Verification of Correctness and Security Properties for CRYSTALS-KYBER

Katharina Kreuzer

May 26, 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 -
```

First, show that the calculations are performed correctly.

```
  have t_correct: "t = A *v s + e" unfolding t_def key_gen_new_def by
simp
  have u_correct: "decompress_vec du u =
    (transpose A) *v r + e1 + cu"
    using u_v_def cu_def unfolding encrypt_new_def
    compress_error_vec_def by simp
  have v_correct: "decompress_poly dv v =
    scalar_product t r + e2 +
    to_module (round((real_of_int q)/2)) * m + cv"
    using u_v_def unfolding encrypt_new_def
    compress_error_poly_def cv_def by simp
  have v_correct': "decompress_poly dv v =
    scalar_product (A *v s + e) r + e2 +
    to_module (round((real_of_int q)/2)) * m + cv"
   using t_correct v_correct
    by (auto simp add: scalar_product_linear_left)
  let ?u = "decompress_vec du u"
  let ?v = "decompress_poly dv v"
```

Define w as the error term of the message encoding. Have $\|w\|_{\infty,q} < \lceil q/4 \rfloor$

```
  define w where "w = scalar_product e r + e2 + cv -
    scalar_product s e1 -
    scalar_product s cu"
```

```
have w_length: "abs_infty_poly w < round (real_of_int q / 4)"
  unfolding w_def using delta by auto
moreover have "abs_infty_poly w = abs_infty_poly (-w)"
  unfolding abs_infty_poly_def
  using neg_mod_plus_minus[OF q_odd q_gt_zero]
  using abs_infty_q_def abs_infty_q_minus by auto
ultimately have minus_w_length:
  "abs_infty_poly (-w) < round (real_of_int q / 4)"
  by auto
have vsu: "?v - scalar_product s ?u =
      w + to_module (round((real_of_int q)/2)) * m"
  unfolding w_def by (auto simp add: u_correct v_correct'
  scalar_product_linear_left scalar_product_linear_right
  scalar_product_assoc)
```

Set m' as the actual result of the decryption. It remains to show that $m' = m$.

```
define m' where "m' = decrypt u v s du dv"
have coeffs_m': "∀i. poly.coeff (of_qr m') i ∈ {0,1}"
  unfolding m'_def decrypt_def using compress_poly_1 by auto
```

Show $\|v - s^T u - \lceil q/2 \rfloor m'\|_{\infty,q} \leq \lceil q/4 \rfloor$

```
have "abs_infty_poly (?v - scalar_product s ?u -
  to_module (round((real_of_int q)/2)) * m')
= abs_infty_poly (?v - scalar_product s ?u -
  decompress_poly 1 (compress_poly 1 (?v - scalar_product s ?u)))"
  by (auto simp flip: decompress_poly_1[of m', OF coeffs_m'])
     (simp add:m'_def decrypt_def)
also have "... ≤ round (real_of_int q / 4)"
  using decompress_compress_poly[of 1 "?v - scalar_product s ?u"]
    q_gt_two by fastforce
finally have "abs_infty_poly (?v - scalar_product s ?u -
  to_module (round((real_of_int q)/2)) * m') ≤
  round (real_of_int q / 4)"
  by auto
```

Show $\|\lceil q/2 \rfloor (m - m'))\|_{\infty,q} < 2\lceil q/4 \rfloor$

```
then have "abs_infty_poly (w + to_module
  (round((real_of_int q)/2)) * m - to_module
  (round((real_of_int q)/2)) * m') ≤ round (real_of_int q / 4)"
  using vsu by auto
then have w_mm': "abs_infty_poly (w +
  to_module (round((real_of_int q)/2)) * (m - m'))
  ≤ round (real_of_int q / 4)"
  by (smt (verit) add_uminus_conv_diff is_num_normalize(1)
    right_diff_distrib')
have "abs_infty_poly (to_module
    (round((real_of_int q)/2)) * (m - m')) =
  abs_infty_poly (w + to_module
```

```
    (round((real_of_int q)/2)) * (m - m') - w)"
  by auto
also have "... ≤ abs_infty_poly
  (w + to_module (round((real_of_int q)/2)) * (m - m'))
   + abs_infty_poly (- w)"
  using abs_infty_poly_triangle_ineq[of
    "w+to_module (round((real_of_int q)/2)) * (m - m')" "-w"]
  by auto
also have "... < 2 * round (real_of_int q / 4)"
  using w_mm' minus_w_length by auto
finally have error_lt: "abs_infty_poly (to_module
  (round((real_of_int q)/2)) * (m - m')) <
  2 * round (real_of_int q / 4)"
  by auto
```

Finally show that $m - m'$ is small enough, ie that it is an integer smaller than one. Here, we need that $q \cong 1 \mod 4$.

```
have coeffs_m':"set ((coeffs ∘ of_qr) m') ⊆ {0,1}"
proof -
  have "compress 1 a ∈ {0,1}" for a
  unfolding compress_def by auto
  then have "poly.coeff (of_qr (compress_poly 1 a)) i ∈ {0,1}"
    for a i
    using compress_poly_1 by presburger
  then have "set (coeffs (of_qr (compress_poly 1 a))) ⊆ {0,1}"
    for a
    using coeffs_in_coeff[of "of_qr (compress_poly 1 a)" "{0,1}"]
    by simp
  then show ?thesis unfolding m'_def decrypt_def by simp
qed
have coeff_0pm1: "set ((coeffs ∘ of_qr) (m-m')) ⊆
  {of_int_mod_ring (-1),0,1}"
proof -
  have "poly.coeff (of_qr m) i ∈ {0,1}"
    for i using m01 coeff_in_coeffs
    by (metis comp_def insertCI le_degree subset_iff
      zero_poly.rep_eq)
  moreover have "poly.coeff (of_qr m') i ∈ {0,1}" for i
    using coeffs_m' coeff_in_coeffs
    by (metis comp_def insertCI le_degree subset_iff zero_poly.rep_eq)
  ultimately have "poly.coeff (of_qr m - of_qr m') i ∈
    {of_int_mod_ring (- 1), 0, 1}" for i
    by (metis (no_types, lifting) coeff_diff diff_zero
      eq_iff_diff_eq_0 insert_iff of_int_hom.hom_one of_int_minus
      of_int_of_int_mod_ring singleton_iff verit_minus_simplify(3))
  then have "set (coeffs (of_qr m - of_qr m')) ⊆
    {of_int_mod_ring (- 1), 0, 1}"
    by (simp add: coeffs_in_coeff)
  then show ?thesis using m01 of_qr_diff[of m m'] by simp
```

```
  qed
  have "set ((coeffs ∘ of_qr) (m-m')) ⊆ {0}"
  proof (rule ccontr)
    assume "¬set ((coeffs ∘ of_qr) (m-m')) ⊆ {0}"
    then have "∃i. poly.coeff (of_qr (m-m')) i ∈
      {of_int_mod_ring (-1),1}"
      using coeff_0pm1
      by (smt (z3) coeff_in_coeffs comp_apply insert_iff
        leading_coeff_0_iff order_refl
        set_coeffs_subset_singleton_0_iff subsetD)
    then have error_ge: "abs_infty_poly (to_module
      (round((real_of_int q)/2)) * (m-m')) ≥
      2 * round (real_of_int q / 4)"
      using abs_infty_poly_ineq_pm_1 by simp
    show False using error_lt error_ge by simp
  qed
  then show ?thesis by (simp flip: m'_def) (metis to_qr_of_qr)
qed

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 -
  obtain *x* **where** *"x∈A"* **and** *"(∑y∈B. f x y) = (MAX x∈A. (∑y∈B. f x y))"* **using** *assms*
  by *(metis (no_types, lifting) Max_in finite_imageI imageE image_is_empty)*
  moreover have *"(∑y∈B. f x y) ≤ (∑y∈B. MAX x∈A. f x y)"*
    by *(intro sum_mono) (simp add: assms(1) calculation(1))*
  ultimately show *?thesis* **by** *metis*
qed

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 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 -
  let ?q = "(CARD('a))"
  let ?n = "degree (qr_poly' TYPE('a))"
  have fin: "finite (UNIV :: 'a mod_ring set)" by simp
  then obtain f where "bij_betw f (UNIV::'a mod_ring set) {0..< ?q}"
    by (metis CARD_mod_ring ex_bij_betw_finite_nat)
  have rew:"(UNIV :: 'a qr set) = (to_qr ∘ Poly ) ` {xs :: 'a mod_ring
list. length xs = ?n}"
  proof (safe, goal_cases)
    case (1 x)
    let ?xs = "coeffs (of_qr x)"
    define ys where "ys = ?xs @ replicate (?n-length ?xs) 0"
    have "length (coeffs (of_qr x)) ≤ degree (qr_poly' TYPE('a))"
    by (metis Suc_le_eq coeffs_eq_Nil deg_of_qr deg_qr_def length_coeffs_degree

      list.size(3) zero_le)
    then have "length (coeffs (of_qr x)) + (degree (qr_poly' TYPE('a))
- length (coeffs (of_qr x))) =
      degree (qr_poly' TYPE('a))" by (subst add_diff_assoc) (auto simp
add: 1)
    then have "length ys = ?n" unfolding ys_def by (auto)
    moreover have "x = to_qr (Poly ys)" unfolding ys_def Poly_append_replicate_zero

      Poly_coeffs to_qr_of_qr by simp
    ultimately have "∃ ys. length ys = ?n ∧ x = to_qr (Poly ys)" by
blast
    then show ?case by force
  qed simp
  have "card (UNIV :: 'a qr set) = card {xs :: 'a mod_ring list. length
xs = ?n}"
  proof (unfold rew, rule card_image, subst comp_inj_on_iff[symmetric],
goal_cases)
    case 1
    then show ?case by (metis (mono_tags, lifting) coeff_Poly_eq inj_onI
map_nth_default
      mem_Collect_eq nat_int)
  next
    case 2
    then show ?case unfolding inj_on_def proof (safe, goal_cases)
      case (1 x xa y xb)
```

```
      moreover have "Poly xa mod qr_poly = Poly xa" using 1(1)
        by (intro deg_mod_qr_poly) (metis coeff_Poly_eq deg_qr_pos degree_0
degree_qr_poly'
          leading_coeff_0_iff nth_default_def)
      moreover have "Poly xb mod qr_poly = Poly xb" using 1(2)
        by (intro deg_mod_qr_poly) (metis coeff_Poly_eq deg_qr_pos degree_0
degree_qr_poly'
          leading_coeff_0_iff nth_default_def)
      ultimately show ?case unfolding to_qr_eq_iff  by (simp add: cong_def)
    qed
  qed
  also have "... = ?q^(?n)" using card_lists_length_eq[OF fin, of "nat
?n"]
    by (auto)
  ultimately show ?thesis by auto
qed


lemma finite_qr [simp]:
  "finite (UNIV::'a::qr_spec qr set)" using card_UNIV_qr
  by (metis card.infinite card_UNIV_option card_option nat.distinct(1)
power_not_zero)

instantiation qr ::(qr_spec) finite
begin
instance
proof
  show "finite (UNIV :: 'a ::qr_spec qr set)" using finite_qr by simp

qed
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 -
  have "bij_betw vec_lambda (UNIV :: ('k ⇒ 'a) set) (UNIV :: ('a, 'k)
vec set)"
    by (metis UNIV_I bijI' vec_lambda_cases vec_lambda_inject)
  show ?thesis
  by (meson <bij vec_lambda> assms bij_betw_finite finite_UNIV_fun finite_class.finite_UNI
qed

lemma finite_mat:
assumes "finite (UNIV :: 'a set)"
shows "finite (UNIV :: (('a, 'k::finite) vec,'k) vec set)"
```

```
using finite_vec[OF finite_vec[OF assms]] by auto


lemma finite_UNIV_vec [simp]:
  "finite (UNIV:: ('a::qr_spec qr, 'k::finite) vec set)"
using finite_vec[OF finite_qr] .

lemma finite_UNIV_mat [simp]:
  "finite (UNIV:: (('a::qr_spec qr, 'k) vec, 'k::finite) vec set)"
using finite_mat[OF finite_qr] .

lemma finite_UNIV_vec_option [simp]:
  "finite (UNIV :: ('a::qr_spec qr,'k::finite option) vec set)"
by (simp add: finite_vec)

lemma finite_UNIV_mat_option [simp]:
  "finite (UNIV:: (('a::qr_spec qr, 'k::finite) vec, 'k option) vec set)"
using finite_vec[OF finite_qr] finite_vec by blast


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)
  }"
unfolding replicate_spmf_def
by (simp add: spmf_of_pmf_bind bind_commute_pmf[of p])

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])
  }"
unfolding replicate_spmf_def replicate_pmf_distrib[of m 1 p]
by (simp add: spmf_of_pmf_bind bind_assoc_pmf bind_return_pmf)
```

Lemmas on *coin_spmf*

```
lemma spmf_coin_spmf: "spmf coin_spmf i = 1/2"
by (simp add: spmf_of_set)

lemma bind_spmf_coin:
assumes "lossless_spmf p"
shows "bind_spmf p (λ_. coin_spmf) = coin_spmf"
proof (rule spmf_eqI, goal_cases)
  case (1 i)
  have weight: "weight_spmf p = 1" using assms by (simp add: lossless_spmf_def)
  have *: "spmf (bind_spmf p (λ_. coin_spmf)) i = 1/2"
    unfolding spmf_bind by (simp add: weight spmf_coin_spmf)
  then show ?case unfolding spmf_coin_spmf * by simp
qed

lemma if_splits_coin:
"(if P then coin_spmf else coin_spmf) = coin_spmf"
by simp
```

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"
by (smt (verit, best) Finite_Cartesian_Product.sum_cong_aux
Groups_Big.comm_monoid_add_class.sum.delta assms(1) assms(2) iso_tuple_UNIV_I
of_bool_def)

lemma (in kyber_spec) surj_add_qr:
"surj (λx. x + (y:: 'a qr))"
unfolding surj_def
by (metis Groups.group_cancel.sub1 add_ac(2) add_diff_cancel_left')

lemma (in kyber_spec) bij_add_qr:
"bij (λx. x + (y::'a qr))"
by (simp add: bij_def surj_add_qr)
```

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"
unfolding Let_def
proof (intro spmf_eqI, goal_cases)
```

```
  case (1 i)
  have "(∑x∈UNIV. ∑xa∈UNIV. of_bool (x + xa = i)) = real CARD('a qr)
^ CARD('k)"
  proof -
    have "∃!xa. x + xa = i" for x
    by (metis add_diff_cancel_left' group_cancel.sub1)
    then have "(∑xa∈UNIV. of_bool (x + xa = i)) = 1" for x by (intro
ex1_sum, auto)
    then show ?thesis
    by (smt (verit, best) CARD_vec of_nat_eq_of_nat_power_cancel_iff real_of_card
sum.cong)
  qed
  then show ?case by (simp add: spmf_bind spmf_of_set integral_spmf_of_set
indicator_def)
qed

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"
unfolding Let_def
proof (intro spmf_eqI, goal_cases)
  case (1 i)
  have "(∑x∈UNIV. ∑xa∈UNIV. of_bool (x - xa = i)) = real CARD('a qr)
^ CARD('k)"
  proof -
    have "∃!xa. x - xa = i" for x
    by (metis (no_types, opaque_lifting) cancel_ab_semigroup_add_class.diff_right_commute

      right_minus_eq)
    then have "(∑xa∈UNIV. of_bool (x - xa = i)) = 1" for x by (intro
ex1_sum, auto)
    then show ?thesis
    by (smt (verit, best) CARD_vec of_nat_eq_of_nat_power_cancel_iff real_of_card
sum.cong)
  qed
  then show ?case by (simp add: spmf_bind spmf_of_set integral_spmf_of_set
indicator_def)
qed


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 exactly 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"
unfolding bit_set_def
by (simp add: finite_lists_length_eq)

lemma bit_set_nonempty:
"bit_set ≠ {}"
proof -
  have "replicate eta 0 ∈ bit_set" unfolding bit_set_def by auto
  then show ?thesis by auto
qed
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)"
by (simp)
```

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 (cases i)
  case None
  then show ?thesis using nth_append_length[of xs x "[]"] that by (simp
add: if_distrib)
next
  case (Some a)
  have "idx a < k" using bij_idx
  by (meson UNIV_I atLeastLessThan_iff bij_betwE)
  then show ?thesis using Some that by (simp add: if_distrib nth_append)
qed


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 using idx'[OF that]
 by presburger
```

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'"
unfolding beta_vec'_def beta_def
by (simp add: replicate_spmf_def)
```

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"
by (metis (mono_tags, lifting) assms bind_pmf_cong mem_Collect_eq set_replicate_pmf)

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)"
unfolding replicate_spmf_def bind_spmf_of_pmf
by (simp add: replicate_pmf_same_length[OF assms])
```

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)))
  }"
unfolding beta_vec'_def idx'_lambda[symmetric] replicate_spmf_Suc_cons

  bind_spmf_assoc return_bind_spmf
by (subst replicate_spmf_same_length[where
  f = "(λy. spmf_of_pmf beta ⤜ (λya. return_spmf (vec_lambda (λi. (ya
# y) ! idx' i))))" and
  g = "(λxs. spmf_of_pmf beta ⤜ (λx. return_spmf (vec_lambda (λi. if
i = None then x else
        xs ! idx (the i)))))"])
   (simp_all add: idx'_lambda)

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"
unfolding mlwe.bit_set_def
  by (simp add: finite_lists_length_eq)


lemma finite_beta:
"finite (set_pmf mlwe.beta)"
unfolding mlwe.beta_def
  by (meson UNIV_I finite_qr finite_subset subsetI)


lemma finite_beta_vec:
"finite (set_pmf mlwe.beta_vec)"
unfolding mlwe.beta_vec_def
  by (meson finite_UNIV_vec finite_subset top.extremum)
```

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"
```
unfolding *Msgs_def*
  by *(meson finite_qr rev_finite_subset subset_UNIV)*

lemma *Msgs_nonempty:*
```
"Msgs ≠ {}"
```
proof -
  have *"to_qr (Poly [0]) ∈ Msgs"* **unfolding** *Msgs_def* **by** *auto*
  then show *?thesis* **by** *auto*
qed

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 .
```

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)"
using kyber_new_correct[OF assms(1-4) _ m01]
using assms(5) by linarith
```

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 -
  have "s = sk" using assms(1) by blast
  define ind1 where "ind1 = (λ r e1 e2. indicat_real
    {e2. decrypt (fst (encrypt_new (snd pk) (fst pk) r e1 e2 du dv m))
    (snd (encrypt_new (snd pk) (fst pk) r e1 e2 du dv m)) sk du dv ≠
m} e2)"
  define ind2 where "ind2 = (λr e1 e2. indicat_real {e2. (round (real_of_int
q / 4)
    ≤ abs_infty_poly (scalar_product (snd pk - fst pk *v sk) r + e2 +
        compress_error_poly dv (scalar_product (snd pk) r + e2 +
        to_module (round (real_of_int q / 2)) * m) - scalar_product sk
e1 -
        scalar_product sk (compress_error_vec du (r v* fst pk + e1))))}
e2)"
  have "ind1 r e1 e2 ≤ ind2 r e1 e2 "
    for r e1 e2
  proof (cases "ind1 r e1 e2 = 0")
    case True
    show ?thesis unfolding True by (auto simp add: ind2_def sum_nonneg)
  next
    case False
    then have one: "ind1 r e1 e2 = 1" unfolding ind1_def indicator_def
by simp
    define u where "u = fst (encrypt_new (snd pk) (fst pk) r e1 e2 du
dv m)"
    define v where "v = snd (encrypt_new (snd pk) (fst pk) r e1 e2 du
dv m)"
    define cu  where "cu = compress_error_vec du ((transpose (fst pk))
*v r + e1)"
    define cv where "cv = compress_error_poly dv (scalar_product (snd
pk) r + e2 +
       to_module (round((real_of_int q)/2)) * m)"
    define e' where "e' = (snd pk) -(fst pk) *v sk"
    have m: "set ((coeffs ∘ of_qr) m) ⊆ {0, 1}" using ‹m∈Msgs› unfold-
ing Msgs_def by simp
    have cipher: "(u,v) = encrypt_new (snd pk) (fst pk) r e1 e2 du dv
m"
      unfolding u_def v_def by simp
    have decrypt: "decrypt u v s du dv ≠ m" using one
      using assms(1) u_def v_def ind1_def by force
    have two: "ind2 r e1 e2 = 1" unfolding ind2_def
      using kyber_new_correct_alt[OF _ cipher cu_def cv_def decrypt m,
of e']
      unfolding ‹s=sk› by (simp add: cu_def cv_def e'_def key_gen_new_def
```

```
pk_sk)
    show ?thesis using one two by simp
  qed
  then have "(∑ e2∈UNIV. pmf mlwe.beta e2 * ind1 r e1 e2)
    ≤ (∑ e2∈UNIV. pmf mlwe.beta e2 *  ind2 r e1 e2)"
    for r e1
  by (intro sum_mono) (simp add: mult_left_mono)
  then have "(∑ e1∈UNIV. pmf mlwe.beta_vec e1 * (∑ e2∈UNIV. pmf mlwe.beta
e2 * ind1 r e1 e2))
    ≤ (∑ e1∈UNIV. pmf mlwe.beta_vec e1 * (∑ e2∈UNIV. pmf mlwe.beta e2
*
       ind2 r e1 e2))"
    for r
  by (intro sum_mono) (simp add: mult_left_mono)
  then have "(∑ r∈UNIV. pmf mlwe.beta_vec r * (∑ e1∈UNIV. pmf mlwe.beta_vec
e1 *
      (∑ e2∈UNIV. pmf mlwe.beta e2 * ind1 r e1 e2)))
    ≤ (∑ r∈UNIV. pmf mlwe.beta_vec r * (∑ e1∈UNIV. pmf mlwe.beta_vec
e1 * (∑ e2∈UNIV. pmf mlwe.beta e2 *
       ind2 r e1 e2 )))"
  by (intro sum_mono) (simp add: mult_left_mono)
  then show ?thesis unfolding delta'_def w_kyber_def pmf_encrypt_def
Let_def ind1_def ind2_def
  by (simp add: bind_assoc_pmf pmf_bind integral_measure_pmf[of UNIV]
sum_distrib_left)
qed
```

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')"
using correct_key_gen[OF assms]
by (meson Lattices_Big.linorder_class.Max.coboundedI assms(2) basic_trans_rules(23)

  finite_Msgs finite_imageI image_eqI)

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')"
using correct_key_gen_max[OF assms]
by (simp add: Msgs_nonempty finite_Msgs)
```

```
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')"
using correct_max[of pk sk] unfolding key_gen_new_def
by (metis (no_types, lifting) Groups.ab_semigroup_add_class.add.commute

  Product_Type.old.prod.exhaust diff_add_cancel)
```

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"
unfolding expect_correct_def delta_kyber_def using correct_max'
proof (subst integral_measure_pmf[of UNIV], goal_cases)
  case 3
  then show ?case proof (subst integral_measure_pmf[of UNIV], goal_cases)
  case 3
  then show ?case
    by (intro sum_mono, unfold real_scaleR_def, intro mult_left_mono)
auto
  qed (auto simp add: finite_prod)
qed (auto simp add: finite_prod)
```

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

```
lemma delta_correct_kyber:
"delta_correct delta_kyber"
using expect_correct delta_correct_def by auto


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))"
unfolding spmf_of_set_def by simp
```

```
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)
  })"
unfolding gpv_key_gen_def pmf_key_gen_def unfolding spmf_pmf_of_set_UNIV
unfolding bind_spmf_of_pmf by (auto simp add: spmf_of_pmf_bind)
```

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 .
```

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 ℐ_full gpv_key_gen"
unfolding gpv_key_gen_def by auto

lemma lossless_encrypt[simp]: "lossless_gpv ℐ_full (gpv_encrypt pk m)"
unfolding gpv_encrypt_def by auto
```

```
lemma lossless_decrypt[simp]: "lossless_gpv ℐ_full (gpv_decrypt sk cipher)"
unfolding gpv_decrypt_def by auto

lemma finite_UNIV_lossless_spmf_of_set:
assumes "finite (UNIV :: 'b set)"
shows "lossless_gpv ℐ_full (lift_spmf (spmf_of_set (UNIV :: 'b set)))"
using assms by simp
```

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 -
  note [cong del] = if_weak_cong
   and [split del] = if_split
   and [simp] = map_lift_spmf gpv.map_id lossless_weight_spmfD
        map_spmf_bind_spmf bind_spmf_const
   and [if_distribs] = if_distrib[where f="λx. try_spmf x _"]
       if_distrib[where f="weight_spmf"]
       if_distrib[where f="λr. scale_spmf r _"]
```

First of all, we can split the IND-CPA adversary $\mathcal{A}$ into two parts, namely the adversary who returns two messages $\mathcal{A}_1$ and the adversary who returns a guess $\mathcal{A}_2$.

```
  obtain 𝒜₁ 𝒜₂ where 𝒜 [simp]: "𝒜 = (𝒜₁, 𝒜₂)" by (cases "𝒜")
  from lossless have lossless1 [simp]: "⋀pk. lossless_gpv ℐ_full (𝒜₁
pk)"
    and lossless2 [simp]: "⋀σ cipher. lossless_gpv ℐ_full (𝒜₂ σ cipher)"
    by(auto simp add: ind_cpa.lossless_def)
```

The initial game $game_0$ is an equivalent formulation of the IND-CPA game.

```
  define game₀ where
  "game₀ = 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 pk = (A, A *v s + e);
      b ← coin_spmf;
      (((msg1, msg2), σ),s) ← exec_gpv ro.oracle (𝒜1 pk) ro.initial;
    if valid_plains msg1 msg2
    then do {
      (c, s1) ← exec_gpv ro.oracle (gpv_encrypt pk (if b then msg1 else
msg2)) s;
      (b', s2) ← exec_gpv ro.oracle (𝒜2 c σ) s1;
      return_spmf (b' = b)
      }
    else  coin_spmf
    }"

  have ind_cpa_game_eq_game0: "run_gpv ro.oracle (ind_cpa.game 𝒜) ro.initial
= game0"
  proof -
    have *: "bind_pmf pmf_key_gen (λx. return_spmf (x, ro.initial)) ⤜
      (λx. f (fst (fst x)) (snd x)) =
    spmf_of_set UNIV ⤜ (λA. bind_pmf mlwe.beta_vec (λs. bind_pmf mlwe.beta_vec
    (λe. f (A, A *v s + e) ro.initial)))" for f
    unfolding pmf_key_gen_def spmf_pmf_of_set_UNIV bind_spmf_of_pmf key_gen_new_def
Let_def
    by (simp add: bind_spmf_pmf_assoc bind_assoc_pmf bind_return_pmf)
    show ?thesis using *[of "(λpk state. exec_gpv ro.oracle (𝒜1 pk) state
⤜
          (λxa. if valid_plains (fst (fst (fst xa))) (snd (fst (fst xa)))
                then coin_spmf ⤜ (λxb. exec_gpv ro.oracle (gpv_encrypt
pk
                     (if xb then fst (fst (fst xa)) else snd (fst (fst
xa)))) (snd xa) ⤜
                     (λx. exec_gpv ro.oracle (𝒜2 (fst x) (snd (fst xa)))
(snd x) ⤜
                     (λx. return_spmf (fst x = xb))))
                else coin_spmf))"]
    unfolding 𝒜 ind_cpa.game.simps unfolding game0_def gpv_key_gen_def
    apply (simp add: split_def try_spmf_bind_spmf_lossless try_bind_assert_spmf
key_gen_def Let_def
      bind_commute_spmf[of coin_spmf] if_distrib_bind_spmf2
      try_gpv_bind_lossless try_bind_assert_gpv lift_bind_spmf comp_def
      if_distrib_exec_gpv exec_gpv_bind_lift_spmf exec_gpv_bind if_distrib_map_spmf
      del: bind_spmf_const)
    apply simp
    done
  qed
```

We define encapsulate the module-LWE instance for generating a public key t in the key generation by the function `is_pk`. The game $game_1$ depends on a function that generates a public key. Indeed, $game_0$ corresponds to $game_1$

```
is_pk.

  define is_pk :: "('a qr,'k) pk spmf" where "is_pk =  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;
    return_spmf (A, A *v s + e)}"

  define game₁ where
  "game₁ f = do {
      pk ← f;
      b ← coin_spmf;
      (((msg1, msg2), σ), s) ← exec_gpv ro.oracle (𝒜₁ pk) ro.initial;
    if valid_plains msg1 msg2
    then do {
      (c, s1) ← exec_gpv ro.oracle (gpv_encrypt pk (if b then msg1 else
msg2)) s;
      (b', s2) ← exec_gpv ro.oracle (𝒜₂ c σ) s1;
      return_spmf (b' = b)
      }
    else  coin_spmf
    }" for f :: "('a qr,'k) pk spmf"

  have game₀_eq_game₁_is_pk: "game₀ = game₁ is_pk"
  by (simp add: game₁_def game₀_def is_pk_def Let_def o_def split_def
if_distrib_map_spmf
    bind_spmf_pmf_assoc bind_assoc_pmf bind_return_pmf)
```

In contrast to the generation of the public key as a module-LWE instance as in *is_pk*, the function *rand_pk* generates a uniformly random public key. We can now use this to do a reduction step to a module-LWE advantage. $game_1$ *is_pk* corresponds to a module-LWE game with module-LWE instance. $game_1$ *rand_pk* corresponds to a module-LWE game with random instance. The difference of their probabilities can then be bounded by the module-LWE advantage in lemma *reduction1*. The reduction function used in this case is *kyber_reduction1*.

```
  define rand_pk :: "('a qr,'k) pk spmf" where
  "rand_pk =  bind_spmf (spmf_of_set UNIV) (λA. spmf_of_set UNIV ⤜ (λt.
return_spmf (A, t)))"

  have red11: "game₁ is_pk = mlwe.game (kyber_reduction1 𝒜)"
  proof -
    have "spmf_of_set UNIV ⤜ (λy. spmf_of_pmf mlwe.beta_vec ⤜ (λya.
spmf_of_pmf mlwe.beta_vec ⤜
      (λyb. exec_gpv ro.oracle (𝒜₁ (y, y *v ya + yb)) ro.initial ⤜
      (λyc. if valid_plains (fst (fst (fst yc))) (snd (fst (fst yc)))
            then coin_spmf ⤜
                 (λb. exec_gpv ro.oracle (gpv_encrypt (y, y *v ya + yb)
                      (if b then fst (fst (fst yc)) else snd (fst (fst
```

```
yc)))) (snd yc) ⤜
                (λp. exec_gpv ro.oracle (𝒜₂ (fst p) (snd (fst yc))) (snd
p) ⤜
                (λp. return_spmf (fst p = b))))
            else coin_spmf)))) =
    spmf_of_set UNIV ⤜(λA. spmf_of_pmf mlwe.beta_vec ⤜(λs. spmf_of_pmf
mlwe.beta_vec ⤜
    (λe. exec_gpv ro.oracle (𝒜₁ (A, A *v s + e)) ro.initial ⤜
    (λp. if valid_plains (fst (fst (fst p))) (snd (fst (fst p)))
         then coin_spmf ⤜
              (λx. TRY exec_gpv ro.oracle (gpv_encrypt (A, A *v s + e)
                   (if x then fst (fst (fst p)) else snd (fst (fst p))))
(snd p) ⤜
              (λpa. exec_gpv ro.oracle (𝒜₂ (fst pa) (snd (fst p))) (snd
pa) ⤜
              (λp. return_spmf (fst p = x))) ELSE coin_spmf)
         else coin_spmf))))"
    apply (subst try_spmf_bind_spmf_lossless)
      subgoal by (subst lossless_exec_gpv, auto)
      subgoal apply (subst try_spmf_bind_spmf_lossless)
        subgoal by (subst lossless_exec_gpv, auto)
        subgoal by (auto simp add: bind_commute_spmf
        [of "spmf_of_set (UNIV :: ('a qr, 'k option) vec set)"])
      done
    done
    then show ?thesis unfolding  mlwe.game_def game₁_def is_pk_def
    by (simp add: try_bind_assert_spmf bind_commute_spmf[of "coin_spmf"]
split_def
      if_distrib_bind_spmf2 try_spmf_bind_spmf_lossless
      bind_spmf_pmf_assoc bind_assoc_pmf bind_return_pmf del: bind_spmf_const)
simp
  qed

  have red12: "game₁ rand_pk = mlwe.game_random (kyber_reduction1 𝒜)"
  proof -
    have "spmf_of_set UNIV ⤜ (λy. spmf_of_set UNIV ⤜
      (λya. exec_gpv ro.oracle (𝒜₁ (y, ya)) ro.initial ⤜
      (λyb. if valid_plains (fst (fst (fst yb))) (snd (fst (fst yb)))
            then coin_spmf ⤜
                 (λb. exec_gpv ro.oracle (gpv_encrypt (y, ya)
                      (if b then fst (fst (fst yb)) else snd (fst (fst
yb)))) (snd yb) ⤜
                 (λp. exec_gpv ro.oracle (𝒜₂ (fst p) (snd (fst yb))) (snd
p) ⤜
                 (λp. return_spmf (fst p = b))))
            else coin_spmf))) =
    spmf_of_set UNIV ⤜ (λA. spmf_of_set UNIV ⤜
      (λb. exec_gpv ro.oracle (𝒜₁ (A, b)) ro.initial ⤜
      (λp. if valid_plains (fst (fst (fst p))) (snd (fst (fst p)))
```

```
          then coin_spmf ⤜
               (λx. TRY exec_gpv ro.oracle (gpv_encrypt (A, b)
                        (if x then fst (fst (fst p)) else snd (fst (fst
p)))) (snd p) ⤜
               (λpa. exec_gpv ro.oracle (𝒜₂ (fst pa) (snd (fst p))) (snd
pa) ⤜
               (λp. return_spmf (fst p = x))) ELSE coin_spmf)
          else coin_spmf)))"
    apply (subst try_spmf_bind_spmf_lossless)
      subgoal by (subst lossless_exec_gpv, auto)
      subgoal apply (subst try_spmf_bind_spmf_lossless)
        subgoal by (subst lossless_exec_gpv, auto)
        subgoal by (auto simp add: bind_commute_spmf
        [of "spmf_of_set (UNIV :: ('a qr, 'k option) vec set)"])
      done
    done
    then show ?thesis unfolding  mlwe.game_random_def game₁_def rand_pk_def
    by (simp add: try_bind_assert_spmf bind_commute_spmf[of "coin_spmf"]
split_def
      if_distrib_bind_spmf2 try_spmf_bind_spmf_lossless
      bind_spmf_pmf_assoc bind_assoc_pmf bind_return_pmf
      del: bind_spmf_const) simp
  qed

  have reduction1: "|spmf (game₁ is_pk) True - spmf (game₁ rand_pk) True|
≤
    mlwe.advantage (kyber_reduction1 𝒜)"
  unfolding mlwe.advantage_def red11 red12 by auto
```

Again, we rewrite our current game such that:

- the generation of the public key is outsourced to a sampling function *?sample*
- the encryption is outsourced to a input function
- *is_encrypt* is the appropriate input function for the encryption
- *rand_encrypt* is the appropriate input function for a uniformly random u and v'

Note that the addition of the message is not changed yet.

```
  define is_encrypt where
  "is_encrypt A t msg = do {
    r ← spmf_of_pmf mlwe.beta_vec;
    e1 ← spmf_of_pmf mlwe.beta_vec;
    e2 ← spmf_of_pmf mlwe.beta;
    let (u,v) = ((((transpose A) *v r + e1), (scalar_product t r + e2
+
```

```
      to_module (round((real_of_int q)/2)) * (plain_to_msg msg))));
    return_spmf (compress_vec du u, compress_poly dv v)
  }" for A ::"(('a qr,'k) vec,'k)vec" and t :: "('a qr,'k) vec" and msg
:: bitstring

  define rand_encrypt where
  "rand_encrypt A t msg = do {
    u ← spmf_of_set (UNIV :: ('a qr,'k) vec set);
    v ← spmf_of_set (UNIV :: 'a qr set);
    let v' = v + to_module (round((real_of_int q)/2)) * (plain_to_msg
msg);
    return_spmf (compress_vec du u, compress_poly dv v')
  }" for A ::"(('a qr,'k) vec,'k)vec" and t :: "('a qr,'k) vec" and msg
:: bitstring

  let ?sample = "λf. bind_spmf (spmf_of_set (UNIV:: (('a qr, 'k) vec,
'k) vec set))
      (λA. bind_spmf (spmf_of_set (UNIV:: ('a qr, 'k) vec set)) (f A))"

  define game₂ where
  "game₂ f A t = bind_spmf coin_spmf
    (λb. bind_spmf (exec_gpv ro.oracle (𝒜₁ (A, t)) ro.initial)
           (λ(((msg1, msg2), σ), s).
               if valid_plains msg1 msg2
               then let msg = if b then msg1 else msg2
                    in bind_spmf (f A t msg)
                       (λc. bind_spmf (exec_gpv ro.oracle (𝒜₂ c σ) s)
                              (λ(b', s1). return_spmf (b' = b)))
               else coin_spmf))"
    for f and A :: "(('a qr, 'k) vec, 'k) vec" and t :: "('a qr, 'k)
vec"
```

Then `game₁ rand_encrypt` is the same as sampling via `?sample` and `game₂ is_encrypt`.

```
  have game₁_rand_pk_eq_game₂_is_encrypt: "game₁ rand_pk = ?sample (game₂
is_encrypt)"
  unfolding game₁_def rand_pk_def game₂_def is_encrypt_def
  by (simp add: game₁_def game₂_def rand_pk_def is_encrypt_def gpv_encrypt_def
    encrypt_new_def pmf_encrypt_def Let_def o_def split_def if_distrib_map_spmf
    bind_spmf_pmf_assoc bind_assoc_pmf bind_return_pmf)
```

To make the reduction work, we need to rewrite `is_encrypt` and `rand_encrypt` such that u and v are in one vector only (since both need to be multiplied with r, thus we cannot split the module-LWE instances). This can be done using the option type to allow for one more index in the vector over the option type.

```
  define is_encrypt1 where
  "is_encrypt1 A t msg = do {
```

```
    r ← spmf_of_pmf mlwe.beta_vec;
    e1 ← spmf_of_pmf mlwe.beta_vec;
    e2 ← spmf_of_pmf mlwe.beta;
    let (e :: ('a qr, 'k option) vec) = (χ i. if i= None then e2 else
e1 $ (the i));
    let (A' :: (('a qr,'k) vec, 'k option) vec) =
      (χ i. if i = None then t else (transpose A) $ (the i));
    let c = A' *v r + e;
    let (u :: ('a qr, 'k) vec) = (χ i. (c $ (Some i)));
    let (v :: 'a qr) = (c $ None +
      to_module (round((real_of_int q)/2)) * (plain_to_msg msg));
    return_spmf (compress_vec du u, compress_poly dv v)
  }" for A :: "(('a qr, 'k) vec, 'k) vec" and t :: "('a qr,'k) vec" and
msg :: bitstring

  define rand_encrypt1 where
  "rand_encrypt1 A t msg = do {
    u' ← spmf_of_set (UNIV :: ('a qr,'k) vec set);
    v' ← spmf_of_set (UNIV :: 'a qr set);
    let (v :: 'a qr) = (v' +
      to_module (round((real_of_int q)/2)) * (plain_to_msg msg));
    return_spmf (compress_vec du u', compress_poly dv v)
  }" for A :"(('a qr,'k) vec,'k)vec" and t :: "('a qr,'k) vec" and msg
:: bitstring
```

Indeed, these functions are the same as the previously defined `is_encrypt` and `rand_encrypt`.

```
  have is_encrypt1: "is_encrypt A t msg = is_encrypt1 A t msg" for A t
msg
  unfolding is_encrypt_def is_encrypt1_def Let_def
  by (simp add: split_def plus_vec_def matrix_vector_mult_def scalar_product_def)

  have rand_encrypt1: "rand_encrypt A t msg = rand_encrypt1 A t msg" for
A t msg
    unfolding rand_encrypt_def rand_encrypt1_def Let_def by simp
```

We also need to rewrite $game_2$ to work over the option type, i.e., that A and t are put in one matix A'. Then, `is_encrypt'` is an adaption to `is_encrypt` working with A' instead of a and t separately.

```
  define game_3 where
  "game_3 f = do {
    b ← coin_spmf;
    A' ← spmf_of_set (UNIV :: (('a qr,'k) vec, 'k option) vec set);
    let A = transpose (vec_lambda (λ i. vec_nth A' (Some i)));
    let t = vec_nth A' None;
    (((msg1, msg2), σ),s) ← exec_gpv ro.oracle (A_1 (A,t)) ro.initial;
    if valid_plains msg1 msg2
    then do {
      let msg = (if b then msg1 else msg2);
```

```
      c ← f A';
      let (u :: ('a qr, 'k) vec) = vec_lambda (λ i. (vec_nth c (Some i)));
      let (v :: 'a qr) = (vec_nth c 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)
    }
  else  coin_spmf
  }" for f
```

```
  define is_encrypt' where
  "is_encrypt' A' = bind_spmf (spmf_of_pmf mlwe.beta_vec)
    (λr. bind_spmf (mlwe.beta_vec')
    (λe. return_spmf (A' *v r + e)))" for A' ::"(('a qr,'k) vec,'k option)
vec"
```

We can write the distribution of the error and the public key as one draw from a probability distribution over the option type. We need this in order to show *?sample* ($game_2$ *is_encrypt) = game*$_3$ *(is_encrypt')*. This corresponds to the module-LWE instance in the module-LWE advantage.

```
  have e_distrib: "do {
    e1 ← spmf_of_pmf mlwe.beta_vec;
    e2 ← spmf_of_pmf mlwe.beta;
    let (e' :: ('a qr, 'k option) vec) =
      (χ i. if i = None then e2 else vec_nth e1 (the i));
    return_spmf e'
  } = mlwe.beta_vec'"
  unfolding mlwe.beta_vec' mlwe.beta_vec_def replicate_spmf_def
  by (simp add: bind_commute_pmf[of "mlwe.beta"] bind_assoc_pmf bind_pmf_return_spmf
    bind_return_pmf)
```

```
  have pk_distrib: "do {
    A ← spmf_of_set (UNIV);
    t ← spmf_of_set (UNIV);
    let (A' :: (('a qr,'k) vec, 'k option) vec) =
      (χ i. if i = None then t else transpose A $ (the i));
    return_spmf A'} =
    spmf_of_set (UNIV)"
  proof (intro spmf_eqI, goal_cases)
    case (1 A')
    let ?P = "(λ (A,t). (χ i. if i = None then t else transpose A $ (the
i)))
      :: ((('a qr, 'k) vec, 'k) vec × ('a qr, 'k) vec) ⇒ (('a qr, 'k)
vec, 'k option) vec"
    define a where "a = transpose (χ i. vec_nth A' (Some i))"
    define b where "b = A' $ None"
```

```
    have "(χ i. if i = None then b else transpose a $ the i) = A'"
      unfolding a_def b_def
      by (smt (verit, ccfv_SIG) Cart_lambda_cong Option.option.collapse
        transpose_transpose vec_lambda_eta vector_component_simps(5))
    moreover have "aa = a ∧ bb = b" if "(χ i. if i = None then bb else
transpose aa $ the i) = A'"
      for aa bb using a_def b_def that by force
    ultimately have "∃! (A, t). ?P(A,t) = A'"
      unfolding Ex1_def by (auto simp add: split_def)
    then have "(∑ x ∈ UNIV. indicat_real {Some A'} (Some (?P (fst x,
snd x)))) = 1"
      by (subst indicator_def, subst ex1_sum, simp_all add: split_def
finite_Prod_UNIV)
    then have "(∑ A∈UNIV. ∑ t∈UNIV. indicat_real {Some A'} (Some (?P
(A,t)))) = 1"
    by (subst sum.cartesian_product'[symmetric]) (simp add: split_def
)
    then show ?case by (simp add: spmf_bind spmf_of_set integral_spmf_of_set)
  qed

  have game2_eq_game3_is_encrypt:
    "?sample (game2 is_encrypt) = game3 (is_encrypt')"
  proof -
    have "?sample (game2 is_encrypt) =
      (spmf_of_set UNIV ⤜
      (λA. spmf_of_set UNIV ⤜
      (λt. let  A' = χ i. if i = None then t else transpose A $ the i
in
        return_spmf A'))) ⤜
       (λA'. let A = transpose (χ i. vec_nth A' (Some i));
                t = A' $ None
        in coin_spmf ⤜
      (λb. exec_gpv ro.oracle (A1 (A, t)) ro.initial ⤜
      (λ(((msg1, msg2), σ), s). if valid_plains msg1 msg2
          then let msg = if b then msg1 else msg2
               in spmf_of_pmf mlwe.beta_vec ⤜
                  (λr. spmf_of_pmf mlwe.beta_vec ⤜
                  (λe1. spmf_of_pmf mlwe.beta ⤜
                  (λe2. let e = χ i. if i = None then e2 else e1 $ the
i;
                            c = A' *v r + e; u = compress_vec du (χ i.
c $ Some i);
                            v = compress_poly dv (c $ None + to_module
                                (round (real_of_int q / 2)) * plain_to_msg
msg)
                        in return_spmf (u, v)))) ⤜
                            (λc. exec_gpv ro.oracle (A2 c σ) s ⤜
                            (λ(b', s1). return_spmf (b' = b)))
          else coin_spmf)))"
```

```
    unfolding is_encrypt1 unfolding game₂_def is_encrypt1_def Let_def
by simp
    also have " ... = spmf_of_set UNIV ⤜
      (λA'. let A = transpose (χ i. vec_nth A' (Some i));
                t = A' $ None
        in coin_spmf ⤜
      (λb. exec_gpv ro.oracle (𝒜₁ (A, t)) ro.initial ⤜
      (λ(((msg1, msg2), σ), s). if valid_plains msg1 msg2
          then let msg = if b then msg1 else msg2
               in spmf_of_pmf mlwe.beta_vec ⤜
                  (λr.
            (spmf_of_pmf mlwe.beta_vec ⤜
            (λe1. spmf_of_pmf mlwe.beta ⤜
            (λe2. let e = χ i. if i = None then e2 else e1 $ the i
            in return_spmf e))) ⤜
                  (λe. let c = A' *v r + e; u = compress_vec du (χ i.
c $ Some i);
                           v = compress_poly dv (c $ None + to_module
                                (round (real_of_int q / 2)) * plain_to_msg
msg)
                       in return_spmf (u, v))) ⤜
                     (λc. exec_gpv ro.oracle (𝒜₂ c σ) s ⤜
                     (λ(b', s1). return_spmf (b' = b)))
          else coin_spmf)))"
      by (subst pk_distrib) (simp add: Let_def del: bind_spmf_of_pmf)
    also have " ... = spmf_of_set UNIV ⤜
      (λA'. let A = transpose (χ i. vec_nth A' (Some i));
                t = A' $ None
        in coin_spmf ⤜
      (λb. exec_gpv ro.oracle (𝒜₁ (A, t)) ro.initial ⤜
      (λ(((msg1, msg2), σ),s). if valid_plains msg1 msg2
          then let msg = if b then msg1 else msg2
               in spmf_of_pmf mlwe.beta_vec ⤜
                  (λr. mlwe.beta_vec' ⤜
                  (λe. let c = A' *v r + e; u = compress_vec du (χ i.
c $ Some i);
                           v = compress_poly dv (c $ None + to_module
                                (round (real_of_int q / 2)) * plain_to_msg
msg)
                       in return_spmf (u, v))) ⤜
                     (λc. exec_gpv ro.oracle (𝒜₂ c σ) s ⤜
                     (λ(b', s1). return_spmf (b' = b)))
          else coin_spmf)))"
      unfolding e_distrib by simp
    also have "... = game₃ is_encrypt'"
      unfolding game₃_def is_encrypt'_def
      by (simp add: bind_commute_spmf[of "coin_spmf"]  bind_spmf_pmf_assoc)
    finally show ?thesis
      by blast
```

qed

We can write the distribution of the cipher test as one draw from a probability distribution over the option type as well. Using this, we can show *?sample (game$_2$ rand_encrypt) = game$_3$ (λ_. spmf_of_set UNIV)*. This corresponds to the uniformly random instance in the module-LWE advantage.

```
  have cipher_distrib:
    "do{
      u ← spmf_of_set (UNIV:: ('a qr, 'k) vec set);
      v ← spmf_of_set (UNIV:: 'a qr set);
      return_spmf (u, v)
    } = do{
      c ← spmf_of_set (UNIV:: ('a qr, 'k option) vec set);
      let u = χ i. c $ Some i;
          v = c $ None in
      return_spmf (u,v)
    }" for msg :: bitstring
  proof (intro spmf_eqI, unfold Let_def, goal_cases)
    case (1 w)
    have "(∑x∈UNIV. ∑xa∈UNIV. indicat_real {Some w} (Some (x, xa)))
= 1"
    proof -
      have "(∑x∈UNIV. ∑xa∈UNIV. indicat_real {Some w} (Some (x, xa)))
=
            (∑x∈UNIV. indicat_real {Some w} (Some x))"
      by (subst sum.cartesian_product'[symmetric]) simp
      also have "... = 1" unfolding indicator_def
        using finite_cartesian_product[OF finite_UNIV_vec finite_qr]
        by (subst ex1_sum) simp_all
      finally show ?thesis by blast
    qed
    moreover have "(∑x∈UNIV. indicat_real {Some w} (Some (χ i. x $
Some i, x $ None))) = 1"
    proof (unfold indicator_def, subst ex1_sum, goal_cases)
      case 1
      define x where "x = (χ i. if i = None then snd w else fst w $ (the
i))"
      have "Some (χ i. x $ Some i, x $ None) ∈ {Some w}" by (simp add:
x_def)
      moreover have "(∀y. Some (χ i. y $ Some i, y $ None) ∈ {Some w}
⟶ y = x)"
      by (metis (mono_tags) Option.option.exhaust Option.option.sel Product_Type.prod.sel(1
        Product_Type.prod.sel(2) calculation singletonD vec_lambda_unique)
      ultimately show ?case unfolding Ex1_def by (intro exI) simp
    qed  simp_all
    ultimately show ?case
      by (simp add: spmf_bind integral_spmf_of_set)
  qed
```

```
  have game2_eq_game3_rand_encrypt:
    "?sample (game2 rand_encrypt) = game3 (λ_. spmf_of_set UNIV)"
  proof -
    have "?sample (game2 rand_encrypt) =
      (spmf_of_set UNIV ≫=
      (λA. spmf_of_set UNIV ≫=
      (λt. let  A' = χ i. if i = None then t else transpose A $ the i
in
        return_spmf A'))) ≫=
       (λA'. let A = transpose (χ i. vec_nth A' (Some i));
                t = A' $ None
        in coin_spmf ≫=
      (λb. exec_gpv ro.oracle (A1 (A, t)) ro.initial ≫=
      (λ(((msg1, msg2), σ),s). if valid_plains msg1 msg2
          then let msg = if b then msg1 else msg2
               in rand_encrypt1 A t msg ≫=
                           (λc. exec_gpv ro.oracle (A2 c σ) s ≫=
                           (λ(b', s1). return_spmf (b' = b)))
          else coin_spmf)))"
    unfolding rand_encrypt1 unfolding game2_def Let_def by simp
    also have " ... = spmf_of_set UNIV ≫=
      (λA'. let A = transpose (χ i. A' $ (Some i));
                t = A' $ None
        in coin_spmf ≫=
      (λb. exec_gpv ro.oracle (A1 (A, t)) ro.initial ≫=
      (λ(((msg1, msg2), σ),s). if valid_plains msg1 msg2
          then let msg = if b then msg1 else msg2
               in do{ u ← spmf_of_set (UNIV:: ('a qr, 'k) vec set);
                      v ← spmf_of_set (UNIV:: 'a qr set);
                      return_spmf (u, v)} ≫=
                  (λ(u,v). let v' = (v + to_module (round((real_of_int
q)/2)) *
                                        (plain_to_msg msg))
                       in exec_gpv ro.oracle (A2 (compress_vec du u,
                            compress_poly dv v') σ) s ≫=
                         (λ(b',s1). return_spmf (b' = b)))
          else coin_spmf)))"
      unfolding rand_encrypt1_def by (subst pk_distrib)(simp add: Let_def
del: bind_spmf_of_pmf)
    also have " ... = spmf_of_set UNIV ≫=
      (λA'. let A = transpose (χ i. vec_nth A' (Some i));
                t = A' $ None
        in coin_spmf ≫=
      (λb. exec_gpv ro.oracle (A1 (A, t)) ro.initial ≫=
      (λ(((msg1, msg2), σ),s). if valid_plains msg1 msg2
          then let msg = if b then msg1 else msg2
               in do{ c ← spmf_of_set (UNIV:: ('a qr, 'k option) vec
set);
                      let u = χ i. c $ Some i; v = c $ None in
```

```
                    return_spmf (u,v) } ≫=
                  (λ(u,v). let v' = (v + to_module (round((real_of_int
q)/2)) *
                                        (plain_to_msg msg))
                       in exec_gpv ro.oracle (𝒜2 (compress_vec du u,
                           compress_poly dv v') σ) s ≫=
                         (λ(b',s1). return_spmf (b' = b)))
          else coin_spmf)))"
    unfolding cipher_distrib by simp
  also have "... = game3 (λ_. spmf_of_set UNIV)"
    unfolding game3_def by (simp add: bind_commute_spmf[of "coin_spmf"])
  finally show ?thesis
    by blast
qed
```

Now, we establish the correspondence between the games in the module-LWE advantage and $game_3$. Here, the rewriting needed to be done manually for some parts, as the automation could not handle it (commutativity only between certain *pmf*s).

```
have game3_eq_mlwe_game': "game3 is_encrypt' = mlwe.game' (kyber_reduction2
𝒜)"
proof -
  have "mlwe.game' (kyber_reduction2 𝒜) =
    spmf_of_set UNIV ≫= (λA. spmf_of_pmf mlwe.beta_vec ≫=
    (λs. exec_gpv ro.oracle (𝒜1 (transpose (χ i. A $ Some i), A $ None))
ro.initial ≫=
    (λy. if valid_plains (fst (fst (fst y))) (snd (fst (fst y)))
         then coin_spmf ≫=
               (λya. mlwe.beta_vec' ≫=
               (λe. TRY exec_gpv ro.oracle (𝒜2 (
                    compress_vec du (χ i. (A *v s) $ Some i + e $ Some
i),
                    compress_poly dv ((A *v s) $ None + e $ None +
                    to_module (round (real_of_int q / 2)) *
                    plain_to_msg (if ya then fst (fst (fst y)) else
snd (fst (fst y)))))
                    (snd (fst y))) (snd y) ≫=
                    (λp. return_spmf (fst p = ya)) ELSE coin_spmf))
         else coin_spmf)))"
  unfolding mlwe.game'_def
  by (simp add: try_bind_assert_spmf split_def bind_commute_spmf[of
"mlwe.beta_vec'"]
    if_distrib_bind_spmf2 try_spmf_bind_spmf_lossless del: bind_spmf_const)
simp
  also have "... =
  spmf_of_set (UNIV :: (('a qr,'k) vec, 'k option) vec set) ≫=
  (λA. spmf_of_pmf mlwe.beta_vec ≫=
  (λs. exec_gpv ro.oracle (𝒜1 (transpose (χ i. A $ Some i), A $ None))
ro.initial ≫=
```

```
    (λy. if valid_plains (fst (fst (fst y))) (snd (fst (fst y)))
         then mlwe.beta_vec' ⤜
              (λe. coin_spmf ⤜
              (λb. exec_gpv ro.oracle (𝒜2 (
                   compress_vec du (χ i. (A *v s) $ Some i + e $ Some
i),
                   compress_poly dv ((A *v s) $ None + e $ None +
                   to_module (round (real_of_int q / 2)) *
                   plain_to_msg (if b then fst (fst (fst y)) else snd
(fst (fst y)))))
                   (snd (fst y))) (snd y) ⤜
              (λp. return_spmf (fst p = b))))
         else  coin_spmf))"
    apply (subst try_spmf_bind_spmf_lossless)
      subgoal by (subst lossless_exec_gpv, auto)
      subgoal by (auto simp add: bind_commute_spmf[of "mlwe.beta_vec'"])
    done
    also have "... =
    spmf_of_set (UNIV :: (('a qr,'k) vec, 'k option) vec set) ⤜
    (λA. exec_gpv ro.oracle (𝒜1 (transpose (χ i. A $ Some i), A $ None))
ro.initial ⤜
    (λp. if valid_plains (fst (fst (fst p))) (snd (fst (fst p)))
         then spmf_of_pmf mlwe.beta_vec ⤜
              (λs. mlwe.beta_vec' ⤜
              (λe. coin_spmf ⤜
              (λb. exec_gpv ro.oracle (𝒜2  (
                       compress_vec du (χ i. (A *v s) $ Some i + e $ Some
i),
                       compress_poly dv ((A *v s) $ None + e $ None +
                        to_module (round (real_of_int q / 2)) *
                        plain_to_msg (if b then fst (fst (fst p)) else
snd (fst (fst p)))))
                      (snd (fst p))) (snd p) ⤜
              (λb'. return_spmf (fst b' = b)))))
          else coin_spmf))"
    apply (subst bind_commute_spmf[of "spmf_of_pmf mlwe.beta_vec"])+
    apply (subst if_distrib_bind_spmf2)
    apply (subst bind_commute_spmf[of "spmf_of_pmf mlwe.beta_vec"])+
    by (simp add: split_def del: bind_spmf_const)
    also have "... = game3 is_encrypt'"
    unfolding game3_def is_encrypt'_def Let_def split_def
    apply (subst bind_commute_spmf[of "coin_spmf"])+
    apply (subst if_distrib_bind_spmf2)
    apply (subst bind_commute_spmf[of "coin_spmf"])+
    apply (simp add: try_bind_assert_spmf split_def bind_commute_spmf[of
"coin_spmf"]
      if_distrib_bind_spmf2 bind_spmf_pmf_assoc del: bind_spmf_const)
    by simp
    ultimately show ?thesis by force
```

```
  qed

  have game3_eq_mlwe_game_random':
    "game3 (λ_. spmf_of_set UNIV) = mlwe.game_random' (kyber_reduction2
𝒜)"
  proof -
    have *: "mlwe.game_random' (kyber_reduction2 𝒜) =
    spmf_of_set UNIV ≫=
    (λA. exec_gpv ro.oracle (𝒜1 (transpose (χ i. A $ Some i), A $ None))
ro.initial ≫=
    (λy. if valid_plains (fst (fst (fst y))) (snd (fst (fst y)))
         then spmf_of_set UNIV ≫=
              (λb. TRY coin_spmf ≫=
              (λba. exec_gpv ro.oracle (𝒜2 (compress_vec du (χ i. b $
Some i),
                 compress_poly dv (b $ None +
                 to_module (round (real_of_int q / 2)) * plain_to_msg
                 (if ba then fst (fst (fst y)) else snd (fst (fst y)))))
(snd (fst y))) (snd y) ≫=
              (λp. return_spmf (fst p = ba))) ELSE coin_spmf)
         else  coin_spmf))"
    unfolding  mlwe.game_random'_def
    using bind_commute_spmf[of "spmf_of_set (UNIV :: ('a qr, 'k option)
vec set)"]
    by (simp add: try_bind_assert_spmf split_def
      bind_commute_spmf[of "spmf_of_set (UNIV :: ('a qr, 'k option) vec
set)"]
      if_distrib_bind_spmf2 del: bind_spmf_const) simp
    also have "... =
    spmf_of_set UNIV ≫=
    (λA. exec_gpv ro.oracle  (𝒜1 (transpose (χ i. A $ Some i), A $ None))
ro.initial ≫=
    (λp. if valid_plains (fst (fst (fst p))) (snd (fst (fst p)))
         then coin_spmf ≫=
              (λba. spmf_of_set UNIV ≫=
              (λb. exec_gpv ro.oracle  (𝒜2 (compress_vec du (χ i. b $
Some i),
                        compress_poly dv (b $ None + to_module
                        (round (real_of_int q / 2)) * plain_to_msg (if
ba then fst (fst (fst p))
                        else snd (fst (fst p))))) (snd (fst p))) (snd
p) ≫=
              (λb'. return_spmf (fst b' = ba))))
         else coin_spmf))"
    apply (subst try_spmf_bind_spmf_lossless, simp)
    apply (subst try_spmf_bind_spmf_lossless)
      subgoal by (subst lossless_exec_gpv, auto)
      subgoal by (auto simp add: bind_commute_spmf
        [of "spmf_of_set (UNIV :: ('a qr, 'k option) vec set)"])
```

```
    done
    then show ?thesis unfolding game3_def *
    by (simp add: try_bind_assert_spmf split_def bind_commute_spmf[of
"coin_spmf"]
      if_distrib_bind_spmf2 del: bind_spmf_const) simp
  qed
```

This finishes the second reduction step. In this case, the reduction function was *kyber_reduction2*.

```
  have reduction2: "|spmf (game3 is_encrypt') True - spmf (game3 (λ_.
spmf_of_set UNIV)) True| ≤
    mlwe.advantage' (kyber_reduction2 𝒜)"
  unfolding game3_eq_mlwe_game' game3_eq_mlwe_game_random' mlwe.advantage'_def
  by simp
```

Now that u and v are generated uniformly at random, we need to show that adding the message will again result in a uniformly random variable. The reason is that we work over the finite space R_q. *game*$_4$ is the game where the message is no longer added, but the ciphertext is uniformly at random.

```
  define game4 where
  "game4 = do {
    b ← coin_spmf;
    A' ← spmf_of_set (UNIV :: (('a qr,'k) vec, 'k option) vec set);
    let A = transpose (χ i. vec_nth A' (Some i));
    let t = A' $ None;
    (((msg1, msg2), σ), s) ← exec_gpv ro.oracle (𝒜1 (A,t)) ro.initial;
    if valid_plains msg1 msg2
    then do {
      let msg = (if b then msg1 else msg2);
      c ← spmf_of_set UNIV;
      let u = (χ i. c $ Some i);
      let v = c $ None;
      (b', s1) ← exec_gpv ro.oracle (𝒜2 (compress_vec du u, compress_poly
dv v) σ) s;
      return_spmf (b' = b)
    }
  else  coin_spmf
  }"
```

Adding the message does not change the uniform distribution. This is needed to show that *game*$_3$ *(λ_. spmf_of_set UNIV) = game*$_4$.

```
  have indep_of_msg:
    "do {c ← spmf_of_set UNIV;
         let u = χ i. c $ Some i;
             v = c $ None + to_module (round (real_of_int q / 2)) * plain_to_msg
msg
          in return_spmf (u, v)} =
    do {c ← spmf_of_set UNIV;
```

```
        let u = χ i. c $ Some i; v = c $ None
         in return_spmf (u, v)}" for msg::bitstring
  proof (intro spmf_eqI, goal_cases)
    case (1 y)
    define msg' where "msg' = to_module (round (real_of_int q / 2)) *
plain_to_msg msg"
    have "(∑x∈UNIV. of_bool ((χ i. x $ Some i, x $ None + msg') = y))
= 1"
    proof (intro ex1_sum, goal_cases)
      case 1
      define x where "x = (χ i. if i = None then snd y - msg' else (fst
y) $ (the i))"
      have "(χ i. x $ Some i, x $ None + msg') = y" unfolding x_def by
simp
      moreover have "(∀ya. (χ i. ya $ Some i, ya $ None + msg') = y
⟶ ya = x)"
        unfolding x_def
        by (metis (mono_tags, lifting) Groups.group_add_class.add.right_cancel
          Option.option.exhaust calculation fst_conv snd_conv vec_lambda_unique
x_def)
      ultimately show ?case unfolding Ex1_def by (intro exI) simp
    qed (simp add: finite_vec)
    moreover have "... = (∑x∈UNIV. of_bool ((χ i. x $ Some i, x $ None)
= y))"
    proof (subst ex1_sum, goal_cases)
      case 1
      define x where "x = (χ i. if i = None then snd y else (fst y) $
(the i))"
      have "(χ i. x $ Some i, x $ None) = y" unfolding x_def by simp
      moreover have "(∀ya. (χ i. ya $ Some i, ya $ None) = y ⟶ ya
= x)"
        unfolding x_def
        by (metis (mono_tags, lifting)
          Option.option.exhaust calculation fst_conv snd_conv vec_lambda_unique
x_def)
      ultimately show ?case unfolding Ex1_def by (intro exI) simp
    qed (simp_all add: finite_vec)
    ultimately have "(∑x∈UNIV. of_bool ((χ i. x $ Some i, x $ None +
msg') = y)) =
    (∑x∈UNIV. of_bool ((χ i. x $ Some i, x $ None) = y))"
      by (smt (verit))
    then show ?case unfolding msg'_def[symmetric]
      by (simp add: spmf_bind integral_spmf_of_set indicator_def del:
sum_of_bool_eq)
  qed

  have game3_eq_game4: "game3 (λ_. spmf_of_set UNIV) = game4"
  proof -
    have "game3 (λ_. spmf_of_set UNIV) = coin_spmf ≫=
```

```
      (λb. spmf_of_set UNIV ⤜
      (λA'. let A = transpose (χ i. A' $ Some i); t = A' $ None
            in exec_gpv ro.oracle (𝒜1 (A, t)) ro.initial ⤜
               (λ(((msg1, msg2), σ),s).
               if valid_plains msg1 msg2
               then let msg = if b then msg1 else msg2 in
       do {c ← spmf_of_set UNIV;
         let u = χ i. c $ Some i;
             v = c $ None + to_module (round (real_of_int q / 2)) * plain_to_msg
msg
          in return_spmf (u, v)} ⤜
                  (λ (u,v). exec_gpv ro.oracle
                    (𝒜2 (compress_vec du u, compress_poly dv v) σ) s ⤜
                  (λ(b', s1). return_spmf (b' = b)))
               else coin_spmf)))"
    unfolding game3_def by simp
  also have "... = coin_spmf ⤜
      (λb. spmf_of_set UNIV ⤜
      (λA'. let A = transpose (χ i. A' $ Some i); t = A' $ None
            in exec_gpv ro.oracle (𝒜1 (A, t)) ro.initial ⤜
               (λ(((msg1, msg2), σ), s).
               if valid_plains msg1 msg2
               then let msg = if b then msg1 else msg2 in
       do {c ← spmf_of_set UNIV;
        let u = χ i. c $ Some i; v = c $ None
        in return_spmf (u, v)} ⤜
                  (λ (u,v). exec_gpv ro.oracle (𝒜2
                    (compress_vec du u, compress_poly dv v) σ) s ⤜
                  (λ(b', s1). return_spmf (b' = b)))
               else coin_spmf)))"
    unfolding indep_of_msg by simp
  also have "... = game4"
    unfolding game4_def by simp
  finally show ?thesis by blast
qed
```

Finally, we can show that $game_4$ is the same as a coin flip. Therefore, the probability to return true is exactly 0.5.

```
  have game4_eq_coin: "game4 = coin_spmf"
  proof -
    have "game4 = spmf_of_set UNIV ⤜
    (λy. exec_gpv ro.oracle (𝒜1 (transpose (χ i. y $ Some i), y $ None))
ro.initial ⤜
    (λy. if valid_plains (fst (fst (fst y))) (snd (fst (fst y)))
         then spmf_of_set UNIV ⤜
              (λya. exec_gpv ro.oracle (𝒜2 (compress_vec du (χ i. ya
$ Some i),
                    compress_poly dv (ya $ None)) (snd (fst y))) (snd
y) ⤜
```

```
            (λy. coin_spmf))
        else coin_spmf))"
    unfolding game4_def by (simp add: bind_commute_spmf[of "coin_spmf"]
split_def
      if_distrib_bind_spmf2 bind_coin_spmf_eq_const bind_spmf_coin del:
bind_spmf_const)
    also have "... = spmf_of_set UNIV ≫=
    (λy. exec_gpv ro.oracle (A1 (transpose (χ i. y $ Some i), y $ None))
ro.initial ≫=
    (λy. if valid_plains (fst (fst (fst y))) (snd (fst (fst y)))
        then coin_spmf
        else coin_spmf))"
    by (subst bind_spmf_coin) (subst lossless_exec_gpv, auto)
    also have "... = spmf_of_set UNIV ≫=
    (λy. exec_gpv ro.oracle (A1 (transpose (χ i. y $ Some i), y $ None))
ro.initial ≫=
    (λy. coin_spmf))"
      by simp
    also have "... = coin_spmf"
      by (subst bind_spmf_coin) (subst lossless_exec_gpv, auto)
    finally show ?thesis by auto
  qed

  have spmf_game4: "spmf (game4) True = 1/2" unfolding game4_eq_coin
    spmf_coin_spmf by simp
```

In the end, we assemble all the steps proven before in order to bound the advantage against IND-CPA.

```
  have "ind_cpa.advantage (ro.oracle, ro.initial) A = |spmf (game0) True
- 1/2|"
    unfolding ind_cpa.advantage.simps ind_cpa_game_eq_game0 ..
  also have "... ≤ |spmf (game1 is_pk) True - spmf (game1 rand_pk) True|
+
                  |spmf (game1 rand_pk) True - 1/2|"
    unfolding game0_eq_game1_is_pk by simp
  also have "... ≤ mlwe.advantage (kyber_reduction1 A) + |spmf (game1
rand_pk) True - 1/2|"
    by (simp add: reduction1 A[symmetric] del: A)
  also have "... ≤ mlwe.advantage (kyber_reduction1 A) +
    |spmf (?sample (game2 is_encrypt)) True - spmf (?sample (game2 is_encrypt))
True | +
    |spmf (?sample (game2 is_encrypt)) True - 1/2|"
    using game1_rand_pk_eq_game2_is_encrypt by simp
  also have "... ≤ mlwe.advantage (kyber_reduction1 A) +
    |spmf (game3 is_encrypt') True - spmf (game3 (λ_. spmf_of_set UNIV))
True | +
    |spmf (game3 (λ_. spmf_of_set UNIV)) True - 1/2|"
    using game2_eq_game3_is_encrypt game2_eq_game3_rand_encrypt by simp
  also have "... ≤ mlwe.advantage (kyber_reduction1 A) +
```

```
    mlwe.advantage' (kyber_reduction2 𝒜) +
    |spmf (game₃ (λ_. spmf_of_set UNIV)) True - 1/2|"
    using reduction2 by simp
  also have "... ≤ mlwe.advantage (kyber_reduction1 𝒜) +
    mlwe.advantage' (kyber_reduction2 𝒜) +
    |spmf (game₄) True - 1/2|"
    unfolding game₃_eq_game₄ by simp
  also have "... ≤ mlwe.advantage (kyber_reduction1 𝒜) +
    mlwe.advantage' (kyber_reduction2 𝒜)"
    unfolding spmf_game₄ by simp
  finally show ?thesis by simp

qed


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
by (rule_tac x = 0 in exI, simp)

setup_lifting type_definition_fin3329


lemma CARD_fin3329 [simp]:
"CARD (fin3329) = 3329"
unfolding type_definition.card [OF type_definition_fin3329]
by simp

lemma fin3329_nontriv [simp]:
"1 < CARD(fin3329)"
unfolding CARD_fin3329 by auto
```

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

```
lemma prime_3329: "prime (3329::nat)" by eval
```

```
instantiation fin3329 :: comm_ring_1
begin

lift_definition zero_fin3329 :: "fin3329" is "0" by simp

lift_definition one_fin3329 :: "fin3329" is "1" by simp

lift_definition plus_fin3329 :: "fin3329 ⇒ fin3329 ⇒ fin3329"
  is "(λx y. (x+y) mod 3329)"
by auto

lift_definition uminus_fin3329 :: "fin3329 ⇒ fin3329"
  is "(λx. (uminus x) mod 3329)"
by auto

lift_definition minus_fin3329 :: "fin3329 ⇒ fin3329 ⇒ fin3329"
  is "(λx y. (x-y) mod 3329)"
by auto

lift_definition times_fin3329 :: "fin3329 ⇒ fin3329 ⇒ fin3329"
  is "(λx y. (x*y) mod 3329)"
by auto


instance
proof
  fix a b c ::fin3329
  show "a * b * c = a * (b * c)"
    by (transfer, simp add: algebra_simps mod_mult_left_eq mod_mult_right_eq)
  show "a + b + c = a + (b + c)"
    by (transfer, simp add: algebra_simps mod_add_left_eq mod_add_right_eq)
  show "(a + b) * c = a * c + b * c"
    by (transfer, simp add: algebra_simps mod_add_right_eq  mod_mult_right_eq)
qed (transfer; simp add: algebra_simps mod_add_right_eq; fail)+

end


instantiation fin3329 :: finite
begin
instance
proof
  show "finite (UNIV :: fin3329 set)" unfolding type_definition.univ
[OF type_definition_fin3329]
  by auto
qed
end
```

```
instantiation fin3329 :: equal
begin
lift_definition equal_fin3329 :: "fin3329 ⇒ fin3329 ⇒ bool" is "(=)" .
instance by (intro_classes, transfer, auto)
end

instantiation fin3329 :: nontriv
begin
instance
proof
  show "1 < CARD(fin3329)" unfolding CARD_fin3329 by auto
qed
end

instantiation fin3329 :: prime_card
begin
instance
proof
  show "prime CARD(fin3329)" unfolding CARD_fin3329 using prime_3329
  by blast
qed
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
have "lead_coeff (qr_poly' TYPE(fin3329)) = 1" unfolding qr_poly'_fin3329_def

by (simp add: degree_add_eq_left degree_monom_eq)
then show "¬ int CARD(fin3329) dvd
       lead_coeff (qr_poly' TYPE(fin3329))"
  unfolding CARD_fin3329 by auto
next
have "degree (qr_poly' TYPE(fin3329)) = 256" unfolding qr_poly'_fin3329_def
by (simp add: degree_add_eq_left degree_monom_eq)
then show "0 < degree (qr_poly' TYPE(fin3329))"  by auto
qed
end

lift_definition to_int_fin3329 :: "fin3329 ⇒ int" is "λx. x" .

lift_definition of_int_fin3329 :: "int ⇒ fin3329" is "λx. (x mod 3329)"
by simp
```

```
interpretation to_int_fin3329_hom: inj_zero_hom to_int_fin3329
  by (unfold_locales; transfer, auto)

interpretation of_int_fin3329_hom: zero_hom of_int_fin3329
  by (unfold_locales, transfer, auto)

lemma to_int_fin3329_of_int_fin3329 [simp]:
"to_int_fin3329 (of_int_fin3329 x) = x mod 3329"
using of_int_fin3329.rep_eq to_int_fin3329.rep_eq by presburger

lemma of_int_fin3329_to_int_fin3329 [simp]:
"of_int_fin3329 (to_int_fin3329 x) = x"
using fin3329_rep to_int_fin3329.rep_eq to_int_fin3329_hom.injectivity

to_int_fin3329_of_int_fin3329 by force

lemma of_int_mod_ring_eq_iff [simp]:
  "(of_int_fin3329 a = of_int_fin3329 b) ⟷
   ((a mod 3329) = (b mod 3329))"
by (metis of_int_fin3329.abs_eq of_int_fin3329.rep_eq)
```

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 (unfold_locales, goal_cases)
  case 4
  then show ?case using prime_3329 prime_int_numeral_eq by blast
next
  case 5
  then show ?case using CARD_fin3329 by auto
next
  case 7
  then show ?case unfolding qr_poly'_fin3329_def by auto
qed auto

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"
unfolding mlwe.bit_set_def
  by (simp add: finite_lists_length_eq)


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

```
unfolding mlwe.beta_def
  by (meson UNIV_I finite_qr finite_subset subsetI)


lemma finite_beta_vec:
"finite (set_pmf mlwe.beta_vec)"
unfolding mlwe.beta_vec_def
  by (meson finite_UNIV_vec finite_subset top.extremum)
```

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"
unfolding Msgs_def
  by (meson finite_qr rev_finite_subset subset_UNIV)

lemma Msgs_nonempty:
"Msgs ≠ {}"
proof -
  have "to_qr (Poly [0]) ∈ Msgs" unfolding Msgs_def by auto
  then show ?thesis by auto
```

qed

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 .

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)"
using kyber_correct[OF assms(1-5) _ m01]
using assms(5)
by (metis (mono_tags, opaque_lifting) add_diff_add arith_simps(50) arith_simps(57)
assms(6)
  verit_la_generic zle_add1_eq_le zless_add1_eq)
```

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 -
  have "s = fst sk" using assms(1) by auto
  have "e = snd sk" using assms(1) by auto
  have snd_pk: "snd pk = compress_vec dt (fst pk *v s + e)"
    using pk_sk unfolding key_gen_def by auto
  have snd_pk': "snd pk = key_gen dt (fst pk) s e" using pk_sk by auto
  define ind1 where "ind1 = (λ r e1 e2. indicat_real
    {e2. decrypt (fst (encrypt (snd pk) (fst pk) r e1 e2 dt du dv m))
    (snd (encrypt (snd pk) (fst pk) r e1 e2 dt du dv m)) (fst sk) du dv
≠ m} e2)"
  define ind2 where "ind2 = (λr e1 e2. indicat_real {e2. (round (real_of_int
q / 4)
    ≤ abs_infty_poly (scalar_product e r + e2 +
         compress_error_poly dv (scalar_product (fst pk *v s + e) r +
           scalar_product (compress_error_vec dt (fst pk *v s + e)) r
+
           e2 + to_module (round (real_of_int q / 2)) * m) +
         scalar_product (compress_error_vec dt ((fst pk) *v s + e)) r
-
         scalar_product s e1 -
         scalar_product s (compress_error_vec du (r v* fst pk + e1))))}
e2)"
  have "ind1 r e1 e2 ≤ ind2 r e1 e2 "
    for r e1 e2
  proof (cases "ind1 r e1 e2 = 0")
    case True
    show ?thesis unfolding True by (auto simp add: ind2_def sum_nonneg)
  next
    case False
    then have one: "ind1 r e1 e2 = 1" unfolding ind1_def indicator_def
by simp
    define u where "u = fst (encrypt (snd pk) (fst pk) r e1 e2 dt du
dv m)"
    define v where "v = snd (encrypt (snd pk) (fst pk) r e1 e2 dt du
dv m)"
    define ct where "ct = compress_error_vec dt ((fst pk) *v s + e)"
    define cu  where "cu = compress_error_vec du ((transpose (fst pk))
*v r + e1)"
    define cv where "cv = compress_error_poly dv (scalar_product (decompress_vec
dt (snd pk)) r +
                          e2 + to_module (round((real_of_int q)/2)) *
m)"
    have cv_alt: "cv = compress_error_poly dv (scalar_product (fst pk
*v s + e) r +
          scalar_product ct r +
```

```
          e2 + to_module (round (real_of_int q / 2)) * m)"
      unfolding cv_def scalar_product_linear_left[symmetric] ct_def compress_error_vec_def
snd_pk
      by auto
    have m: "set ((coeffs ∘ of_qr) m) ⊆ {0, 1}" using ‹m∈Msgs› unfold-
ing Msgs_def by simp
    have cipher: "(u,v) = encrypt (snd pk) (fst pk) r e1 e2 dt du dv m"
      unfolding u_def v_def by simp
    have decrypt: "decrypt u v s du dv ≠ m" using one
      using assms(1) u_def v_def ind1_def by force
    have two: "ind2 r e1 e2 = 1" unfolding ind2_def
      using kyber_correct_alt[OF snd_pk' cipher ct_def cu_def cv_def decrypt
m]
      unfolding ct_def[symmetric] snd_pk[symmetric]
      unfolding transpose_matrix_vector[symmetric] cu_def[symmetric]
      using cv_alt by auto
    show ?thesis using one two by simp
  qed
  then have "(∑ e2∈UNIV. pmf mlwe.beta e2 * ind1 r e1 e2)
    ≤ (∑ e2∈UNIV. pmf mlwe.beta e2 *  ind2 r e1 e2)"
    for r e1
  by (intro sum_mono) (simp add: mult_left_mono)
  then have "(∑ e1∈UNIV. pmf mlwe.beta_vec e1 * (∑ e2∈UNIV. pmf mlwe.beta
e2 * ind1 r e1 e2))
    ≤ (∑ e1∈UNIV. pmf mlwe.beta_vec e1 * (∑ e2∈UNIV. pmf mlwe.beta e2
*
       ind2 r e1 e2))"
    for r
  by (intro sum_mono) (simp add: mult_left_mono)
  then have "(∑ r∈UNIV. pmf mlwe.beta_vec r * (∑ e1∈UNIV. pmf mlwe.beta_vec
e1 *
      (∑ e2∈UNIV. pmf mlwe.beta e2 * ind1 r e1 e2)))
    ≤ (∑ r∈UNIV. pmf mlwe.beta_vec r * (∑ e1∈UNIV. pmf mlwe.beta_vec
e1 *
      (∑ e2∈UNIV. pmf mlwe.beta e2 *  ind2 r e1 e2 )))"
  by (intro sum_mono) (simp add: mult_left_mono)
  then show ?thesis unfolding delta'_def w_kyber_def w_kyber'_def pmf_encrypt_def
Let_def
    ind1_def ind2_def ‹e=snd sk›[symmetric] ‹s=fst sk›[symmetric]
  by (simp add: bind_assoc_pmf pmf_bind integral_measure_pmf[of UNIV]
sum_distrib_left )
qed
```

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')"
using correct_key_gen[OF assms]
by (meson Lattices_Big.linorder_class.Max.coboundedI assms(2) basic_trans_rules(23)

  finite_Msgs finite_imageI image_eqI)

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')"
using correct_key_gen_max[OF assms]
by (simp add: Msgs_nonempty finite_Msgs)

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 -
  define A where "A = fst pk"
  define s where "s= fst sk"
  define e where "e = snd sk"
  have *: "(pk, sk) = ((A, key_gen dt A s e), s, e)"
    unfolding key_gen_def using A_def s_def e_def assms
    by (metis Product_Type.prod.collapse)
  show ?thesis using correct_max[OF *] by auto
qed
```

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

```
lemma expect_correct:
"expect_correct ≤ delta_kyber'"
unfolding delta_kyber'_def expect_correct_def
proof (subst integral_measure_pmf[of UNIV], goal_cases)
  case 3
  then show ?case proof (subst integral_measure_pmf[of UNIV], goal_cases)
  case 3
  have "pmf pmf_key_gen a * (MAX m∈Msgs. pmf (pmf_encrypt (fst a) m ⤜
        (λc. return_pmf (decrypt (fst c) (snd c) (fst (snd a)) du dv
≠ m))) True)
       ≤ pmf pmf_key_gen a * Max (delta' (snd a) (fst a) ` Msgs)" for
a
  proof (cases "pmf pmf_key_gen a = 0")
    case False
```

```
    obtain sk pk where ak: "a = (pk,sk)" by (meson surj_pair)
    have "(∑aa∈UNIV. (∑ab∈UNIV. pmf mlwe.beta_vec ab * sum (pmf mlwe.beta_vec)
      {ac. (((aa, compress_vec dt (aa *v ab + ac)), ab, ac) = a)}) /
      (real CARD('a qr) ^ CARD('k)) ^ CARD('k)) ≠ 0 ⟹
      ∃A s e. a = ((A, compress_vec dt (A *v s + e)), s, e)"
      by (metis (mono_tags, lifting) Cartesian_Space.vector_space_over_itself.scale_zero_ri

        Groups_Big.comm_monoid_add_class.sum.neutral div_0 mem_Collect_eq)
    then have "∃A s e. a = ((A, compress_vec dt (A *v s +e)), s,e)"
      using False unfolding pmf_key_gen_def key_gen_def
      by (auto simp add: integral_measure_pmf[of UNIV] pmf_bind pmf_expectation_bind)

    then obtain A s e where a: "a = ((A, compress_vec dt (A *v s +e)),
s,e)"
      by auto
    then have *: "snd pk = compress_vec dt (fst pk *v fst sk + snd sk)
"
      using a ak by auto
    show ?thesis using correct_max'[OF *] by (intro mult_left_mono, auto
simp add: ak)
  qed auto
  then show ?case
    by (intro sum_mono, unfold real_scaleR_def) (simp add: split_def)
  qed (auto simp add: finite_prod)
qed (auto simp add: finite_prod)
```

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

```
lemma delta_correct_kyber:
"delta_correct delta_kyber'"
using expect_correct delta_correct_def by auto


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.