Formally verifying Kyber

Part I: Implementation correctness

José Bacelar Almeida[1,3], Manuel Barbosa[2,3], Gilles Barthe[4,5], Benjamin Grégoire[6], Vincent Laporte[7], Jean-Christophe Léchenet[6], Tiago Oliveira[4], Hugo Pacheco[2,3], Miguel Quaresma[4], Peter Schwabe[4,8], Antoine Séré[9] and Pierre-Yves Strub[10]

[1] Universidade do Minho, Braga, Portugal
jba@di.uminho.pt
[2] University of Porto (FCUP), Porto, Portugal
mbb@fc.up.pt
[3] INESC TEC, Porto, Portugal
[4] Max Planck Institute for Security and Privacy, Bochum, Germany
gilles.barthe@mpi-sp.org, tiago.oliveira@mpi-sp.org, miguel.quaresma@mpi-sp.org, peter@cryptojedi.org
[5] IMDEA Software Institute, Madrid, Spain
[6] Université Côte dAzur, Inria, France
benjamin.gregoire@inria.fr, jean-christophe.lechenet@inria.fr
[7] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
vincent.laporte@inria.fr
[8] Radboud University, Nijmegen, The Netherlands
[9] École Polytechnique, Paris, France
antoine.sere@protonmail.com
[10] pierre-yves@strub.nu

Abstract. In this paper we present the first formally verified implementations of Kyber and, to the best of our knowledge, the first such implementations of any post-quantum cryptosystem. We give a (readable) formal specification of Kyber in the EasyCrypt proof assistant, which is syntactically very close to the pseudocode description of the scheme as given in the most recent version of the NIST submission. We present high-assurance open-source implementations of Kyber written in the Jasmin language, along with machine-checked proofs that they are functionally correct with respect to the EasyCrypt specification. We describe a number of improvements to the EasyCrypt and Jasmin frameworks that were needed for this implementation and verification effort, and we present detailed benchmarks of our implementations, showing that our code achieves performance close to existing hand-optimized implementations in C and assembly.

Keywords: High-assurance cryptography, lattice-based KEMs, NIST PQC, Jasmin, EasyCrypt

1 Introduction

In July 2022, NIST announced the first batch of "winners" of the post-quantum project, i.e., schemes that will be forwarded to standardization [AAC+22]. To the surprise of many, this first batch contained three signature schemes (CRYSTALS-Dilithium [BDK+21, DKL+18], Falcon [FHK+20], and SPHINCS$^+$ [ABB+22, BHK+19]), but only one key-encapsulation mechanism (KEM): the lattice-based scheme CRYSTALS-Kyber (or in short just KYBER) [ABD$^+$21, BDK$^+$18]. This is remarkable in particular because most experts see the biggest urgency in post-quantum deployment for KEMs rather than signatures. The reason for this urgency is the *store-now-decrypt-later* attack scenario that breaks the confidentiality of ciphertexts sent today when, some time in the future, attackers have a quantum computer in their arsenal. Consequently, most early-adopter experiments with post-quantum cryptography focus on key encapsulation [Lan16, Lan18, KV19, Ope22] and we also expect that the first big wave of post-quantum deployment in the next few years—certainly after the NIST standards are ready—will lead to widespread use of the one KEM covered by those standards: KYBER.

This deployment, and the migration to post-quantum cryptography in general, presents a massive challenge for both technical and organizational reasons. Technical reasons include the fact that KEMs cannot in general be used as a drop-in replacement for ubiquitous Diffie-Hellman key exchange, the significantly larger sizes of public keys and ciphertexts compared to ECC, and the relative immaturity of side-channel protections. Examples of organizational reasons are modifications of protocol standards, updates to certification criteria, and strategies for updates of devices in the field.

However, this deployment of the next generation of asymmetric crypto also presents a huge opportunity to improve the deployed cryptographic software. Over the last decade, the field of *computer-aided cryptography* [BBB$^+$21] has brought the use of formal methods into cryptographic engineering. This resulted in high-performance cryptographic implementations with strong formal guarantees of functional correctness and implementation-specific security properties such as memory safety or absence of timing leaks. Indeed, high-assurance crypto software is already being used in, for example, the NSS library used in the Firefox web browser [ZBPB17, Beu17] and the BoringSSL library by Google [EPG$^+$19]. While these examples show that high-assurance cryptography is ready for real-world deployment, a more widespread deployment requires replacing existing implementations. The costs of this shift are often hard to justify, despite the stronger security guarantees of formally verified code.

The situation is different for the deployment of post-quantum crypto, where new software *must* be rolled out. In this work we give an important step to enable the deployment of the next generation of cryptographic primitives relying also on a new generation of high-assurance post-quantum cryptographic software.

Contributions. In this paper we present the first fully formally verified implementations of KYBER and, to the best of our knowledge, the first such implementations of any post-quantum cryptosystem. Completing this work represented a significant challenge that implied non-trivial improvements to pre-existing formal verification tools. The challenges resided not only in the scale of the verified code — which, for example, makes it impossible to work with fully inlined functions — but also in a number of optimizations that do not appear in pre-quantum cryptography, such as a fast Number Theoretic Transform (NTT), noise generation and rejection sampling, and data compression and decompression routines that have an impact in the correctness and security semantics.

More specifically, our contributions are the following:

1. We formalize the specification of KYBER [ABD$^+$21, Sec. 1] in EASYCRYPT [BGHZ11]. Great care (and effort) was dedicated to ensure that the EASYCRYPT specification is syntactically close enough to the KYBER specification to enable efficient human validation. This means, in particular, that we use *algorithmic* definitions of the randomness sampling routines and provide computer-verified proofs that these algorithms generate outputs that follow the distributions defined in the KYBER specification.
2. We extend the Jasmin programming language [ABB$^+$17, ABB$^+$20] with local functions, sub-arrays, pointer types required for call-by-reference, and system calls to obtain randomness. We communicated these extensions to the Jasmin developers and they have meanwhile been adopted upstream.

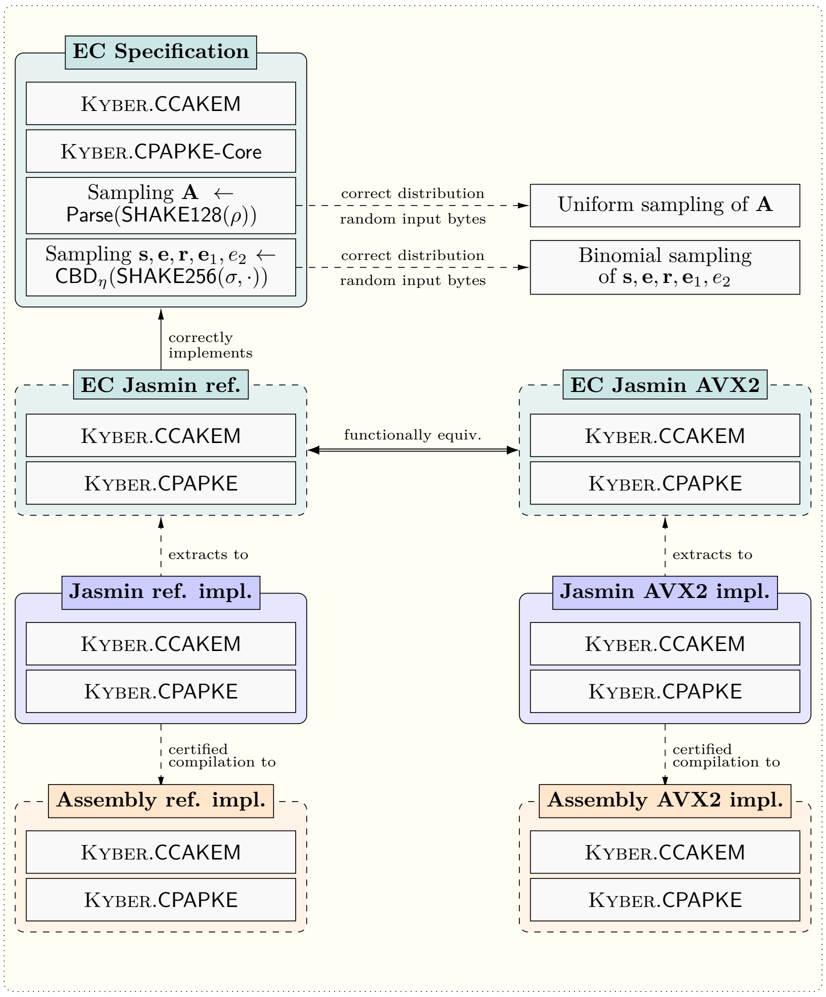

Figure 1: Overview of the proof structure and tooling. Elements in dashed boxes are automatically generated; elements in solid boxes need to be written by hand.

3. We present two implementations of Kyber-768 in Jasmin targeting the AMD64 architecture: one reference implementation and one optimized implementation using the AVX2 vector instruction set. We use the automated extraction provided by the Jasmin compiler to obtain EasyCrypt models of the two Jasmin implementations.
4. We extend EasyCrypt's standard library with new features required by Kyber, such as support for polynomial rings, signed representations of field elements, dealing with sub-arrays, etc. We created a library that specifies and validates several optimizations of NTT computations used in Kyber, but which is of interest for other post-quantum cryptosystems. We also add better support for reasoning about nested for loops, and automatic inference of functional specifications for simple EasyCrypt procedures. All of these enhancements reduce the need for human input when writing the proofs. We communicated these extensions to the EasyCrypt developers and they have also been adopted upstream.
5. We then provide two computer-verified proofs to establish the functional correctness of our implementations: First we prove that the reference Jasmin implementation correctly implements the specification of Kyber in EasyCrypt; Second we prove that the optimized AVX2 Jasmin implementation is equivalent to the reference im-

plementation.

6. Finally we provide comparative benchmarks of our implementations on different generations of Intel CPUs. As we did not prove correctness of the rejection-sampling routine in the AVX2 implementation, we benchmark two variants of our AVX2 implementation: one that is fully optimized and one that is fully verified. (The latter uses rejection sampling from the reference implementation.) These benchmarks show that the fully optimized AVX2 Jasmin implementation is within about 10% of the performance of the hand-optimized C/asm implementation by the Kyber team.

An overview of our implementation and verification effort and the tooling we use is given in Fig. 1.

What does our result mean? The use of the formally verified Jasmin compiler guarantees correct translation to assembly, so the behavior of the assembly code is guaranteed to match the behavior of the source Jasmin code. The computer-verified proofs of functional correctness establish the link between the implementations and the specification of Kyber and, ultimately, guarantee that the assembly code is indeed performing the computations specified by the Kyber specification.

However, there are two aspects that still require careful manual inspection by expert auditors: the formal specification of Kyber in EasyCrypt and the statements of the theorems that we prove. We therefore dedicate most of the paper to explaining our formalization choices with respect to these two aspects, and present only a summarized view of the proof itself, focusing on the most innovative aspects and documenting lessons learned for the future.

With regards to verification of implementation security, we refer to [ASBG+23], which reports formally proven protection against classical timing attacks and Spectre v1 attacks [KHF+19] for Jasmin implementations of multiple cryptographic routines, including Kyber. The implementations of Kyber considered in that paper are very close to the ones proven correct here. We are confident that we can use the same approach to prove constant-timeness and protect against Spectre v1, once the security type system in the Jasmin compiler from [ASBG+23] is integrated into the main branch. Also, we are confident that the performance overhead for Spectre v1 protections will be just as low (i.e., $< 1\%$) as reported in [ASBG+23].

Limitations and TCB. Our computer-verified proofs rely on a trusted code base (TCB) and are limited in scope to the Kyber specification. The correctness proofs of the Jasmin compiler rely on the assumption that the semantics of individual machine instructions are correctly formalized in Coq and EasyCrypt. The mechanism that extracts EasyCrypt code from Jasmin code is also part of the TCB. Finally, the EasyCrypt tool itself is part of the TCB. In terms of scope, our proofs do not cover the correctness of the SHA3 implementations, which come from the libjade library[1]. Some of this code has been verified in [ABB+19], but we do not establish a formal connection to this work.

We note also that the correctness theorem for the Jasmin compiler and the extraction to EasyCrypt only apply to memory-safe programs. However, automatically proving memory safety of our Kyber implementations is currently out of scope for the static safety analyzer in the Jasmin compiler. For this reason, we check memory safety using a combination of static analysis and dynamic analysis: the functional correctness proof guarantees that the memory accesses which are relevant for the computation of the scheme are safe, but they do not exclude accesses that cause unsafety due to, e.g., misalignment or simply accessing memory that is subsequently not used. For this reason we use the Jasmin interpreter and valgrind to reinforce the memory-safety guarantees.

The bigger picture. As the subtitle of this paper suggests, formally verifying the functional correctness and certain aspects of implementation security of Kyber is only part

[1] https://https://github.com/formosa-crypto/libjade

of the whole story. The use of EasyCrypt as specification language and verification tool allows us to also conduct mechanized reductionist proofs of security, i.e., establish through computer-verified proofs that Kyber is indeed an IND-CCA2 secure KEM. Our development is fully prepared so that, when the ongoing machine-checked proof of security is concluded, this yields a security theorem for the Kyber specification that, by the proofs presented in this paper, extends all the way down to the implementations at assembly level.

Related Work. While this paper is the first to present a fully computer-verified implementation of a post-quantum crypto scheme, it is not the first to present verification efforts in the realm of post-quantum cryptography. Hwang et al. [HLS+22] use CryptoLine[2] to formally verify the AVX2 implementation of the NTT transform used in the Kyber NIST submission. This is essentially the same code used for NTT in our AVX2 implementation, so our results corroborate the findings in [HLS+22]. However, our proof goes beyond NTT and covers two full Kyber implementations. We further discuss [HLS+22] in the concluding remarks of the paper. Kreuzer [Kre23] focuses on machine-checking the correctness of the Kyber specification as a cryptographic construction, i.e., a proof that encryption and decryption commute. Other works have focused on machine-checking proofs of security for post-quantum cryptographic constructions, such as [Unr20, BBF+21, HMS22]. These works are orthogonal to what we present in this paper, since here we are focusing on the correctness of the Kyber implementations. A comprehensive description of related work in computer-aided cryptography can be found in [BBB+21], including prior uses of Jasmin to obtain verified high-speed implementations of symmetric and elliptic-curve cryptographic primitives, as well as alternative frameworks designed to achieve the same goal.

Artifact. An artifact including our extended versions of Jasmin and EasyCrypt, the Jasmin implementations of Kyber, and the EasyCrypt proofs is available from `https://artifacts.formosa-crypto.org`. The artifact also includes a README with detailed instructions how to build the tools, build the software, validate the proofs in EasyCrypt, and run our benchmarks. The contents of this artifact are under permissive licenses (MIT license for all modifications to EasyCrypt and CC0 for all Jasmin implementations and associated proofs).

2 Specification of Kyber in EasyCrypt

The EasyCrypt specification of Kyber follows closely the description in the latest version (3.02) of the CRYSTALS-Kyber proposal [ABD+21], omitting some features that are not relevant for Kyber-768, namely noise sampling and compression algorithms that are only used for other Kyber parameter sets[3]. We will gradually introduce the various definitions and intermediate results leading to the full code in Figures 2 to 5. This section was written so that it is easy to match our code to the pseudocode specification of Kyber in [ABD+21, Sec. 1].

2.1 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`

[2]CryptoLine is a formal-verification framework that takes low-level code with user annotations for pre-conditions, post-conditions and intermediate assertions, and generates inputs for SMT solvers and computer algebra systems that can then be used to automatically check correctness. It is available from `https://github.com/fmlab-iis/cryptoline`.

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

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

  proc kg_derand(seed : W8.t Array32.t ∗ W8.t Array32.t) :
       pkey ∗ (skey ∗ pkey ∗ W8.t Array32.t ∗ W8.t Array32.t) = {
     kgs ← seed_1; z ← seed_2;
     (pk,sk) ← Kyber(HS,XOF,PRF,O).kg_derand(kgs);
     hpk ← KemH(O).pkH(pk);
     return (pk, (sk,pk,hpk,z));
  }

  proc enc_derand(pk : pkey, prem : W8.t Array32.t) :
      ciphertext ∗ W8.t Array32.t = {
     m ← KemG(KemHS).sample(prem);
     hpk ← KemH(O).pkH(pk);
     (_Kt,r) ← KemH(O).g(m,hpk);
     c ← Kyber(HS,XOF,PRF,O).enc_derand(pk,m,r);
     hc ← KemH(O).cH(c);
     _K ← KemH(O).kdf(_Kt,hc);
     return (c,_K);
  }

  proc dec(cph : ciphertext, sk : skey ∗ pkey ∗ W8.t Array32.t ∗ W8.t Array32.t) :
      W8.t Array32.t = {
     (skp,pk,hpk,z) ← sk;
     m ← Kyber(HS,XOF,PRF,O).dec(skp,cph);
     (_Kt,r) ← KemH(O).g(oget m,hpk);
     c ← Kyber(HS,XOF,PRF,O).enc_derand(pk,oget m,r);
     hc ← KemH(O).cH(cph);
     if (c = cph) {  _K ← KemH(O).kdf(_Kt,hc); }
     else { _K ← KemH(O).kdf(z,hc); }
     return _K;
  }
}.
```

Figure 2: KyberEasyCrypt Specification: IND-CCA KEM

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, as indicated in the Kyber specification. Compression and decompression map field elements into integers in the range $[0, 2^{\mathsf{d}})$ and back, with a loss that is captured by the following lemma stated and proved in EasyCrypt.

```
op compress_err(d : int, c: Fq) : Fq = decompress d (compress d c) − c.
lemma compress_err_bound (c:Fq) d :
          0 < d ⇒ 2^d < q ⇒  | as_sint (compress_err d c) | ≤ round (q /_R (2^(d+1))).
```

We pave the way to connecting the Kyber specification to Jasmin code by giving alternative definitions of compress and decompress—these match what is done in code using machine instructions—and prove that these are equivalent to the specifications above. The code computes compress using compress_alt for $d = 1$ and $d = 4$, and it uses compress_alt_large for $d = 10$.

```
op compress_alt(d : int, c : Fq) : int = (asint c ∗ 2^d + (q+1)/2) ∗ (2^28/q) / 2^28 % 2^d.
op compress_alt_large(c : Fq) : int = (asint c ∗ 2^10 + (q+1)/2) ∗ (2^32/q) / 2^32 % 2^10.
op decompress_alt(d : int, c : int) : Fq = inFq ((c∗q + 2^(d−1)) / 2^d).
```

Using these operators we can also capture the encoding/decoding of bits as finite-field

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

proc dec(sk : skey, cph : ciphertext) : plaintext option = {
  (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 Some 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(O).init(rho,W8.of_int i, W8.of_int j);
      c ← Parse(XOF,O).sample();
      aT[(i,j)] ← c;
      j ← j + 1;
    }
    i ← i + 1;
  }
  i ← 0;
  while (i < kvec) {
    c ← CBD2(PRF).sample(r,_N);
    rv ← set rv i c;
    _N ← _N + 1;
    i ← i + 1;
  }
  i ← 0;
  while (i < kvec) {
    c ← CBD2(PRF).sample(r,_N);
    e1 ← set e1 i c;
    _N ← _N + 1;
    i ← i + 1;
  }
  e2 ← CBD2(PRF).sample(r,_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: KyberEasyCrypt Specification: IND-CPA PKE

elements and relate the semantics of these operations to that of the compression and decompression operators, as is done in the Kyber spec.

```
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 that are used by Kyber. 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 (this is important for proving correctness of the sampling procedures used by the implementation). We omit the details for brevity.

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

The ring $\mathcal{R}_q$, matrices and vectors. At this point we can formalize the ring $\mathcal{R}_q$ of polynomials over which Kyber operates. We start by defining a concrete representation of

```
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: KyberEasyCrypt Specification: Rejection sampling a uniform element in $\mathcal{R}_q$.

these polynomials using arrays of size 256 of field elements. As in the Kyber 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 $\mathbb{Z}_q[X]/(X^{256}+1)$ as follows. We first instantiate the generic polynomial theory PolyReduce recently added to EasyCrypt with $n = 256$ and $q = 3329$, to obtain a description of the corresponding mathematical ring and its operations. We then define a bijection between our poly type above and the type of ring elements in the resulting theory, which allows us to easily prove that our ring operations above are correct with respect to the explicit formulae defined by PolyReduce.

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 Kyber-768, and the 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 Kyber. We define the NTT forward and inverse operations as literal translations of what is stated in the Kyber specifications (including the use of the same representation for inputs, and outputs, and the operation that commutes with the NTT transform for fast multiplication in $\mathcal{R}_q$). We omit here the definition of NTT for vectors and matrices of

```
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: KyberEasyCrypt Specification: Sampling noise from the Binomial distribution.

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.

```
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 $\sum_{j=0}^{127}$ p[2∗$j$] ∗ zroot$^{(2*\text{br ii}+1)*j}$
                  else $\sum_{j=0}^{127}$ p[2∗$j$+1] ∗ zroot $^{(2*\text{br ii}+1)*j}$.

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

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$^{(2*\text{br ii}+1)}$)$_1$
                  else (cmplx_mul (a[2∗ii],a[2∗ii+1]) (b[2∗ii],b[2∗ii+1]) (zroot$^{(2*\text{br ii}+1)}$)$_2$
```

In Section 5.3 we describe the EasyCrypt library that was developed with general results about the NTT transform, which we used to establish the following properties, as well as natural extensions for vectors and matrices of ring elements.

```
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 (see Section 4). We note that these are well known results that are taken as a given in the Kyber specification and, to the best of our knowledge, have not been previously formally verified in the context of cryptographic proofs.[4] As stated in the introduction,we believe that these results are of independent interest, since the NTT is pervasively used in post-quantum cryptography and will surely be used in future EasyCrypt developments.

2.2 Algorithmic Sampling, Encoding and Decoding

The Kyber specification relies on a number of auxiliary algorithms. We follow the same structure and define them as independent EasyCrypt modules used by the main Kyber 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. In Section 7 we further discuss how our formalization and an ongoing machine-checked security proof of the specification clarify the assumptions that Kyber makes about SHA3 algorithms.

Rejection sampling a uniform ring element. The first auxiliary algorithm used by Kyber is a rejection sampling procedure called Parse, which we give in Figure 4. 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 $\mathcal{R}_q$. We note that in Kyber 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 Kyber uses SHAKE-128 for this purpose.

Using SHA3 to hash, smooth randomness, and sample noise. Kyber uses SHA3 algorithms over strings of different input lengths to implement the various hash functions required for the KEM (Fujisaki-Okamoto-like) construction. Our specification clarifies exactly how the SHA3 algorithms (a combination of SHA3-256 and SHAKE-256) are being used to "instantiate" these hash functions, as discussed in Section 7.

Kyber also uses SHA3 algorithms to expand the raw randomness fed to key generation into two 32-byte seeds, one for the matrix rejection sampling, and another to be used as PRF key in the noise sampling procedure above. Here SHA3-256 and SHA3-512 are used as a pseudorandom generators (PRG), referred in the spec as G. Since this PRG maps fixed-length input strings to fixed-length output strings, we model it as a PRF that can be evaluated on a single point.

Finally, Kyber 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 Kyber specification for the concrete parameter of $\eta = 2$, is used by the implementation of Kyber-768. Here we model SHAKE-256 as a Pseudo Random Function (PRF), using the corresponding EasyCrypt library and an operator SHA3_PRF.

In the proofs we describe in this paper we do not assume any security semantics to these uses of SHA3; this will be subject of a follow-up ongoing work to prove the security of

[4]In independent works [Kre23, HLS+22] have also looked at the NTT transformation. In [Kre23] the specification of the transform is similar to the one we use; the authors use this to show that the Kyber specification is *cryptographically* a correct KEM and, for this, the authors need to prove a set of properties about the NTT that is close to what we also prove. The specification used in [HLS+22] is the mathematical view based on the Chinese Remainder Theorem, rather than the convolution given in the Kyber specification. This specification is used to show correctness of the NTT AVX2 computations, something that we also do as an intermediate result. Our results could be formally related to those in [Kre23, HLS+22] by showing that the NTT specifications are the same.

```
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.

the Kyber specification and, by transitivity, security of the implementations. Instead, we simply assume that the Kyber specification is using SHA3 algorithms in exactly the same way that the implementation is using SHA3 algorithms and, prove, under this assumption, that the implementation is functionally correct with respect to the specification. We note that some of the implementations of SHA3 algorithms that we use in the development have been proved correct in [ABB+19], but we do not establish a formal connection between the two verification results. This means that, for the time being, correctness of the SHA3 library is an assumption in our proof.

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 Kyber specification [ABD+21], 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) = \sum_{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))).
```

The Kyber specification [ABD+21] leaves encoding implicitly defined as the inverse of the decoding operation. 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 Kyber specification, and finally prove that encoding inverts decoding. We discuss this in more detail in Section 3.2.

The full specification. The code given in Figures 2 to 5 and the code snippets we provide throughout this section closely match the pseudo-code and definitions in the Kyber specification [ABD+21]. As discussed in the introduction, this is relevant as this specification is the centerpiece of our effort that all formal-verification results refer to. Hence, their applicability to Kyber itself depends on how faithfully it matches what is written in the handwritten proposal. For this reason, our goal was to simplify the task of a human checking by inspection that the EC specification we described in this section matches the hand-written description[5].

3 Our toolchain: Jasmin and EasyCrypt

3.1 Jasmin before and after this paper

Jasmin is a programming language and a set of accompanying tools designed to allow the efficient implementation and the formal verification of cryptographic primitives. It empowers programmers with precise control over low-level features of the assembly code produced by the Jasmin compiler, such as instruction selection and register spilling. Yet, the language provides high-level abstractions such as functions and arrays that help structuring programs and proofs. For instance, arrays have a functional semantics and functions are call-by-value, vastly simplifying modular reasoning. In accordance with the high-efficiency objective, most abstractions are eliminated at compile-time; the compiler checks that arrays are used linearly, which in turn allows to modify them in-place and ensure that there are no run-time copies.

Before it was improved with the extensions described in this work, the Jasmin compiler had a few limitations. First of all, running a computation on a slice of an array (i.e., a contiguous region within an array as opposed to the full array) was very tedious and pollutes the proofs. Second, the compiler inlined all functions at every call-site, which dramatically increased the code size. For instance, the Jasmin implementations of the SHA3 algorithms from previous work [ABB+19] are already quite large (hundreds or thousands of assembly instructions, depending on the parameters). The Kyber implementations rely on this primitive for various purposes, such that inlining it at every call-site would inevitably result in an unaffordable code blow-up.

To overcome these limitations and allow the efficient implementation of large routines such as Kyber, the Jasmin infrastructure has been extended with three new features described below: local functions, pointers and sub-arrays, and a `randombytes` system call.

Local functions. Jasmin programs now are made of functions: at the source level, they now come in three flavors: export functions, inline functions, and local functions. Export functions correspond to entry-points, meant to be called from other programs. Therefore they must comply, after compilation, with the targeted ABI. In particular, the number of arguments and returned values is limited and callee-saved registers must be preserved. Moreover, their argument and return types are limited to machine words. At the other end of the spectrum, inline functions disappear early in the compilation process. They thus enjoy a lot of flexibility without incurring any run-time cost: they can take any number of arguments of any types, including arrays and return as many as necessary.

Finally, local functions are kept as separate entities after compilation. Yet, they are not exposed to external users and therefore need not comply with any convention. For instance, they can return many results and the programmer chooses whether the return address is passed through the stack memory or through a register. Interestingly, there is no fixed list of callee-saved registers, i.e., registers whose contents must be preserved by function calls. This ensures that there will be no hidden cost to using a function. This is made possible by a rather unique method to register allocation: all functions are compiled together and all their `reg` local variables are allocated simultaneously.

Arrays, sub-arrays and implicit pointers to stack. In Jasmin, each variable is labeled either `reg` or `stack`, specifying that the compiler must allocate it in a register or on the

[5]The specification of Kyber described in Section 2 can be found in our artifact in the file `proof/spec/Kyber.ec` starting from line 1193.

stack respectively. This is the case also for arrays. reg arrays are a convenient way to manipulate several reg variables together. They are compiled to individual registers. stack arrays are allocated as contiguous memory zones in the stack. The compiler, at the stack allocation pass, decides on a shape for the stack, and replaces accesses to arrays with memory accesses at positions relative to SP, the stack pointer. For instance, if array b is allocated on the stack at address SP + 5, an access b[r+3], where r is a register, is compiled to a memory access at address SP + r + 8.

A new feature that represented a major change to the compiler was allowing local functions to accept stack arrays as arguments. This implied a redesign of the stack allocation pass: there are now arrays whose address is not known at compile time. To achieve that, we impose that arrays in the stack are taken by local functions using a new type called reg ptr. At the source level, such an array behaves as the other arrays. Jasmin semantics makes no distinction between arrays of different kinds, thus reasoning is not more complex than before. However reg ptr is compiled as a pointer, to a register holding its address. The stack allocation pass checks that this transformation is valid, meaning that all modifications to the array are done in place. If an array is changed in the body of a local function, it has to be returned to abide by the high-level semantics of Jasmin. This spurious copy will be eliminated during stack allocation. Since reg ptr arrays are actually registers, we need a way to let the user spill them if needed. For that matter, we introduced stack ptr array.

In the version of Jasmin prior to this work, arrays could be passed as arguments to local functions, but there was no way to call a function with only part of the array without making an explicit copy. To make the best use of local functionswe allow the user to take slices of arrays called sub-arrays. This allows to call a local function on a small part of a large array. The starting position and length must for now be known at compile time. If the array is a reg ptr, this is compiled as pointer arithmetic.

Note that the features described in this section are the only facility given to the user to manipulate (indirectly) pointers. The fact that the user never gets direct access to pointers ensures that formally reasoning on the source remains tractable.

Randombytes. Many cryptographic routines process random data. A common practice is to receive randomness as input. However, KYBER (as well as other post-quantum schemes) uses a rejection sampling procedure which may consume an unbounded stream of random data. To overcome this limitation it proves convenient to sample fresh randomness on demand. To this end there is now a randombytes interface in Jasmin. The usual method of taking random data as input remains of course available. Moreover, the user of the Jasmin program is responsible for implementing this interface: no actual implementation is needed before link time. Possible implementation methods include asking the OS through a system call, or calling some dedicated library, e.g., to access a hardware-security module.

Semantically, this is modeled as an unspecified deterministic function in a state monad. This ensures that the compiler correctness theorem applies seamlessly. The EASYCRYPT formalization uses a slightly different view: the EASYCRYPT code is parameterized by a module providing the implementation of this system call. Said implementation can indeed be stateful. The functional correctness of the KYBER implementations can be established for any instantiation of the parameter, since they will match the semantics of the specification for any input random bytes. Security analysis (part II of this paper that is ongoing work) however will need to make assumptions about the distribution.

More generally, there is a general facility to call external functions from Jasmin programs. Currently only randombytes is available through this mechanism but, should the need arise, other operations can be seamlessly added.

3.2 EasyCrypt and extensions added in this work

EasyCrypt is a proof assistant geared towards proving the correctness and security of implementations of cryptographic primitives. The Jasmin compiler can output either assembly or EasyCrypt code with semantics equivalent to its input. Properties about this EasyCrypt code can then be proven in EasyCrypt using one of the variants of Hoare logic it features:

- HL is a deterministic Hoare logic that allows to prove result on deterministic programs. Statements in this logic are of the form **hoare** [M.foo : P ⇛ Q], where M.foo is a procedure in module M, P is a precondition and Q is a post condition.
- pHL is a probabilistic Hoare logic, that allows to reason about the probability of occurrence of events in potentially probabilistic programs. Statements in this logic are of the form **phoare** [M.foo : P ⇛ Q] = p, where p is now the probability that Q holds in the post-state. In the current paper we typically have p=1 and the pHL statement can be proved by first establishing that the program always terminates, and then proving the statement using HL.
- pRHL is a probabilistic relational Hoare logic, that allows to prove relational results about the lock-step execution of a pair of programs. Statements in this logic are of the form **equiv** [M.foo ~ N.foo : rP ⇛ rQ], where rP and rQ are now relational pre- and post- conditions that can refer to the initial and final states of both procedures.

In particular, pRHL can be used in conjunction with pHL to prove results by *game-hopping*: using a correctness result about a procedure P_1 in pHL and an equivalence result between procedures P_1 and P_2 in pRHL, one can prove a correctness result about P_2. Also, in a deterministic setting, results in HL can be proven using results in pHL, as well as the other way around. The proof of correctness for the Kyber Jasmin code relies on these two mechanisms. In particular, the game-hopping strategy allows us to build intermediate programs between the EasyCrypt code produced by Jasmin and our desired specification. We use it to remove optimizations and to abstract operations on algebraic structures.

EasyCrypt also features a standard library formalizing some basic algebraic structures, which we use to capture the algebraic structure underlying Kyber. This library is based on the EasyCrypt cloning mechanism, originally introduced to manage abstract types in cryptographic primitives. One thing that became clear during this project is that cloning has proven to be a source of complexity and limitations as more algebraic structures were added to the hierarchy, namely polynomial rings, vectors and matrices thereof needed for Kyber. We have communicated this to the EasyCrypt developers and learned that future versions of EasyCrypt will use type classes for this purpose.

The most innovative use of EasyCrypt in this development was, perhaps, the functional correctness of the Number Theoretic Transform (NTT) implementations, which we will describe in detail later in the paper. However, a number of enhancement to EasyCrypt were created to support this development, which we summarize next.

Additions to the standard library. Kyber is the first cryptographic primitive studied using EasyCrypt to involve algebraic structures such as polynomials arithmetic, modular rings and ring quotients. As a result, many parts of the standard library had to be created or improved to lay foundations underlying the Kyber specification. These included results about lists, bit-vector manipulations, ring quotients, logarithm and p-adic valuations.

Automatic inference of functional specifications. One of the tedious aspects of some EasyCrypt proofs arises from the fact that some parts of the proofs are easier to do with functional specifications, whereas other parts are easier to do with imperative specifications. The sweet spot usually resides in using both approaches at the same time, which means writing both specifications by hand and proving that they are equivalent. In Kyber, a number of simple procedures are used to encode and decode data into byte arrays, so applying this approach would be quite time consuming. To avoid this, a new mechanism was developed that can automatically infer a functional specification from an imperative program, provided that the latter has as reasonably simple control flow.

Dealing with nested for loops in EasyCrypt. The reference implementation of the NTT in Kyber involves multiple nested for loops whose formal study is tedious in EasyCrypt. Indeed, for dealing with loops, EasyCrypt only comes with a restricted set of general purposes proof rules that trade expressiveness for ease of use. In consequence, we designed a new library that recasts the aforementioned general purpose rules as simplest ones but that only apply to more restricted forms of while loops, as the ones that can be found in Kyber. This library targets the different flavors of EasyCrypt's Hoare logic and has proven particularly helpful in *game-hopping* based proofs on programs that contain multiple nested for loops.

4 Implementation

This paper presents two different implementations of Kyber-768, a reference implementation and an optimized implementation using AVX2 vector instructions. Those implementations follow largely the structure and implementation strategies of the `ref` and `avx2` implementations by the Kyber team[6], which are also included in the submission package to NIST and integrated into PQClean[7].

Reference implementation. The structure of the reference implementation largely follows the specification. On the lowest level they implement arithmetic in $\mathbb{F}_q$, i.e. on integers modulo the prime q; these are represented as signed 16-bit integers. The most interesting routines on this level are Montgomery and Barrett reductions. The Montgomery reduction takes as input a signed 32-bit integer a and produces a signed 16-bit output r with $r \equiv a \cdot 2^{16} \pmod{q}$. This function does not actually work for arbitrary inputs and is also no producing fully reduced results: more specifically, it requires the input to be in $\{-q \cdot 2^{15}, \dots, q \cdot 2^{15} - 1\}$ and guarantees that the output is in $\{-q+1, \dots, q-1\}$. Barrett reduction takes an arbitrary signed 16-bit integer a as input and produces $r \in \{-(q-1)/2, \dots, (q-1)/2\}$, such that $r \equiv a \pmod{q}$.

On top of this is arithmetic in $\mathcal{R}_q$, i.e., on univariate polynomials over $\mathbb{F}_q$ modulo $(X^{256}+1)$. Addition and subtraction in $\mathcal{R}_q$ are simply coefficient-wise addition and subtraction of arrays of 16-bit integers. The implementation follows a lazy-reduction approach here and does not apply modular reduction after addition and subtraction. The more interesting routines on polynomials are forward and inverse NTT, and "pointwise" multiplication of polynomials in NTT domain. The forward NTT takes an array of 256 coefficients in normal order and produces the output inplace in bitreversed order. It uses 7 layers of Cooley-Tukey butterfly operations. Twiddle factors are precomputed in Montgomery and multiplication by those twiddle factors is followed by Montgomery reduction, bringing the result back to normal domain. Addition and subtraction inside the butterflies is not followed by modular reduction, i.e., also here the implementation makes use of the lazy-reduction optimization. The inverse NTT takes input in bitreversed order and uses 7 layers of Gentleman-Sande butterflies to produce output inplace in normal order. Multiplication by twiddle factors is again using Montgomery arithmetic. The result of addition inside the butterflies is reduced using the Barrett reduction routine. The result of subtraction is again left unreduced. As the NTT in Kyber is incomplete (see [ABD+21, Sec. 1.1]), the pointwise multiplication of polynomials in NTT domain corresponds to 128

[6]See `https://github.com/pq-crystals/kyber`.
[7]See `https://github.com/PQClean/PQClean`

multiplications of 2-coefficient polynomials modulo $X^2 - \zeta^i$, where ζ^i are precomputed powers of $\zeta = 17$, the smallest 256-th root of unity modulo q. This operation is computed through Montgomery multiplications and additions that again omit modular reduction. Note that Montgomery reduction moves resulting coefficients to "inverse Montgomery domain", i.e., they get multiplied by a factor of 2^{-16}. This factor is removed again at the very end of the inverse NTT. For the public key, which is transmitted in NTT domain, the factor is removed in a dedicated function that multiplies each coefficient by $2^{32} \mod q$ and then applies Montgomery reduction.

Also on this layer are functions to sample polynomials with coefficients that are uniformly distributed modulo q and to sample noise polynomials with coefficients that follow a centered binomial distribution with parameter $\eta = 2$. The former is using simple rejection sampling on output of SHAKE-128. The latter is a bit more involved and treats 32-bit words (in the C reference implementation) or bytes (in the Jasmin reference implementation) as vectors of 2-bit integers to compute Hamming weight and subsequent subtraction. Finally, the implementation of $\mathcal{R}_q$ includes functions for serialization and deserialization and for "LWR-style" rounding to a smaller, power-of-two modulus to compress ciphertexts. For the reference implementations, these functions are rather straightforward translations of the specification into C respectively Jasmin.

The highest level of arithmetic implements operations on vectors and matrices over $\mathcal{R}_q$. These functions are all rather straightforward loops over calls to functions performing arithmetic in $\mathcal{R}_q$. These functions are first combined to implement the KYBER.CPAPKE passively secure public-key encryption scheme and then, together with multiple functions derived from Keccak (SHA3-256, SHA3-512, SHAKE-128, SHAKE-256) to build the full IND-CCA2 secure KEM KYBER.

AVX2 implementation. As most optimized implementations, the main idea of the AVX2 implementation code is to identify the functions that consume most of the CPU cycles in the reference implementation and, one-by-one, replace those by more optimized versions. One might think that verification "simply" proves equivalence between the reference and the optimized implementations function-by-function and that this verification would mostly mean showing equivalence of longer loops over scalar operations and shorter loops over vector operations. While this is true for some simple functions like polynomial addition and subtraction, the differences are much more substantial for the bulk of the AVX2 implementation. The most interesting differences between the reference and the AVX2 implementation are the following:

- Whenever possible, the AVX2 implementation uses a Keccak implementation that processes 4 independent streams of input and output in parallel. Using this more efficient $4\times$ parallel implementation requires the high-level code to batch calls; the vectorization is thus not a local change encapsulated inside a low-level function anymore.

- The AVX2 implementation of the NTT produces output in a "permuted bitreversed" order; this order is also used by the pointwise multiplication, so all polynomials in NTT domain use a different ordering of coefficients in the AVX2 implementation compared to the reference implementation. As the public key of KYBER is transmitted in NTT domain, serialization of the public key includes a permutation to move to standard bitreversed order.

- The AVX2 NTT code is also an illustrative example of where the aggressive optimization strategies, together with the willingness to maximize the impact of the available AVX2 registers and instruction-set, often leads to ingenious but contrived code, intertwining arithmetic and bitwise operations. Correctness verification is challenging, since one needs to keep track of what is happening at each step with significant detail, but the deep understanding of the underlying arguments occasionally suggested tweaks for additional speed-ups that were incorporated into the code.

- The AVX2 modular reductions are subtly different, so they produce output that is equivalent modulo q, but not equal to those produced by the reference implementation. This also means that the whole lazy-reduction strategy of the AVX2 implementation is different, which requires updates to the range analysis to make sure that no overflows occur.
- The compression function for vectors of polynomials uses a different algebraic expression to compute the rounding to the smaller power-of-two modulus. More specifically, while the reference implementation computes rounding from $q = 3329$ to $q' = 1024$ as `((1024*i + (Q-1)/2)/Q) & 1023`, the AVX2 implementation uses `(8*V*i - (i+15) + (1 << 18)) >> 19`, with the value `V` set to 20159. The equivalence of these two expressions for all relevant inputs is in principle not hard to prove, but the situation is further complicated by the way how results of vectorized multiplications are split into low and high parts of the result. In other words the complication is extracting this algebraic expression from the code in the first place.
- Finally, the rejection sampling to generate the matrix $\mathbf{A}$ is using a rather involved vectorization approach inspired by the techniques introduced for NewHope in [GS16, Sec. 4.2]. This routine is the only part of the AVX2 implementation that is not yet proven correct; this is why Section 6 reports performance results for a fully optimized (with this involved AVX2-based sampling) and a fully proven (with the sampling from the reference implementation) implementation. This faster rejection sampling routine is challenging to verify because it takes a totally different approach to generating the necessary coefficients than the reference implementation: rather than going sequentially through the output bytes produced by the XOF, it generates a large number of candidates and checks their bounds in parallel, to compute what intuitively is a bit map of the good elements; the code then uses this information to *pack* these good elements in a contiguous memory region, and then takes the necessary prefix. Since the used set of SIMD instructions and optimization thechniques have little in common with the rest of the code, verifying this part of the implementation represents significant additional effort.

5 Proof of functional correctness

The strategy we adopted for the functional correctness proof divided the effort into a first phase, in which we constructed a proof for the reference implementation, and a second phase where we addressed the AVX2 implementation. The first phase included the development of a number of EasyCrypt libraries to deal with various data structures and operators used by the EasyCrypt code extracted from the Jasmin implementations, namely distributions over arrays, conversions between arrays of field elements and their representations, etc. One important library that may be reused in future developments establishes results for field elements in Montgomery representation, including not only the expected congruence relations, but also the ranges of the representations of these elements when stored as microprocessor words by the implementation. For the latter, a formalization and proof of the Barrett reduction operation correctness. Another one establishes results for the NTT, which we will describe in detail at the end of this section.

5.1 Verifying the Reference Implementation

The proof for the reference implementation follows the hierarchical structure of the code described in Section 4. We present the results we establish in a bottom-up order.

Field operations. The lowest-level results concern field-level operations for addition, multiplication, subtraction, and Barrett reduction. The following example shows the proved lemma for field multiplication in Montgomery representation.

```
lemma fqmul_corr _a _b : phoare [ M.__fqmul :
  W16.to_sint a = _a  ∧  W16.to_sint b = _b ⇒  W16.to_sint res = SREDC (_a ∗ _b)] = 1.
```

The lemma states that the Jasmin function `__fqmul`, given two 16-bit words `a` and `b`, produces a result **res** with probability 1, meaning that it always terminates, satisfying the expected relation to the input parameter values: the value stored in the output 16-bit word, when seen as a signed integer, contains the output of a specification operator `SREDC` computed over the inputs. The operator is defined in the `Montgomery` theory as follows:

```
op SREDC (a: int) : int =
  let u = smod (a ∗ qinv ∗ R) (R^2) in
  let t = smod (a − u / R ∗ q) (R^2) in smod (t / R % (R^2)) R.

lemma SREDCp_corr a:
  0 < q < R / 2 ⇒
  −R / 2 ∗ q ≤ a < R / 2 ∗ q ⇒  −q ≤ SREDC a ≤ q  ∧ SREDC a % q = (a ∗ Rinv) % q.
```

Note that the operator semantics is given by the above lemma both in terms of its algebraic meaning and input/output value ranges. (Here `R` is 2^{16}, the range of 16-bit words.) Note also that this lemma imposes a precondition on the ranges of input values, which must be met by caller functions.

At the field level we also state and prove auxiliary lemmas that relate some sequences of processor operations used by higher-level functions to the field element compression operators introduced in Section 2. These are then used in the proofs of correctness for ring element and vector operations.

Ring (polynomial) operations. Operations over ring elements come in three flavors: i. those that follow from the pointwise application of some operation to each coefficient; ii. those that involve reading and writing polynomials from memory; and iii. NTT-related operations. The first type of operations includes addition, subtraction, Barrett reduction, etc. The correctness proofs for these Jasmin functions are straightforward loop arguments that rely on the lower level results described above and stated similarly to the **phoare** judgment given as an example for field multiplication.

The operations involving memory access typically perform compression followed by memory write operations, or memory read operations followed by decompression. The values read from/written to memory are decoded/encoded into byte arrays, as described in Section 2. This motivates a different form of functional correctness statement, which we illustrate with the Jasmin function that writes a compressed ring element to memory.

```
lemma poly_compress_corr _a _p mem :
  equiv [ M._poly_compress ~ EncDec.encode4 :
    pos_bound256_cxq a{1} 0 256 2  ∧ lift_array256 a{1} = _a  ∧ p{2} = compress_poly 4 _a  ∧
    valid_ptr _p 128  ∧ Glob.mem_1 = mem  ∧ to_uint rp{1} = _p    ⇒
      lift_array256 res{1} = _a  ∧ pos_bound256_cxq res{1} 0 256 1  ∧
      touches mem Glob.mem{1} _p 128  ∧ load_array128 Glob.mem{1} _p = res{2}].
```

The above lemma establishes the equivalence between the specification of the encoding procedure `EncDec.encode4` and the `_poly_compress` Jasmin function. It states that, when the input parameter `a{1}` to the Jasmin function contains the representation (the `lift_array256` operator) of a ring element _a with coefficients in the $[0..2q)$ range (pos_bound_256_cxq predicate), and if it starts executing in some initial memory mem, then the final state of the memory will satisfy two properties: i. memory is only modified in a specific region identified via the touches predicate; and ii. reading from that region yields the correct encoding result (this is stated by fixing the input p{2} to EncDec.encode4 on the right to be the result of compressing ring element _a. Expressing correctness in this way has two advantages: i. we can match the control flow of the encoding specification to the control flow of the implementation (see Section 2) and rely on relational Hoare logic to avoid dealing with closed forms for the encoding mechanisms in this part of the proof and; ii. we can directly plug-in these **equiv** results into the top level equivalence proof for the IND-CPA encryption scheme.

Finally, the most challenging steps in the proof of ring operations concern the three functions used for NTT-based multiplication: NTT, inverse NTT and pointwise multiplication of degree one polynomials. We prove correctness of the NTT and inverse NTT functions using the same sequence of steps. We first show that the NTT function does not introduce computation errors at the representation level; to this end, we define an EasyCrypt procedure that has the same control flow as the Jasmin implementation, but performs all operations in $\mathbb{F}_q$. This is expressed as the following equivalence in EasyCrypt between our idealized NTT implementation NTT.ntt and the actual extracted code _poly_ntt.

```
equiv ntt_correct_aux :
  NTT.ntt ~ M._poly_ntt :
  r{1} = lift_array256 rp{2}  ∧ zetas{1} = zetas  ∧ signed_bound_cxq rp{2} 0 256 2  ⇒
    res{1} = lift_array256 res{2}  ∧ pos_bound256_cxq res{2} 0 256 2.
```

This equivalence imposes a contract on the ranges of input/output representations for the implementation, and it shows that our idealized code, when called with the ring-element represented by the input to the Jasmin code, and a set of global constants (zetas) used by the Jasmin code, computes the same result over the ring.

We then prove that the global constants used by the implementation are correct representations of the necessary roots of unity, and we prove that our idealized algorithm computes the mathematical NTT operator. This part of the proof relies on an NTT library developed for this project, which we describe in more detail in Section 5.3. The final statement looks as follows:

```
lemma ntt_correct _r :
  phoare[ M._poly_ntt :
  _r = lift_array256 rp  ∧ signed_bound_cxq rp 0 256 2  ⇒
    ntt _r = lift_array256 res  ∧ pos_bound256_cxq res{2} 0 256 2] = 1.
```

The correctness proof for the _poly_basemul and NTT inverse operations come with an interesting twist. Rather than computing the pointwise multiplication of the degree one polynomials, the former actually uses the _fqmul field multiplication operation in a way that leaves an extra Montgomery inverse multiplicative term in the result. Conversely, the NTT inverse operation produces a result that carries an uncancelled Montgomery multiplicative term. This is achieved by introducing an additional Montgomery multiplicative term in the precomputed constants, and therefore has no impact on performance. Since the NTT transform preserves multiplicative scalars (a property we have proved), the composition of these two functions actually produces a correct result without any spurious terms. In Section 5.3 we give some more detail on the EasyCrypt library we developed to support the NTT proofs.

Vector operations. Most operations over vectors do not bring any additional challenges: some of them work coefficient-wise over all the coefficients of all ring elements in the vector, and therefore almost identical to the corresponding ring operations, or they are point-wise operations over ring elements (e.g., vector addition, NTT, etc.) which we simply handle by plugging in the results described above three times. The remaining and most interesting operation is the inner product between two vectors in the NTT domain, which we prove correct by relying on the correctness of the basemul and poly_add functions, to get the following result. (Here again we note the extra Montgomery inverse multiplicative term (inFq 169) that will be subsequently canceled by the inverse NTT operation.)

```
lemma polyvec_pointwise_acc_corr_alg va vb :
  phoare [ M.___polyvec_pointwise_acc :
    nttv va = lift_vector a  ∧ nttv vb = lift_vector b  ∧
    signed_bound768_cxq a 0 768 2  ∧ signed_bound768_cxq b 0 768 2  ⇒
      signed_bound_cxq res 0 256 2  ∧
      lift_array256 res = scale (ntt (dotp va vb)) (inFq 169) ] = 1
```

Top-level correctness. At the top level of the correctness proof reside the lemmas that establish an equivalence between the Kyber specifications of the IND-CPA encryption scheme and the KEM construction described in Section 2 and the corresponding Jasmin implementations. The proof of the KEM implementation correctness follows directly from the fact that the IND-CPA implementation is correct and the various calls to the hash functions in the Fujisaki-Okamoto-like construction are done in the correct sequence. Two additional simple results are needed for comparing two arrays and conditionally copying an array in constant-time.

The proof of the IND-CPA encryption scheme implementation is done in two steps. We first create an EasyCrypt auxiliary implementation of the scheme that is very close to the extracted Jasmin code, bar the fact that we restructure it slightly to better match the structure of the specification and we create additional mock versions of the matrix and noise generation functions, as shown in Figure 7. These functions have the same type as the extracted ones, but exactly match the specification modulo the representation of outputs, which here comes as arrays of processor words. As an intermediate step we prove that these functions are equivalent to the corresponding ones in the extracted Jasmin code, assuming that both use SHA3 in the same way: to this end we parameterize the mock implementations in Figure 7 with concrete XOF and a PRF modules that call SHA3 code in exactly the same way as the Jasmin implementations do. Looking back at Figure 1, this explains why we show the EC specification parameterized by SHAKE-128 and SHAKE256. In the same figure we also show that we have proved as important side-results that, should the XOF and PRF modules we show above be replaced with purely random coins, then the noise sampling and matrix sampling procedures in the Kyber specification produce the correct output distribution.

The top-level IND-CPA proof then proceeds by connecting our intermediate implementation to the Kyber specification by using the lower level results described above for ring elements and vectors thereof. The proof boils down to guaranteeing that plugging the outputs of one function into the next always satisfies the contracts that constrain the ranges of the representations. When it comes to noise sampling and matrix sampling, we first replace the extracted code with the mock implementation, which allows us to conclude the equivalence proof in a trivial way. An example of a top-level result is given in Figure 8 for encapsulation.

In addition to the validity of the input memory regions, the precondition binds these inputs to symbolic constants representing pointers to the input public key provided to the implementations, and specifies that the input public key provided to the specification

```
proc ___gen_matrix(seed : W8.t Array32.t, trans : bool) : W16.t Array2304.t = {
  i ← 0;
  while (i < kvec) {
    j ← 0;
    while (j < kvec) {
      XOF(H).init(seed, if trans then W8.of_int i else W8.of_int j,
                    if trans then W8.of_int j else W8.of_int i);
      c ← Parse(XOF, H).sample();
      a ← (a[i, j ← c])%Kyber;
      j ← j + 1;
    }
    i ← i + 1;
  }
  return unlift_matrix a;
}
proc sample_noise2(noiseseed:W8.t Array32.t) : W16.t Array768.t ∗ W16.t Array768.t ∗ int= {
  _N ← 0;
  i ← 0;
  while (i < kvec) {
    c ← CBD2(KPRF).sample(noiseseed,_N);
    noise1 ← set noise1 i c;
    _N ← _N + 1; i ← i + 1;
  }
  i ← 0;
  while (i < kvec) {
    c ← CBD2(KPRF).sample(noiseseed,_N);
    noise2 ← set noise2 i c;
    _N ← _N + 1; i ← i + 1;
  }
  return (unlift_vector noise1,unlift_vector noise2,_N);
}
```

Figure 7: Mock matrix and noise generation functions.

```
lemma kyber_kem_correct_enc mem _ctp _pkp _kp :
   equiv [ Jkem.M(Jkem.Syscall).__crypto_kem_enc_jazz ~
                                KyberKEM(KHS,XOF,KPRF,KHS_KEM,KemH,H).enc_derand:
     valid_ptr _pkp (384∗3 + 32) ∧ valid_disj_reg _ctp (3∗320+128) _kp (32) ∧
     Glob.mem{1} = mem ∧ to_uint ctp{1} = _ctp ∧
     to_uint pkp{1} = _pkp ∧  to_uint shkp{1} = _kp ∧
     pk{2}₁ = load_array1152 mem _pkp ∧
     pk{2}₂ = load_array32 mem (_pkp + 3∗384)
     randomnessp{1} = prem{2} ∧
       ⇛
        touches2 Glob.mem{1} mem _ctp (3∗320+128) _kp (32) ∧
        let (c,k) = res{2} in
            c₁ = load_array960 Glob.mem{1} _ctp ∧
            c₂ = load_array128 Glob.mem{1} (_ctp + 960) ∧
            k = load_array32 Glob.mem{1} _kp
].
```

Figure 8: Example of a top-level correctness result.

should match the one stored in the starting memory for the implementation. Furthermore, the random coins provided to both algorithms are also assumed to be the same. The post condition specifies which part of the memory is modified, and states that reading from that memory region correctly recovers the ciphertext and key computed by the specification.

5.2 Verifying the AVX2 Implementation

In light of the work described in the previous section for the reference implementation, the process for the AVX2 implementation can also be described as having two parts: 1) showing that AVX2 operations over ring elements and vectors and equivalent to their reference counterparts and 2) proving that this implies that the AVX2 implementations of the IND-CPA PKE and KEM, which call these functions, are equivalent to their reference implementation counterparts. The proof of correctness then follows by transitivity, since we have already proved that the reference implementation is correct with respect to the Kyber specification. We now describe these steps in more detail.

Top-level Correctness. We prove functional correctness statements for the KEM and

IND-CPA encryption scheme constructions that are identical to those given above for the reference implementation. This is natural, since the AVX2 implementation is meant to be a drop-in replacement, whenever the target platform supports the necessary extensions. The proof is by transitivity: we have that the reference implementation is correct with respect to the KYBER specification, so it suffices to prove that the AVX2 implementation is equivalent to the reference implementation. We propagate this transitivity argument using relational Hoare logic down to lower level operations, as far as the structure of the code allows. For example, it is straightforward to prove that the IND-CPA encryption operations are equivalent, assuming that the intermediate level operations over vectors and ring elements on each side are, in turn, equivalent. We end up with a series of proof goals that look like the statement below.

```
equiv reduceequiv :
  Jkem_avx.M.__poly_reduce ~ Jkem.M.__poly_reduce :
    lift_array256 arg{1} = nttunpack (lift_array256 arg{2})  ⇒
      lift_array256 res{1} = nttunpack (lift_array256 res{2})  ∧
      pos_bound256_cxq res{1} 0 256 2  ∧ pos_bound256_cxq res{2} 0 256 2.
```

Here, we have on the left an intermediate-level AVX2 function that performs Barrett reduction of polynomial coefficients and, on the right, the corresponding reference implementation function. Note that we express the input/output relation modulo a permutation ntt_unpack, since this particular function is executed over ring elements in the NTT domain, which the AVX2 code stores differently (see Section 4).

Lower-level equivalences. As one proceeds down the call tree, the code of the AVX implementation becomes significantly different from the reference implementation: there is no-longer a one-to-one correspondence between inner functions and the control-flow is completely different. This is because the optimization based on vector operations relies on a totally different instruction schedule. The most extreme cases are, perhaps, the NTT and inverse NTT operations, where the order in which the computations are performed is indeed totally different: after a first round where the full array is transformed, the algorithm processes the bottom half of the array before moving to the top one.

To prove these lower-level equivalences, we therefore proceed in two steps. We first prove that the AVX2 function satisfies the same contract that we proved for the corresponding reference implementation function. For simple functions, such as addition, subtraction, etc., this task takes similar (small) effort when comparing to the reference implementation counterparts. The only difference lies in the data types and operations used, but the proof steps are essentially the same.

For more complex functions the proof effort was sometimes significantly greater, as SIMD operations often require sequences of intricate permutations before a semantically meaningful arithmetic or bitwise operation can be executed. Again, the most challenging cases were the NTT-related operations, where the proof effort was divided into three additional steps: i. defining an intermediate EASYCRYPT implementation that performs field operation in the same order as the AVX implementation; ii. proving that this code is equivalent to the AVX implementation by excluding the possibility of representation errors; and iii. proving that the resulting sequence of field operations is indeed computing the NTT as prescribed by the specification. Our NTT library was extended with additional variants of the NTT computation, all proved equivalent to the specification, to facilitate the connection to the AVX implementation, as we describe next.

5.3 Verifying the NTT

As described above, to manage complexity, the proof of NTT correctness has two main parts: 1) dealing with the representation of finite field elements and 2) proving that the

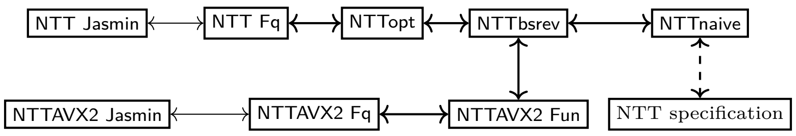

Figure 9: Structure of the NTT correctness proof.

computations performed over field elements is correct with respect to the specification. Step 1) is carried out as a simple game hop where we replace all the Jasmin register and stack variables that represent group elements, as well as computations over these variables, with computations over $\mathbb{F}_q$. Proving the correctness of this step is done by relying on a series of low level results for computations over $\mathbb{F}_q$.

In step 2), we take care of the most interesting part of the correctness proof: dealing with various optimizations such as the use of an array to store the values of the 256^{th} roots of the unity, the efficient computation of the index used to access this array, and the computation in reverse bit order. The proof is again done by game-hopping and its structure is summed up in Figure 9. The thin double arrow represents step 1) discussed above. The thick double arrows correspond to pRHL equivalence proofs. The dashed double arrow corresponds to the final HL correctness proof, where the EasyCrypt program NTTnaive is proven to implement the EasyCrypt functional specification of the NTT, described in Section 2. This proof is then lifted to pHL and carried by transitivity to all the other versions of the NTT via the other equivalence proofs. The intermediate NTTs are characterized by the following:

- NTT Fq replaces the calls to auxiliary functions in NTT Jasmin by their mathematical specification
- NTTAVX2 Fq replaces the calls to auxiliary functions in NTTAVX2 Jasmin by their mathematical specification
- NTTopt splits the NTT into several small functions to allow for modular reasoning
- NTTbsrev replaces the incremental computation of the index used to access the array of roots of unity by the computation of it's value from the loop indexes, and the index variable of the innermost loop by it's final value when it's used after the innermost loop
- NTTnaive has it's two innermost loops in reverse bit order when compared to NTTbsrev, and the accesses to the array of roots of the unity are replaced by the computation of the value of the desired root
- NTTAVX2 Fun replaces the two innermost loops by the evaluation of a function, and is designed to be easy to be shown equivalent to both NTTbsrev and NTTAVX2

The most challenging proof was the equivalence between NTTbsrev and NTTnaive, as it is in this transformation that the loops on both sides do not have the same order. As pRHL was made to reason about the lock-step execution of two programs, the invariants used were more complex than in other proofs. The AVX2 NTT implementation, NTTAVX2 Fq, is shown in pRHL to be equivalent to NTTAVX2 Fun, which is itself shown to be equivalent to NTTbsrev. The structure of the correctness proof of the inverse NTT, and the challenges it poses are essentially the same.

6 Benchmarks

In this Section we present benchmark results of our Jasmin implementations and compare to the original implementations written in C and assembly. We benchmark on three different generations of Intel CPUs that all implement the AMD64 architecture and support the AVX2 instruction set. Specifically, we benchmark on an Intel Core i7-6500U (Skylake), an Intel Core i7-4770K (Haswell) and on an Intel Core i7-10700K (Comet Lake) CPU. All benchmarks are using only a single core of these CPUs; we followed the standard practice of disabling Hyperthreading and TurboBoost; the C compiler used for all benchmarks was Clang, version 14.[8]

Table 1 reports the results of these benchmarks. All cycle counts in this table are the median of 10 000 runs with different inputs. All implementations used the same implementation of `randombytes`, which is implemented through the Linux `getrandom` system call. We see that the Jasmin reference implementation is considerably slower than the C reference implementation. This is expected, because the Jasmin compiler, unlike C compilers, does not optimize the code: this is by design the job of the programmer. The goal of this implementation was not to achieve high performance, but rather to facilitate the EasyCrypt proof of correctness.

More relevant is the performance comparison of the optimized AVX2 implementations. We first comment that, for the same C/asm implementation and very similar CPUs, SUPERCOP [BL] reports somewhat lower cycle counts. The main reason for this difference is that our benchmarks obtain random bytes through a system call, while SUPERCOP uses a fast userspace randomness generator. We see that the fully optimized Jasmin code is within about 10% of the performance of the optimized C and assembly implementation by the Kyber team. We also see the rather dramatic performance impact of our two implementations of rejection sampling, i.e., the only difference between the fully verified and the fully optimized AVX2 implementations. Clearly this difference motivates further effort on verifying a faster rejection sampling routine, but, as we will discuss in Section 7, such a proof might want to wait until EasyCrypt supports a higher degree of proof automation.

7 Discussion

The development we present in this paper took almost three years to complete due to the sheer scale of the effort, the fact that different tools needed to be improved in order to tackle the example at hand, and to the large number of people involved, their availabily to work on the project and their varying sets of skills and expertise.

If an equivalent project were to be started now with the same team, e.g., to formally verify the implementation of another NIST candidate, the expected development time would be cut down significantly due to the accumulated experience and improved tooling, but it would still imply a significant resource investment. An obvious conclusion we draw from this fact is that we need to be able to increase automation in the available tools.

The approach in [HLS+22] is promising but, at the moment, there are significant differences in the level of assurance that our work provides in comparison, namely: (1) our coverage of the implementation (excluding SHA3) is complete, whereas in [HLS+22] only the NTT transform is covered; and (2) the trusted computing base (TCB) for the tooling in our development is arguably smaller, since EasyCrypt has been around for a long time and is reasonably self-contained, and the Jasmin compiler is verified in Coq. Nevertheless, like in [HLS+22], our TCB includes hand-written semantics for assembly operations and the the part of the Jasmin compiler that extracts Jasmin code to EasyCrypt. In any

[8] clang-14 -march=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE

Table 1: Cycle counts for C, C/assembly, and our Jasmin implementations of Kyber-768 on different generations on Intel CPUs

Implementation	**operation**	**Skylake**	**Haswell**	**Comet Lake**
C ref	keygen	200302	187172	184374
	encaps	251384	242424	235714
	decaps	287724	278160	272296
Jasmin ref	keygen	411676	394636	384948
	encaps	488904	471680	458640
	decaps	562426	534420	527266
C/asm AVX2	keygen	49572	47280	41682
	encaps	60018	62900	55956
	decaps	45854	47784	43906
Jasmin AVX2 (fully verified)	keygen	106578	96296	93244
	encaps	119308	111536	107474
	decaps	105336	98328	96564
Jasmin AVX2 (fully optimized)	keygen	50004	48800	45046
	encaps	65132	63988	59496
	decaps	50340	51444	48172

case, future research will certainly include an effort to integrate the ideas of [HLS+22] (or even the tooling) into the current toolchain.

Given the necessary resources, the natural question to ask is whether the benefits justify the means. We believe that, for security-critical code that will be used and replicated in a vast number of applications — which is the case for cryptographic standards — the following take-aways from our work indeed show that the effort is justified:

1. The proof clarifies how the implementation, which has a formal semantics, interprets the specification which is given in paper. Indeed, providing a formal semantics for the specification is an important side result of this line of work, and it allows a complementary effort of verifying the security and correctness of the specification itself.

2. The proof looks at properties that cannot be reasonably checked by testing, such as the fact that the sampling procedures are generating uniform matrices or correctly distributed noise.

3. The proof highlights sub-optimal parts of the code, e.g., the fact that bounds are explicitly stated and proved at the function level permit identifying the points where these bounds are too tight for what is needed, and hence can be relaxed by removing unnecessary computations. This was indeed the case in this development, where the final version of the AVX2 code is faster because we were able to eliminate redundant reduction steps that became evident when formally proving the ranges of intermediate computations.

As an example of the importance of point (1) above, which is perhaps the less obvious one, let us consider the declarations in Figure 10, which pinpoint how SHA3 algorithms are used in Kyber. We note that a total of 9 different types of calls to the SHA3 library are performed: these are declared as operators, which we axiomatize as the semantics of the corresponding functions in the SHA3 library for concrete input and output sizes.

We can see that we also annotate these declarations with the security semantics we will assume for these operators in the ongoing follow-up work where we analyze the security of the Kyber specification. These will be either standard PRG/PRF assumptions or, when nothing is stated in the code, they will be modeled as random oracles because the attacker will potentially know the full input. For that future analysis it is important to check that all of these calls to SHA3 algorithms can be considered to be independent hash functions. The argument to justify this claim will rely on the fact that each use of SHA3 algorithms is domain-separated from the other for one of two reasons: 1) using SHAKE vs SHA3 hashing, or 2) if using the same family of algorithms, by using different input and/or output sizes. We do not need to make this argument formal for the purpose of this paper, but it is important to emphasize that our specification is precise enough that we *can* formally state it.

Follow-up work includes tying the functional correctness results we established to formal proofs of correctness for SHA3 implementations, and extending the correctness results to cover other Kyber implementations, namely the faster AVX2 rejection sampling routine and implementations offering protection against timing attacks and Spectre v1 attacks [ASBG+23]. These results will also be connected to machine-checked security proofs that apply to the Kyber specification we present here.

Acknowledgments

This research was supported by Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972; by the European Commission through the ERC Starting Grant 805031 (EPOQUE); by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038; and by the *Agence Nationale de la Recherche* (French National Research Agency) as part of the France 2030 programme – ANR-22-PECY-0006. This work was supported by European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nž 047264; Funding Reference: POCI-01-0247-FEDER-047264].

```
(* PRG that cleans up randomness in PKE *)
op SHA3_512_32_64 : W8.t Array32.t → unit → W8.t Array64.t.
clone import HS_DEFS.PseudoRF as HSF with
  type K ← W8.t Array32.t,
  op dK ← srand,
  op F ← SHA3_512_32_64.

(* XOF used to rejection−sample matrix in PKE *)
op SHAKE128_ABSORB_34 : W8.t Array34.t →  W64.t Array25.t.
op SHAKE128_SQUEEZE_168 : W64.t Array25.t → W64.t Array25.t *  W8.t Array168.t.
module (XOF : XOF_t) (O : KyberPKE.RO.POracle) = {
  var state : W64.t Array25.t
  proc init(rho : W8.t Array32.t, i j : W8.t) : unit = {
       extseed ← Array34.init
        (fun k ⇒  if k < 32 then rho[k] else if k=32 then i else j);
       state ← SHAKE128_ABSORB_34 extseed;
  }
  proc next_bytes() : W8.t Array168.t = {
       (state,buf) ← SHAKE128_SQUEEZE_168 state;
       return buf;
  }
}.

(* PRF used to sample noise in PKE *)
op SHAKE256_33_128 : 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 ← SHAKE256_33_128.

(* PRG that cleans up randomness in KEM *)
op SHA3_256_32_32 : W8.t Array32.t → unit → W8.t Array32.t.
clone import HS_KEM_DEFS.PseudoRF as HSF_KEM with
  type K ← W8.t Array32.t,
  op dK ← srand,
  op F ← SHA3_256_32_32.

(* Various Fujisaki−Okamoto hashes, plus KDF used by KEM. *)
op SHA3_256_64_64   : W8.t Array64.t → W8.t Array64.t.
op SHA3_256_1088_32 : W8.t Array1088.t → W8.t Array32.t.
op SHA3_256_1184_32 : W8.t Array1184.t → W8.t Array32.t.
op SHAKE256_64_32 : W8.t Array64.t → W8.t Array32.t.
module (KemH : KEMHashes) (RO : RO.POracle) = {
  proc pkH(pk : W8.t Array1152.t * W8.t Array32.t) : W8.t Array32.t = {
        return SHA3_256_1184_32 (Array1184.init (fun k ⇒  if (k < 1152) then pk.`1[k] else pk.`2[k−1152]));
  }
  proc cH(c : W8.t Array960.t * W8.t Array128.t) : W8.t Array32.t = {
        return SHA3_256_1088_32 (Array1088.init (fun k ⇒  if (k < 960) then c.`1[k] else c.`2[k−960]));

  }
  proc g(m : W8.t Array32.t, pkh : W8.t Array32.t) : W8.t Array32.t * W8.t Array32.t  = {
      var ktr;
      ktr ← SHA3_256_64_64 (Array64.init (fun k ⇒  if (k < 32) then m[k] else pkh[k−32]));
      return (Array32.init (fun i⇒  ktr[i]), Array32.init (fun i ⇒  ktr[i + 32]));
  }
  proc kdf(kt : W8.t Array32.t, ch : W8.t Array32.t) : W8.t Array32.t = {
        return SHAKE256_64_32 (Array64.init (fun k ⇒  if (k < 32) then kt[k] else ch[k−32]));
  }
}.
```

Figure 10: SHA3 usage in the KYBER development.

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. Status report on the third round of the NIST post-quantum cryptography standardization process. NISTIR 8413, 2022. https://csrc.nist.gov/publications/detail/nistir/8413/final.

[ABB+17] 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.

In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1807–1823. ACM Press, October / November 2017.

[ABB+19] José Bacelar Almeida, Cecile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1607–1622. ACM Press, November 2019.

[ABB+20] 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*, pages 965–982. IEEE Computer Society Press, May 2020.

[ABB+22] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS$^+$ – submission to the nist post-quantum project, v.3.1. Round-3 submission to the NIST PQC standardization project, 2022. https://sphincs.org/data/sphincs+-r3.1-specification.pdf.

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation (version 3.02). Round-3 submission to the NIST PQC standardization project, 2021. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[ASBG+23] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. Typing high-speed cryptography against Spectre v1. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023. (to appear) https://eprint.iacr.org/2022/1270.

[BBB+21] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy*, pages 777–795. IEEE Computer Society Press, May 2021.

[BBF+21] Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. EasyPQC: Verifying post-quantum cryptography. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2564–2586. ACM Press, November 2021.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*, pages 353–367. IEEE, 2018. https://eprint.iacr.org/2017/634.

[BDK$^+$21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm specifications and supporting documentation (version 3.1). Round-3 submission to the NIST PQC standardization project, 2021. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

[Beu17] Benjamin Beurdouche. Verified cryptography for Firefox 57. Post on the Mozilla Security Blog, 2017. https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/ (accessed 2023-01-08).

[BGHZ11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Heidelberg, August 2011.

[BHK$^+$19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ signature framework. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2129–2146. ACM Press, November 2019.

[BL] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to/ (accessed 2023-01-15).

[DKL$^+$18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR TCHES*, 2018(1):238–268, 2018. https://tches.iacr.org/index.php/TCHES/article/view/839.

[EPG$^+$19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE Computer Society Press, May 2019.

[FHK$^+$20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over NTRU (specification v1.2). Round-3 submission to the NIST PQC standardization project, 2020. https://falcon-sign.info/falcon.pdf.

[GS16] Shay Gueron and Fabian Schlieker. Speeding up R-LWE post-quantum key exchange. In Billy Bob Brumley and Juha Röning, editors, *Secure IT Systems*, volume 10014 of *LNCS*, pages 187–198. Springer, 2016. https://eprint.iacr.org/2016/467.

[HLS$^+$22] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT multiplications for NIST-PQC KEM lattice finalists: Kyber, SABER, and NTRU. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):718–750, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9838.

[HMS22] Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. Formal verification of Saber's public-key encryption scheme in EasyCrypt. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, volume 13507 of *LNCS*, pages 622–653. Springer, 2022. https://eprint.iacr.org/2022/351.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19. IEEE Computer Society Press, May 2019.

[Kre23] Katharina Kreuzer. Verification of the $(1 - \delta)$-correctness proof of CRYSTALS-KYBER with number theoretic transform. IACR ePrint Archive report 2023/27, 2023. https://eprint.iacr.org/2023/27.

[KV19] Kris Kwiatkowski and Luke Valenta. The TLS post-quantum experiment. Post on the Cloudflare blog, 2019. https://blog.cloudflare.com/the-tls-post-quantum-experiment/.

[Lan16] Adam Langley. CECPQ1 results. Blog post, 2016. https://www.imperialviolet.org/2016/11/28/cecpq1.html.

[Lan18] Adam Langley. CECPQ2. Blog post, 2018. https://www.imperialviolet.org/2018/12/12/cecpq2.html.

[Ope22] OpenSSH. OpenSSH 9.0 release notes, 2022. https://www.openssh.com/txt/release-9.0.

[Unr20] Dominique Unruh. Post-quantum verification of Fujisaki-Okamoto. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 321–352. Springer, Heidelberg, December 2020.

[ZBPB17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1789–1806. ACM Press, October / November 2017.