Formally Verified Correctness Bounds for Lattice-Based Cryptography

EC Frodo KEM Team
Formosa Crypto + CQT

***Abstract*—The abstract goes here.**

1. Introduction

The transition to post-quantum cryptography (PQC) has seen an important development in 2024 with the publication of the first PQC standards FIPS-203, FIPS-204 and FIPS-204 by NIST []. The standardised algorithms are called, ML-KEM, ML-DSA and SLH-DSA, and they essentially match the Kyber, Dilithium and SPHINCS+ submissions with minor changes, respectively. These algorithms will see large-scale deployment in the near future in many practical applications as mitigation for the potential arrival of a quantum computer. Key Encapsulation Mechanisms (KEM), such as ML-KEM are arguably the most critical components in the PQC transition, as they guarantee protection against so-called *harvest now, decrypt later* attacks with respect to which data exchanged today is already at risk. For this reason, ML-KEM is already being deployed by software giants Apple, Google and Amazon [], and the number of deployed implementations is expected to grow fast in the near future. Another competitor in the NIST PQC competition for KEMs is called FrodoKEM. Although not selected by NIST for standardisation, FrodoKEM's conservative design—its security is based on the standard Learning With Errors (LWE) assumption, rather than the Module LWE (MLWE) assumption used by ML-KEM—has led to endorsement of entities such as the German Bundesamt für Sicherheit in der Informationstechnik (BSI) for adoption in the transition to PQC.

Widely deployed cryptographic (de facto) standards such as ML-KEM and FrodoKEM will be critical security components in the ITC infrastructure of the coming decades, and so it is crucial that their design is validated to the highest level of assurance. For ML-KEM, several recent works have looked at formally verifying both the design and efficient implementation of the standard. In particular, Almeida et al. [**?**], [**?**] presented formally verified proofs of cryptographic security (IND-CCA) and correctness—the guantee that decapsulation inverts encapsulation—in EasyCrypt. Alternative proofs of IND-CPA security and correctness were given by Kreuzer et al. [] in Isabelle. However, in both of these works, there is one aspect of the security and correctness claims that support the ML-KEM design that is not formally verified: the concrete values for the probability of a failed decryption. Both works account for the probability of a failed decryption by defining a statistical event and proving that bounding the probability of such an event yields an upper bound for decryption failure. But neither work provides a means to compute or even upper-bound this concrete probability to a high-level of assurance. In this paper we address this gap. We begin by recalling the importance of decryption errors in post-quantum KEM security.

The importance of decryption errors. Unlike Diffie-Hellman and RSA-based constructions, which typically yield perfectly correct cryptographic constructions, lattice-based constructions often allow for a low probability of error in order to optimize the compromise between security and performance. One might think that a decryption error would represent only an inconvenience for practical applications, e.g., in that it would cause message transmission to sometimes fail. However, it is well known that, when freely exposed to an adversary, decryption errors can lead to devastating attacks on lattice-based constructions [**?**], [**?**], [**?**]. Put differently, lattice-based KEM constructions such as ML-KEM and FrodoKEM are supported by IND-CCA security proofs where the overall bound on an attacker's advantage in breaking the KEM must typically account for the probability that the attacker can cause a decryption error to occur. This means that, in order to have a concrete security bound for the construction, one must bound the probability of a decryption error.

Intuitively, it is easy to explain why this is the case. ML-KEM and FrodoKEM internally use the Fujisaki-Okamoto [] transformation, where IND-CCA security is achieved by having the decapsulation algorithm check consistency of a recovered decryption result via re-encryption. I.e., informally, decapsulation checks that $C = \mathrm{Enc}(pk, M; H(M))$, where $M = \mathrm{Dec}(sk, C)$. If the check succeeds, then decapsulation proceeds; otherwise the ciphertext is rejected. Indeed, correct decryption and re-encryption is taken as evidence that C was honestly constructed by the adversary starting from M, rather than mauling another ciphertext from which it is trying to extract information. The soundness of this technique crucially depends on the adversary not being able exploit decryption correctness errors, which is why the probability of a correctness error appears in the security bound for the IND-CCA construction.

Bounding the probability of decryption error. Among the algorithms considered for the last round of the NIST PQC competition, three of them were very close in structure: Kyber, Saber and FrodoKEM.[1] All of these schemes

[1] FrodoKEM was not a finalist, but kept as an alternate candidate.

start from a lattice-based IND-CPA encryption scheme and then apply the Fujisaki-Okamoto transform outlined above. Moreover, all three proposals support the soundness of their designs and parameter choices by computing exact bounds for statistical events that permit setting upper bounds for the probability of a decryption failure of the IND-CPA scheme, which also yields a bound for the decryption failure of the IND-CCA scheme. Interestingly, all three bounds were computed using adaptations of the same Python script, which was originally produced for the first version of the FrodoKEM proposal [**?**].[2]

The computation performed by this script can be described as follows. The IND-CPA scheme decryption procedure ub FrodoKEM recovers $M' = C_2 - C_1 S$ where M, C_1 and C_2 are matrices of (binary) field elements and M' encodes a message in the most significant bits of its entries. Here, (C_1, C_2) are produced by the encryption procedure as $C_1 = S'A + E'$ and $C_2 = S'B + E'' + M$, where matrices A and $B = AS + E$ are fixed by the public encryption key, the S matrix is the secret key, and S', E, E' and E'' are noise matrices sampled from distributions with very small support—every finite field element produced by these distributions is an element close to 0 chosen from a small set of possibilities. A straightforward linear algebra argument shows that $M' = M$ if the noise expression $E''' = S'E + E'' - E'S$ results in a matrix where *all* entries are field elements with a small norm, i.e., they are small enough that the entries in M and M' have the same most significant bits. The python scrypt brute-force computes the probability mass function of a coefficient in E''' and computes the tail probability of a value exceeding the correctness threshold. The overall correctness bound follows from arguing that all entries in E''', individually, have the same distribution, and computing a union bound. We note that these computations are performed using floating point arithmetic and result in values of the order of 2^{-200}. The cases of ML-KEM (i.e., Kyber) and Saber are slightly more intricate due to the use of rounding, but the principle is the same.

As it stands, the correctness of the computed bounds that support the ML-KEM standard, and the FrodoKEM and Saber proposals have not been subject to formal verification.

Our Contributions. Our main contribution is an extension to EasyCrypt that permits computing formally verified upper bounds for the tail probabilities associated with decryption errors in lattice-based KEMs. More in detail:

- We provide a general framework to reason in EasyCrypt about distributions over a restricted class of matrix expressions, and proving that the relevant events related to decryption errors can be expressed as a union bound of an event thay can be checked for only one of the matrix entries. We extend this result to cases where matrix entries are expressions in a certain class of polynomial rings, in which the event is checked for only one of the polynomial coefficients. This framework reduces the problem of bounding the probability of decryption errors to the problem of comparing the absolute value of a finite field element sampled from a distribution, to a fixed threshold.
- We extend EasyCrypt to carry out approximate computations over the reals, which are guaranteed by construction to provide an upper bound, i.e., they always round up. We then build on this feature to allow the explicit computation of the probabilities described above. The algorithm has a reasonable execution time, whenever the distributions have a simple description and small enough support. In particular, for ML-KEM and FrodoKEM parameters, the execution time is in the range of a few minutes. This feature can be seen as a formally verified implementation of the Python script used in the NIST postquantum submissions.
- We show that, for FrodoKEM, the new EasyCrypt features permit providing a fully concrete correctness bound where all statistical terms are computed. We stress that we can do this end-to-end inside the same tool: our correctness theorem relates the adversary's advantage in winning the correctness game to the concrete probability value we compute. As a side contribution, we give a security proof for the IND-CPA component of FrodoKEM that goes down to a variant of the standard LWE problem (rather than MLWE as in Kyber [] or LWE with rounding as in Saber []). This proof is similar in structure to those given in [] but, to the best of our knowledge, such a proof had not been previously verified. In particular, our proof includes a hybrid argument that reduces the LWE problem to the multi-instance LWE problem required for FrodoKEM.
- We revisit the correctness proof for Kyber in [] and resolve one of the proof goals left for future work: formally verifying that the simplified (heuristic) computation for the correctness bound given in the Kyber submission for the NIST PQC competition is correct. This shows the generality of our method and extends the formal verification results for Kyber [] to cover all of the formal claims that support its security.[3].

Decide on Saber

Structure of this paper.

2. Preliminaries

HERE STARTS THE EPISODE V PRELIMINARIES, FROM WHERE WE CAN REUSE SOME PARTS.

We now briefly discuss the mechanized reasoning tools we use for our proofs and give an overview of ML-KEM and how it evolved from KYBER. The cryptographic definitions and notation we use are fairly standard and are given in [**?**].

[2] https://github.com/lwe-frodo/parameter-selection

[3] Proving the claim that the heuristic bound, which is computed over a simplified distribution, applies to Kyber is an open problem. Details are given in [] and we recap them in Section **??**

2.1. The EasyCrypt proof assistant

EasyCrypt[4] [?] is a proof assistant for formalizing proofs of cryptographic properties. Its primary feature is the Probabilistic Relational Hoare Logic (pRHL), which we use throughout to prove equivalences between games, sometimes conditioned by the non-occurrence of a failure event. We call such conditional equivalence steps *up-to-bad* reasoning in later section. pRHL is designed to support reasoning about equivalences of probabilistic programs while reasoning only locally (within oracles) and without reasoning about the distribution of specific variables—essentially keeping track only of the fact that variables in one program are distributed identically to variables in the other, but not keeping track of what that distribution may be. This logic has proved highly expressive for the bulk of cryptographic proof work. However, some steps require more global reasoning (about the entire execution) or keeping track of the distribution of individual variables. Logical rules to support such reasoning steps are implemented in EasyCrypt, but are often unwieldy to apply in concrete context. The EasyCrypt team has, over the years, developed a number of generic libraries that abstract those more complex reasoning rules into "game transformations" or equivalence results that can be instantiated as part of other proofs.

Our proof makes extensive use of three of those generic libraries, which we outline below and discuss more in depth in [?]:

plug-and-pray provides a formalized generic argument for bounding the security loss of reductions that guess an instance, session or query (among a bounded-sized set) will lead to a successful run;

hybrid provides a formalized and generic argument for bounding the distance between two games that differ only in one oracle, but where the transition must be done query-by-query for the purpose of the proof;

PROM provides a generic argument, initially intended to apply to programmable random oracles, that encapsulates the widely used argument that a "value is random and independent of the adversary's view".

2.2. Background on ML-KEM

From Kyber to ML-KEM. The original NIST PQC round-1 submission of Kyber described the scheme as a two-stage construction consisting of a module-lattice instantiation of the IND-CPA-secure LPR encryption scheme [?], [?] and a *"slightly tweaked"* Fujisaki-Okamoto (FO) transform [?], [?] to build an IND-CCA-secure KEM. The reduction linking IND-CPA security of the encryption scheme to MLWE was not spelled out in the NIST submission document. For the proof of IND-CCA security of the full scheme the submission mostly referred to previous work on the security of the FO transform, most notably [?] and [?]. Less than two months after the submission documents were made public, D'Anvers pointed out[5] that there are major obstacles to proving IND-CPA security from MLWE in Kyber. The reason is that unlike the original LPR scheme, round-1 Kyber included a public-key-compression step, which invalidates an assumption in the LPR proof. As a consequence, the Kyber submission team decided to tweak the submission for round 2 of NIST PQC by removing the public-key compression (and adjusting some other parameters to obtain similar performance); some more tweaks to parameters were proposed for the round-3 version of Kyber.

Algorithm 1 K-PKE.KeyGen(): key generation

```
Ensure: Secret key $sk \in \mathcal{R}_q^k$ and public key $pk \in \hat{\mathcal{R}_q}^k \times \{0,1\}^{256}$
$d \xleftarrow{\$} \{0,1\}^{256}$
$(\rho, \sigma) \leftarrow \mathrm{G}(d)$
$\hat{\mathbf{A}} \leftarrow \mathsf{Parse}(\mathsf{XOF}(\rho))$
$\mathbf{s}, \mathbf{e} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}_\sigma)$    $\triangleright\ \mathbf{s}, \mathbf{e} \in \mathcal{R}_q^k$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
return $sk = \hat{\mathbf{s}}$ and $pk = (\hat{\mathbf{t}}, \rho)$
```

Algorithm 2 K-PKE.Enc(pk, m): encryption

```
Require: Public key $pk = (\hat{\mathbf{t}}, \rho) \in \mathcal{R}_q^k \times \{0,1\}^{256}$, message $m \in \{0,1\}^{256}$
Ensure: Ciphertext $c \in \mathcal{R}_{d_u}^k \times \mathcal{R}_{d_v}$
$r \xleftarrow{\$} \{0,1\}^{256}$
$\hat{\mathbf{A}} \leftarrow \mathsf{Parse}(\mathsf{XOF}(\rho))$
$\mathbf{r} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}_r)$    $\triangleright\ \mathbf{r} \in \mathcal{R}_q^k$
$\mathbf{e}_1, e_2 \leftarrow \mathsf{CBD}_{\eta_2}(\mathsf{PRF}_r)$    $\triangleright\ \mathbf{e}_1 \in \mathcal{R}_q^k, e_2 \in \mathcal{R}_q$
$\hat{\mathbf{r}} \leftarrow \mathsf{NTT}(\mathbf{r})$
$\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
$v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{ToPoly}(m)$
$\mathbf{c}_1 \leftarrow \mathsf{Compress}_q(\mathbf{u}, d_u)$
$c_2 \leftarrow \mathsf{Compress}_q(v, d_v)$
return $c = (\mathbf{c}_1, c_2)$
```

Through the further course of the NIST PQC project, issues were identified also in the second part of Kyber's security argument, which establishes IND-CCA security

Algorithm 3 K-PKE.Dec(sk, c): decryption

```
Require: Secret key $sk = \hat{\mathbf{s}} \in \mathcal{R}_q^k$ and ciphertext $c = (\mathbf{c}_1, c_2) \in \mathcal{R}_{d_u}^k \times \mathcal{R}_{d_v}$
Ensure: Message $m \in \{0,1\}^{256}$
$\tilde{\mathbf{u}} \leftarrow \mathsf{Decompress}_q(\mathbf{c}_1, d_u)$
$\tilde{v} \leftarrow \mathsf{Decompress}_q(c_2, d_v)$
$m \leftarrow \mathsf{ToMsg}(\tilde{v} - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\tilde{\mathbf{u}})))$
return $m$
```

[4] https://easycrypt.info

[5] See https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/PX_Wd11BecI.

Algorithm 4 ML-KEM.KeyGen(): key generation

```
Ensure: Secret key $sk = (sk', pk, \mathrm{H}(pk), z) \in \mathcal{R}_q^k \times (\hat{\mathcal{R}_q}^k \times \{0,1\}^{256}) \times \{0,1\}^{256} \times \{0,1\}^{256}$ and public key $pk \in \hat{\mathcal{R}_q}^k \times \{0,1\}^{256}$
$(pk, sk') \leftarrow$ K-PKE.KeyGen()
$z \leftarrow \{0,1\}^{256}$
$sk \leftarrow (sk' \| pk \| \mathrm{H}(pk) \| z)$
return $(sk, pk)$
```

Algorithm 5 ML-KEM.Encap(pk): encapsulation

```
Require: Public key $pk \in \hat{\mathcal{R}_q}^k \times \{0,1\}^{256}$
Ensure: Ciphertext $c \in \mathcal{R}_{d_u}^k \times \mathcal{R}_{d_v}$ and shared key $K \in \{0,1\}^{256}$
$m \leftarrow \{0,1\}^{256}$
$(K, r) \leftarrow \mathrm{G}(m \| \mathrm{H}(pk))$ ▷ $(K, r) \in \{0,1\}^{256} \times \{0,1\}^{256}$
$c \leftarrow$ K-PKE.Enc$(pk, m, r)$
return $(c, K)$
```

through a "tweaked" FO transform. It turned out that existing QROM proofs of the FO transform, e.g., from [**?**], did not apply and that proving the particular variant of the FO transform is not straightforward [**?**, Sec. 5.4]. Although KYBER's variant of the FO transform was eventually proven secure [**?**], [**?**], NIST decided, after discussion on the pqc-forum mailing list, to revert to a more "vanilla" version of the FO transform for ML-KEM.

Our mechanized proof validates the final outcome of these evolutions, which led to the draft specification of ML-KEM. In our models we do *not* include the public-key validation step at the beginning of encapsulation, which NIST introduced in the ML-KEM draft standard. The reason is that, as far as we know, this is the last major open discussion topic for the final standard and it seems rather unlikely that it will be included in the final version of ML-KEM.

TECHNICAL DESCRIPTION OF ML-KEM. We give high-level algorithmic descriptions of K-PKE, the IND-CPA-secure public-key encryption scheme underlying ML-KEM, in algorithms 1 to 3 and of the full IND-CCA-secure KEM in algorithms 4 to 6. For a more implementation-oriented description that operates on byte arrays, see [**?**, Algs. 12–14].

ML-KEM works in the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n+1)$ with $q = 3329$ and $n = 256$. The core operations are on small-dimension vectors and matrices over $\mathcal{R}_q$; the dimension depends on the parameter k, which is different for different parameter sets of ML-KEM: ML-KEM (NIST security level 1) uses $k = 2$, ML-KEM (NIST security level 3) uses $k = 3$, and ML-KEM (NIST security level 5) uses $k = 4$. We denote elements in $\mathcal{R}_q$ with regular lower-case letters (e.g., v); vectors over $\mathcal{R}_q$ with bold-face lower-case letters (e.g., $\mathbf{u}$), and matrices over $\mathcal{R}_q$ with bold-face upper-case letters (e.g., $\mathbf{A}$).

In these descriptions, XOF is an extendable output function that in ML-KEM is instantiated with SHAKE-128 [**?**]. Parse interprets outputs of XOF as sequence of 12-bit unsigned integers and runs rejection sampling to obtain coefficients that look uniformly random modulo q. The function H is instantiated with SHA3-256, G is instantiated with SHA3-512 and J is instantiated with SHAKE256 with output length fixed to 32 bytes. CBD_η denotes sampling coefficients from a centered binomial distribution with parameter η;[6] extension from coefficients to (vectors of) polynomials is done by sampling each coefficient independently from CBD_η. For example, ML-KEM-768 uses $\eta_1 = \eta_2 = 2$. The sampling routine is parameterized by a pseudorandom function PRF_k with key k. NTT is the number-theoretic transform of a polynomial in $\mathcal{R}_q$. Both input and output of NTT can be written as a sequence of 256 coefficients in $\mathbb{Z}_q$ and typical implementations perform the transform inplace. However, output coefficients do not have any meaning as a polynomial in $\mathcal{R}_q$. We therefore denote the output domain as $\hat{\mathcal{R}_q}$; we apply the same notation for elements in $\hat{\mathcal{R}_q}$, e.g., $\hat{u} = \mathsf{NTT}(u)$. Application of NTT to vectors and matrices over $\mathcal{R}_q$ is done element-wise. $\mathsf{Compress}_q$ compresses elements in $\mathcal{R}_q$ (or $\mathcal{R}_q^k$) by rounding coefficients to a smaller modulus d_v (or d_u). $\mathsf{Decompress}_q$ is an approximate inverse of $\mathsf{Compress}_q$. ToPoly maps 256-bit strings to elements in $\mathcal{R}_q$ by mapping a zero bit to a zero coefficient and mapping a one bit to a $\frac{q}{2}$ coefficient; ToMsg rounds coefficients to bits to recover a message from a noisy version of a polynomial generated by ToPoly.

Algorithm 6 ML-KEM.Decap(c, sk): decapsulation

```
Require: Ciphertext $c \in \mathcal{R}_{d_u}^k \times \mathcal{R}_{d_v}$ and secret key $sk = (sk', pk, \mathrm{H}(pk), z) \in \mathcal{R}_q^k \times (\hat{\mathcal{R}_q}^k \times \{0,1\}^{256}) \times \{0,1\}^{256} \times \{0,1\}^{256}$
Ensure: Shared key $K \in \{0,1\}^{256}$
$m' \leftarrow$ K-PKE.Dec$(sk', c)$
$(K', r') \leftarrow \mathrm{G}(m' \| \mathrm{H}(pk))$
$c' \leftarrow$ K-PKE.Enc$(pk, m', r')$
$K^- \leftarrow \mathrm{J}(z \| c)$
if $c = c'$ then then $K \leftarrow K'$ else $K \leftarrow K^-$ ▷ Requires branch-free conditional
return $K$
```

[6] This means we have $B(n, p)$ with $p = 1/2$, $n = 2\eta$ and expected value shifted to 0.

3. Correctness of FrodoKEM and Kyber

4. Computing formally-verified upper-bounds in EasyCrypt

5. Formally verified proof of FrodoKEM

6. Completing the proof of ML-KEM

7. Related Work

8. Conclusions and Future Work