Formal Specifications for Certifiable Cryptography

M. Barbosa
University of Porto

K. Bhargavan
Cryspen

F. Kiefer
Cryspen

P. Schwabe
MPI-SP

P-Y. Strub
PQShield

B. E. Westerbaan
Cloudflare

Abstract

Cryptographic standards, as published by the NIST or IETF, are typically written in semi-formal English, with mathematical concepts defined using formulas, and algorithms written in pseudocode. Their primary goal is clarity and precision, and to enable interoperable implementations on a variety of hardware architectures and in multiple programming languages. However, these specifications are often not suitable for security proofs or the formal verification of implementations. In this paper, we investigate whether and how formal specification languages can be used to provide strong links between cryptographic standards, their security proofs, and their optimized implementations. We use the ongoing standardization and research around the upcoming ML-KEM standard as a case study for this work.

1 Specifying Cryptographic Algorithms

Cryptography is a crucial component of the Trusted Computing Base of the Internet, and consequently, new crypto standards provide an opportunity for improved real-world security and privacy, while at the same time, they may introduce new risks and vulnerabilities due to insecure designs or implementations. Consequently, new standards and their deployments are carefully scrutinized by both practitioners and researchers.

With the recent uptick in interest in novel cryptographic constructions for lightweight cryptography, post-quantum cryptography, homomorphic encryption, multi-party computation, and zero-knowledge proofs, a variety of groups are working on new standards that will likely be published in the coming years. While some of these standards are based on classic constructions that have been studied for decades, many details are being worked out for the first time during the standardization process, and will need to be carefully scrutinized. Furthermore, all of these standards require new implementations that bear little resemblance to existing crypto code and are likely to suffer from new classes of bugs and vulnerabilities.

Cryptographic standards are typically written in semi-formal English, interspersed with mathematical formulas and algorithmic pseudocode, and their main goal is precision and unambiguity, so that software developers and hardware manufacturers can implement the algorithm securely and correctly, and in a way that interoperates with other implementations. Each standard also usually has a reference implementation, written by the cryptographers who originally designed the algorithm in a low-level language like C, which is used to generate test vectors and serves as a guide for other implementations.

How can we be certain that the algorithm specified in a new cryptographic standard is secure enough to deploy? Do the pen-and-paper proofs from the literature apply to the concrete instantiation in the standard? Are the novel bit-level encodings specified in the standard correct and unambiguous? Must all implementations validate public keys, or can it be optional? Would a naive implementation of the standard be vulnerable to side-channel attacks?

These questions can be hard to answer for even simple, well-understood cryptographic primitives, let alone for new standards that compose multiple constructions and offer many optional features over dozens of pages. It can be even harder to guarantee that a heavily optimized implementation correctly and securely implements the novel cryptographic mechanisms defined in the standard.

To help evaluate cryptographic designs and implementations, a field of research, sometimes called Computer-Aided Cryptography [4], advocates the use of formal specification languages and semi-automated verification tools to build machine-checked formal proofs of security and correctness. These methods can be seen as complementary to and extending the scope of pen-and-paper proofs and manual security audits.

The survey paper cited above considers many different tools and techniques. Tools like EasyCrypt [5] can be used to formalize and prove the security of cryptographic constructions, and serve as specifications for optimized assembly implementations written in Jasmin [1, 2]. Proof-oriented programming languages like F* [12] have be used to specify and implement cryptographic libraries like HACL [13]. General-purpose proof assistants like Coq have been used to develop verified field arithmetic in Fiat-Crypto [7] against mathematical specifications written in Gallina, the input language of Coq.

No matter which verification framework one may use, each of these approaches begins with the need for a formal specification of the cryptographic algorithm, and each tool supports its own specification language. The formal specification differs from the published standard in several ways. First, it is written in a language with a well-defined mathematical semantics, so that there can be no ambiguity about its meaning. Second, it is succinct, and therefore suitable as an artifact to be formally studied. Third, it is precise, in that it carefully details all its assumptions, functional dependencies, and byte-level encodings, which may be spread across (or left implicit in) the published standard. Fourth, some formal specifications are executable and hence can be used to generate test vectors and test other implementations for interoperability. Fifth, these specifications are supported by a verification tool and hence can be used as a basis for machine-checked security proofs and implementation correctness verification.

However, the need to manually write a formal specification in one of these formal languages introduces an extra work step for the analyst and potentially opens a gap between the published standard and its proofs. It becomes hard to compare proofs done in different frameworks against different specifications, and to link them with pen-and-paper proofs. It would clearly be preferable if we could close this gap by directly using the pseudocode in the standard as a specification for proofs.

In the rest of this paper, we investigate various approaches towards the goal of bringing cryptographic standards closer to formal specifications. Using the upcoming ML-KEM standard as a case study, we look at various formal and semi-formal specifications and open up a discussion on how such specifications could be improved and included in the standard. We hope that this discussion can lead to a larger effort around the use of formal specifications in NIST and IETF standards.

2 Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)

ML-KEM is a key encapsulation mechanism currently undergoing standardization as FIPS 203 at NIST [10]. A previous version of ML-KEM was known by the name Kyber.

The goal of ML-KEM is to allow a sender to encapsulate a fresh symmetric key and send it to a recipient, in a way that the key is indistinguishable from a fresh random value to anyone other than the sender and recipient, even under chosen-ciphertext attacks (IND-CCA). To this end, ML-KEM uses a lattice-based cryptographic construction that relies on the difficulty of the Module Learning with Errors (MLWE) problem.

2.1 Mathematical Operations

The ML-KEM standard uses a combination of pseudocode and mathematical notation to describe the KEM construction. Section 4 introduces various hash functions that ML-KEM relies on, and then defines a series of algorithms for field, polynomial, and matrix arithmetic, and for sampling (Algorithms 2-11).

For example, the compression function takes a field element in $\mathbb{Z}_q$ and converts it to a d-bit integer (in $\mathbb{Z}_{2^d}$) using the following formula (Equation 4.5):

$$\begin{aligned} \text{Compress}_d: \quad & \mathbb{Z}_q \longrightarrow \mathbb{Z}_{2^d} \\ & x \longrightarrow \lceil (2d/q) \cdot x \rfloor \end{aligned}$$

where $\lceil \ldots \rfloor$ refers to a rounding function (where 0.5 rounds up to 1).

The ML-KEM standard defines polynomial multiplication directly in terms of the Number Theoretic Transform (NTT), a somewhat unusual choice from the point of view of a specification, since the NTT is

Algorithm 9 $\mathsf{NTT}^{-1}(\hat{f})$

Computes the polynomial $f \in R_q$ corresponding to the given NTT representation $\hat{f} \in T_q$.

Input: array $\hat{f} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of input NTT representation

Output: array $f \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the inverse-NTT of the input

1: $f \leftarrow \hat{f}$ ▷ will compute in-place on a copy of input array
2: $k \leftarrow 127$
3: **for** ($len \leftarrow 2$; $len \leq 128$; $len \leftarrow 2 \cdot len$)
4: **for** ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
5: $zeta \leftarrow \zeta^{\mathsf{BitRev}_7(k)} \bmod q$
6: $k \leftarrow k - 1$
7: **for** ($j \leftarrow start$; $j < start + len$; j++)
8: $t \leftarrow f[j]$
9: $f[j] \leftarrow t + f[j + len]$ ▷ steps 9-10 done modulo q
10: $f[j + len] \leftarrow zeta \cdot (f[j + len] - t)$
11: **end for**
12: **end for**
13: **end for**
14: $f \leftarrow f \cdot 3303 \bmod q$ ▷ multiply every entry by $3303 \equiv 128^{-1} \bmod q$
15: **return** f

Figure 1: Inverse NTT Transform from FIPS 203

typically seen as an optimization technique for implementations. Algorithms 8 and 9 define conversion to and from the NTT domain, and Algorithms 10 and 11 define multiplication in the NTT domain.

Algorithm 9 is depicted in Figure 1. As a specification, this algorithm is very precise and low-level, almost at the level of an implementation. It defines the back-conversion of a polynomial from the NTT domain to the normal domain, but it does so operationally, not mathematically. So, it is not immediately apparent that this function is an inverse of the NTT function in Algorithm 8. Furthermore, if an implementation were to choose an optimized implementation that were to be structured differently, it would again not be trivial to prove that it matched this low-level algorithmic specification.

2.2 Cryptographic APIs

Section 5 describes the ML-KEM public key encryption (PKE) construction as three algorithms: key generation, encryption, and decryption (Algorithms 12-14). These algorithms make extensive use of the sampling, matrix operations, and NTT multiplication defined in Section 4. This API is expected to satisfy indistinguishability against chosen-plaintext attacks (IND-CPA).

Section 6 then defines the main KEM API: key generation, encapsulation, and decapsulation (Algorithms 15-17). These algorithms wrap around the PKE API to implement a variant of the Fujisaki-Okamoto transform. This API is expected to satisfy indistinguishability against chosen-plaintext attacks (IND-CCA).

All these cryptographic functions are also written in pseudocode, similar in style to Algorithm 9. They provide a concrete specification of the input-output behavior of ML-KEM and nothing more. They do not sketch or outline any security arguments. They also do not concern themselves with which implementation concerns such as side-channel resistance.

2.3 Machine-readable Executable Specifications

Although the pseudocode in the ML-KEM standard is precise, it uses both standard and custom mathematical notation, and is not executable. Consequently, these algorithms cannot be used to generate test vectors, and is is hard to check if an implementation conforms to the standard. To achieve these goals, it is much better to have an executable, testable specification that can be read, checked, and run on multiple platforms.

2.3.1 Python Pseudocode in IETF Draft

The CFRG internet draft standard for ML-KEM (Kyber) [11] takes a different approach from the NIST spec by using python as a specification language for the ML-KEM pseudocode. For example, the definition of the Compress_d function in this draft is as follows:

```
Compress(x, d) = Round( (2^d / q) x ) umod 2^d
```

NTT conversion and multiplication are described mathematically, but Section 13 includes a complete machine-readable specification in Python, where Algorithm 9 is written as follows:

```
def InvNTT(self):
    cs = list(self.cs)
    layer = 2
    zi = n//2
    while layer < n:
        for offset in range(0, n-layer, 2*layer):
            zi -= 1
            z = pow(zeta, brv(zi), q)

            for j in range(offset, offset+layer):
                t = (cs[j+layer] - cs[j]) % q
                cs[j] = (inv2*(cs[j] + cs[j+layer])) % q
                cs[j+layer] = (inv2 * z * t) % q
        layer *= 2
    return Poly(cs)
```

The full Python specification is remarkably succinct (323 lines) and looks quite similar to the algorithm pseudocode in the NIST spec. It takes full advantage of the notation and mathematical functions available in Python. It is also executable and has been used to generate test vectors for the draft standard.

However, this specification is not strongly typed and uses short variable names, and so it can be hard to see and check the range of each value. This also makes it a bit far from the kinds of formal specifications that are needed for verification in tools like EasyCrypt and F*. If the Python code is annotated with types, it may be possible to translate such a specification to a formal model, as demonstrated in the original hacspec-python tool [6].

2.3.2 ML-KEM in hacspec

The hacspec approach focuses on executable specifications that can serve as a bridge between a standard and its implementations. The current version of hacspec [9] is a functional subset of Rust with a set of helper libraries that provide abstract mathematical integers and other common operations that are useful in formal specifications, but not commonly used in implementations.

Since Rust is an efficient systems programming language, hacspec specifications can be easily used to test optimized implementations and may even be used as reference implementations when performance is not crucial. The Rust ecosystem further makes it easy to write, deploy, and maintain documentation of the specification using tools like rustdoc.

The Rust type system enforce a strict discipline on hacspec specifications. For example, all integer conversions must done explicitly: when an 8-bit integer and a 16-bit integer are combined, the 8-bit integer must be explicitly cast to 16-bits for the Rust type checker to approve the operation. We use const generics to write a single specification that covers all variants of ML-KEM. We also use types to capture the mathematical structures in the spec, such as field elements and polynomial ring elements:

```
type KyberFieldElement = PrimeFieldElement<3329>;
type KyberPolynomialRingElement = PolynomialRingElement<KyberFieldElement, 256>;
```

With the above definition, all arithmetic on field elements and polynomials implicitly reduces values modulo the ML-KEM prime (3329). The following listing shows an example of the specification of the inverse NTT from Figure 1. The comments map the operations to FIPS 203 where the syntax is different.

```
fn ntt_inverse(f_hat: KyberPolynomialRingElement) -> KyberPolynomialRingElement {
    let mut f = f_hat;
    let mut k: u8 = 127;
    // for (len <- 2; len <= 128; len <- 2*len)
    for len in NTT_LAYERS {
        // for (start <- 0; start < 256; start <- start + 2*len)
        for start in (0..(COEFFICIENTS_IN_RING_ELEMENT - len)).step_by(2 * len) {
            // zeta <- Zeta^(BitRev_7(k)) mod q
            let zeta = ZETA.pow(bit_rev_7(k));
            k -= 1;

            for j in start..start + len {
                let t = f[j];
                f[j] = t + f[j + len];
                f[j + len] = zeta * (f[j + len] - t);
            }
        }
    }
    // f <- f*3303 mod q
    for i in 0..f.coefficients().len() {
        f[i] = f[i] * INVERSE_OF_128;
    }
    f
}
```

The full ML-KEM specification in hacspec can be found in the libcrux repository:

https://github.com/cryspen/libcrux/tree/main/specs/kyber

This specification has been compiled via the hax tool to F*, and hence can be used as the basis for formal proofs using the F* framework.

2.3.3 Reference Implementations

Another source of "truth" for cryptographic specifications is in their reference implementations. Many implementations of the standard start by copying the reference implementation and then adapting them for different scenarios and optimizing them for different platforms, which makes the reference implementation highly influential in how the standard is perceived and understood.

The official reference implementation of ML-KEM is available from the PQ-Crystals repository, and variants of this implementation have been integrated into many other libraries such as PQClean and liboqs, integrated into applications like Signal's PQXDH etc.

These reference implementations are typically written in C, which makes them even more low-level than Python. For example, the code for 1-bit message compression (Compress_1) in this implementation is:

```
// t += ((int16_t)t >> 15) & KYBER_Q;
// t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
t <<= 1;
t += 1665;
t *= 80635;
t >>= 28;
t &= 1;
```

This function is used to compress plaintext (secret) messages, and so using the formula with division (as in the comments above) would result in a side-channel vulnerability, hence the more complicated sequence of operations used here. (Indeed, the original version of this implementation did contain such a vulnerability, but it was discovered as part of a formal verification project.)

While the reference implementation is as precise as it gets, it is way too low-level to serve as a formal specification for proofs. Indeed, it is often not even obvious that the C reference implementation itself is correct or secure.

2.4 Towards Verifiable Formal Specifications

There are at least three verification-oriented formal specifications of ML-KEM that we are aware of.

The first is an EasyCrypt specification of Kyber (later adapted to ML-KEM) that has been used as the basis for both security proofs and implementation verification [3]. We describe this specification in detail in the next section.

The second is an F* specification used as the basis for verifying the Rust implementation of ML-KEM in libcrux. This specification is currently written by hand to be succinct, but in the future, it may be possible to use the F* generated from the hacspec specification described above.

The third is a specification written in Cryptol [8], a domain-specific functional language for cryptography, which it has been used to specify a number of cryptographic algorithms, including ML-KEM:

https://github.com/GaloisInc/cryptol-specs

Cryptol is a functional language but the syntax is flexible enough to allow for a quasi-imperative presentation. Except for minor syntactic differences, many of the core algorithms in the Cryptol spec are quite similar to the EasyCrypt counterpart. This includes the description of encode and decoding algorithms, compression and decompression, and mathematical operations. On the other hand, the parts of the specification that are more naturally written in imperative form, namely the randomness sampling loops, cannot be literally transcribed into Cryptol, and may look more natural in EasyCrypt.

3 A Complete EasyCrypt Specification of ML-KEM

The EasyCrypt specification of ML-KEM closely follows the draft standard proposal for ML-KEM-768.[1]. We will gradually introduce the various definitions and intermediate results leading to the full code in Figures 2 to 5. The text that follows is taken with minor changes from [3].

3.1 Algebraic Structure

Operations over values in field $\mathbb{F}_q$. We begin by fixing the prime $q = 3329$ and instantiating the EasyCrypt theory ZModField with this parameter to obtain type Fq that represents elements in the finite field $\mathbb{F}_q$. We extend this theory with an operator as_sint that maps elements in Fq to integers in the range $[-(q-1)/2, (q-1)/2]$ and define compression and decompression operators as follows:

```
op as_sint(x : Fq) = if (q−1) / 2 < asint x then asint x − q else asint x.
op compress(d : int, x : Fq) : int = round (asint x * 2^d /ℝ q) % 2^d.
op decompress(d : int, x : int) : Fq = inFq (round (x * q /ℝ 2^d)).
```

Here, asint is the operator that takes a field element and returns an integer in the range $[0, q)$. We use $/_{\mathbb{R}}$ to denote real division, use / for integer division and % for the mod operation. round denotes rounding to the closest integer with ties being rounded up. Compression and decompression map field elements into integers in the range $[0, 2^{\mathsf{d}})$ and back. Using these operators we can also capture the encoding/decoding of bits as finite-field elements and relate the semantics of these operations to that of the compression and decompression operators.

```
op b_encode(b: bool): Fq = if b then inFq ((q+1)/2) else inFq 0.
op b_decode(c: Fq) : bool = ¬ |as_sint c| < q / 4 + 1.
lemma b_encode_sem c : b_encode c = decompress 1 (if c then 1 else 0).
lemma b_decode_sem c : compress 1 c = if b_decode c then 1 else 0.
```

The next step is to specify the distributions over field elements. We define the Binomial distribution and the uniform distribution over field elements by declaring them as follows, and then deriving auxiliary lemmas that describe the mass function of each of them in an explicit way.

```
op dshort_elem : Fq distr = dmap (dcbd 2) inFq. (* binomial distribution *)
op duni_elem : Fq distr = DZmodP.dunifin. (* uniform distribution *)
```

[1] Extending the specification for other variants is straightforward.

```
module MLKEM(HS : HSF.PseudoRF, XOF : XOF_t, PRF : PseudoRF,
     KemHS : HSF_KEM.PseudoRF, KemH : KEMHashes, O : RO.POracle)  = {

  proc kg_derand(coins: W8.t Array32.t * W8.t Array32.t) : publickey * secretkey = {
    kgs     ← coins_1;
    z       ← coins_2;
    (pk,sk) ← InnerPKE.kg_derand(kgs);
    hpk     ← H_pk pk;
    return (pk, (sk,pk,hpk,z));
  }

  proc enc_derand(pk : publickey, coins: W8.t Array32.t) : ciphertext * sharedsecret = {
    m       ← coins;
    hpk     ← H_pk pk;
    (_K,r) ← G_mhpk m hpk;
    c       ← InnerPKE.enc_derand(pk,m,r);
    return (c,_K);
  }

  proc dec(cph : ciphertext, sk : secretkey) : sharedsecret = {
    (skp,pk,hpk,z) ← sk;
    m              ← InnerPKE.dec(skp,cph);
    (_K,r)         ← G_mhpk m hpk;
    _K'            ← J z cph;
    c              ← InnerPKE.enc_derand(pk,m,r);
    if (c ≠ cph) { _K ← _K'; }
    return _K;
  }
}.
```

Figure 2: ML-KEM EasyCrypt Specification: IND-CCA KEM

THE RING R_q, MATRICES AND VECTORS. At this point we can formalize the ring R_q of polynomials over which ML-KEM operates. We start by defining a concrete representation of these polynomials using arrays of size 256 of field elements. As in the ML-KEM specification, we will reuse this representation for ring elements in the NTT domain.

We write explicit formulae for the basic ring operations over this array representation (shown below) and extend the definitions of compression/decompression and binomial/uniform distributions to ring elements by applying them pointwise to each coefficient.

```
type poly = Fq Array256.t.
op zero : poly = Array256.create Zq.zero.
op one : poly = zero[0←Zq.one].
op (+) (pa pb : poly) : poly = map2 (fun a b : Fq ⇒ Zq.(+) a b) pa pb.
op (−) (p : poly) : poly = map Zq.([−]) p.
op ( * ) (pa pb : poly) : poly =
  Array256.init (fun (i : int) ⇒ foldr (fun (k : int) (ci : Fq) ⇒
    if (0 ≤ i − k) then ci + pa[k] * pb[i − k]
    else ci − pa[k] * pb[256 + (i − k)]) Zq.zero (iota_ 0 256)).
```

We prove that these definitions implement the correct operations over $\mathbb{Z}_q[X]/(X^{256}+1)$. We can now define matrices and vectors over ring elements in our representation by cloning the Matrix theory in the EasyCrypt library with the correct dimension kvec=3 for ML-KEM mid, and instantiating the ring operations with those we described above. Note that, when instantiating the Matrix theory, we must prove that the provided operators indeed constitute a ring, which we can do by leveraging results in the PolyReduce theory.

NUMBER THEORETIC TRANSFORM. Finally, we complete the algebraic definitions by specifying the Number Theoretic Transform (NTT) in EasyCrypt and proving some fundamental results about this transform that are essential to the security and correctness of ML-KEM . We define the NTT forward and inverse operations as follows. We omit here the definition of NTT for vectors and matrices of ring elements, as this is simply the pointwise application of NTT to each component. We use notation $(\cdot)_1$ and $(\cdot)_2$ for the first and second element in a pair.

```
proc kg_derand(seed: W8.t Array32.t) : pkey * skey = {
  (rho,sig) ← G_coins coins;
  _N ← 0;
  i ← 0;
  while (i < kvec) {
    j ← 0;
    while (j < kvec) {
      XOF.init(rho, j, i);
      c ← Parse(XOF).sample();
      a[(i,j)] ← c;
      j ← j + 1;
    }
    i ← i + 1;
  }
  i ← 0;
  while (i < kvec) {
    c ← CBD2.sample(PRF sig (W8.of_int _N));
    s ← s[i←c];
    _N ← _N + 1;
    i ← i + 1;
  }
  i ← 0;
  while (i < kvec) {
    c ← CBD2.sample(PRF sig (W8.of_int _N));
    e ← e[i←c];
    _N ← _N + 1;
    i ← i + 1;
  }
  s ← nttv s;
  e ← nttv e;
  t ← ntt_mmul a s + e;
  tv ← EncDec.encode12_vec(toipolyvec t); (* minimum residues *)
  sv ← EncDec.encode12_vec(toipolyvec s); (* minimum residues *)
  return ((tv,rho),sv);
 }

proc dec(sk : skey, cph : ciphertext) : plaintext = {
  (c1,c2) ← cph;
  ui ← EncDec.decode10_vec(c1);
  u ← decompress_polyvec 10 ui;
  vi ← EncDec.decode4(c2);
  v ← decompress_poly 4 vi;
  si ← EncDec.decode12_vec(sk);
  s ← ofipolyvec si;
  mp ← v − invntt (ntt_dotp s (nttv u));
  m ← EncDec.encode1(compress_poly 1 mp);
  return m;
}
```

```
proc enc_derand(pk : pkey, m : plaintext, r : W8.t Array32.t)
                                                    : ciphertext = {
 (tv,rho) ← pk;
  _N ← 0;
  thati ← EncDec.decode12_vec(tv);
  that ← ofipolyvec thati;
  i ← 0;
  while (i < kvec) {
    j ← 0;
    while (j < kvec) {
      XOF.init(rho, i, j);
      c ← Parse(XOF).sample();
      aT[(i,j)] ← c; (* this is the transposed matrix *)
      j ← j + 1;
    }
    i ← i + 1;
  }
  i ← 0;
  while (i < kvec) {
    c ← CBD2.sample(PRF coins (W8.of_int _N));
    rv ← rv[i←c];
    _N ← _N + 1;
    i ← i + 1;
  }
  i ← 0;
  while (i < kvec) {
    c ← CBD2.sample(PRF coins (W8.of_int _N));
    e1 ← e1[i←c];
    _N ← _N + 1;
    i ← i + 1;
  }
  e2 ← CBD2.sample(PRF coins (W8.of_int _N));
  rhat ← nttv rv;
  u ← invnttv (ntt_mmul aT rhat) + e1;
  mp ← EncDec.decode1(m);
  v ← invntt (ntt_dotp that rhat) + e2 + decompress_poly 1 mp;
  c1 ← EncDec.encode10_vec(compress_polyvec 10 u);
  c2 ← EncDec.encode4(compress_poly 4 v);
  return (c1,c2);

}
```

Figure 3: ML-KEM EasyCrypt Specification: IND-CPA PKE

```
module type XOF_t(O : RO.POracle) = {
  proc init(rho :  W8.t Array32.t, i j : W8.t) : unit
  proc next_bytes() : W8.t Array168.t
}.

module Parse(XOF : XOF_t, O : RO.POracle) = {
  proc sample() : poly = {
    j ← 0;
    while (j < 256) {
      b168 ← XOF(O).next_bytes();
      k ← 0;
      while ((j < 256) && (k < 168)) {
        bi  ← b168[k]; bi1 ← b168[k+1]; bi2 ← b168[k+2];
        k ← k + 3;
        d1 ← to_uint bi        + 256 * (to_uint bi1 % 16);
        d2 ← to_uint bi1  / 16 + 16  * to_uint bi2;
        if (d1 < q)                    { aa[j] ← inFq d1; j ← j + 1; }
        if ((d2 < q) && (j < 256)) { aa[j] ← inFq d2; j ← j + 1; }
      }
    }
    return aa;
  }
}.
```

Figure 4: ML-KEM EasyCrypt Specification: Rejection sampling a uniform element in R_q.

```
op SHA3_PRF : W8.t Array32.t → W8.t →  W8.t Array128.t.

clone import PRF_DEFS.PseudoRF as PRF_ with
  type K ← W8.t Array32.t,
  op dK ← srand,
  op F ← SHA3_PRF.

module CBD2(PRF : PseudoRF) = {
  proc sample(sig : W8.t Array32.t, _N : int) : poly = {
    bytes ← PRF.f(sig, W8.of_int _N);
    bits ← BytesToBits (to_list bytes);
    i ← 0;
    while (i < 256) {
      a ← b2i (nth false bits (4*i)) + b2i (nth false bits (4*i+1));
      b ← b2i (nth false bits (4*i+2)) + b2i (nth false bits (4*i+3));
      rr[i] ← inFq  (a − b);
      i ← i + 1;
    }
    return rr;
  }
}.
```

Figure 5: ML-KEM EasyCrypt Specification: Sampling noise from the Binomial distribution.

```
op zroot = inFq 17.
op br = BitEncoding.BitReverse.bsrev 7.

op ntt(p : poly) = Array256.init (fun i ⇒  let ii = i / 2 in
  if i % 2 = 0 then ∑_{j=0}^{127} p[2*j] * zroot^{(2*br ii+1)*j}
               else ∑_{j=0}^{127} p[2*j+1] * zroot^{(2*br ii+1)*j}).

op invntt(p : poly) = Array256.init (fun i ⇒  let ii = i / 2 in
  if i % 2 = 0 then ∑_{j=0}^{127} inv (inFq 128) * p[2*j] * zroot^{−(2*br j+1)*ii}
               else ∑_{j=0}^{127} inv (inFq 128) * p[2*j+1] * zroot^{−(2*br j+1)*ii}).

op cmplx_mul (a :Fq * Fq, b : Fq * Fq, zzeta : Fq) : Fq * Fq =
  (a_2 * b_2 * zzeta + a_1*b_1, a_1 * b_2 + a_2 * b_1).

op basemul(a b : poly) : poly = Array256.init (fun i ⇒  let ii = i / 2 in
  if i % 2 = 0 then (cmplx_mul (a[2*ii],a[2*ii+1]) (b[2*ii],b[2*ii+1]) (zroot^{(2*br ii+1)}))_1
               else (cmplx_mul (a[2*ii],a[2*ii+1]) (b[2*ii],b[2*ii+1]) (zroot^{(2*br ii+1)}))_2).

op scale(p : poly, c : Fq) : poly = Array256.map (fun x ⇒  x * c) p.
lemma ntt_scale p c : ntt (scale p c) = scale (ntt p) c.
lemma invntt_scale p c : invntt (scale p c) = scale (invntt p) c.
lemma nttK : cancel invntt ntt.
lemma mul_comm_ntt (pa pb : poly): ntt (pa * pb) = basemul (ntt pa) (ntt pb).
lemma add_comm_ntt (pa pb : poly): ntt (pa + pb) = (ntt pa) + (ntt pb).
lemma nttZero : ntt KPoly.zero = KPoly.zero.
```

These properties play an important role in the proof of correctness and in the ongoing proof of security, and they include the expected cancellation (invertibility) properties, commuting with ring addition and multiplication. Additionally, the preservation of scaling factors is used to deal with multiplicative terms used in Montgomery representation.

3.2 Algorithmic Sampling, Encoding and Decoding

The ML-KEM specification relies on a number of auxiliary algorithms. We follow the same structure and define them as independent EasyCrypt modules used by the main ML-KEM specification in EasyCrypt. In this section we mention SHA3 algorithms several times, when the specification refers to it. By this we mean algorithms in the SHA-3 standard, including the SHA3 hash functions and the SHAKE extendable-output functions.

```
type ipoly = int Array256.t.
op toipoly(p : poly) : ipoly = map asint p.
op ofipoly(p : ipoly)  : poly = map inFq p.

module EncDec = {

  proc decode4(a : W8.t Array128.t) : ipoly = {
    i ← 0;
    while (i < 128) {
      r[i*2+0]  ← to_uint a[i] % 16;
      r[i*2+1]  ← to_uint a[i] / 16;
      i ← i + 1;
    }
    return r;
  }

  proc encode4(p : ipoly) : W8.t Array128.t = {
    j ← 0; i ← 0;
    while (i < 128) {
      fi ← p[j]; j ← j + 1;
      fi1 ← p[j]; j ← j + 1;
      r[i] ← W8.of_int (fi + fi1 * 2^4); i ← i + 1;
    }
    return r;
  } (···) }.
```

Figure 6: Imperative specification of encoding/decoding.

Rejection sampling a uniform ring element. The first auxiliary algorithm used by ML-KEM is a rejection sampling procedure called Parse, which we give in Figure 4, and which corresponds to SampleNTT in the ML-KEM specification. The algorithm is parametric on an Extendable-Output Function (XOF), which must be initialized prior to the execution of Parse, and is then used by this algorithm to obtain as many random bytes as needed to complete the 256 coefficients that make up an element in R_q. We note that in ML-KEM the output of this procedure is interpreted as being in the NTT domain, which has implications in various parts of the correctness and security proofs. We model the XOF as producing a sequence of byte blocks of length 168 (each call to next_bytes is the next batch of XOF output bytes), since this matches the way ML-KEM uses SHAKE-128 for this purpose.

Using SHA3 to hash, smooth randomness, and sample noise. ML-KEM uses SHA3 algorithms over strings of different input lengths to implement the various hash functions required for the KEM (Fujisaki-Okamoto) construction. Concretely, function J corresponds to using SHAKE-256 as a PRF when producing pseudorandom keys returned for ciphertexts rejected in decryption. Function G_mhpk corresponds to using SHA-512 to compute the shared key in encapsulation. ML-KEM also uses SHA3-512 to expand the raw randomness fed to key generation into two 32-byte seeds, one for the matrix rejection sampling, and another to key SHAKE-256 used as PRF in the noise sampling procedure above. Here SHA3-512 (G_coins) os used as a pseudorandom generator (PRG).

Finally, ML-KEM relies on SHA3 algorithms to sample pseudorandom ring elements of small norm (and vectors thereof). The procedure used to sample ring elements is specified in Figure 5, which exactly matches the ML-KEM specification for the concrete parameter of $\eta = 2$, which is used by the implementation of ML-KEM mid. Here we model SHAKE-256 as a Pseudo Random Function (PRF), using the corresponding EasyCrypt library and an operator SHA3_PRF.

Encoding and Decoding. The last auxiliary module is called EncDec and it specifies various forms of serializing and de-serializing keys, messages and ciphertexts to/from byte arrays. We specify these operations by defining the canonical bijection toipoly/ofipoly between (possibly compressed) ring elements and arrays of integers representing the polynomial coefficients, and a number of procedures that encode/decode polynomials that have been previously compressed to varying lengths. We show in Figure 6 the case of encoding a polynomial that has been compressed to 4-bits per coefficient into 128 bytes.

These encoding/decoding procedures are specified in imperative form, with a loop structure and output/input byte array types that simplify the connection to the implementation and are also more readable. Our choice here is motivated by the fact that we use fixed-sized arrays for the encoded bytes, which means that we must independently specify an explicit encoding/decoding procedure for each compression size. In the ML-KEM draft standard, decoding is defined by first mapping the compressed polynomial to a bit string (BytesToBits), which is subsequently partitioned into the various coefficients. We also formalize this version of the operation for completeness.

```
op BytesToBits(bytes : W8.t list) : bool list = flatten (map W8.w2bits bytes).
op decode(l i : int, bits : bool list) =∑_{j=0}^{l} b2i (nth false bits (i*l+j)) *2^j.
op sem_decode4(a : W8.t Array128.t) : ipoly =
    Array256.init (fun i ⇒  decode 4 i (BytesToBits (to_list a))).
```

We rely on a new EasyCrypt feature to lift our imperative versions of the encoding and decoding operations to functional operators, prove that decoding matches the ML-KEM specification, and finally prove that encoding inverts decoding.

3.3 Uses of the EasyCrypt specification

The EasyCrypt specification presented in the previous sections stands at the core of a large formal verification effort for ML-KEM that includes the following components:

- A proof of security of the ML-KEM specification, showing that it is indeed an IND-CCA secure KEM.
- An extended set of semantic properties about the specification itself, including 1) a description of the probability mass functions for the various distributions used in ML-KEM and a proof that the various sampling procedures indeed produce the specified distributions; and 2) proofs that the NTT computation procedures described in the draft standard indeed compute the Number Theoretic Transform and has the required algebraic properties.
- Functional correctness of Jasmin implementations, include both reference and highly optimized AVX2 code, that is also proved to correctly deploy constant-time countermeasures. Here, functional correctness means that the Jasmin code is proved to be functional equivalent to the specification: for small valid inputs, it produces the same outputs.

Note that, even though the EasyCrypt specification is not directly executable, the Jasmin reference implementation plays this role in the above development.

4 Discussion Points

We have studied various artifacts around the ML-KEM standard: the NIST and IETF draft specifications, executable specifications in Python and hacspec, implementations in C, Rust, and assembly, and formal verification-friendly specifications in F*, Cryptol, and EasyCrypt.

This investigation raises several interesting questions that are worth discussing at the workshop:

- Would it be desirable to embed formal specifications directly within NIST and IETF crypto standards?
- If not, would it be possible to link the pseudocode used in these standards with formal specifications?
- Is it more valuable to have an executable specification for testing or a formal spec for verification?
- Are specifications written in languages like Python and Rust more accessible, readable, usable than specifications written in formal languages like F* or EasyCrypt?
- Should formal specifications describe high-level mathematical concepts like polynomial multiplication or should they detail low-level algorithms like NTT multiplication?
- Should specifications in standards be targeted towards security proofs or implementation correctness, and can they do both?
- Should standards and their formal specifications include indications for secure implementations, such as algorithms that may be at risk of side-channel attacks?

Secret Independence

It is worth expanding on the last of these discussion points. The ML-KEM specification does not say anything about side-channel attacks, and as we have seen, it specifies the Compress function using a division operation that can potentially leak the value of its numerator, which may in some cases be the secret plaintext.

This side-channel leak was found during the verification of the Rust implementation in libcrux and thereafter identified in a number of ML-KEM implementations including the reference implementation. The libcrux implementation uses types to enforce *secret independence*, where potentially leaking operations are

not permitted on secret values. This includes arithmetic operations with input-dependent timing such as division, comparison, branching, and memory access.

Similarly to what is done in libcrux, in the Jasmin/EasyCrypt ecosystem, an automatic type-based method is used to check that implementations adhere to the constant-time programming discipline. Also there developers must tag values as secret or public. Clearly it would make sense that in both developments these annotations are consistent, so that both implementations satisfy the same security definition. However, at the moment, the standard does not provide such guidance.

Would labeling the ML-KEM spec with secret/public annotations and explicitly requiring implementations to enforce a secret independent discipline have prevented the side-channel leak in various ML-KEM implementations? Would this be a desirable feature of a formal specification?

We invite workshop attendees to consider all these questions with us and try to find a constructive way forward towards designing and deploying formally verifiable cryptographic standards.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. pages 1807–1823. ACM, 2017.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 965–982. IEEE, 2020.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying kyber episode IV: implementation correctness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):164–193, 2023.

[4] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 777–795. IEEE, 2021.

[5] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 71–90, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Karthikeyan Bhargavan, Franziskus Kiefer, and Pierre-Yves Strub. hacspec: Towards verifiable crypto standards. In *Security Standardisation Research (SSR)*, volume 11322 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2018.

[7] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. pages 1202–1219. IEEE, 2019.

[8] Jeffrey R. Lewis and Brad Martin. Cryptol: high assurance, retargetable crypto development and validation. In *Proceedings of the 2003 IEEE Conference on Military Communications - Volume II*, MILCOM'03, page 820–825. IEEE Computer Society, 2003.

[9] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Technical report, Inria, March 2021.

[10] NIST. FIPS 203 (Initial Public Draft): Module-Lattice-Based Key-Encapsulation Mechanism Standard, 2023. https://csrc.nist.gov/pubs/fips/203/ipd.

[11] P. Schwabe and B. E. Westerbaan. Kyber Post-Quantum KEM, 2024. https://datatracker.ietf.org/doc/html/draft-cfrg-schwabe-kyber-04.

[12] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. pages 256–270. ACM, 2016.

[13] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. pages 1789–1806. ACM, 2017.