format for these is not standardized, which again introduces the potential for invalid keys.

Algorithm 9 ML-DSA.Verify_internal(pk, M', σ)

Internal function to verify a signature σ for a formatted message M'.
Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$, message $M' \in \{0,1\}^*$, signature $\sigma \in \mathbb{B}^{\lambda/4+32\ell(1+\text{bitlen}(\gamma_1-1))+\omega+k}$
Output: Boolean
1: $(\rho, t_1) \leftarrow \text{pkDecode}(pk)$
2: $(\tilde{c}, z, h) \leftarrow \text{sigDecode}(\sigma)$
3: **if** $h = \perp$ **then return** false
4: **end if**
5: $\hat{A} \leftarrow \text{ExpandA}(\rho)$
6: $tr \leftarrow H(pk, 64)$
7: $\mu \leftarrow H(\text{BytesToBits}(tr)||M', 64)$
8: $R_q \ni c \leftarrow \text{SampleInBall}(\tilde{c})$
9: $c \leftarrow \text{SampleInBall}(\tilde{c}_1)$
10: $w'_{\text{Approx}} \leftarrow \text{NTT}^{-1}(\hat{A} \circ \text{NTT}(z) - \text{NTT}(c) \circ \text{NTT}(t_1 \cdot 2^d))$
11: $w'_1 \leftarrow \text{UseHint}(h, w'_{\text{Approx}})$
12: $\tilde{c}' \leftarrow H(\mu||\text{w1Encode}(w'_1), \lambda/4)$
13: **return** $[[||z||_\infty < \gamma_1 - \beta]]$ and $[[\tilde{c} = \tilde{c}']]$

Algorithm 10 pkEncode(ρ, t_1)

Encodes a public key for ML-DSA into a byte string.
Input: $\rho \in \mathbb{B}^{32}$, $t_1 \in R^k$ with coefficients in $[0, 2^{\text{bitlen}(q-1)-d} - 1]$
Output: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$
1: $pk \leftarrow \rho$
2: **for** i **from** 0 **to** $k-1$ **do**
3: $\quad pk \leftarrow pk \,\|\, \text{SimpleBitPack}(t_1[i], 2^{\text{bitlen}(q-1)-d} - 1)$
4: **end for**
5: **return** pk

seen to be incomplete. The kinds of lints we describe to test for randomness do not detect all cases of non-uniformness, e.g., there are no pattern tests. This incompleteness is due to the fact that many known distribution tests are not suited to be used as lints as the latter are supposed to be small and easily verifiable checks. However, we provide a first set of randomness lints that meet these requirements while excluding some obvious cases of non-uniformity. For the remaining classes (interoperability and generic), it is hard to define completeness as they comprise more generic concepts. In general, we want to emphasize that we view our paper as a starting point for post-quantum linting with the objective that more lints can be added over time using the proposed formal description.

3.3 Implementation

Lints are implemented in Java and Rust. Lints which examine the properties of the certificate related to the public key, like the key usage, rather than properties of the public key itself are implemented in Java. This is done using methods provided by BouncyCastle [3], an open-source cryptographic library.

Algorithm 11 pkDecode(pk)

Reverses the procedure pkEncode.
Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$
Output: $\rho \in \mathbb{B}^{32}$, $t_1 \in \mathbb{R}^k$ with coefficients in $[0, 2^{\text{bitlen}(q-1)-d} - 1]$
$\mathbb{B}^{32} \times (\mathbb{B}^{32(\text{bitlen}(q-1)-d)})^k \ni (y_0, z_0, \ldots, z_{k-1}) \leftarrow pk$
for i **from** 0 **to** $k-1$ **do**
    $t_1[i] \leftarrow \text{SimpleBitUnpack}(z_i, 2^{\text{bitlen}(q-1)-d} - 1)$
end for
return (ρ, t_1)

Algorithm 12 ExpandA(ρ)

Samples a $k \times l$ matrix $\hat{A}$ of elements of $\mathbb{Z}_q^{256}$.
Input: $\rho \in \mathbb{B}^{32}$
Output: Matrix $\hat{A} \in (\mathbb{Z}_q^{256})^{k \times l}$
for r **from** 0 **to** $k-1$ **do**
    for s **from** 0 **to** $\ell - 1$ **do**
        $\rho' = \rho || \text{IntegerToBits}(s, 1) || \text{IntegerToBits}(r, 1)$
        $\hat{A}[r, s] \leftarrow \text{RejNTTPoly}(\rho')$
    end for
end for
return $\hat{A}$

Lints that check the correctness of the key within the certificate are implemented in Rust. Rust allows us to enforce the size of inputs on the type level, such that dimension lints get covered by the API and do not need additional checks. The caller needs to ensure that the array is of the correct size and the compilation will fail, or the conversion of the unsized slice to the fixed-sized array will fail even calling the implementation if the key has the wrong size. The lint GEN_02 corresponds to the domain check described in FIPS 203 [18] for ML-KEM and is implemented as such. We further implement the distribution checks on the seed. They are implemented in order and hence fail with the first offending lint.

Listing 1.1 shows how our linting tool can be called from the command line to lint a certificate. The output of this call is given in Listing 1.2 (cf. Appendix B). Some lints (like e_ml_dsa_key_usage) are not applicable (na), since the key contained in the certificate is an ML-KEM key, while all others pass since the public key contained in the certificate is a valid one and does not contain any errors. In our implementation, test vectors for keys with and without errors exist.

Listing 1.1: Calling the command-line tool. It uses Java to start the tool. The certificate mlkem.crt, which contains an ML-KEM public key signed with ML-DSA, is provided. The flag -includeSources PQC is used to include lints only from this paper. Lastly, flag -p provides an easy-to-read output of the tool.

```
java -jar jzlint-cli-X.Y.Z.jar \
mlkem.crt -p -includeSources PQC
```

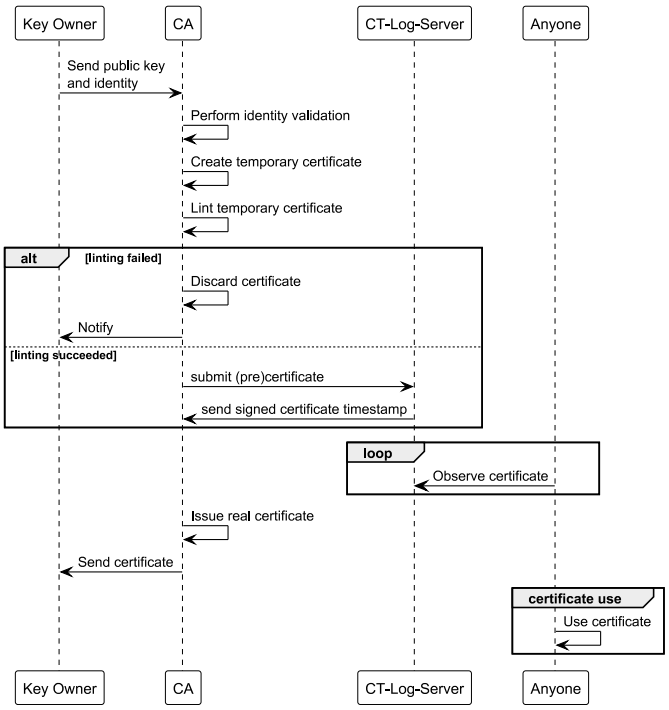


Fig. 1: Simplified certificate request and linting process. The user sends the public key to the CA. If the linting fails, the CA notifies the user to correct the request, especially the error in the public key. If linting is successful, then the certificate is submitted to a certificate transparency (CT) log server. A CT log server holds a large number of certificates and provides public accessibility to them.

The lints for ML-DSA are essentially identical to those for ML-KEM, and their implementation is deferred to future work, as it does not add significant value to the work. The implementation is provided under `https://github.com/MTG-AG/jzlint` and `https://github.com/cryspen/libcrux/`.

4 Lints for ML-KEM

In this section, we present all lints that we have identified for ML-KEM. An overview of the lints is given in Table 1. The formal description of the lints for ML-KEM is provided in Appendix A.

The main sources for the ML-KEM lints are the NIST standard FIPS 203 [18] and the IETF draft [22]. We also take into account known ML-KEM test vectors that are relevant for linting the public keys.[19]

[19] There was a discussion about this topic in the pqc-forum `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/aCAX-2QrUFw/m/rLoIz6duCwAJ`.

While the standard FIPS 203 describes multiple tests for input validation, not all of them are relevant for public key linting. The tests *seed consistency* and *pair-wise consistency* are not applicable in our setting, as a CA does not obtain the randomness (used during key generation) and the secret key, but only the public key. For the same reason, the *ciphertext type check*, *decapsulation key type check*, and *hash check* do not apply. The remaining two tests, the *type check* and the *modulus check* concern the public key and are implemented as lints in our work: The *type check* (ML_KEM_DIM_01) concerns the size of the encoded encapsulation key and is hence assigned to the dimension check lints in Section 4.2. The *modulus check* (ML_KEM_GEN_02) relates to the de- and encoding process and is presented as a generic lint in Section 4.5. The lints related to the IETF draft focus on certificate properties that target interoperability, like the encoding of the algorithms used to identify the cryptographic schemes and parameters or meaningful values for the usage of the key.

In the following sections we mark the lints that we implement, i.e., all lints that are not application lints, by appending the symbol * to the lint identifier.

The Binding Notions. Recently, there have been many discussions about the binding properties of ML-KEM[20] and it was shown that the scheme does not achieve three of the binding notions [11, 20]. For these three notions (and more generally, for the so-called MAL notions), the adversary can maliciously generate key pairs. Depending on the attack, however, a key pair cannot always be recognized as faulty—even given both the public *and* the secret key. We are not aware of binding attacks against ML-KEM that can be solely detected given the public keys (and hence through public key linting). However, if such arise in the future, corresponding lints should be added to the collection below.

4.1 Interoperability Lints

ML_KEM_INTER_01*. The key usage extension [4, Sec. 4.2.1.3] in a certificate specifies for which types of use, like digital signature or encryption of data, the certificate's public key can be used. This lint checks that the values of the key usage extension are compatible with the ML-KEM algorithm. Only values related to key encryption are compatible.

ML_KEM_INTER_02*. This lint checks that the encoding of the algorithm identifier of the ML-KEM public key in a certificate is correct. This means that the object identifier is one of ML-KEM-512, ML-KEM-768, or ML-KEM-1024 and that the parameters of the algorithm are absent.

[20] `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/8cNYhg23B9k/m/KoAO1xmUAAAJ`

Table 1: Lints for ML-KEM extracted from [18, 22]. The first column shows the name of the lint, and column two shows what the lint checks. Column three shows the identifier of the lint, where the prefix ML_KEM_ needs to be added. Column four shows a ✓ if the lint is an application lint and an ✗ otherwise. Column five shows whether the public key is linted in encoded (enc), extracted (extr), or expanded (exp) form. All lints are described formally in Appendix A.1, and their implementation is presented in Section 3.3.

Lint	Classification	Identifier	Application lint?	Form of public key
key usage	interoperability	INTER_01	✗	-
pk aid enc	interoperability	INTER_02	✗	-
ek length	dimension	DIM_01	✗	enc
ek seed length	dimension	DIM_02	✗	enc[a]
ek matrix dimension	dimension	DIM_03	✓	exp
ek vector dimension	dimension	DIM_04	✓	extr
ek matrix entries	domain	DOM_01	✓	exp
ek vector entries	domain	DOM_02	✓	extr
ek seed entry frequency	distribution	DIS_01	✗	enc
ek seed entry run	distribution	DIS_02	✗	enc
ek seed small/large entries	distribution	DIS_03	✗	enc
ek matrix entry frequency	distribution	DIS_04	✓	exp
ek matrix entry run	distribution	DIS_05	✓	exp
ek matrix small/large entries	distribution	DIS_06	✓	exp
known enc key	generic	GEN_01	✗	enc/extr/exp
ek encoding	generic	GEN_02	✗	enc

[a]Note that for ML-KEM the encapsulation key is $\text{ByteEncode}_{12}(\hat{t})||\rho$, i.e., to check properties of the seed ρ it is not necessary to decode.

4.2 Dimension Check Lints

ML_KEM_DIM_01*. This lint checks the length of the encoded encapsulation key ek. The correct length is $384k + 32$ bytes for $k \in \{2, 3, 4\}$ depending on the ML-KEM parameter set (see Algorithm 2). If the encoded encapsulation key is too long, the computation $G(m||H(ek))$ used to derive the shared key K and randomness r (see Algorithm 3, l.1), can become inefficient. If it is too short, the seed ρ and vector $\hat{t}$, contained and extracted, respectively, from the encoded key will not have full length (see Algorithm 4)—possible consequences of this are described in the corresponding application lints.

ML_KEM_DIM_02. This lint checks the length of the seed ρ that is contained in the encoded ML-KEM encapsulation key ek. The seed makes up the last 32 bytes of the encoded encapsulation key, i.e., $\rho \in \mathbb{B}^{32}$ (see Algorithm 2). A too

long seed can slow down the XOF computation in SampleNTT (see Algorithm 7).

The two lints below check the dimensions of the matrix $\hat{A}$ and the vector $\hat{t}$. Depending on the implementation, $\hat{A}$ and $\hat{t}$ of the wrong dimension can impair functionality but also security, as encapsulation and decapsulation key are related via $\hat{A}$ and $\hat{t}$. More precisely $\hat{t} = \hat{A} \circ \hat{s} + \hat{e}$ for e an error term sampled from a centered binomial distribution and $\mathrm{dk} = \mathrm{ByteEncode}_{12}(\hat{s})$ (see Algorithm 2).

ML_KEM_DIM_03. This lint checks the dimensions of the matrix $\hat{A}$, that is obtained by expanding the seed ρ, which is in turn part of the encoded ML-KEM encapsulation key ek. The matrix is computed as $\hat{A}[i,j] = \mathrm{SampleNTT}(\rho||j||i)$ and should have $k \in \{2,3,4\}$ rows and columns (see Algorithm 2).

ML_KEM_DIM_04. This lint checks the size of the vector $\hat{t}$, which is contained in the extracted ML-KEM encapsulation key. The correct length of $\hat{t} = \mathrm{ByteDecode}_{12}(ek[0:384k])$ is k for $k \in \{2,3,4\}$ (see Algorithm 2).

4.3 Domain Check Lints

ML_KEM_DOM_01. This lint checks if expanding the seed ρ, that is part of the encoded ML-KEM encapsulation key ek, yields a matrix with entries over the correct space. More precisely, we consider $\hat{A}$ with $\hat{A}[i,j] = \mathrm{SampleNTT}(\rho||j||i)$ and test whether $\hat{A}[i,j] \in \mathbb{Z}_q^{256}$ for $q = 3329$ (see Algorithm 7).

Depending on the concrete implementation, a matrix $\hat{A}$ that is not reduced modulo q can cause interoperability problems. We give an example, in which encapsulation using $\hat{A}$ will result in a ciphertext c that leads to implicit rejection during decapsulation with overwhelming probability. For this, assume that $\hat{A}$ has entries greater q. During an ML-KEM encapsulation, $\mathrm{NTT}^{-1}(\hat{A}^\top \circ \hat{y})$ is used in the underlying encryption $c = \text{K-PKE.Encrypt}(ek, m, r)$ (see Algorithms 3, 4). Here y is sampled from a centered binomial distribution, hence we can assume $\hat{A}^\top \circ \hat{y}$ to still have entries greater q. Note that we assume that the implementation does not explicitly reduce $\hat{A}^\top \hat{y}$—this is sensible as $\hat{A}^\top \hat{y}$ is directly inputted into NTT^{-1}, which contains a reduction modulo q (see [18, Algorithm 10]). Then during decapsulation, the value $\mathrm{NTT}(\mathrm{NTT}^{-1}(\hat{A}^\top \circ \hat{y}))$ is computed (see [18, Algorithm 15, l.6]), which is reduced modulo q and hence different from $\hat{A}^\top \circ \hat{y}$. However, then $m' = \text{K-PKE.Decrypt}(\mathrm{dk}, c)$ is likely to differ from m. This implies that the re-encrypt step will fail with overwhelming probability, which in turn induces implicit rejection (see [18, Algorithm 18]).

ML_KEM_DOM_02. This lint checks if the vector $\hat{t}$ contained in the extracted ML-KEM public key has entries from the correct space. We test whether $\hat{t}[i] \in \mathbb{Z}_q^{256}$ for $q = 3329$ and $\hat{t} = \mathrm{ByteDecode}_{12}(ek[0:384k])$ (see Algorithm 6).

4.4 Distribution Lints

The following lints concern the distribution of the components contained in (and computed from) the public key, more precisely the seed ρ, the matrix $\hat{A}$, and the vector $\hat{t}$ (see Algorithms 2, 4). As ρ and $\hat{A}$ are assumed to be pseudorandom in FIPS 203 [18], we give three lints that exclude some obvious cases of non-uniformness for the entries of ρ and $\hat{A}$. Note that non-randomness of the public key matrix can negatively influence security. For example, in the extreme case that a sparse matrix is used, hardness of the standard LWE problem is not guaranteed, as is the case if binary matrices are used [10].

To determine the rejection value x for which a lint should output an error, we roughly use the following procedure: We choose the bound $\alpha = 2^{-128}$ and compute x as a value such that for uniformly distributed values, the probability that the lint outputs an error is smaller α. Details on the computation of the rejection values can be found in Appendix C.

ML_KEM_DIS_01*. This lint checks if the seed ρ, that is contained in the encoded ML-KEM encapsulation key ek, contains the same element an amount of times that is unlikely for a pseudorandomly sampled value from $\mathbb{B}^{32}$. The lint outputs an error if there are at least $x = 20$ occurrences of the same element.

ML_KEM_DIS_02*. This lint checks if the seed ρ, that is contained in the encoded ML-KEM encapsulation key ek, contains a long run of the same element, which is unlikely for a pseudorandomly sampled value from $\mathbb{B}^{32}$. An error is outputted if the length of the longest run of the same element has length at least 18.

ML_KEM_DIS_03*. This lint checks if the seed $\rho \in \mathbb{B}^{32}$, that is contained in the encoded ML-KEM encapsulation key ek, has too small or too large entries. An error is outputted if N_s, the number of entries smaller than 128, and N_g, the number of entries greater than 128, fulfill $N_s \geq x$ or $N_g \geq x$ for $x = 29$.

Note that for this lint we cannot achieve a probability smaller 2^{-128} (unless we reject no seeds at all) due to the fact that only 32 elements are drawn. Hence we change the bound α to $\frac{1}{2^{18}}$ to obtain a sensible rejection bound. In particular, this means that this lint will output an error far more often than the other distribution lints. Thus, before deploying the lint, its relevance should be weighed against the disadvantage of more false positives—taking into account the use case at hand. Here, one should be aware that the frequent rejection of valid keys, can weaken the perceived reliability of a CA. In our implementation, lints can be configurably disabled. Thus, one CA can easily exclude this particular lint to avoid rejecting valid keys, while another CA, which is interested in performing this check, may include it.

Remark. Note that during the computation of the matrix $\hat{A}$, the seed ρ is expanded using an extendable-output function XOF (see Algorithm 2). However,

XOF does not have a fixed output length, but will produce more bytes until $\hat{A}$ is filled with entries that have the desired properties (being reduced modulo q, see Algorithm 7). Hence, for seeds ρ such that XOF(ρ) has many large entries, the generation of $\hat{A}$ will take longer. This in turn would influence efficiency of encapsulation and decapsulation in a negative manner.

The following three lints test the uniformness of the entries in the matrix A that is contained in the expanded public key. Note that these lints can only output an error in exceptional cases, e.g., if "weak seeds" for SampleNTT exist. By this, we mean seeds ρ for which the matrix expansion using SampleNTT yields a matrix that is biased in some way—which in turn can enable attacks, as the one for sparse or binary matrices [10]. However, we are not aware of any existing examples of such weak seeds, and consider this as an open question that has to be further analyzed in the future.

ML_KEM_DIS_04*. This lint checks if the matrix $\hat{A}$ that is generated from the seed ρ (contained in the encoded ML-KEM encapsulation key ek) contains each element from $\mathbb{Z}_q$ only a number of times that is likely for a pseudorandomly generated matrix. The matrix $\hat{A}$ is computed as $\hat{A}[i,j] = \text{SampleNTT}(\rho||j||i)$ for $i,j \in \{1,\ldots,k\}$ and $k \in \{2,3,4\}$ and has entries in $\mathbb{Z}_q^{256}$ with $q = 3329$. The lint outputs an error if there are at least x occurrences of the same element from $\mathbb{Z}_q$, where $x \in \{26, 31, 36\}$ (corresponding to the three parameter sets).

ML_KEM_DIS_05*. This lint checks if the matrix $\hat{A}$ that is generated from the seed ρ (contained in the encoded ML-KEM encapsulation key ek) contains a long run of the same element, which is unlikely for a pseudorandomly generated matrix. The matrix $\hat{A}$ is computed as $\hat{A}[i,j] = \text{SampleNTT}(\rho||j||i)$ for $i,j \in \{1,\ldots,k\}$ and $k \in \{2,3,4\}$ and has entries in $\mathbb{Z}_q^{256}$ with $q = 3329$. The lint outputs an error if the length of the longest run[21] of the same element from $\mathbb{Z}_q$ has length at least $x = 13$.

ML_KEM_DIS_06*. This lint checks whether the matrix $\hat{A}$ that is generated from the seed ρ (contained in the encoded ML-KEM encapsulation key ek) has too small or too large entries. The matrix $\hat{A}$ is computed as $\hat{A}[i,j] = \text{SampleNTT}(\rho||j||i)$ for $i,j \in \{1,\ldots,k\}$ and $k \in \{2,3,4\}$ and has entries in $\mathbb{Z}_q^{256}$. Note that we can consider $\hat{A}$ to be a $(256k \times 256k)$-matrix over $\mathbb{Z}_q$. The lint outputs an error if the number of entries N_s smaller than $\frac{q}{2}$ and then the number N_g of ones greater than $\frac{q}{2}$ fulfill $N_s \geq x$ or $N_g \geq x$ for $x \in \{720, 1466, 2467\}$ (depending on the parameter set).

[21] Note that the ordering of the elements in the matrix should coincide with the order in which the entries of $\hat{A}$ are generated during the key generation.

4.5 Generic Lints

ML_KEM_GEN_01*. The lint checks for public keys of publicly known key pairs, e.g., key pairs that are published for demonstration purposes (in standards, specifications, etc.) or stem from key leaks. Public keys of such key pairs should not be used in practice, as the corresponding secret keys are publicly available.

ML_KEM_GEN_02*. This lint checks if first decoding and then encoding the encapsulation key ek will give back the same ek.[22] First, we observe that this is not automatically given [18, Section 7.2]: ByteDecode_{12} takes a byte array of length 384 as input and computes from each 12-bit segment of this input an integer smaller than $2^{12} = 4096$. The resulting values are reduced modulo $q = 3329$ and ByteDecode_{12} outputs the corresponding element in $(\mathbb{Z}_q)^{256}$. Now assume the input B to ByteDecode_{12} is chosen in a way such that at least one of its 12-bit segments will yield a value $3329 \leq x < 4096$. Note that this can easily be achieved by setting $B = (255, 255, \ldots, 255)$. However, such an output cannot be produced by ByteEncode_{12}, i.e., $\text{ByteEncode}_{12}(\text{ByteDecode}_{12}(B)) \neq B$. Moreover, in this case the input B cannot be the result of an honest ML-KEM key generation as first applying ByteEncode_{12} and then ByteDecode_{12} is the identity mapping. Thus, this test is of great importance, as it can identify encapsulation keys that might have the correct length, but still do not stem from an honest key generation.

5 Lints for ML-DSA

In this section, we present all lints that we have identified for ML-DSA. An overview of the lints is given in Table 2, and their formal description is provided in Appendix A.2. The main sources for the ML-DSA lints are the NIST standard FIPS 204 [17] and the IETF draft [15]. Note that FIPS 204 requires a length check for public keys and signatures. We implement the former as a dimension check lint. Like in the case of ML-KEM, the lints related to the IETF draft focus on certificate properties that target interoperability. In the following sections we mark the lints that we implement, i.e., all lints that are not application lints, by appending the symbol * to the lint identifier.

The Buff Notions. Note that ML-DSA was proven to achieve the so-called BUFF notions, which formalize security against maliciously generated public keys for signature schemes [7]. While these notions exclude four attack scenarios, there are still many basic ways in which the public keys can be malformed that are not covered and can enable further attacks (like for sparse matrices as described in Section 4).

[22] Note that for ML-DSA no such lint is necessary as pkEncode can hit the full output space, which equals the input space of pkDecode. In contrast for ML-KEM, one can input values into ByteDecode_{12} that are never outputted by ByteEncode_{12}.

5.1 Interoperability Lints

ML_DSA_INTER_01*. This lint checks that the values of the key usage extension are compatible with the ML-DSA algorithm. Only values related to signatures are compatible.

ML_DSA_INTER_02*. This lint checks that the encoding of the algorithm identifier of the ML-DSA public key in a certificate is correct. This means that the object identifier is the one identifying an ML-DSA public key and that the parameters of the algorithm are absent.

ML_DSA_INTER_03*. This lint checks that the encoding of the signature algorithm in a certificate signed by an ML-DSA private key is correct. This means that the object identifier is one of ML-DSA-44, ML-DSA-65, or ML-DSA-87 and that the parameters of the algorithm are absent.

5.2 Dimension Check Lints

ML_DSA_DIM_01*. This lint checks the length of the encoded ML-DSA public key pk, i.e., checks if $\text{pk} \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$ for $\mathbb{B} = \{0, 1, \ldots, 255\}$, $q = 8380417$, $d = 13$, and $k \in \{4, 6, 8\}$ (see Algorithm 8).[23] A too short public key will result in a seed ρ and vector t_1 of incorrect length after extraction (see Algorithm 11).

ML_DSA_DIM_02. This lint checks the length of the seed ρ which is part of the extracted ML-DSA public key. More precisely, the seed is obtained as $(\rho, t_1) = \text{pkDecode}(\text{pk})$ and it is verified whether $\rho \in \mathbb{B}^{32}$ (see Algorithm 11). For a too long seed ρ, the hashing done in the subalgorithm RejNTTPoly of ExpandA (Algorithm 12) can become inefficient.

The following two lints check the dimension of the matrix A and the vector t_1. Note that the private and public key are related via $t = As_1 + s_2$ for s_1 and s_2 two vector with short coefficients, that are part of the private key (see Algorithm 8). Hence, wrong dimensions of A and t_1 can impair functionality as well as security—depending on the concrete implementation.

ML_DSA_DIM_03. This lint checks the size of t_1, which is contained in the extracted ML-DSA encapsulation key. It is computed as $(\rho, t_1) = \text{pkDecode}(\text{pk})$ and the lint verifies if $t_1 \in R^k$ is a vector of length k for $k \in \{4, 6, 8\}$ (see Algorithm 11).

[23] Note that [15] defines $\text{pk} \in \mathbb{B}^{32+320k}$, which matches the description from [17] that is used in this lint as $32 + 32k(\text{bitlen}(q-1) - d = 32 + 32k \cdot 10 = 32 + 320k$ for $q = 8380417$, $\text{bitlen}(q-1) = 23$ and $d = 13$.

Table 2: Lints for ML-DSA extracted from [15, 17]. The first column shows the name of the lint, and column two shows what the lint checks. Column three shows the identifier of the lint, where the prefix ML_DSA_ needs to be added. Column four shows a ✓ if the lint is an application lint and an ✗ otherwise. Column five shows whether the public key is linted in encoded (enc), extracted (extr), or expanded (exp) form. All lints are described formally in Appendix A.2.

Lint	Classification	Identifier	Application lint?	Form of public key
key usage	interoperability	INTER_01	✗	-
pk aid enc	interoperability	INTER_02	✗	-
signature aid enc	interoperability	INTER_03	✗	-
pk length	dimension	DIM_01	✗	enc
pk seed length	dimension	DIM_02	✓	extr[a]
pk vector dimension	dimension	DIM_03	✓	extr
pk matrix dimension	dimension	DIM_04	✓	exp
pk matrix entries	domain	DOM_01	✓	exp
pk vector	domain	DOM_02	✓	extr
pk seed entry frequency	distribution	DIS_01	✓	extr
pk seed entry run	distribution	DIS_02	✓	extr
pk seed small/large entries	distribution	DIS_03	✓	extr
pk matrix entry frequency	distribution	DIS_04	✓	exp
pk matrix entry run	distribution	DIS_05	✓	exp
pk matrix small/large entries	distribution	DIS_06	✓	exp
known enc key	generic	GEN_01	✗	enc/extr/exp

[a]Note that for ML-DSA the public key is $\mathsf{pkEncode}(\rho, t_1)$, i.e., to check properties of the seed ρ we need to consider the extracted public key.

ML_DSA_DIM_04. This lint checks if expanding the seed ρ, that is part of the extracted ML-DSA public key, yields a matrix of the correct dimensions. More precisely, the matrix $\hat{A} \coloneqq \mathrm{ExpandA}(\rho)$ needs to have $k \in \{4, 6, 8\}$ rows and $l \in \{4, 5, 7\}$ columns (see Algorithm 12).

5.3 Domain Check Lints

ML_DSA_DOM_01. This lint checks if expanding the seed ρ, that is part of the extracted ML-DSA public key, yields a matrix with entries over the correct space. More precisely, one considers $\hat{A} = \mathrm{ExpandA}(\rho)$ and checks if $\hat{A}[i, j] \in \mathbb{Z}_q^{256}$ with $q = 8380417$ (see Algorithm 12).

ML_DSA_DOM_02. This lint checks if t_1 contained in the extracted ML-DSA public key is from the correct space. More precisely, we consider $(\rho, t_1) = \mathsf{pkDecode}(pk)$ and test whether $t[i] \in R$ (see Algorithm 11).

Note that a vector t_1 with entries over $\mathbb{Z}[X]$ that are not reduced modulo $(X^n + 1)$ can lead to severe consequences of the kind described in lint ML_KEM_DOM_01. The main problem is that $\text{NTT}^{-1}(\text{NTT}(t_1))$ will not equal t_1 for a non-reduced vector. However, this is necessary for signing and verifying to work properly (see Algorithm 9, l.10).[24]

5.4 Distribution Lints

The following lints test the randomness of the public key components A, t_1, and ρ (see Algorithms 8, 9). We follow the same convention as described in Section 4.4 regarding the rejection values. Further, the consequences regarding non-random matrices A given there, are also relevant for this section.

ML_DSA_DIS_01*. This lint checks if the seed ρ (contained in the extracted ML-DSA public key) contains the same element an amount of times that is unlikely for a pseudorandomly sampled value from $\mathbb{B}^{32}$. Note that the seed is obtained via $(\rho, t_1) = \text{pkDecode}(pk)$ (see Algorithm 11). The lint outputs an error if there are at least $x = 20$ occurrences of the same element.

ML_DSA_DIS_02*. This lint checks if the seed ρ (contained in the extracted ML-DSA public key) contains consecutive occurrences of the same element, which is unlikely for a pseudorandomly sampled value from $\mathbb{B}^{32}$. Note that the seed is obtained via $(\rho, t_1) = \text{pkDecode}(pk)$ (see Algorithm 11). The lint outputs an error if an element occurs consecutively more than $x = 18$ times.

ML_DSA_DIS_03*. This lint checks if the seed ρ (contained in the extracted ML-DSA public key) has too small or too large entries. An error is outputted if the number of entries N_s smaller than 128 and the number N_g of ones greater than 128 fulfill $N_s \geq x$ or $N_g \geq x$ for $x = 29$.

Note that this lint (as well as the two preceding ones) completely agree with the corresponding lints for ML-KEM as for both schemes $\rho \in \mathbb{B}^{32}$. Thus, the disclaimer concerning α in ML_KEM_DIS_03 must also be taken into account for ML_DSA_DIS_03.

Remark. During the computation of the matrix $\hat{A}$ from the seed ρ with ExpandA (see Algorithm 12), the algorithm RejNTTPoly is used to compute the entries of $\hat{A}$ [17, Algorithm 30]. RejNTTPoly in turn calls the algorithm CoefFromThreeBytes on (three bytes of) SHAKE128(ρ) [17, Algorithm 14], which outputs $\perp$ if the final result is greater than q—this case will occur for seeds ρ such that SHAKE128(ρ) has large entries. Then, RejNTTPoly will execute

[24] In contrast to ML-KEM, $\text{NTT}(\text{NTT}^{-1}(\hat{A}))$ is not computed in ML-DSA, hence the problem regarding unreduced matrix entries described in ML_KEM_DOM_01 cannot occur here.

CoefFromThreeBytes again (using the next three bytes of SHAKE128(ρ) and this process is continued until suitable values are found. Thus, seeds ρ for which SHAKE128(ρ) has large values, will lead to slowed down signing and verifying as both algorithms deploy ExpandA.

ML_DSA_DIS_04*. This lint checks if the matrix $\hat{A}$ that is generated from the seed ρ (contained in the extracted ML-DSA public key) contains elements more frequently than is likely for a pseudorandomly generated $k \times l$-matrix over $\mathbb{Z}_q^{256}$, for $k \in \{4, 6, 8\}$, $l \in \{4, 5, 7\}$, and $q = 8380417$. Note that the matrix is computed as $\hat{A} \coloneqq \text{ExpandA}(\rho)$ (see Algorithm 12). The lint outputs an error if there are at least $x \in \{10, 11, 12\}$ (corresponding to the three parameter set) occurrences of the same element from $\mathbb{Z}_q$.

ML_DSA_DIS_05*. This lint checks if the matrix $\hat{A}$ that is generated from the seed ρ (contained in the extracted ML-DSA public key) contains a long run of the same element, which is unlikely for a pseudorandomly $k \times l$-matrix over $\mathbb{Z}_q^{256}$, for $k \in \{4, 6, 8\}$, $l \in \{4, 5, 7\}$, and $q = 8380417$. The matrix is computed as $\hat{A} \coloneqq \text{ExpandA}(\rho)$ (see Algorithm 12). The lint outputs an error if the length of the longest run[25] of the same element from $\mathbb{Z}_q$ has length at least 8.

ML_DSA_DIS_06*. This lint checks whether the matrix $\hat{A}$ that is generated from the seed ρ (contained in the extracted ML-DSA public key) has too small or too large entries. $\hat{A}$ is a $k \times l$-matrix over $\mathbb{Z}_q^{256}$, where $k \in \{4, 6, 8\}$, $l \in \{4, 5, 7\}$, and $q = 8380417$ is computed as $\hat{A} \coloneqq \text{ExpandA}(\rho)$ (see Algorithm 12). Note that $\hat{A}$ can also be considered to be a $(256k \times 256l)$-matrix over $\mathbb{Z}_q$. We check if the number of matrix entries smaller $\frac{q}{2}$ and the number of ones greater $\frac{q}{2}$ differ in a way that is unlikely for a pseudorandom matrix over $\mathbb{Z}_q$. The lint outputs an error if the number of entries N_s smaller than $\frac{q}{2}$ and then the number N_g of ones greater than $\frac{q}{2}$ fulfill $N_s \geq x$ or $N_g \geq x$ for $x \in \{2467, 4414, 7953\}$ (corresponding to the three parameter sets).

5.5 Generic Lints

ML_DSA_GEN_01*. This lint checks for public keys of publicly known key pairs, e.g. key pairs that are published in standards, specs etc. for demonstration purposes or stem from key leaks.

Public keys of such key pairs should not be used in practice, as the corresponding secret keys are publicly available.

[25] Note that the ordering of the elements in the matrix should coincide with the order in which the entries of $\hat{A}$ are generated during the key generation.

6 Conclusion

In this paper, we initiated the study of lints for PQC and gave the first lints for the lattice-based standards ML-KEM and ML-DSA. Additionally, we provided a general framework consisting of formal descriptions and a classification for lints.

While linting is not novel, applying it to post-quantum schemes is. From this, challenges that did not occur when linting classical schemes arise. One of them is due to the fact that for post-quantum schemes, security is often related to certain elements looking random or following a pre-defined distribution. While there are also vulnerabilities in classical schemes that are due to badly distributed elements (e.g., the RSA primes p and q being too close), such issues occur even more predominantly in post-quantum schemes. Hence, finding ways to check distribution properties using lints, i.e., small and easily verifiable checks, is one of the challenges that has to be met when linting post-quantum schemes. Therefore, we introduced the class of distribution lints and equipped it with first lints for ML-KEM and ML-DSA. However, these lints do not detect all cases of non-uniformness and further lints should be added in the future.

Another hurdle stems from the usage of seeds, as is also done in ML-KEM and ML-DSA. Instead of saving the public key in the form that is used for computations during the verification of signatures, only the seed from which this full version can be expanded, is saved. Hence, properties of the expanded public key cannot be checked by the CA when seeds are used, and the correctness of the expanded public key depends on the implementation of the expansion algorithm used by the party expanding the seed. In practice, different or even faulty implementations of the expanding functions might be used, which can lead to different results when expanding the same seed, and thereby induce, among others, interoperability problems. Thus, the usage of seeds requires testing of the implementations used for expanding, which lies outside the scope of linting, which is the responsibility of the CA. More generally, the decision between using seeds or "full" keys relates to a trade-off between implementation testing and key linting.

Since linting heavily depends on scheme-specific details, one cannot give a complete set of lints covering all lattice-based or even post-quantum schemes, both now and in the future. Instead, with the ongoing standardization and increased usage of PQC, dedicated analyses of the relevant schemes have to be carried out. The lints we gave for ML-KEM and ML-DSA serve as a starting point and example for future analyses, and the provided framework guides the introduction of further lints in the future.

References

1. Martin R. Albrecht. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 103–129. Springer, Heidelberg, April / May 2017. 2
2. Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part I*, volume 6755 of *LNCS*, pages 403–415. Springer, Heidelberg, July 2011. 2
3. Bouncy Castle – Open-source cryptographic APIs. `https://www.bouncycastle.org/`. 13
4. Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008. 16
5. Peter Bowen. certlint. `https://github.com/amazon-archives/certlint`. GitHub repository. 3
6. Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. Provably weak instances of ring-LWE revisited. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 147–167. Springer, Heidelberg, May 2016. 2
7. Cas Cremers, Samed Düzlü, Rune Fiedler, Marc Fischlin, and Christian Janson. BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures. In *2021 IEEE Symposium on Security and Privacy*, pages 1696–1714. IEEE Computer Society Press, May 2021. 21
8. DigiCert. pkilint. `https://github.com/digicert/pkilint`. GitHub repository. 3
9. Yara Elias, Kristin E. Lauter, Ekin Ozman, and Katherine E. Stange. Provably weak instances of ring-LWE. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 63–92. Springer, Heidelberg, August 2015. 2
10. Gottfried Herold and Alexander May. LP solutions of vectorial integer subset sums — cryptanalysis of Galbraith's binary matrix LWE. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 3–15. Springer, Heidelberg, March 2017. 19, 20
11. Juliane Krämer, Patrick Struck, and Maximiliane Weishäupl. Binding security of implicitly-rejecting KEMs and application to BIKE and HQC. Cryptology ePrint Archive, Paper 2024/1233, 2024. 16
12. Deepak Kumar, Zhengping Wang, Matthew Hyder, Joseph Dickinson, Gabrielle Beck, David Adrian, Joshua Mason, Zakir Durumeric, J. Alex Halderman, and Michael Bailey. Zlint: Tracking certificate misissuance in the wild. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018. 3
13. Kurt Roeckx. x509lint. `https://github.com/kroeckx/x509lint`. GitHub repository. 3
14. Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. 2
15. Jake Massimo, Panos Kampanakis, Sean Turner, and Bas Westerbaan. Internet X.509 Public Key Infrastructure: Algorithm Identifiers for ML-DSA. Internet-Draft draft-ietf-lamps-dilithium-certificates-03, Internet Engineering Task Force, February 2024. Work in Progress. 21, 22, 23, 30, 31

16. MTG AG. JZLint. `https://github.com/MTG-AG/jzlint`. GitHub repository. 3
17. National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard. `https://doi.org/10.6028/NIST.FIPS.204`, 2024. 2, 5, 7, 21, 22, 23, 24, 31, 32
18. National Institute of Standards and Technology. Module-Lattice-based Key-Encapsulation Mechanism Standard. `https://doi.org/10.6028/NIST.FIPS.203`, 2024. 2, 5, 6, 14, 15, 17, 18, 19, 21, 28, 29, 30
19. Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Dr. Carlisle Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960, June 2013. 3
20. Sophie Schmieg. Unbindable Kemmy Schmidt: ML-KEM is neither MAL-BIND-k-CT nor MAL-BIND-k-PK. Cryptology ePrint Archive, Paper 2024/523, 2024. 16
21. Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. 2
22. Sean Turner, Panos Kampanakis, Jake Massimo, and Bas Westerbaan. Internet X.509 Public Key Infrastructure - Algorithm Identifiers for Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM). Internet-Draft draft-ietf-lamps-kyber-certificates-03, Internet Engineering Task Force, March 2024. Work in Progress. 15, 17, 28
23. Wycheproof. `https://github.com/C2SP/wycheproof`. GitHub repository. 3
24. ZLint. `https://github.com/zmap/zlint`. GitHub repository. 3

A Formal Description of Lints

A.1 ML-KEM Lints

Name	e_ml_kem_key_usage
ID	ML_KEM_INTER_01
Citation	[22, Sec. 4]
Description	A certificate with an ML-KEM public key must only have the keyEncipherment key usage value.

Name	e_ml_kem_public_key_aid_encoding
ID	ML_KEM_INTER_02
Citation	[22, Sec. 4]
Description	The algorithm identifier in the public key of a certificate with an ML-KEM public key must have the correct encoding.

Name	e_ml_kem_ek_length
ID	ML_KEM_DIM_01
Citation	[18]
Description	An encoded ML-KEM encapsulation key must be of the correct length.

Name	e_ml_kem_ek_seed_length
ID	ML_KEM_DIM_02
Citation	[18]
Description	The seed contained in the ML-KEM encapsulation key must be of the correct length.

Name	e_ml_kem_ek_matrix_dimension
ID	ML_KEM_DIM_03
Citation	[18]
Description	The matrix expanded from the seed, that is contained in the ML-KEM encapsulation key, must have the correct dimensions.

Name	e_ml_kem_ek_vector_dimension
ID	ML_KEM_DIM_04
Citation	[18]
Description	The vector contained in the extracted ML-KEM encapsulation key, must have the correct dimension.

Name	e_ml_kem_ek_matrix_entries
ID	ML_KEM_DOM_01
Citation	[18]
Description	The matrix expanded from the seed, that is contained in the ML-KEM encapsulation key, must have entries from the correct space.

Name	e_ml_kem_ek_vector_entries
ID	ML_KEM_DOM_02
Citation	[18]
Description	The vector contained in the extracted ML-KEM encapsulation key, must have entries from the correct space.

Name	e_ml_kem_ek_seed_entry_frequency
ID	ML_KEM_DIS_01
Citation	[18]
Description	The seed contained in the ML-KEM encapsulation key must not have too many occurrences of the same element.

Name	e_ml_kem_ek_seed_entry_run
ID	ML_KEM_DIS_02
Citation	[18]
Description	The seed contained in the ML-KEM encapsulation key must not have a too long run of the same element.

Name	e_ml_kem_ek_seed_sl_entries
ID	ML_KEM_DIS_03
Citation	[18]
Description	The seed contained in the ML-KEM encapsulation key must not have too small or too large entries.

Name	e_ml_kem_ek_matrix_entry_frequency
ID	ML_KEM_DIS_04
Citation	[18]
Description	The matrix expanded from the seed, that is contained in the ML-KEM encapsulation key must not have too many occurrences of the same element.

Name	e_ml_kem_ek_matrix_entry_run
ID	ML_KEM_DIS_05
Citation	[18]
Description	The matrix expanded from the seed, that is contained in the ML-KEM encapsulation key must not have a too long run of the same element.

Name	e_ml_kem_ek_matrix_sl_entries
ID	ML_KEM_DIS_06
Citation	[18]
Description	The matrix expanded from the seed, that is contained in the ML-KEM encapsulation key must not have too small or too large entries.

Name	e_known_encoded_key
ID	ML_KEM_GEN_01
Citation	Databases, documents etc.,containing public keys whose private key is suspected to be known.
Description	A public key whose corresponding private key is known to be compromised, is weak, or is leaked must not be placed in a certificate.

Name	e_ml_kem_ek_encoding
ID	ML_KEM_GEN_02
Citation	[18]
Description	An ML-KEM encryption key must be correctly encoded.

A.2 ML-DSA Lints

Name	e_ml_dsa_key_usage
ID	ML_DSA_INTER_01
Citation	[15, Sec. 5]
Description	A certificate with an ML-DSA public key must have only one or more of the following key usage values: digitalSignature, nonRepudiation, keyCertSign, cRLSign.

Name	e_ml_dsa_public_key_aid_encoding
ID	ML_DSA_INTER_02
Citation	[15, Sec. 4]
Description	The algorithm identifier in the public key of a certificate with an ML-DSA public key must have the correct encoding.

Name	e_ml_dsa_signature_aid_encoding
ID	ML_DSA_INTER_03
Citation	[15, Sec. 4]
Description	The algorithm identifier in the signature of a certificate signed by an ML-DSA public key must have the correct encoding.

Name	e_ml_dsa_pk_length
ID	ML_DSA_DIM_01
Citation	[17]
Description	An encoded ML-DSA public key must be of the correct length.

Name	e_ml_dsa_pk_seed_length
ID	ML_DSA_DIM_02
Citation	[17]
Description	The seed contained in the extracted public key must be of the correct length.

Name	e_ml_dsa_pk_vector_dimension
ID	ML_DSA_DIM_03
Citation	[17]
Description	The vector contained in the extracted ML-DSA public key must have the correct dimension.

Name	e_ml_dsa_pk_matrix_dimension
ID	ML_DSA_DIM_04
Citation	[17]
Description	The matrix expanded from the seed, that is contained in the extracted ML-DSA public key, must have the correct dimensions.

Name	e_ml_dsa_pk_matrix_entries
ID	ML_DSA_DOM_01
Citation	[17]
Description	The matrix expanded from the seed, that is contained in the extracted ML-DSA public key, must have entries from the correct space.

Name	e_ml_dsa_pk_vector
ID	ML_DSA_DOM_02
Citation	[17]
Description	The vector contained in the extracted ML-DSA public key, must have entries from the correct space.

Name	e_ml_dsa_pk_seed_entry_frequency
ID	ML_DSA_DIS_01
Citation	[17]
Description	The seed contained in the extracted ML-DSA public key must not have too many occurrences of the same element.

Name	e_ml_dsa_pk_seed_entry_run
ID	ML_DSA_DIS_02
Citation	[17]
Description	The seed contained in the extracted ML-DSA public key must not have a too long run of the same element.

Name	e_ml_dsa_pk_seed_sl_entries
ID	ML_DSA_DIS_03
Citation	[17]
Description	The seed contained in the extracted ML-DSA public key must not have too small or too large entries.

Name	e_ml_dsa_pk_matrix_entry_frequency
ID	ML_DSA_DIS_04
Citation	[17]
Description	The matrix expanded from the seed, that is contained in the extracted ML-DSA public key must not have too many occurrences of the same element.

Name	e_ml_dsa_pk_matrix_entry_run
ID	ML_DSA_DIS_05
Citation	[17]
Description	The matrix expanded from the seed, that is contained in the extracted ML-DSA public key must not have a too long run of the same element.

Name	e_ml_dsa_pk_matrix_sl_entries
ID	ML_DSA_DIS_06
Citation	[17]
Description	The matrix expanded from the seed, that is contained in the extracted ML-DSA public key must not have too small or too large entries.

Name	e_known_encoded_key
ID	ML_DSA_GEN_01
Citation	Databases, documents etc.,containing public keys whose private key is suspected to be known.
Description	A public key whose corresponding private key is known to be compromised, is weak, or is leaked must not be placed in a certificate.

B Postponed Listing

Listing 1.2: Output of the tool in JSON format. Each key is the name of the lint and in the result key the result of the linting process for this lint is given. For example, the public key in the certificate passed the lint e_ml_kem_ek_seed_entry_run.


```
{
        "e_ml_kem_ek_encoding" : {
                "result" : "pass"
        },
        "e_ml_kem_ek_seed_entry_run" : {
                "result" : "pass"
        },
        "e_ml_kem_ek_length" : {
                "result" : "pass"
        },
        "e_ml_dsa_signature_aid_encoding" : {
                "result" : "pass"
        },
        "e_known_encoded_key" : {
                "result" : "pass"
        },
        "e_ml_dsa_key_usage" : {
                "result" : "na"
        },
        "e_ml_dsa_public_key_aid_encoding" : {
                "result" : "na"
        },
        "e_ml_kem_ek_seed_sl_entries" : {
                "result" : "pass"
        },
        "e_ml_kem_ek_seed_entry_frequency" : {
                "result" : "pass"
        },
        "e_ml_kem_key_usage" : {
                "result" : "pass"
        },
        "e_ml_kem_public_key_aid_encoding" : {
                "result" : "pass"
        }
}
```


C Computation of Rejection Values

For all lints described below, we want to determine the rejection value x in a way such that for uniformly sampled values, the probability that the lint outputs an error is smaller $\alpha = 2^{-128}$.

C.1 Entry-Frequency-Lints

These lints check whether a string of length n with entries from a set M of size m, contains the same element an amount of times that is unlikely for pseudo-randomly sampled values.

Let E denote the event that the lint outputs an error, i.e., there is some element that appears at least x times. Further, for each $\mu \in M$, consider the event E_μ that the value μ appears at least x times. Then, for a uniformly sampled string, we obtain the following upper bound

$$\begin{aligned}\Pr(E) &= \Pr\left(\bigcup_\mu E_\mu\right) \leq \sum_\mu \Pr(E_\mu) \\ &= \sum_\mu \left(\sum_{i=x}^{n} \binom{n}{i} \left(\frac{1}{m}\right)^i \left(1-\frac{1}{m}\right)^{n-i}\right) \\ &= m \cdot \underbrace{\left(\sum_{i=x}^{n} \binom{n}{i} \left(\frac{1}{m}\right)^i \left(1-\frac{1}{m}\right)^{n-i}\right)}_{=:P} .\end{aligned}$$

For concrete values for n and m, this formula allows to determine x such that $P \leq \frac{1}{m} \cdot \alpha$, which yields the desired rejection value.

C.2 Entry-Run-Lints

These lints check whether a string of length n with entries from a set M of size m, contains a long run (i.e., consecutive occurences) of the same element, which is unlikely for pseudorandomly sampled values.

Let E denote the event that the lint outputs an error, i.e., that the longest run has length at least x. Further, for each $i \in \{1, \ldots, n\}$, consider the event E_i that, starting in position i of the string, there is a run of length at least x. Note that $\Pr(E_i) = m \cdot \left(\frac{1}{m}\right)^x = \frac{1}{m^{x-1}}$ as there are m possibilities for the element in the run, and the probability that this element appears at x chosen positions is $\left(\frac{1}{m}\right)^x$.[26] Then, for a uniformly sampled string, we obtain

$$\begin{aligned}\Pr(E) &= \Pr\left(\bigcup_i E_i\right) \leq \sum_i \Pr(E_i) \\ &\leq \sum_i \frac{1}{m^{x-1}} = \frac{n}{m^{x-1}} .\end{aligned}$$

Thus, for $x \geq \log_m(n2^{128}) + 1$, it holds that $\Pr(E) \leq \alpha$, which yields the desired rejection value.

[26] Note that this computation holds only if a run of length x starting at positions i is possible, otherwise the probability is 0.

C.3 Small/Large-Entries-Lints

These lints check whether a string of length n with entries from $M = \{1, \ldots, m\}$, has too small or too large entries—which is unlikely to hold for pseudorandomly sampled values.

Denote by N_s the number of entries that belong to the first $\lfloor \frac{m}{2} \rfloor$ elements of M and by N_g the number of ones that belong to the last $\lfloor \frac{m}{2} \rfloor$ elements of M.[27] Let E denote the event that the lint outputs an error, i.e., that $N_s \geq x$ or $N_g \geq x$. Note that this condition can be reformulated into $x \leq N_s$ or $N_s \leq n - x$ holding. Then, for a uniformly sampled string, we obtain

$$\Pr(E) = \left(\sum_{i=x}^{n} \binom{n}{i} \left(\frac{1}{2}\right)^{i} \left(1 - \frac{1}{2}\right)^{n-i} \right) + \left(\sum_{i=0}^{n-x} \binom{n}{i} \left(\frac{1}{2}\right)^{i} \left(1 - \frac{1}{2}\right)^{n-i} \right) .$$

For concrete values for n and m, this formula allows to determine x such that $\Pr(E) \leq \alpha$, which yields the desired rejection value.

[27] We assume wlog that m is even as the odd case can be reduced to the even case using the estimate $\Pr(E \text{ in the odd case}) \leq \Pr(E \text{ in the even case})$.