Verification of Correctness and Security Properties for CRYSTALS-KYBER

Katharina Kreuzer

April 18, 2024

Abstract

This article builds upon the formalization of the deterministic part of the original Kyber algorithms [6]. The correctness proof is expanded to cover both the deterministic part (from [6]) and the probabilistic part of randomly chosen inputs. Indeed, the probabilistic version of the δ-correctness [5] was flawed and could only be formalized for a modified δ'.

The authors [5] also remarked, that the security proof against indistinguishability under chosen plaintext attack (IND-CPA) does not hold for the original version of Kyber. Thus, the newer version [4, 2] was formalized as well, including the adapted deterministic and probabilistic correctness theorems. Moreover, the IND-CPA security proof against the new version of Kyber has been verified using the CryptHOL library [10, 9]. Since the new version also included a change of parameters, the Kyber algorithms have been instantiated with the new parameter set as well.

Together with the entry "CRYSTALS-Kyber"[6], this entry formalises the paper [7].

Contents

1 Introduction

CRYSTALS-KYBER is a cryptographic key encapsulation mechanism (KEM) and the winner of the NIST standardization project for post-quantum cryptography [1]. That is, even with feasible quantum computers, Kyber is thought to be hard to crack.

The original version of the Kyber algorithms was introduced in [5, 3] and formalized in [6]. During the rounds of the NIST specification process, several changes to the KEM and the underlying public key encryption scheme (PKE) were made [4, 2]. The most noteworthy change is the omission of the compression of the public key. The reason is that the compression of the public key induced an error in the security proof against the indistinguishability against chosen plaintext attack (IND-CPA). When omitting the compression, the advantage against IND-CPA can be reduced to the advantage against the module Learning-with-Errors (module LWE). The module-LWE has been shown to be a NP-hard problem using probabilistic reductions [8].

In this article, we extend the prior formalization of Kyber [6] by formalizing and verifying the following points:

- Kyber algorithms without compression of the public key
- Exemplary parameter set for Round 2 and 3 (using modulus $q = 3329$)
- Deterministic correctness for Kyber without compression of the public key
- Probabilistic correctness for both versions of Kyber but only for modified error bound (original bound could not be formalized due to the compression error in the reduction to module-LWE)
- IND-CPA security proof for Kyber without compression of the public key

The last point, the security proof against IND-CPA, is a major contribution of this work. Using the game-based proof techniques for security analysis under the standard random oracle model as defined in CryptHOL [9, 10], the advantage against Kyber's IND-CPA game was bounded by the advantage against the module-LWE.

All in all, this entry formalizes claims for correctness and IND-CPA security of Kyber and uncovers flaws in the relevant proofs. More details can be found in the corresponding paper [7]. Since Kyber was chosen by NIST for standardisation, a formal proof of correctness and security properties is essential.

theory *Crypto_Scheme_new*

imports *"CRYSTALS-Kyber.Crypto_Scheme"*

begin

2 Deterministic Part of Correctness Proof for Kyber without Compression of the Public Key

context *kyber_spec*
begin

In the following the key generation and encryption algorithms of Kyber without compression of the public key are stated. Here, the variables have the meaning:

- A: matrix, part of Alices public key
- s: vector, Alices secret key
- t: is the key generated by Alice qrom A and s in `key_gen`
- r: Bobs "secret" key, randomly picked vector
- m: message bits, $m \in \{0,1\}^{256}$
- (u,v): encrypted message
- du, dv: the compression parameters for u and v respectively. Notice that `0 < d < ⌈log_2 q⌉`. The d values are public knowledge.
- e, $e1$ and $e2$: error parameters to obscure the message. We need to make certain that an eavesdropper cannot distinguish the encrypted message qrom uniformly random input. Notice that e and $e1$ are vectors while $e2$ is a mere element in `ℤ_q[X]/(X^n+1).`

The decryption algorithm is the same as in the original Kyber algorithms, thus we do not need to redefine it.

```
definition key_gen_new ::
  "(('a qr, 'k) vec, 'k) vec ⇒ ('a qr, 'k) vec ⇒
   ('a qr, 'k) vec ⇒ ('a qr, 'k) vec" where
"key_gen_new A s e = A *v s + e"

definition encrypt_new ::
  "('a qr, 'k) vec ⇒ (('a qr, 'k) vec, 'k) vec ⇒
   ('a qr, 'k) vec ⇒ ('a qr, 'k) vec ⇒ ('a qr) ⇒
   nat ⇒ nat ⇒ 'a qr ⇒
   (('a qr, 'k) vec) * ('a qr)" where
"encrypt_new t A r e1 e2 du dv m =
```

```
(compress_vec du ((transpose A) *v r + e1),
 compress_poly dv (scalar_product t r +
  e2 + to_module (round((real_of_int q)/2)) * m)) "
```

We now want to show the deterministic correctness of the algorithm. That means, for fixed input variables, after generating the public key, encrypting and decrypting, we get back the original message.

```
lemma kyber_new_correct:
  fixes A s r e e1 e2 du dv cu cv t u v
  assumes
      t_def:   "t = key_gen_new A s e"
  and u_v_def: "(u,v) = encrypt_new t A r e1 e2 du dv m"
  and cu_def:  "cu = compress_error_vec du
                ((transpose A) *v r + e1)"
  and cv_def:  "cv = compress_error_poly dv
                (scalar_product t r + e2 +
                 to_module (round((real_of_int q)/2)) * m)"
  and delta:   "abs_infty_poly (scalar_product e r + e2 + cv -
                scalar_product s e1 -
                scalar_product s cu) < round (real_of_int q / 4)"
  and m01:     "set ((coeffs ∘ of_qr) m) ⊆ {0,1}"
  shows "decrypt u v s du dv = m"
⟨proof⟩

end

end
theory Delta_Correct

imports "HOL-Probability.Probability"

begin
```

3 δ-Correctness of PKEs

The following locale defines the δ-correctness of a public key encryption (PKE) scheme given by the algorithms `key_gen` `encrypt` and `decrypt`. `Msgs` is the set of all possible messages that can be encoded with the PKE. Since some PKE have a small failure probability, the definition of correctness has to be adapted to cover the case of failures as well. The standard definition of such δ-correctness is given in the function `expect_correct`.

```
locale pke_delta_correct =
fixes key_gen :: "('pk × 'sk) pmf"
  and encrypt :: "'pk ⇒ 'plain ⇒ 'cipher pmf"
  and decrypt :: "'sk ⇒ 'cipher ⇒ 'plain"
  and Msgs :: "'plain set"
begin
```

```
type_synonym ('pk', 'sk') cor_adversary = "('pk' ⇒ 'sk' ⇒ bool pmf)"

definition expect_correct where
"expect_correct = measure_pmf.expectation key_gen
  (λ(pk,sk). MAX m∈Msgs. pmf (bind_pmf (encrypt pk m)
  (λc. return_pmf (decrypt sk c ≠ m))) True)"

definition delta_correct where
"delta_correct delta = (expect_correct ≤ delta)"
```

game_correct is the game played to guarantee correctness. If an adversary *Adv* has a non-negligible advantage in the correctness game, he might have enough information to break the PKE. However, the definition of this correctness game is somewhat questionable, since the adversary *Adv* is given the secret key as well, thus enabling him to break the encryption and the PKE.

```
definition game_correct where
"game_correct Adv = do{
      (pk,sk) ← key_gen;
      m ← Adv pk sk;
      c ← encrypt pk m;
      let m' = decrypt sk c;
      return_pmf (m' ≠ m)
    }"


end
```

An auxiliary lemma to handle the maximum over a sum.

```
lemma max_sum:
fixes A B and f :: "'a ⇒ 'b ⇒ 'c :: {ordered_comm_monoid_add, linorder}"
assumes "finite A " "A ≠ {}"
shows "(MAX x∈A. (∑y∈B. f x y)) ≤ (∑y∈B. (MAX x∈A. f x y))"
⟨proof⟩

end
theory Finite_UNIV
imports
  "HOL-Analysis.Finite_Cartesian_Product"
  "CRYSTALS-Kyber.Kyber_spec"

begin
```

4 R_q is Finite

The module R_q is finite. This can be reasoned in two steps: One, the set of possible coefficients of a polynomial in R_q is finite since coefficients are in $\mathbb{F}_q$. Two, the polynomials in R_q have degree less than n. Together, this implies that R_q itself is a finite set.

lemma *card_UNIV_qr:*
```
 "card (UNIV :: 'a::qr_spec qr set) = (CARD('a)) ^ (degree (qr_poly' TYPE('a)))"
```
⟨*proof*⟩

lemma *finite_qr [simp]:*
```
  "finite (UNIV::'a::qr_spec qr set)"
```
⟨*proof*⟩

instantiation *qr ::(qr_spec) finite*
begin
instance
⟨*proof*⟩
end

Moreover, there are only finitely many vectors (of fixed length) over a finite types and only finitely many matrices (of fixed dimension) over a finite type. This yields that R_q^k and $R_q^{k\times k}$ are both finite.

lemma *finite_vec:*
assumes *"finite (UNIV :: 'a set)"*
shows *"finite (UNIV :: ('a, 'k::finite) vec set)"*
⟨*proof*⟩

lemma *finite_mat:*
assumes *"finite (UNIV :: 'a set)"*
shows *"finite (UNIV :: (('a, 'k::finite) vec,'k) vec set)"*
⟨*proof*⟩

lemma *finite_UNIV_vec [simp]:*
```
  "finite (UNIV:: ('a::qr_spec qr, 'k::finite) vec set)"
```
⟨*proof*⟩

lemma *finite_UNIV_mat [simp]:*
```
  "finite (UNIV:: (('a::qr_spec qr, 'k) vec, 'k::finite) vec set)"
```
⟨*proof*⟩

lemma *finite_UNIV_vec_option [simp]:*
```
  "finite (UNIV :: ('a::qr_spec qr,'k::finite option) vec set)"
```
⟨*proof*⟩

lemma *finite_UNIV_mat_option [simp]:*
```
  "finite (UNIV:: (('a::qr_spec qr, 'k::finite) vec, 'k option) vec set)"
```
⟨*proof*⟩

end
theory *Lemmas_for_spmf*

imports *CryptHOL.CryptHOL*
Finite_UNIV

begin

5 Auxiliary Lemmas on *spmf*

Replicate function for spmf.

definition *replicate_spmf :: "nat ⇒ 'b pmf ⇒ 'b list spmf"* **where**
"replicate_spmf m p = spmf_of_pmf (replicate_pmf m p)"

lemma *replicate_spmf_Suc_cons:*
```
"replicate_spmf (m + 1) p =
  do {
    xs ← replicate_spmf m p;
    x ← spmf_of_pmf p;
    return_spmf (x # xs)
  }"
```
⟨*proof*⟩

lemma *replicate_spmf_Suc_app:*
```
"replicate_spmf (m + 1) p =
  do {
    xs ← replicate_spmf m p;
    x ← spmf_of_pmf p;
    return_spmf (xs @ [x])
  }"
```
⟨*proof*⟩

Lemmas on *coin_spmf*

lemma *spmf_coin_spmf: "spmf coin_spmf i = 1/2"*
⟨*proof*⟩

lemma *bind_spmf_coin:*
assumes *"lossless_spmf p"*
shows *"bind_spmf p (λ_. coin_spmf) = coin_spmf"*
⟨*proof*⟩

lemma *if_splits_coin:*
"(if P then coin_spmf else coin_spmf) = coin_spmf"
⟨*proof*⟩

Lemmas for rewriting of discrete probabilities.

lemma *ex1_sum:*
assumes *"∃! (x ::'a). P x" "finite (UNIV :: 'a set)"*
shows *"sum (λx. of_bool (P x)) UNIV = 1"*
⟨*proof*⟩

lemma (**in** *kyber_spec*) *surj_add_qr:*
"surj (λx. x + (y:: 'a qr))"
⟨*proof*⟩

lemma (**in** *kyber_spec*) *bij_add_qr:*
"bij (λx. x + (y::'a qr))"
⟨*proof*⟩

Lemmas for addition and difference of uniform distributions

lemma (**in** *kyber_spec*) *spmf_of_set_add:*
"let A = (UNIV :: ('a qr, 'k) vec set) in
do {x ← spmf_of_set A; y ← spmf_of_set A; return_spmf (x+y)} = spmf_of_set A"
⟨*proof*⟩

lemma (**in** *kyber_spec*) *spmf_of_set_diff:*
"let A = (UNIV :: ('a qr, 'k) vec set) in
do {x ← spmf_of_set A; y ← spmf_of_set A; return_spmf (x-y)} = spmf_of_set A"
⟨*proof*⟩

end
theory *MLWE*

imports *Lemmas_for_spmf*
"Game_Based_Crypto.CryptHOL_Tutorial"

begin

6 Module Learning-with-Errors Problem (module-LWE)

Berlekamp_Zassenhaus loads the vector type *'a vec* from *Jordan_Normal_Form.Matrix*. This doubles the symbols *\$* and χ for *vec_nth* and *vec_lambda*. Thus we delete the *vec_index* for type *'a vec*. Still some type ambiguities remain.

Here the actual theory starts.

We introduce a locale `module_lwe` that represents the module-Learning-with-Errors (module-LWE) problem in the setting of Kyber. The locale takes as input:

- *type_a* the type of the quotient ring of Kyber. (This is a side effect of the Harrison trick in the Kyber locale.)

- `type_k` the finite type for indexing vectors in Kyber. The cardinality is exactely k. (This is a side effect of the Harrison trick in the Kyber locale.)
- `idx` an indexing function from `'k` to `{0..<k}`
- `eta` the specification value for the centered binomial distribution β_η

```
locale module_lwe =
fixes type_a :: "('a :: qr_spec) itself"
  and type_k :: "('k ::finite) itself"
  and k :: nat
  and idx :: "'k::finite ⇒ nat"
  and eta :: nat
assumes "k = CARD('k)"
  and bij_idx: "bij_betw idx (UNIV::'k set) {0..<k}"

begin
```

The adversary in the module-LWE takes a matrix `A::(('b, 'n) vec, 'm) vec` and a vector `t::('b, 'm) vec` and returns a probability distribution on `bool` guessing whether the given input was randomly generated or a valid module-LWE instance.

```
type_synonym ('b, 'n, 'm) adversary =
  "(('b, 'n) vec, 'm) vec ⇒ ('b, 'm) vec ⇒  bool spmf"
```

Next, we want to define the centered binomial distributions β_η. `bit_set` returns the set of all bit lists of length `eta`. `beta` is the centered binomial distribution β_η as a `pmf` on the quotient ring R_q. `beta_vec` is then centered binomial distribution β_η^k on vectors in R_q^k.

```
definition bit_set :: "int list set" where
"bit_set = {xs:: int list. set xs ⊆ {0,1} ∧ length xs = eta}"

lemma finite_bit_set:
"finite bit_set"
⟨proof⟩

lemma bit_set_nonempty:
"bit_set ≠ {}"
⟨proof⟩
definition beta :: "'a qr pmf" where
"beta = do {
    as ← pmf_of_set (bit_set);
    bs ← pmf_of_set (bit_set);
    return_pmf (to_module (∑i<eta. as ! i - bs! i))
  } "

definition beta_vec :: "('a qr , 'k) vec pmf" where
```

```
"beta_vec = do {
    (xs :: 'a qr list) ← replicate_pmf (k) (beta);
    return_pmf (χ i.  xs ! (idx i))
  }"
```

Since we work over `spmf`, we need to show that `beta_vec` is lossless.

lemma `lossless_beta_vec[simp]:`
```
  "lossless_spmf (spmf_of_pmf beta_vec)"
```
⟨*proof*⟩

We define the game versions of module-LWE. Given an adversary $\mathcal{A}$, we have two games: in `game`, the instance given to the adversary is a module-LWE instance, whereas in `game_random`, the instance is chosen randomly.

definition `game :: "('a qr,'k,'k) adversary ⇒ bool spmf"` **where**
```
  "game 𝒜  = do {
    A ← spmf_of_set (UNIV:: (('a qr, 'k) vec, 'k) vec set);
    s ← beta_vec;
    e ← beta_vec;
    b' ← 𝒜 A (A *v s + e);
    return_spmf (b')
  }"
```

definition `game_random :: "('a qr,'k,'k) adversary ⇒ bool spmf"` **where**
```
  "game_random 𝒜  = do {
    A ← spmf_of_set (UNIV:: (('a qr, 'k) vec, 'k) vec set);
    b ← spmf_of_set (UNIV:: ('a qr, 'k) vec set);
    b' ← 𝒜 A b;
    return_spmf (b')
  }"
```

The advantage of an adversary $\mathcal{A}$ returns a value how good the adversary is at guessing whether the instance is generated by the module-LWE or uniformly at random.

definition `advantage :: "('a qr,'k,'k) adversary ⇒ real"` **where**
```
"advantage 𝒜 = |spmf (game 𝒜) True - spmf (game_random 𝒜) True |"
```

Since the reduction proof of Kyber uses the module-LWE problem for two different dimensions (ie. $A \in R_q^{(k+1)\times k}$ and $A \in R_q^{k\times k}$), we need a second definition of the index function, the centered binomial distribution, the game and random game, and the advantage. Here the problem is that the dimension k of the vectors is hard-coded in the type `'k`. That makes it hard to “simply add” another dimension. A trick how this can be formalised nevertheless is to use the option type on `'k` to encode a type with $k+1$ elements. With the option type, we can embed a vector of dimension k indexed by the type `'k` into a vector of dimension $k+1$ by adding a value for the index `None` (an element `a :: 'k` is mapped to `Some a`). Note also that the additional index appears only in one dimension of A, resulting in a non-quadratic matrix.

Index function of the option type *'k option*.

```
fun idx' :: "'k option ⇒ nat" where
  "idx' None = 0" |
  "idx' (Some x) = idx x + 1"

lemma idx': "((x # xs) ! idx' i) =
  ( if i = None then x else xs ! idx (the i))"
  if "length xs = k" for xs and i::"'k option"
⟨proof⟩


lemma idx'_lambda:
  "(χ i. (x # xs) ! idx' i) =
   (χ i. if i = None then x else xs ! idx (the i))"
   if "length xs = k" for xs ⟨proof⟩
```

Definition of the centered binomial distribution β_η^{k+1} and lossless property.

```
definition beta_vec' :: "('a qr , 'k option) vec spmf" where
"beta_vec' = do {
    (xs :: 'a qr list) ← replicate_spmf (k+1) (beta);
    return_spmf (χ i.  xs ! (idx' i))
  }"

lemma lossless_beta_vec'[simp]:
  "lossless_spmf beta_vec'"
⟨proof⟩
```

Some lemmas on replicate.

```
lemma replicate_pmf_same_length:
assumes "⋀ xs. length xs = m ⟹ f xs = g xs"
shows "bind_pmf (replicate_pmf m p) f = bind_pmf (replicate_pmf m p) g"
⟨proof⟩

lemma replicate_spmf_same_length:
assumes "⋀ xs. length xs = m ⟹ f xs = g xs"
shows "(replicate_spmf m p ⤜ f) = (replicate_spmf m p ⤜ g)"
⟨proof⟩
```

Lemma to split the `replicate (k+1)` function in `beta_vec'` into two parts: `replicate k` and a separate value. Note, that the `xs` in the `do` notation below are always of length k.

```
no_adhoc_overloading Monad_Syntax.bind bind_pmf

lemma beta_vec':
  "beta_vec' = do {
    (xs :: 'a qr list) ← replicate_spmf (k) (beta);
```

```
    (x :: 'a qr) ← spmf_of_pmf beta;
    return_spmf (χ i.  if i = None then x else xs ! (idx (the i)))
  }"
```
⟨proof⟩

adhoc_overloading *Monad_Syntax.bind bind_pmf*

Definition of the two games for the option type.

definition *game' :: "('a qr,'k,'k option) adversary ⇒ bool spmf"* **where**
```
  "game' 𝒜  = do {
    A ← spmf_of_set (UNIV:: (('a qr, 'k) vec, 'k option) vec set);
    s ← beta_vec;
    e ← beta_vec';
    b' ← 𝒜 A (A *v s + e);
    return_spmf (b')
  }"
```

definition *game_random' :: "('a qr,'k,'k option) adversary ⇒ bool spmf"* **where**
```
  "game_random' 𝒜  = do {
    A ← spmf_of_set (UNIV:: (('a qr, 'k) vec, 'k option) vec set);
    b ← spmf_of_set (UNIV:: ('a qr, 'k option) vec set);
    b' ← 𝒜 A b;
    return_spmf (b')
  }"
```

Definition of the advantage for the option type.

definition *advantage' :: "('a qr,'k,'k option) adversary ⇒ real"* **where**

"advantage' 𝒜 = |spmf (game' 𝒜) True - spmf (game_random' 𝒜) True |"

Game and random game for finite type with one element only

definition *beta1 :: "('a qr , 1) vec pmf"* **where**
"beta1 = bind_pmf beta (λx. return_pmf (χ i. x))"

definition *game1 :: "('a qr, 1, 1) adversary ⇒ bool spmf"* **where**
```
  "game1 𝒜  = do {
    A ← spmf_of_set (UNIV:: (('a qr, 1) vec, 1) vec set);
    s ← spmf_of_pmf beta1;
    e ← spmf_of_pmf beta1;
    b' ← 𝒜 A (A *v s + e);
    return_spmf (b')
  }"
```

definition *game_random1 :: "('a qr,1,1) adversary ⇒ bool spmf"* **where**
```
  "game_random1 𝒜  = do {
    A ← spmf_of_set (UNIV:: (('a qr, 1) vec, 1) vec set);
    b ← spmf_of_set (UNIV:: ('a qr, 1) vec set);
    b' ← 𝒜 A b;
```

```
    return_spmf (b')
  }"
```

The advantage of an adversary $\mathcal{A}$ returns a value how good the adversary is at guessing whether the instance is generated by the module-LWE or uniformly at random.

```
definition advantage1 :: "('a qr,1,1) adversary ⇒ real" where
"advantage1 𝒜 = |spmf (game1 𝒜) True - spmf (game_random1 𝒜) True |"

end
end
theory Correct_new

imports Crypto_Scheme_new
    Delta_Correct
    MLWE

begin
```

7 δ-Correctness of Kyber without Compression of the Public Key

The functions *key_gen_new*, *encrypt_new* and *decrypt* are deterministic functions that calculate the output of the Kyber algorithms for a given input. To completely model the Kyber algorithms, we need to model the random choice of the input as well. This results in probabilistic programs that first choose the input according the the input distributions and then calculate the output. Probabilistic programs are modeled by the Giry monad of *pmf*'s. The correspond to the probability mass functions of the output.

7.1 Definition of Probabilistic Kyber without Key Compression and δ

The correctness of Kyber is formulated in a locale that defines the necessary assumptions on the parameter set. For the correctness analysis we need to import the definitions of the probability distribution β_η from the module-LWE and the Kyber locale itself. Moreover, we fix the compression depths for the outputs u and v. Note that in this case the output t of the key generation is uncompressed.

```
locale kyber_cor_new = mlwe: module_lwe "(TYPE('a ::qr_spec))" "TYPE('k::finite)"
k  +
kyber_spec _ _ _ _ "(TYPE('a ::qr_spec))" "TYPE('k::finite)"  +
fixes type_a :: "('a :: qr_spec) itself"
  and type_k :: "('k ::finite) itself"
```

and *du dv ::nat*
begin

We define types for the private and public keys, as well as plain and cipher texts. The public key consists of a matrix $A \in R_q^{k\times k}$ and a vector $t \in R_q^k$. The private key is the secret vector $s \in R_q$ such that there is an error vector $e \in R_q^k$ such that $A \cdot s + e = t$ (uncompressed). The plaintext consists of a bitstring (ie. a list of booleans). The ciphertext is an element of R_q^{k+1}represented by a vector u in R_q^k and a value $v \in R_q$ (both compressed).

```
type_synonym ('b,'l) pk = "((('b,'l) vec,'l) vec) × (('b,'l) vec)"
type_synonym ('b,'l) sk = "('b,'l) vec"
type_synonym plain = bitstring
type_synonym ('b,'l) cipher = "('b,'l) vec × 'b"
```

First, we need to show properties on the probability distributions needed. `beta` is the centered binomial distribution defined in `mlwe`.

```
lemma finite_bit_set:
"finite mlwe.bit_set"
⟨proof⟩
```

```
lemma finite_beta:
"finite (set_pmf mlwe.beta)"
⟨proof⟩
```

```
lemma finite_beta_vec:
"finite (set_pmf mlwe.beta_vec)"
⟨proof⟩
```

Next, we define the key generation, encryption and decryption as probabilistic programs which first generate random variables according to their distributions and then call the key generation, encryption or decryption functions accordingly. Since we look at Kyber without compression of the public key, the output of the key generation is uncompressed.

Note that in comparison to Kyber with public key compression, we do not need to output the error term e. Since t is uncompressed, we can easily recompute e using the secret key s.

```
definition pmf_key_gen  where
"pmf_key_gen = do {
  A ← pmf_of_set (UNIV:: (('a qr,'k) vec,'k) vec set);
  s ← mlwe.beta_vec;
  e ← mlwe.beta_vec;
  let t = key_gen_new A s e;
  return_pmf ((A, t), s)
}"
```

definition *pmf_encrypt* **where**
```
"pmf_encrypt pk m = do{
  r ← mlwe.beta_vec;
  e1 ← mlwe.beta_vec;
  e2 ← mlwe.beta;
  let c = encrypt_new (snd pk) (fst pk) r e1 e2 du dv m;
  return_pmf c
}"
```

Msgs is the space of all possible messages to be encrypted. It is non-empty and finite.

definition
```
"Msgs = {m::'a qr. set ((coeffs ∘ of_qr) m) ⊆ {0,1}}"
```

lemma *finite_Msgs:*
```
"finite Msgs"
```
⟨*proof*⟩

lemma *Msgs_nonempty:*
```
"Msgs ≠ {}"
```
⟨*proof*⟩

Since Kyber is a PKE, we can instantiate the PKE correctness locale with the Kyber algorithms without compression of the public key.

no_adhoc_overloading *Monad_Syntax.bind bind_pmf*

sublocale *pke_delta_correct pmf_key_gen pmf_encrypt*
  "(λ sk c. decrypt (fst c) (snd c) sk du dv)" Msgs ⟨*proof*⟩

adhoc_overloading *Monad_Syntax.bind bind_pmf*

In order to measure and estimate the errors introduced by the compression and decompression of the output of the encryption, we introduce *error_dist_vec* on vectors and *error_dist_poly* on polynomials.

definition
```
"error_dist_vec d = do{
  y ← pmf_of_set (UNIV :: ('a qr,'k) vec set);
  return_pmf (decompress_vec d (compress_vec d y)-y)
}"
```

definition
```
"error_dist_poly d = do{
  y ← pmf_of_set (UNIV :: 'a qr set);
  return_pmf (decompress_poly d (compress_poly d y)-y)
}"
```

The functions *w_distrib'*, *w_distrib* and *delta* define the originally claimed δ for the correctness of Kyber. However, the *delta*-correctness of Kyber could not be formalized.

The reason is that the values of *cu* and *cv* in *w_distrib'* rely on the compression error of uniformly random generated values. In truth, these values are not uniformly generated but instances of the module-LWE. *delta* also adds the additional error due to the module-learning with error instances. However, we cannot use the module-LWE assumption to reduce these values to uniformly generated ones since we would lose all information about the secret key otherwise. This is needed to perform the decryption in order to check whether the original message and the decryption of the ciphertext are indeed the same.

Therefore, we modified the given δ and defined a new value *delta'* in order to prove at least *delta'*-correctness.

definition *w_distrib'* **where**
```
"w_distrib' s e = do{
  r ← mlwe.beta_vec;
  e1 ← mlwe.beta_vec;
  e2 ← mlwe.beta;
  cu ← error_dist_vec du;
  cv ← error_dist_poly dv;
  let w = (scalar_product e r + e2 + cv - scalar_product s e1 - scalar_product
s cu);
  return_pmf (abs_infty_poly w ≥ round (q/4))}"
```

definition *w_distrib* **where**
```
"w_distrib = do{
  s ← mlwe.beta_vec;
  e ← mlwe.beta_vec;
  w_distrib' s e}"
```

definition *delta* **where**
```
"delta Adv0 Adv1 = pmf w_distrib True + mlwe.advantage Adv0 + mlwe.advantage1
Adv1"
```

This is the modified δ' which makes the correctness arguments to go through.

The functions *w_kyber*, *delta'* and *delta_kyber* define the modified δ for the correctness proof. Note the in *w_kyber*, the values *yu* and *yv* are generated according to their corresponding module-LWE instances and are not uniformly random. *delta'* is still dependent on the public and secret keys and the message. This dependency is eliminated in *delta_kyber* by taking the expectation over the key pair and the maximum over all messages, similar to the definition of δ-correctness.

definition *w_kyber* **where**
```
"w_kyber A s e m = do{
  r ← mlwe.beta_vec;
  e1 ← mlwe.beta_vec;
  e2 ← mlwe.beta;
  let t = A *v s + e;
```

```
  let yu = transpose A *v r + e1;
  let yv = (scalar_product t r + e2 +
           to_module (round (real_of_int q / 2)) * m);
  let cu = compress_error_vec du yu;
  let cv = compress_error_poly dv yv;
  let w = (scalar_product e r + e2 + cv - scalar_product s e1 - scalar_product
s cu);
  return_pmf (abs_infty_poly w ≥ round (q/4))}"
```

definition *delta'* **where**
```
"delta' sk pk m = pmf (w_kyber (fst pk) sk (snd pk - (fst pk) *v sk) m)
True"
```

definition *delta_kyber* **where**
```
"delta_kyber = measure_pmf.expectation pmf_key_gen
     (λ(pk, sk). MAX m∈Msgs. delta' sk pk m)"
```

7.2 δ-Correctness Proof

The idea to bound the probabilistic Kyber algorithms by *delta_kyber* is the following: First use the deterministic part given by *Crypto_Scheme_new.kyber_new_correct* to bound the correctness by *delta'* depending on a fixed key pair and message. Then bound the message by the maximum over all messages. Finally bound the key pair by using the expectation over the key pair. The result is that the correctness error of the Kyber PKE is bounded by *delta_kyber*.

First of all, we rewrite the deterministic part of the correctness proof *kyber_new_correct* from *Crypto_Scheme_new*.

```
lemma kyber_new_correct_alt:
  fixes A s r e e1 e2 cu cv t u v
  assumes t_def:   "t = key_gen_new A s e"
  and u_v_def: "(u,v) = encrypt_new t A r e1 e2 du dv m"
  and cu_def:  "cu = compress_error_vec du  ((transpose A) *v r + e1)"
  and cv_def:  "cv = compress_error_poly dv  (scalar_product t r + e2
+
                to_module (round((real_of_int q)/2)) * m)"
  and error: "decrypt u v s du dv ≠ m"
  and m01:     "set ((coeffs ∘ of_qr) m) ⊆ {0,1}"
  shows "abs_infty_poly (scalar_product e r + e2 + cv - scalar_product
s e1 -
           scalar_product s cu) ≥ round (real_of_int q / 4)"
⟨proof⟩
```

Then we show the correctness in the probabilistic program for a fixed key pair and message. The bound we use is *delta'*.

```
lemma correct_key_gen:
fixes A s e m
assumes pk_sk: "(pk, sk) = ((A, key_gen_new A s e), s)"
```

and *m_Msgs: "m∈Msgs"*
shows *"pmf (do{c ← pmf_encrypt pk m;*
return_pmf (decrypt (fst c) (snd c) sk du dv ≠ m)}) True ≤ delta' sk pk m"
⟨*proof*⟩

Now take the maximum over all messages. We rewrite this in order to be able to instantiate it nicely.

lemma *correct_key_gen_max:*
fixes *A s e m*
assumes *"(pk, sk) = ((A, key_gen_new A s e), s)"*
and *"m∈Msgs"*
shows *"pmf (do{c ← pmf_encrypt pk m;*
return_pmf (decrypt (fst c) (snd c) sk du dv ≠ m)}) True ≤ (MAX m'∈Msgs. delta' sk pk m')"
⟨*proof*⟩

lemma *correct_max:*
fixes *A s e*
assumes *"(pk, sk) = ((A, key_gen_new A s e), s)"*
shows *"(MAX m∈Msgs. pmf (do{c ← pmf_encrypt pk m;*
return_pmf (decrypt (fst c) (snd c) sk du dv ≠ m)}) True) ≤ (MAX m'∈Msgs. delta' sk pk m')"
⟨*proof*⟩

lemma *correct_max':*
fixes *pk sk*
shows *"(MAX m∈Msgs. pmf (do{c ← pmf_encrypt pk m;*
return_pmf (decrypt (fst c) (snd c) sk du dv ≠ m)}) True) ≤
(MAX m'∈Msgs. delta' sk pk m')"
⟨*proof*⟩

Finally show the overall bound `delta_kyber` for the correctness error of the Kyber PKE without compression of the public key.

lemma *expect_correct:*
"expect_correct ≤ delta_kyber"
⟨*proof*⟩

This yields the overall `delta_kyber`-correctness of Kyber without compression of the public key.

lemma *delta_correct_kyber:*
"delta_correct delta_kyber"
⟨*proof*⟩

end
end
theory *Kyber_gpv_IND_CPA*

```
imports "Game_Based_Crypto.CryptHOL_Tutorial"
        Correct_new

begin
```

8 IND-CPA Security of Kyber

The IND-CPA security of the Kyber PKE is based on the module-LWE. It takes the length *len_plain* of the plaintexts in the security games as an input. Note that the security proof is for the uncompressed scheme only! That means that the output of the key generation is not compressed and the input of the encryption is not decompressed. The compression/decompression would entail that the decompression of the value *t* from the key generation is not distributed uniformly at random any more (because of the compression error). This prohibits the second reduction to module-LWE. In order to avoid this, the compression and decompression in key generation and encryption have been omitted from the second round of the NIST standardisation process onwards.

```
locale kyber_new_security =  kyber_cor_new _ _ _ _ _ _ "TYPE('a::qr_spec)"
"TYPE('k::finite)" +
  ro: random_oracle len_plain
for len_plain :: nat +
fixes type_a :: "('a :: qr_spec) itself"
  and type_k :: "('k ::finite) itself"
begin
```

The given plaintext as a bitstring needs to be converted to an element in R_q. The bitstring is respresented as an integer value by interpreting the bitstring as a binary number. The integer is then converted to an element in R_q by the function *to_module*. Conversely, the bitstring representation can by extracted from the coefficient of the element in R_q.

```
definition bitstring_to_int:
"bitstring_to_int msg = (∑i<length msg. if msg!i then  2^i else 0)"

definition plain_to_msg :: "bitstring ⇒ 'a qr" where
"plain_to_msg msg = to_module (bitstring_to_int msg)"

definition msg_to_plain :: "'a qr ⇒ bitstring" where
"msg_to_plain msg = map (λi. i=0) (coeffs (of_qr msg))"
```

8.1 Instantiation of *ind_cpa* Locale with Kyber

We only look at the uncompressed version of Kyber. As the IND-CPA locale works over the generative probabilistic values type *gpv*, we need to lift our definitions to *gpv*'s.

The lifting of the key generation:

```
definition gpv_key_gen where
"gpv_key_gen = lift_spmf (spmf_of_pmf pmf_key_gen)"

lemma spmf_pmf_of_set_UNIV:
"spmf_of_set (UNIV:: (('a qr,'k) vec,'k) vec set) =
  spmf_of_pmf (pmf_of_set (UNIV:: (('a qr,'k) vec,'k) vec set))"
⟨proof⟩

lemma key_gen:
"gpv_key_gen = lift_spmf ( do {
    A ← spmf_of_set (UNIV:: (('a qr, 'k) vec, 'k) vec set);
    s ← spmf_of_pmf mlwe.beta_vec;
    e ← spmf_of_pmf mlwe.beta_vec;
    let t = key_gen_new A s e;
    return_spmf ((A, t),s)
  })"
⟨proof⟩
```

The lifting of the encryption:

```
definition gpv_encrypt ::
    "('a qr, 'k) pk ⇒ plain ⇒ (('a qr, 'k) vec × 'a qr, 'b, 'c) gpv"
where
"gpv_encrypt pk m = lift_spmf (spmf_of_pmf (pmf_encrypt pk (plain_to_msg
m)))"
```

The lifting of the decryption:

```
definition gpv_decrypt ::
  "('a qr, 'k) sk ⇒ ('a qr, 'k) cipher ⇒ (plain, ('a qr,'k) vec, bitstring)
gpv" where
"gpv_decrypt sk cipher = lift_spmf (do {
    let msg' = decrypt (fst cipher) (snd cipher) sk du dv ;
    return_spmf (msg_to_plain (msg'))
  })"
```

In order to verify that the plaintexts given by the adversary in the IND-CPA security game have indeed the same length, we define the test *valid_plains*.

```
definition valid_plains :: "plain ⇒ plain ⇒ bool" where
"valid_plains msg1 msg2 ⟷ (length msg1 = len_plain ∧ length msg2 =
len_plain)"
```

Now we can instantiate the IND-CPA locale with the lifted Kyber algorithms.

sublocale *ind_cpa: ind_cpa_pk "gpv_key_gen" "gpv_encrypt" "gpv_decrypt" valid_plains* ⟨*proof*⟩

8.2 Reduction Functions

Since we lifted the key generation and encryption functions to *gpv*'s, we need to show that they are lossless, i.e., that they have no failure.

lemma *lossless_key_gen[simp]: "lossless_gpv $\mathcal{I}$_full gpv_key_gen"*
⟨*proof*⟩

lemma *lossless_encrypt[simp]: "lossless_gpv $\mathcal{I}$_full (gpv_encrypt pk m)"*
⟨*proof*⟩

lemma *lossless_decrypt[simp]: "lossless_gpv $\mathcal{I}$_full (gpv_decrypt sk cipher)"*
⟨*proof*⟩

lemma *finite_UNIV_lossless_spmf_of_set:*
assumes *"finite (UNIV :: 'b set)"*
shows *"lossless_gpv $\mathcal{I}$_full (lift_spmf (spmf_of_set (UNIV :: 'b set)))"*
⟨*proof*⟩

The reduction functions give the concrete reduction of a IND-CPA adversary to a module-LWE adversary. The first function is for the reduction in the key generation using $m = k$, whereas the second reduction is used in the encryption with $m = k + 1$ (using the option type).

```
fun kyber_reduction1 ::
"(('a qr, 'k) pk, plain, ('a qr, 'k) cipher, ('a qr,'k) vec, bitstring,
'state) ind_cpa.adversary
  ⇒ ('a qr, 'k,'k) mlwe.adversary"
where
  "kyber_reduction1 (𝒜₁, 𝒜₂) A t = do {
    (((msg1, msg2), σ), s) ← exec_gpv ro.oracle (𝒜₁ (A, t)) ro.initial;
    try_spmf (do {
      _ :: unit ← assert_spmf (valid_plains msg1 msg2);
      b ← coin_spmf;
      (c, s1) ← exec_gpv ro.oracle (gpv_encrypt (A,t) (if b then msg1
else msg2)) s;
      (b', s2) ← exec_gpv ro.oracle (𝒜₂ c σ) s1;
      return_spmf (b' = b)
    }) (coin_spmf)
  }"

fun kyber_reduction2 ::
"(('a qr, 'k) pk, plain, ('a qr, 'k) cipher, ('a qr,'k) vec, bitstring,
'state) ind_cpa.adversary
  ⇒ ('a qr, 'k,'k option) mlwe.adversary"
where
  "kyber_reduction2 (𝒜₁, 𝒜₂) A' t' = do {
```

```
    let A = transpose (χ i. A' $ (Some i));
    let t = A' $ None;
    (((msg1, msg2), σ),s) ← exec_gpv ro.oracle (𝒜₁ (A, t)) ro.initial;
    try_spmf (do {
      _ :: unit ← assert_spmf (valid_plains msg1 msg2);
      b ← coin_spmf;
      let msg = (if b then msg1 else msg2);
      let u = (χ i. t' $ (Some i));
      let v = (t' $ None) + to_module (round((real_of_int q)/2)) * (plain_to_msg
msg);
      (b', s1) ← exec_gpv ro.oracle (𝒜₂ (compress_vec du u, compress_poly
dv v) σ) s;
      return_spmf (b'=b)
    }) (coin_spmf)
  }"
```

8.3 IND-CPA Security Proof

The following theorem states that if the adversary against the IND-CPA game is lossless (that is it does not act maliciously), then the advantage in the IND-CPA game can be bounded by two advantages against the module-LWE game. Under the module-LWE hardness assumption, the advantage against the module-LWE is negligible.

The proof proceeds in several steps, also called game-hops. Initially, the IND-CPA game is considered. Then we gradually alter the games and show that either the alteration has no effect on the resulting probabilities or we can bound the change by an advantage against the module-LWE. In the end, the game is a simple coin toss, which we know has probability 0.5 to guess the correct outcome. Finally, we can estimate the advantage against IND-CPA using the game-hops found before, and bounding it against the advantage against module-LWE.

```
theorem concrete_security_kyber:
assumes lossless: "ind_cpa.lossless 𝒜"
shows "ind_cpa.advantage (ro.oracle, ro.initial) 𝒜 ≤
  mlwe.advantage (kyber_reduction1 𝒜) + mlwe.advantage' (kyber_reduction2
𝒜)"
⟨proof⟩


end

end
theory Kyber_new_Values
imports
  Crypto_Scheme_new

begin
```

9 Specification for Kyber with $q = 3329$

Since NIST round 2, Kyber changed the modulus q from 7981 to 3329. In the following, a finite type with 3329 elements is defined and shown to fulfil the `prime_card` property.

typedef `fin3329 = "{0..<3329::int}"`
morphisms `fin3329_rep fin3329_abs`
⟨*proof*⟩

setup_lifting `type_definition_fin3329`

lemma `CARD_fin3329 [simp]:`
`"CARD (fin3329) = 3329"`
⟨*proof*⟩

lemma `fin3329_nontriv [simp]:`
`"1 < CARD(fin3329)"`
⟨*proof*⟩

The type $fin3329$ fulfils the `prime_card` property required by the `kyber_spec` locale.

lemma `prime_3329: "prime (3329::nat)"` ⟨*proof*⟩

instantiation `fin3329 :: comm_ring_1`
begin

lift_definition `zero_fin3329 :: "fin3329"` **is** `"0"` ⟨*proof*⟩

lift_definition `one_fin3329 :: "fin3329"` **is** `"1"` ⟨*proof*⟩

lift_definition `plus_fin3329 :: "fin3329 ⇒ fin3329 ⇒ fin3329"`
  is `"(λx y. (x+y) mod 3329)"`
⟨*proof*⟩

lift_definition `uminus_fin3329 :: "fin3329 ⇒ fin3329"`
  is `"(λx. (uminus x) mod 3329)"`
⟨*proof*⟩

lift_definition `minus_fin3329 :: "fin3329 ⇒ fin3329 ⇒ fin3329"`
  is `"(λx y. (x-y) mod 3329)"`
⟨*proof*⟩

lift_definition `times_fin3329 :: "fin3329 ⇒ fin3329 ⇒ fin3329"`
  is `"(λx y. (x*y) mod 3329)"`
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *fin3329 :: finite*
begin
instance
⟨*proof*⟩
end

instantiation *fin3329 :: equal*
begin
lift_definition *equal_fin3329 :: "fin3329 ⇒ fin3329 ⇒ bool"* **is** *"(=)"* ⟨*proof*⟩
instance ⟨*proof*⟩
end

instantiation *fin3329 :: nontriv*
begin
instance
⟨*proof*⟩
end

instantiation *fin3329 :: prime_card*
begin
instance
⟨*proof*⟩
end

Now, we can define the quotient type of R_{3329} over `fin3329`.

instantiation *fin3329 :: qr_spec*
begin

definition *qr_poly'_fin3329:: "fin3329 itself ⇒ int poly"* **where**
"qr_poly'_fin3329 ≡ (λ_. Polynomial.monom (1::int) 256 + 1)"

instance ⟨*proof*⟩
end

lift_definition *to_int_fin3329 :: "fin3329 ⇒ int"* **is** *"λx. x"* ⟨*proof*⟩

lift_definition *of_int_fin3329 :: "int ⇒ fin3329"* **is** *"λx. (x mod 3329)"*
⟨*proof*⟩

interpretation *to_int_fin3329_hom: inj_zero_hom to_int_fin3329*
⟨*proof*⟩

```
interpretation of_int_fin3329_hom: zero_hom of_int_fin3329
  ⟨proof⟩

lemma to_int_fin3329_of_int_fin3329 [simp]:
"to_int_fin3329 (of_int_fin3329 x) = x mod 3329"
⟨proof⟩

lemma of_int_fin3329_to_int_fin3329 [simp]:
"of_int_fin3329 (to_int_fin3329 x) = x"
⟨proof⟩

lemma of_int_mod_ring_eq_iff [simp]:
  "(of_int_fin3329 a = of_int_fin3329 b) ⟷
   ((a mod 3329) = (b mod 3329))"
⟨proof⟩
```

Finally, we show that the Kyber algorithms can be instantiated with $q = 3329$.

```
interpretation kyber3329: kyber_spec  256 3329 3 8 "TYPE(fin3329)" "TYPE(3)"
⟨proof⟩

end
theory Correct

imports "CRYSTALS-Kyber.Crypto_Scheme"
    Delta_Correct
    MLWE

begin
```

10 δ-Correctness of Kyber's Probabilistic Algorithms

The functions *key_gen*, *encrypt* and *decrypt* are deterministic functions that calculate the output of the Kyber algorithms for a given input. To completely model the Kyber algorithms, we need to model the random choice of the input as well. This results in probabilistic programs that first choose the input according the the input distributions and then calculate the output. Probabilistic programs are modeled by the Giry monad of *pmf*'s. The correspond to the probability mass functions of the output.

10.1 Definition of Probabilistic Kyber and δ

The correctness of Kyber is formulated in a locale that defines the necessary assumptions on the parameter set. For the correctness analysis we need to import the definitions of the probability distribution β_η from the module-

LWE and the Kyber locale itself. Moreover, we fix the compression depths for the outputs t, u and v.

```
locale kyber_cor = mlwe: module_lwe "(TYPE('a ::qr_spec))" "TYPE('k::finite)"
k  +
kyber_spec _ _ _ _"(TYPE('a ::qr_spec))" "TYPE('k::finite)"  +
fixes type_a :: "('a :: qr_spec) itself"
  and type_k :: "('k ::finite) itself"
  and dt du dv ::nat
begin
```

We define types for the private and public keys, as well as plain and cipher texts. The public key consists of a matrix $A \in R_q^{k\times k}$ and a (compressed) vector $t \in R_q^k$. The private key is the secret vector $s \in R_q$ such that there is an error vector $e \in R_q^k$ such that $A \cdot s + e = t$. The plaintext consists of a bitstring (ie. a list of booleans). The ciphertext is an element of R_q^{k+1}represented by a vector u in R_q^k and a value $v \in R_q$ (both compressed).

```
type_synonym ('b,'l) pk = "((('b,'l) vec,'l) vec) × (('b,'l) vec)"
type_synonym ('b,'l) sk = "('b,'l) vec"
type_synonym plain = bitstring
type_synonym ('b,'l) cipher = "('b,'l) vec × 'b"
```

Some finiteness properties.

```
lemma finite_bit_set:
"finite mlwe.bit_set"
⟨proof⟩


lemma finite_beta:
"finite (set_pmf mlwe.beta)"
⟨proof⟩


lemma finite_beta_vec:
"finite (set_pmf mlwe.beta_vec)"
⟨proof⟩
```

The probabilistic program for key generation and encryption. The decryption does not need a probabilistic program, since there is no random choice involved.

We need to give back the error term as part of the secret key since otherwise we lose this information and cannot recalculate it. This is needed in the proof of correctness. Since the δ was modified for the originally claimed one, this could be improved.

```
definition pmf_key_gen  where
"pmf_key_gen = do {
  A ← pmf_of_set (UNIV:: (('a qr,'k) vec,'k) vec set);
```

```
  s ← mlwe.beta_vec;
  e ← mlwe.beta_vec;
  let t = key_gen dt A s e;
  return_pmf ((A, t),(s,e))
}"
```

definition *pmf_encrypt* **where**

```
"pmf_encrypt pk m = do{
  r ← mlwe.beta_vec;
  e1 ← mlwe.beta_vec;
  e2 ← mlwe.beta;
  let c = encrypt (snd pk) (fst pk) r e1 e2 dt du dv m;
  return_pmf c
}"
```

The message space is *Msgs*. It is finite and non-empty.

definition

```
"Msgs = {m::'a qr. set ((coeffs ∘ of_qr) m) ⊆ {0,1}}"
```

lemma *finite_Msgs:*

```
"finite Msgs"
```

⟨*proof*⟩

lemma *Msgs_nonempty:*

```
"Msgs ≠ {}"
```

⟨*proof*⟩

Now we can instantiate the public key encryption scheme correctness locale with the probabilistic algorithms of Kyber. This hands us the definition of δ-correctness.

no_adhoc_overloading *Monad_Syntax.bind bind_pmf*

sublocale *pke_delta_correct pmf_key_gen pmf_encrypt*
"(λ sk c. decrypt (fst c) (snd c) (fst sk) du dv)" Msgs ⟨*proof*⟩

adhoc_overloading *Monad_Syntax.bind bind_pmf*

The following functions return the distribution of the compression error (for vectors and polynomials).

definition

```
"error_dist_vec d = do{
  y ← pmf_of_set (UNIV :: ('a qr,'k) vec set);
  return_pmf (decompress_vec d (compress_vec d y)-y)
}"
```

definition

```
"error_dist_poly d = do{
  y ← pmf_of_set (UNIV :: 'a qr set);
```

```
  return_pmf (decompress_poly d (compress_poly d y)-y)
}"
```

The functions *w_distrib'*, *w_distrib* and *w_dist* define the originally claimed δ (here *delta_kyber*) for the correctness of Kyber. However, the *delta*-correctness of Kyber could not be formalized.

The reason is that the values of *ct*, *cu* and *cv* in *w_distrib'* rely on the compression error of uniformly random generated values. In truth, these values are not uniformly generated but instances of the module-LWE. However, we cannot use the module-LWE assumption to reduce these values to uniformly generated ones since we would lose all information about the secret key otherwise. This is needed to perform the decryption in order to check whether the original message and the decryption of the ciphertext are indeed the same. The *delta_kyber* with additional module-LWE errors are calculated in *delta*.

Therefore, we modified the given δ and defined a new value *delta'* in order to prove at least *delta'*-correctness.

```
definition w_distrib' where
"w_distrib' s e r e1 e2 = do{
  ct ← error_dist_vec dt;
  cu ← error_dist_vec du;
  cv ← error_dist_poly dv;
  let w = (scalar_product e r + e2 + cv + scalar_product ct r
    - scalar_product s e1 - scalar_product s cu);
  return_pmf (abs_infty_poly w ≥ round (q/4))}"

definition w_distrib where
"w_distrib s e = do{
  r ← mlwe.beta_vec;
  e1 ← mlwe.beta_vec;
  e2 ← mlwe.beta;
  w_distrib' s e r e1 e2}"

definition w_dist where
"w_dist = do{
  s ← mlwe.beta_vec;
  e ← mlwe.beta_vec;
  w_distrib s e}"

definition delta_kyber where
"delta_kyber =  pmf w_dist True"

definition delta where
"delta Adv0 Adv1 = delta_kyber + mlwe.advantage Adv0 + mlwe.advantage1
Adv1"
```

The functions *w_kyber'*, *w_kyber*, *delta'* and *delta_kyber'* define the modified δ for the correctness proof. Note the in *w_kyber'*, the values *t*, *yu* and *yv* are generated according to their corresponding module-LWE instances and are not uniformly random. *delta'* is still dependent on the public and secret keys and the message. This dependency is eliminated in *delta_kyber'* by taking the expectation over the key pair and the maximum over all messages, similar to the definition of δ-correctness.

```
definition w_kyber' where
"w_kyber' A s e m r e1 e2 = do{
  let t = A *v s + e;
  let ct = compress_error_vec dt t;
  let yu = transpose A *v r + e1;
  let yv = (scalar_product t r + scalar_product ct r + e2 +
           to_module (round (real_of_int q / 2)) * m);
  let cu = compress_error_vec du yu;
  let cv = compress_error_poly dv yv;
  let w = (scalar_product e r + e2 + cv + scalar_product ct r - scalar_product
s e1 -
    scalar_product s cu);
  return_pmf (abs_infty_poly w ≥ round (q/4))}"

definition w_kyber where
"w_kyber A s e m = do{
  r ← mlwe.beta_vec;
  e1 ← mlwe.beta_vec;
  e2 ← mlwe.beta;
  w_kyber' A s e m r e1 e2}"

definition delta' where
"delta' sk pk m = pmf (w_kyber (fst pk) (fst sk) (snd sk) m) True"

definition delta_kyber' where
"delta_kyber' = measure_pmf.expectation pmf_key_gen
     (λ(pk, sk). MAX m∈Msgs. delta' sk pk m)"
```

10.2 δ-Correctness Proof

The idea to bound the probabilistic Kyber algorithms by *delta_kyber'* is the following: First use the deterministic part given by *CRYSTALS-Kyber.Crypto_Scheme.kyber_correct* to bound the correctness by *delta'* depending on a fixed key pair and message. Then bound the message by the maximum over all messages. Finally bound the key pair by using the expectation over the key pair. The result is that the correctness error of the Kyber PKE is bounded by *delta_kyber'*.

First of all, we rewrite the deterministic part of the correctness proof *kyber_correct* from *CRYSTALS-Kyber.Crypto_Scheme*.

```
lemma kyber_correct_alt:
  fixes A s r e e1 e2 cu cv t u v
```

```
  assumes t_def:   "t = key_gen dt A s e"
  and u_v_def: "(u,v) = encrypt t A r e1 e2 dt du dv m"
  and ct_def:  "ct = compress_error_vec dt (A *v s + e)"
  and cu_def:  "cu = compress_error_vec du
               ((transpose A) *v r + e1)"
  and cv_def:  "cv = compress_error_poly dv
               (scalar_product (decompress_vec dt t) r + e2 +
                to_module (round((real_of_int q)/2)) * m)"
  and error: "decrypt u v s du dv ≠ m"
  and m01:     "set ((coeffs ∘ of_qr) m) ⊆ {0,1}"
  shows "abs_infty_poly (scalar_product e r + e2 + cv + scalar_product
ct r
    - scalar_product s e1 - scalar_product s cu) ≥ round (real_of_int
q / 4)"
⟨proof⟩
```

Then we show the correctness in the probabilistic program for a fixed key pair and message. The bound we use is `delta'`.

```
lemma correct_key_gen:
fixes A s e m
assumes pk_sk: "(pk, sk) = ((A, key_gen dt A s e), (s,e))"
  and m_Msgs: "m∈Msgs"
shows "pmf (do{c ← pmf_encrypt pk m;
  return_pmf (decrypt (fst c) (snd c) (fst sk) du dv ≠ m)}) True ≤ delta'
sk pk m"
⟨proof⟩
```

Now take the maximum over all messages. We rewrite this in order to be able to instantiate it nicely.

```
lemma correct_key_gen_max:
fixes A s e m
assumes "(pk, sk) = ((A, key_gen dt A s e), (s,e))"
  and "m∈Msgs"
shows "pmf (do{c ← pmf_encrypt pk m;
  return_pmf (decrypt (fst c) (snd c) (fst sk) du dv ≠ m)}) True ≤ (MAX
m'∈Msgs. delta' sk pk m')"
⟨proof⟩

lemma correct_max:
fixes A s e
assumes "(pk, sk) = ((A, key_gen dt A s e), (s,e))"
shows "(MAX m∈Msgs. pmf (do{c ← pmf_encrypt pk m;
  return_pmf (decrypt (fst c) (snd c) (fst sk) du dv ≠ m)}) True) ≤ (MAX
m'∈Msgs. delta' sk pk m')"
⟨proof⟩

lemma correct_max':
fixes pk sk
assumes "snd pk = compress_vec dt ((fst pk) *v (fst sk) + (snd sk))"
```

```
shows "(MAX m∈Msgs. pmf (do{c ← pmf_encrypt pk m;
  return_pmf (decrypt (fst c) (snd c) (fst sk) du dv ≠ m)}) True) ≤
  (MAX m'∈Msgs. delta' sk pk m')"
⟨proof⟩
```

Finally show the overall bound `delta_kyber'` for the correctness error of the Kyber PKE.

```
lemma expect_correct:
"expect_correct ≤ delta_kyber'"
⟨proof⟩
```

This yields the overall `delta_kyber'`-correctness of Kyber.

```
lemma delta_correct_kyber:
"delta_correct delta_kyber'"
⟨proof⟩


end
end
```

References

[1] G. Alagic, D. A. Cooper, Q. Dang, T. Dang, J. M. Kelsey, J. Lichtinger, Y.-K. Liu, C. A. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and D. Apon. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process, 2022-07-05 04:07:00 2022.

[2] R. M. Avanzi, J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 3.0). 01/10/2020.

[3] R. M. Avanzi, J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation. 2017.

[4] R. M. Avanzi, J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 2.0). 30/03/2019.

[5] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS — Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy*, pages 353–367, 2018.

[6] K. Kreuzer. CRYSTALS-Kyber. *Archive of Formal Proofs*, September 2022. https://isa-afp.org/entries/CRYSTALS-Kyber.html, Formal proof development.

[7] K. Kreuzer. Verification of Correctness and Security Properties for CRYSTALS-KYBER. In *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*, page TBD, Los Alamitos, CA, USA, 2024. IEEE Computer Society.

[8] A. Langlois and D. Stehlé. Worst-Case to Average-Case Reductions for Module Lattices. *Des. Codes Cryptogr.*, 75(3):565–599, June 2015.

[9] A. Lochbihler. CryptHOL. *Archive of Formal Proofs*, May 2017. https://isa-afp.org/entries/CryptHOL.html, Formal proof development.

[10] A. Lochbihler and S. R. Sefidgar. A tutorial introduction to CryptHOL. Cryptology ePrint Archive, Paper 2018/941, 2018. https://eprint.iacr.org/2018/941.