```
#include <stdint.h>
#include <string.h>
#include "aes256ctr.h"

static inline uint32_t br_dec32le(const uint8_t *src)
{
        return (uint32_t)src[0]
                | ((uint32_t)src[1] << 8)
                | ((uint32_t)src[2] << 16)
                | ((uint32_t)src[3] << 24);
}

static void br_range_dec32le(uint32_t *v, size_t num, const uint8_t *src)
{
        while (num-- > 0) {
                *v ++ = br_dec32le(src);
                src += 4;
        }
}

static inline uint32_t br_swap32(uint32_t x)
{
        x = ((x & (uint32_t)0x00FF00FF) << 8)
                | ((x >> 8) & (uint32_t)0x00FF00FF);
        return (x << 16) | (x >> 16);
}

static inline void br_enc32le(uint8_t *dst, uint32_t x)
{
        dst[0] = (uint8_t)x;
        dst[1] = (uint8_t)(x >> 8);
        dst[2] = (uint8_t)(x >> 16);
        dst[3] = (uint8_t)(x >> 24);
}

static void br_range_enc32le(uint8_t *dst, const uint32_t *v, size_t num)
{
        while (num-- > 0) {
                br_enc32le(dst, *v ++);
                dst += 4;
        }
}

static void br_aes_ct64_bitslice_Sbox(uint64_t *q)
```

```
{
        /*
         * This S-box implementation is a straightforward translation of
         * the circuit described by Boyar and Peralta in "A new
         * combinational logic minimization technique with applications
         * to cryptology" (https://eprint.iacr.org/2009/191.pdf).
         *
         * Note that variables x* (input) and s* (output) are numbered
         * in "reverse" order (x0 is the high bit, x7 is the low bit).
         */

        uint64_t x0, x1, x2, x3, x4, x5, x6, x7;
        uint64_t y1, y2, y3, y4, y5, y6, y7, y8, y9;
        uint64_t y10, y11, y12, y13, y14, y15, y16, y17, y18, y19;
        uint64_t y20, y21;
        uint64_t z0, z1, z2, z3, z4, z5, z6, z7, z8, z9;
        uint64_t z10, z11, z12, z13, z14, z15, z16, z17;
        uint64_t t0, t1, t2, t3, t4, t5, t6, t7, t8, t9;
        uint64_t t10, t11, t12, t13, t14, t15, t16, t17, t18, t19;
        uint64_t t20, t21, t22, t23, t24, t25, t26, t27, t28, t29;
        uint64_t t30, t31, t32, t33, t34, t35, t36, t37, t38, t39;
        uint64_t t40, t41, t42, t43, t44, t45, t46, t47, t48, t49;
        uint64_t t50, t51, t52, t53, t54, t55, t56, t57, t58, t59;
        uint64_t t60, t61, t62, t63, t64, t65, t66, t67;
        uint64_t s0, s1, s2, s3, s4, s5, s6, s7;

        x0 = q[7];
        x1 = q[6];
        x2 = q[5];
        x3 = q[4];
        x4 = q[3];
        x5 = q[2];
        x6 = q[1];
        x7 = q[0];

        /*
         * Top linear transformation.
         */
        y14 = x3 ^ x5;
        y13 = x0 ^ x6;
        y9 = x0 ^ x3;
        y8 = x0 ^ x5;
        t0 = x1 ^ x2;
        y1 = t0 ^ x7;
        y4 = y1 ^ x3;
        y12 = y13 ^ y14;
        y2 = y1 ^ x0;
        y5 = y1 ^ x6;
        y3 = y5 ^ y8;
        t1 = x4 ^ y12;
        y15 = t1 ^ x5;
        y20 = t1 ^ x1;
        y6 = y15 ^ x7;
        y10 = y15 ^ t0;
        y11 = y20 ^ y9;
        y7 = x7 ^ y11;
        y17 = y10 ^ y11;
        y19 = y10 ^ y8;
        y16 = t0 ^ y11;
        y21 = y13 ^ y16;
        y18 = x0 ^ y16;

        /*
         * Non-linear section.
         */
        t2 = y12 & y15;
        t3 = y3 & y6;
        t4 = t3 ^ t2;
```

```
        t5 = y4 & x7;
        t6 = t5 ^ t2;
        t7 = y13 & y16;
        t8 = y5 & y1;
        t9 = t8 ^ t7;
        t10 = y2 & y7;
        t11 = t10 ^ t7;
        t12 = y9 & y11;
        t13 = y14 & y17;
        t14 = t13 ^ t12;
        t15 = y8 & y10;
        t16 = t15 ^ t12;
        t17 = t4 ^ t14;
        t18 = t6 ^ t16;
        t19 = t9 ^ t14;
        t20 = t11 ^ t16;
        t21 = t17 ^ y20;
        t22 = t18 ^ y19;
        t23 = t19 ^ y21;
        t24 = t20 ^ y18;

        t25 = t21 ^ t22;
        t26 = t21 & t23;
        t27 = t24 ^ t26;
        t28 = t25 & t27;
        t29 = t28 ^ t22;
        t30 = t23 ^ t24;
        t31 = t22 ^ t26;
        t32 = t31 & t30;
        t33 = t32 ^ t24;
        t34 = t23 ^ t33;
        t35 = t27 ^ t33;
        t36 = t24 & t35;
        t37 = t36 ^ t34;
        t38 = t27 ^ t36;
        t39 = t29 & t38;
        t40 = t25 ^ t39;

        t41 = t40 ^ t37;
        t42 = t29 ^ t33;
        t43 = t29 ^ t40;
        t44 = t33 ^ t37;
        t45 = t42 ^ t41;
        z0 = t44 & y15;
        z1 = t37 & y6;
        z2 = t33 & x7;
        z3 = t43 & y16;
        z4 = t40 & y1;
        z5 = t29 & y7;
        z6 = t42 & y11;
        z7 = t45 & y17;
        z8 = t41 & y10;
        z9 = t44 & y12;
        z10 = t37 & y3;
        z11 = t33 & y4;
        z12 = t43 & y13;
        z13 = t40 & y5;
        z14 = t29 & y2;
        z15 = t42 & y9;
        z16 = t45 & y14;
        z17 = t41 & y8;

        /*
         * Bottom linear transformation.
         */
        t46 = z15 ^ z16;
        t47 = z10 ^ z11;
        t48 = z5 ^ z13;
```

```
        t49 = z9 ^ z10;
        t50 = z2 ^ z12;
        t51 = z2 ^ z5;
        t52 = z7 ^ z8;
        t53 = z0 ^ z3;
        t54 = z6 ^ z7;
        t55 = z16 ^ z17;
        t56 = z12 ^ t48;
        t57 = t50 ^ t53;
        t58 = z4 ^ t46;
        t59 = z3 ^ t54;
        t60 = t46 ^ t57;
        t61 = z14 ^ t57;
        t62 = t52 ^ t58;
        t63 = t49 ^ t58;
        t64 = z4 ^ t59;
        t65 = t61 ^ t62;
        t66 = z1 ^ t63;
        s0 = t59 ^ t63;
        s6 = t56 ^ ~t62;
        s7 = t48 ^ ~t60;
        t67 = t64 ^ t65;
        s3 = t53 ^ t66;
        s4 = t51 ^ t66;
        s5 = t47 ^ t65;
        s1 = t64 ^ ~s3;
        s2 = t55 ^ ~t67;

        q[7] = s0;
        q[6] = s1;
        q[5] = s2;
        q[4] = s3;
        q[3] = s4;
        q[2] = s5;
        q[1] = s6;
        q[0] = s7;
}

static void br_aes_ct64_ortho(uint64_t *q)
{
#define SWAPN(cl, ch, s, x, y)    do { \
                uint64_t a, b; \
                a = (x); \
                b = (y); \
                (x) = (a & (uint64_t)cl) | ((b & (uint64_t)cl) << (s)); \
                (y) = ((a & (uint64_t)ch) >> (s)) | (b & (uint64_t)ch); \
        } while (0)

#define SWAP2(x, y)    SWAPN(0x5555555555555555, 0xAAAAAAAAAAAAAAAA,  1, x, y)
#define SWAP4(x, y)    SWAPN(0x3333333333333333, 0xCCCCCCCCCCCCCCCC,  2, x, y)
#define SWAP8(x, y)    SWAPN(0x0F0F0F0F0F0F0F0F, 0xF0F0F0F0F0F0F0F0,  4, x, y)

        SWAP2(q[0], q[1]);
        SWAP2(q[2], q[3]);
        SWAP2(q[4], q[5]);
        SWAP2(q[6], q[7]);

        SWAP4(q[0], q[2]);
        SWAP4(q[1], q[3]);
        SWAP4(q[4], q[6]);
        SWAP4(q[5], q[7]);

        SWAP8(q[0], q[4]);
        SWAP8(q[1], q[5]);
        SWAP8(q[2], q[6]);
        SWAP8(q[3], q[7]);
}
```

```
static void br_aes_ct64_interleave_in(uint64_t *q0, uint64_t *q1, const uint32_t *w)
{
        uint64_t x0, x1, x2, x3;

        x0 = w[0];
        x1 = w[1];
        x2 = w[2];
        x3 = w[3];
        x0 |= (x0 << 16);
        x1 |= (x1 << 16);
        x2 |= (x2 << 16);
        x3 |= (x3 << 16);
        x0 &= (uint64_t)0x0000FFFF0000FFFF;
        x1 &= (uint64_t)0x0000FFFF0000FFFF;
        x2 &= (uint64_t)0x0000FFFF0000FFFF;
        x3 &= (uint64_t)0x0000FFFF0000FFFF;
        x0 |= (x0 << 8);
        x1 |= (x1 << 8);
        x2 |= (x2 << 8);
        x3 |= (x3 << 8);
        x0 &= (uint64_t)0x00FF00FF00FF00FF;
        x1 &= (uint64_t)0x00FF00FF00FF00FF;
        x2 &= (uint64_t)0x00FF00FF00FF00FF;
        x3 &= (uint64_t)0x00FF00FF00FF00FF;
        *q0 = x0 | (x2 << 8);
        *q1 = x1 | (x3 << 8);
}

static void br_aes_ct64_interleave_out(uint32_t *w, uint64_t q0, uint64_t q1)
{
        uint64_t x0, x1, x2, x3;

        x0 = q0 & (uint64_t)0x00FF00FF00FF00FF;
        x1 = q1 & (uint64_t)0x00FF00FF00FF00FF;
        x2 = (q0 >> 8) & (uint64_t)0x00FF00FF00FF00FF;
        x3 = (q1 >> 8) & (uint64_t)0x00FF00FF00FF00FF;
        x0 |= (x0 >> 8);
        x1 |= (x1 >> 8);
        x2 |= (x2 >> 8);
        x3 |= (x3 >> 8);
        x0 &= (uint64_t)0x0000FFFF0000FFFF;
        x1 &= (uint64_t)0x0000FFFF0000FFFF;
        x2 &= (uint64_t)0x0000FFFF0000FFFF;
        x3 &= (uint64_t)0x0000FFFF0000FFFF;
        w[0] = (uint32_t)x0 | (uint32_t)(x0 >> 16);
        w[1] = (uint32_t)x1 | (uint32_t)(x1 >> 16);
        w[2] = (uint32_t)x2 | (uint32_t)(x2 >> 16);
        w[3] = (uint32_t)x3 | (uint32_t)(x3 >> 16);
}

static const uint8_t Rcon[] = {
        0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36
};

static uint32_t sub_word(uint32_t x)
{
        uint64_t q[8];

        memset(q, 0, sizeof q);
        q[0] = x;
        br_aes_ct64_ortho(q);
        br_aes_ct64_bitslice_Sbox(q);
        br_aes_ct64_ortho(q);
        return (uint32_t)q[0];
}

static void br_aes_ct64_keysched(uint64_t *comp_skey, const uint8_t *key)
{
```

```
        int i, j, k, nk, nkf;
        uint32_t tmp;
        uint32_t skey[60];

        int key_len = 32;

        nk = (int)(key_len >> 2);
        nkf = (int)((14 + 1) << 2);
        br_range_dec32le(skey, (key_len >> 2), key);
        tmp = skey[(key_len >> 2) - 1];
        for (i = nk, j = 0, k = 0; i < nkf; i ++) {
                if (j == 0) {
                        tmp = (tmp << 24) | (tmp >> 8);
                        tmp = sub_word(tmp) ^ Rcon[k];
                } else if (nk > 6 && j == 4) {
                        tmp = sub_word(tmp);
                }
                tmp ^= skey[i - nk];
                skey[i] = tmp;
                if (++ j == nk) {
                        j = 0;
                        k ++;
                }
        }

        for (i = 0, j = 0; i < nkf; i += 4, j += 2) {
                uint64_t q[8];

                br_aes_ct64_interleave_in(&q[0], &q[4], skey + i);
                q[1] = q[0];
                q[2] = q[0];
                q[3] = q[0];
                q[5] = q[4];
                q[6] = q[4];
                q[7] = q[4];
                br_aes_ct64_ortho(q);
                comp_skey[j + 0] =
                          (q[0] & (uint64_t)0x1111111111111111)
                        | (q[1] & (uint64_t)0x2222222222222222)
                        | (q[2] & (uint64_t)0x4444444444444444)
                        | (q[3] & (uint64_t)0x8888888888888888);
                comp_skey[j + 1] =
                          (q[4] & (uint64_t)0x1111111111111111)
                        | (q[5] & (uint64_t)0x2222222222222222)
                        | (q[6] & (uint64_t)0x4444444444444444)
                        | (q[7] & (uint64_t)0x8888888888888888);
        }
}

static void br_aes_ct64_skey_expand(uint64_t *skey, const uint64_t *comp_skey)
{
        unsigned u, v, n;

        n = (14 + 1) << 1;
        for (u = 0, v = 0; u < n; u ++, v += 4) {
                uint64_t x0, x1, x2, x3;

                x0 = x1 = x2 = x3 = comp_skey[u];
                x0 &= (uint64_t)0x1111111111111111;
                x1 &= (uint64_t)0x2222222222222222;
                x2 &= (uint64_t)0x4444444444444444;
                x3 &= (uint64_t)0x8888888888888888;
                x1 >>= 1;
                x2 >>= 2;
                x3 >>= 3;
                skey[v + 0] = (x0 << 4) - x0;
                skey[v + 1] = (x1 << 4) - x1;
                skey[v + 2] = (x2 << 4) - x2;
```

```
                skey[v + 3] = (x3 << 4) - x3;
        }
}

static inline void add_round_key(uint64_t *q, const uint64_t *sk)
{
        q[0] ^= sk[0];
        q[1] ^= sk[1];
        q[2] ^= sk[2];
        q[3] ^= sk[3];
        q[4] ^= sk[4];
        q[5] ^= sk[5];
        q[6] ^= sk[6];
        q[7] ^= sk[7];
}

static inline void shift_rows(uint64_t *q)
{
        int i;

        for (i = 0; i < 8; i ++) {
                uint64_t x;

                x = q[i];
                q[i] = (x & (uint64_t)0x000000000000FFFF)
                     | ((x & (uint64_t)0x00000000FFF00000) >> 4)
                     | ((x & (uint64_t)0x00000000000F0000) << 12)
                     | ((x & (uint64_t)0x0000FF0000000000) >> 8)
                     | ((x & (uint64_t)0x000000FF00000000) << 8)
                     | ((x & (uint64_t)0xF000000000000000) >> 12)
                     | ((x & (uint64_t)0x0FFF000000000000) << 4);
        }
}

static inline uint64_t rotr32(uint64_t x)
{
        return (x << 32) | (x >> 32);
}

static inline void mix_columns(uint64_t *q)
{
        uint64_t q0, q1, q2, q3, q4, q5, q6, q7;
        uint64_t r0, r1, r2, r3, r4, r5, r6, r7;

        q0 = q[0];
        q1 = q[1];
        q2 = q[2];
        q3 = q[3];
        q4 = q[4];
        q5 = q[5];
        q6 = q[6];
        q7 = q[7];
        r0 = (q0 >> 16) | (q0 << 48);
        r1 = (q1 >> 16) | (q1 << 48);
        r2 = (q2 >> 16) | (q2 << 48);
        r3 = (q3 >> 16) | (q3 << 48);
        r4 = (q4 >> 16) | (q4 << 48);
        r5 = (q5 >> 16) | (q5 << 48);
        r6 = (q6 >> 16) | (q6 << 48);
        r7 = (q7 >> 16) | (q7 << 48);

        q[0] = q7 ^ r7 ^ r0 ^ rotr32(q0 ^ r0);
        q[1] = q0 ^ r0 ^ q7 ^ r7 ^ r1 ^ rotr32(q1 ^ r1);
        q[2] = q1 ^ r1 ^ r2 ^ rotr32(q2 ^ r2);
        q[3] = q2 ^ r2 ^ q7 ^ r7 ^ r3 ^ rotr32(q3 ^ r3);
        q[4] = q3 ^ r3 ^ q7 ^ r7 ^ r4 ^ rotr32(q4 ^ r4);
        q[5] = q4 ^ r4 ^ r5 ^ rotr32(q5 ^ r5);
        q[6] = q5 ^ r5 ^ r6 ^ rotr32(q6 ^ r6);
```

```
        q[7] = q6 ^ r6 ^ r7 ^ rotr32(q7 ^ r7);
}

static void inc4_be(uint32_t *x)
{
  *x = br_swap32(*x)+4;
  *x = br_swap32(*x);
}

static void aes_ctr4x(uint8_t out[64], uint32_t ivw[16], uint64_t sk_exp[120])
{
  uint32_t w[16];
  uint64_t q[8];
  int i;

  memcpy(w, ivw, sizeof(w));
  for (i = 0; i < 4; i++) {
    br_aes_ct64_interleave_in(&q[i], &q[i + 4], w + (i << 2));
  }
  br_aes_ct64_ortho(q);

  add_round_key(q, sk_exp);
  for (i = 1; i < 14; i++) {
    br_aes_ct64_bitslice_Sbox(q);
    shift_rows(q);
    mix_columns(q);
    add_round_key(q, sk_exp + (i << 3));
  }
  br_aes_ct64_bitslice_Sbox(q);
  shift_rows(q);
  add_round_key(q, sk_exp + 112);

  br_aes_ct64_ortho(q);
  for (i = 0; i < 4; i ++) {
    br_aes_ct64_interleave_out(w + (i << 2), q[i], q[i + 4]);
  }
  br_range_enc32le(out, w, 16);

  /* Increase counter for next 4 blocks */
  inc4_be(ivw+3);
  inc4_be(ivw+7);
  inc4_be(ivw+11);
  inc4_be(ivw+15);
}

static void br_aes_ct64_ctr_init(uint64_t sk_exp[120], const uint8_t *key)
{
        uint64_t skey[30];

        br_aes_ct64_keysched(skey, key);
        br_aes_ct64_skey_expand(sk_exp, skey);
}

static void br_aes_ct64_ctr_run(uint64_t sk_exp[120], const uint8_t *iv, uint32_t cc, uint8
_t *data, size_t len)
{
        uint32_t ivw[16];
        size_t i;

        br_range_dec32le(ivw, 3, iv);
        memcpy(ivw +  4, ivw, 3 * sizeof(uint32_t));
        memcpy(ivw +  8, ivw, 3 * sizeof(uint32_t));
        memcpy(ivw + 12, ivw, 3 * sizeof(uint32_t));
        ivw[ 3] = br_swap32(cc);
        ivw[ 7] = br_swap32(cc + 1);
        ivw[11] = br_swap32(cc + 2);
        ivw[15] = br_swap32(cc + 3);
```

```
        while (len > 64) {
                aes_ctr4x(data, ivw, sk_exp);
                data += 64;
                len -= 64;
        }
        if(len > 0) {
                uint8_t tmp[64];
                aes_ctr4x(tmp, ivw, sk_exp);
                for(i=0;i<len;i++)
                        data[i] = tmp[i];
        }
}

void aes256ctr_prf(uint8_t *out, size_t outlen, const uint8_t key[32], const uint8_t nonce[
12])
{
  uint64_t sk_exp[120];

  br_aes_ct64_ctr_init(sk_exp, key);
  br_aes_ct64_ctr_run(sk_exp, nonce, 0, out, outlen);
}

void aes256ctr_init(aes256ctr_ctx *s, const uint8_t key[32], const uint8_t nonce[12])
{
  br_aes_ct64_ctr_init(s->sk_exp, key);

  br_range_dec32le(s->ivw, 3, nonce);
  memcpy(s->ivw +  4, s->ivw, 3 * sizeof(uint32_t));
  memcpy(s->ivw +  8, s->ivw, 3 * sizeof(uint32_t));
  memcpy(s->ivw + 12, s->ivw, 3 * sizeof(uint32_t));
  s->ivw[ 3] = br_swap32(0);
  s->ivw[ 7] = br_swap32(1);
  s->ivw[11] = br_swap32(2);
  s->ivw[15] = br_swap32(3);
}

void aes256ctr_squeezeblocks(uint8_t *out, size_t nblocks, aes256ctr_ctx *s)
{
  while (nblocks > 0) {
    aes_ctr4x(out, s->ivw, s->sk_exp);
    out += 64;
    nblocks--;
  }
}
```

```c
#include <stdint.h>
#include "params.h"
#include "cbd.h"

/*************************************************
* Name:        load32_littleendian
*
* Description: load 4 bytes into a 32-bit integer
*              in little-endian order
*
* Arguments:   - const uint8_t *x: pointer to input byte array
*
* Returns 32-bit unsigned integer loaded from x
**************************************************/
static uint32_t load32_littleendian(const uint8_t x[4])
{
  uint32_t r;
  r  = (uint32_t)x[0];
  r |= (uint32_t)x[1] << 8;
  r |= (uint32_t)x[2] << 16;
  r |= (uint32_t)x[3] << 24;
  return r;
}

/*************************************************
* Name:        load24_littleendian
*
* Description: load 3 bytes into a 32-bit integer
*              in little-endian order.
*              This function is only needed for Kyber-512
*
* Arguments:   - const uint8_t *x: pointer to input byte array
*
* Returns 32-bit unsigned integer loaded from x (most significant byte is zero)
**************************************************/
#if KYBER_ETA1 == 3
static uint32_t load24_littleendian(const uint8_t x[3])
{
  uint32_t r;
  r  = (uint32_t)x[0];
  r |= (uint32_t)x[1] << 8;
  r |= (uint32_t)x[2] << 16;
  return r;
}
#endif


/*************************************************
* Name:        cbd2
*
* Description: Given an array of uniformly random bytes, compute
*              polynomial with coefficients distributed according to
*              a centered binomial distribution with parameter eta=2
*
* Arguments:   - poly *r: pointer to output polynomial
*              - const uint8_t *buf: pointer to input byte array
**************************************************/
static void cbd2(poly *r, const uint8_t buf[2*KYBER_N/4])
{
  unsigned int i,j;
  uint32_t t,d;
  int16_t a,b;

  for(i=0;i<KYBER_N/8;i++) {
    t  = load32_littleendian(buf+4*i);
    d  = t & 0x55555555;
    d += (t>>1) & 0x55555555;
```

```
    for(j=0;j<8;j++) {
      a = (d >> (4*j+0)) & 0x3;
      b = (d >> (4*j+2)) & 0x3;
      r->coeffs[8*i+j] = a - b;
    }
  }
}

/*************************************************
* Name:        cbd3
*
* Description: Given an array of uniformly random bytes, compute
*              polynomial with coefficients distributed according to
*              a centered binomial distribution with parameter eta=3.
*              This function is only needed for Kyber-512
*
* Arguments:   - poly *r: pointer to output polynomial
*              - const uint8_t *buf: pointer to input byte array
**************************************************/
#if KYBER_ETA1 == 3
static void cbd3(poly *r, const uint8_t buf[3*KYBER_N/4])
{
  unsigned int i,j;
  uint32_t t,d;
  int16_t a,b;

  for(i=0;i<KYBER_N/4;i++) {
    t  = load24_littleendian(buf+3*i);
    d  = t & 0x00249249;
    d += (t>>1) & 0x00249249;
    d += (t>>2) & 0x00249249;

    for(j=0;j<4;j++) {
      a = (d >> (6*j+0)) & 0x7;
      b = (d >> (6*j+3)) & 0x7;
      r->coeffs[4*i+j] = a - b;
    }
  }
}
#endif

void poly_cbd_eta1(poly *r, const uint8_t buf[KYBER_ETA1*KYBER_N/4])
{
#if KYBER_ETA1 == 2
  cbd2(r, buf);
#elif KYBER_ETA1 == 3
  cbd3(r, buf);
#else
#error "This implementation requires eta1 in {2,3}"
#endif
}

void poly_cbd_eta2(poly *r, const uint8_t buf[KYBER_ETA2*KYBER_N/4])
{
#if KYBER_ETA2 == 2
  cbd2(r, buf);
#else
#error "This implementation requires eta2 = 2"
#endif
}
```

```c
#include <stdint.h>
#include "cpucycles.h"

uint64_t cpucycles_overhead(void) {
  uint64_t t0, t1, overhead = -1LL;
  unsigned int i;

  for(i=0;i<100000;i++) {
    t0 = cpucycles();
    __asm__ volatile ("");
    t1 = cpucycles();
    if(t1 - t0 < overhead)
      overhead = t1 - t0;
  }

  return overhead;
}
```

```c
/* Based on the public domain implementation in crypto_hash/keccakc512/simple/ from
 * http://bench.cr.yp.to/supercop.html by Ronny Van Keer and the public domain "TweetFips20
2"
 * implementation from https://twitter.com/tweetfips202 by Gilles Van Assche, Daniel J. Ber
nstein,
 * and Peter Schwabe */

#include <stddef.h>
#include <stdint.h>
#include "fips202.h"

#define NROUNDS 24
#define ROL(a, offset) ((a << offset) ^ (a >> (64-offset)))

/*************************************************
* Name:        load64
*
* Description: Load 8 bytes into uint64_t in little-endian order
*
* Arguments:   - const uint8_t *x: pointer to input byte array
*
* Returns the loaded 64-bit unsigned integer
**************************************************/
static uint64_t load64(const uint8_t x[8]) {
  unsigned int i;
  uint64_t r = 0;

  for(i=0;i<8;i++)
    r |= (uint64_t)x[i] << 8*i;

  return r;
}

/*************************************************
* Name:        store64
*
* Description: Store a 64-bit integer to array of 8 bytes in little-endian order
*
* Arguments:   - uint8_t *x: pointer to the output byte array (allocated)
*              - uint64_t u: input 64-bit unsigned integer
**************************************************/
static void store64(uint8_t x[8], uint64_t u) {
  unsigned int i;

  for(i=0;i<8;i++)
    x[i] = u >> 8*i;
}

/* Keccak round constants */
static const uint64_t KeccakF_RoundConstants[NROUNDS] = {
  (uint64_t)0x0000000000000001ULL,
  (uint64_t)0x0000000000008082ULL,
  (uint64_t)0x800000000000808aULL,
  (uint64_t)0x8000000080008000ULL,
  (uint64_t)0x000000000000808bULL,
  (uint64_t)0x0000000080000001ULL,
  (uint64_t)0x8000000080008081ULL,
  (uint64_t)0x8000000000008009ULL,
  (uint64_t)0x000000000000008aULL,
  (uint64_t)0x0000000000000088ULL,
  (uint64_t)0x0000000080008009ULL,
  (uint64_t)0x000000008000000aULL,
  (uint64_t)0x000000008000808bULL,
  (uint64_t)0x800000000000008bULL,
  (uint64_t)0x8000000000008089ULL,
  (uint64_t)0x8000000000008003ULL,
  (uint64_t)0x8000000000008002ULL,
  (uint64_t)0x8000000000000080ULL,
```

```c
  (uint64_t)0x000000000000800aULL,
  (uint64_t)0x800000008000000aULL,
  (uint64_t)0x8000000080008081ULL,
  (uint64_t)0x8000000000008080ULL,
  (uint64_t)0x0000000080000001ULL,
  (uint64_t)0x8000000080008008ULL
};

/*************************************************
* Name:        KeccakF1600_StatePermute
*
* Description: The Keccak F1600 Permutation
*
* Arguments:   - uint64_t *state: pointer to input/output Keccak state
**************************************************/
static void KeccakF1600_StatePermute(uint64_t state[25])
{
        int round;

        uint64_t Aba, Abe, Abi, Abo, Abu;
        uint64_t Aga, Age, Agi, Ago, Agu;
        uint64_t Aka, Ake, Aki, Ako, Aku;
        uint64_t Ama, Ame, Ami, Amo, Amu;
        uint64_t Asa, Ase, Asi, Aso, Asu;
        uint64_t BCa, BCe, BCi, BCo, BCu;
        uint64_t Da, De, Di, Do, Du;
        uint64_t Eba, Ebe, Ebi, Ebo, Ebu;
        uint64_t Ega, Ege, Egi, Ego, Egu;
        uint64_t Eka, Eke, Eki, Eko, Eku;
        uint64_t Ema, Eme, Emi, Emo, Emu;
        uint64_t Esa, Ese, Esi, Eso, Esu;

        //copyFromState(A, state)
        Aba = state[ 0];
        Abe = state[ 1];
        Abi = state[ 2];
        Abo = state[ 3];
        Abu = state[ 4];
        Aga = state[ 5];
        Age = state[ 6];
        Agi = state[ 7];
        Ago = state[ 8];
        Agu = state[ 9];
        Aka = state[10];
        Ake = state[11];
        Aki = state[12];
        Ako = state[13];
        Aku = state[14];
        Ama = state[15];
        Ame = state[16];
        Ami = state[17];
        Amo = state[18];
        Amu = state[19];
        Asa = state[20];
        Ase = state[21];
        Asi = state[22];
        Aso = state[23];
        Asu = state[24];

        for(round = 0; round < NROUNDS; round += 2) {
            //    prepareTheta
            BCa = Aba^Aga^Aka^Ama^Asa;
            BCe = Abe^Age^Ake^Ame^Ase;
            BCi = Abi^Agi^Aki^Ami^Asi;
            BCo = Abo^Ago^Ako^Amo^Aso;
            BCu = Abu^Agu^Aku^Amu^Asu;

            //thetaRhoPiChiIotaPrepareTheta(round, A, E)
```

```
        Da = BCu^ROL(BCe, 1);
        De = BCa^ROL(BCi, 1);
        Di = BCe^ROL(BCo, 1);
        Do = BCi^ROL(BCu, 1);
        Du = BCo^ROL(BCa, 1);

        Aba ^= Da;
        BCa = Aba;
        Age ^= De;
        BCe = ROL(Age, 44);
        Aki ^= Di;
        BCi = ROL(Aki, 43);
        Amo ^= Do;
        BCo = ROL(Amo, 21);
        Asu ^= Du;
        BCu = ROL(Asu, 14);
        Eba =   BCa ^((~BCe)&  BCi );
        Eba ^= (uint64_t)KeccakF_RoundConstants[round];
        Ebe =   BCe ^((~BCi)&  BCo );
        Ebi =   BCi ^((~BCo)&  BCu );
        Ebo =   BCo ^((~BCu)&  BCa );
        Ebu =   BCu ^((~BCa)&  BCe );

        Abo ^= Do;
        BCa = ROL(Abo, 28);
        Agu ^= Du;
        BCe = ROL(Agu, 20);
        Aka ^= Da;
        BCi = ROL(Aka,  3);
        Ame ^= De;
        BCo = ROL(Ame, 45);
        Asi ^= Di;
        BCu = ROL(Asi, 61);
        Ega =   BCa ^((~BCe)&  BCi );
        Ege =   BCe ^((~BCi)&  BCo );
        Egi =   BCi ^((~BCo)&  BCu );
        Ego =   BCo ^((~BCu)&  BCa );
        Egu =   BCu ^((~BCa)&  BCe );

        Abe ^= De;
        BCa = ROL(Abe,  1);
        Agi ^= Di;
        BCe = ROL(Agi,  6);
        Ako ^= Do;
        BCi = ROL(Ako, 25);
        Amu ^= Du;
        BCo = ROL(Amu,  8);
        Asa ^= Da;
        BCu = ROL(Asa, 18);
        Eka =   BCa ^((~BCe)&  BCi );
        Eke =   BCe ^((~BCi)&  BCo );
        Eki =   BCi ^((~BCo)&  BCu );
        Eko =   BCo ^((~BCu)&  BCa );
        Eku =   BCu ^((~BCa)&  BCe );

        Abu ^= Du;
        BCa = ROL(Abu, 27);
        Aga ^= Da;
        BCe = ROL(Aga, 36);
        Ake ^= De;
        BCi = ROL(Ake, 10);
        Ami ^= Di;
        BCo = ROL(Ami, 15);
        Aso ^= Do;
        BCu = ROL(Aso, 56);
        Ema =   BCa ^((~BCe)&  BCi );
        Eme =   BCe ^((~BCi)&  BCo );
        Emi =   BCi ^((~BCo)&  BCu );
```

```
        Emo =   BCo ^((~BCu)&  BCa );
        Emu =   BCu ^((~BCa)&  BCe );

        Abi ^= Di;
        BCa = ROL(Abi, 62);
        Ago ^= Do;
        BCe = ROL(Ago, 55);
        Aku ^= Du;
        BCi = ROL(Aku, 39);
        Ama ^= Da;
        BCo = ROL(Ama, 41);
        Ase ^= De;
        BCu = ROL(Ase,  2);
        Esa =   BCa ^((~BCe)&  BCi );
        Ese =   BCe ^((~BCi)&  BCo );
        Esi =   BCi ^((~BCo)&  BCu );
        Eso =   BCo ^((~BCu)&  BCa );
        Esu =   BCu ^((~BCa)&  BCe );

        //    prepareTheta
        BCa = Eba^Ega^Eka^Ema^Esa;
        BCe = Ebe^Ege^Eke^Eme^Ese;
        BCi = Ebi^Egi^Eki^Emi^Esi;
        BCo = Ebo^Ego^Eko^Emo^Eso;
        BCu = Ebu^Egu^Eku^Emu^Esu;

        //thetaRhoPiChiIotaPrepareTheta(round+1, E, A)
        Da = BCu^ROL(BCe, 1);
        De = BCa^ROL(BCi, 1);
        Di = BCe^ROL(BCo, 1);
        Do = BCi^ROL(BCu, 1);
        Du = BCo^ROL(BCa, 1);

        Eba ^= Da;
        BCa = Eba;
        Ege ^= De;
        BCe = ROL(Ege, 44);
        Eki ^= Di;
        BCi = ROL(Eki, 43);
        Emo ^= Do;
        BCo = ROL(Emo, 21);
        Esu ^= Du;
        BCu = ROL(Esu, 14);
        Aba =   BCa ^((~BCe)&  BCi );
        Aba ^= (uint64_t)KeccakF_RoundConstants[round+1];
        Abe =   BCe ^((~BCi)&  BCo );
        Abi =   BCi ^((~BCo)&  BCu );
        Abo =   BCo ^((~BCu)&  BCa );
        Abu =   BCu ^((~BCa)&  BCe );

        Ebo ^= Do;
        BCa = ROL(Ebo, 28);
        Egu ^= Du;
        BCe = ROL(Egu, 20);
        Eka ^= Da;
        BCi = ROL(Eka, 3);
        Eme ^= De;
        BCo = ROL(Eme, 45);
        Esi ^= Di;
        BCu = ROL(Esi, 61);
        Aga =   BCa ^((~BCe)&  BCi );
        Age =   BCe ^((~BCi)&  BCo );
        Agi =   BCi ^((~BCo)&  BCu );
        Ago =   BCo ^((~BCu)&  BCa );
        Agu =   BCu ^((~BCa)&  BCe );

        Ebe ^= De;
        BCa = ROL(Ebe, 1);
```

```
        Egi ^= Di;
        BCe = ROL(Egi, 6);
        Eko ^= Do;
        BCi = ROL(Eko, 25);
        Emu ^= Du;
        BCo = ROL(Emu, 8);
        Esa ^= Da;
        BCu = ROL(Esa, 18);
        Aka =   BCa ^((~BCe)&  BCi );
        Ake =   BCe ^((~BCi)&  BCo );
        Aki =   BCi ^((~BCo)&  BCu );
        Ako =   BCo ^((~BCu)&  BCa );
        Aku =   BCu ^((~BCa)&  BCe );

        Ebu ^= Du;
        BCa = ROL(Ebu, 27);
        Ega ^= Da;
        BCe = ROL(Ega, 36);
        Eke ^= De;
        BCi = ROL(Eke, 10);
        Emi ^= Di;
        BCo = ROL(Emi, 15);
        Eso ^= Do;
        BCu = ROL(Eso, 56);
        Ama =   BCa ^((~BCe)&  BCi );
        Ame =   BCe ^((~BCi)&  BCo );
        Ami =   BCi ^((~BCo)&  BCu );
        Amo =   BCo ^((~BCu)&  BCa );
        Amu =   BCu ^((~BCa)&  BCe );

        Ebi ^= Di;
        BCa = ROL(Ebi, 62);
        Ego ^= Do;
        BCe = ROL(Ego, 55);
        Eku ^= Du;
        BCi = ROL(Eku, 39);
        Ema ^= Da;
        BCo = ROL(Ema, 41);
        Ese ^= De;
        BCu = ROL(Ese, 2);
        Asa =   BCa ^((~BCe)&  BCi );
        Ase =   BCe ^((~BCi)&  BCo );
        Asi =   BCi ^((~BCo)&  BCu );
        Aso =   BCo ^((~BCu)&  BCa );
        Asu =   BCu ^((~BCa)&  BCe );
    }

    //copyToState(state, A)
    state[ 0] = Aba;
    state[ 1] = Abe;
    state[ 2] = Abi;
    state[ 3] = Abo;
    state[ 4] = Abu;
    state[ 5] = Aga;
    state[ 6] = Age;
    state[ 7] = Agi;
    state[ 8] = Ago;
    state[ 9] = Agu;
    state[10] = Aka;
    state[11] = Ake;
    state[12] = Aki;
    state[13] = Ako;
    state[14] = Aku;
    state[15] = Ama;
    state[16] = Ame;
    state[17] = Ami;
    state[18] = Amo;
    state[19] = Amu;
```

```c
        state[20] = Asa;
        state[21] = Ase;
        state[22] = Asi;
        state[23] = Aso;
        state[24] = Asu;
}

/*************************************************
* Name:        keccak_init
*
* Description: Initializes the Keccak state.
*
* Arguments:   - uint64_t *s: pointer to Keccak state
**************************************************/
static void keccak_init(uint64_t s[25])
{
  unsigned int i;
  for(i=0;i<25;i++)
    s[i] = 0;
}

/*************************************************
* Name:        keccak_absorb
*
* Description: Absorb step of Keccak; incremental.
*
* Arguments:   - uint64_t *s: pointer to Keccak state
*              - unsigned int pos: position in current block to be absorbed
*              - unsigned int r: rate in bytes (e.g., 168 for SHAKE128)
*              - const uint8_t *in: pointer to input to be absorbed into s
*              - size_t inlen: length of input in bytes
*
* Returns new position pos in current block
**************************************************/
static unsigned int keccak_absorb(uint64_t s[25],
                                  unsigned int pos,
                                  unsigned int r,
                                  const uint8_t *in,
                                  size_t inlen)
{
  unsigned int i;

  while(pos+inlen >= r) {
    for(i=pos;i<r;i++)
      s[i/8] ^= (uint64_t)*in++ << 8*(i%8);
    inlen -= r-pos;
    KeccakF1600_StatePermute(s);
    pos = 0;
  }

  for(i=pos;i<pos+inlen;i++)
    s[i/8] ^= (uint64_t)*in++ << 8*(i%8);

  return i;
}

/*************************************************
* Name:        keccak_finalize
*
* Description: Finalize absorb step.
*
* Arguments:   - uint64_t *s: pointer to Keccak state
*              - unsigned int pos: position in current block to be absorbed
*              - unsigned int r: rate in bytes (e.g., 168 for SHAKE128)
*              - uint8_t p: domain separation byte
**************************************************/
static void keccak_finalize(uint64_t s[25], unsigned int pos, unsigned int r, uint8_t p)
{
```

```
  s[pos/8] ^= (uint64_t)p << 8*(pos%8);
  s[r/8-1] ^= 1ULL << 63;
}

/*************************************************
* Name:        keccak_squeeze
*
* Description: Squeeze step of Keccak. Squeezes arbitratrily many bytes.
*              Modifies the state. Can be called multiple times to keep
*              squeezing, i.e., is incremental.
*
* Arguments:   - uint8_t *out: pointer to output
*              - size_t outlen: number of bytes to be squeezed (written to out)
*              - uint64_t *s: pointer to input/output Keccak state
*              - unsigned int pos: number of bytes in current block already squeezed
*              - unsigned int r: rate in bytes (e.g., 168 for SHAKE128)
*
* Returns new position pos in current block
**************************************************/
static unsigned int keccak_squeeze(uint8_t *out,
                                   size_t outlen,
                                   uint64_t s[25],
                                   unsigned int pos,
                                   unsigned int r)
{
  unsigned int i;

  while(outlen) {
    if(pos == r) {
      KeccakF1600_StatePermute(s);
      pos = 0;
    }
    for(i=pos;i < r && i < pos+outlen; i++)
      *out++ = s[i/8] >> 8*(i%8);
    outlen -= i-pos;
    pos = i;
  }

  return pos;
}


/*************************************************
* Name:        keccak_absorb_once
*
* Description: Absorb step of Keccak;
*              non-incremental, starts by zeroeing the state.
*
* Arguments:   - uint64_t *s: pointer to (uninitialized) output Keccak state
*              - unsigned int r: rate in bytes (e.g., 168 for SHAKE128)
*              - const uint8_t *in: pointer to input to be absorbed into s
*              - size_t inlen: length of input in bytes
*              - uint8_t p: domain-separation byte for different Keccak-derived functions
**************************************************/
static void keccak_absorb_once(uint64_t s[25],
                               unsigned int r,
                               const uint8_t *in,
                               size_t inlen,
                               uint8_t p)
{
  unsigned int i;

  for(i=0;i<25;i++)
    s[i] = 0;

  while(inlen >= r) {
    for(i=0;i<r/8;i++)
      s[i] ^= load64(in+8*i);
```

```
    in += r;
    inlen -= r;
    KeccakF1600_StatePermute(s);
  }

  for(i=0;i<inlen;i++)
    s[i/8] ^= (uint64_t)in[i] << 8*(i%8);

  s[i/8] ^= (uint64_t)p << 8*(i%8);
  s[(r-1)/8] ^= 1ULL << 63;
}

/*************************************************
* Name:        keccak_squeezeblocks
*
* Description: Squeeze step of Keccak. Squeezes full blocks of r bytes each.
*              Modifies the state. Can be called multiple times to keep
*              squeezing, i.e., is incremental. Assumes zero bytes of current
*              block have already been squeezed.
*
* Arguments:   - uint8_t *out: pointer to output blocks
*              - size_t nblocks: number of blocks to be squeezed (written to out)
*              - uint64_t *s: pointer to input/output Keccak state
*              - unsigned int r: rate in bytes (e.g., 168 for SHAKE128)
**************************************************/
static void keccak_squeezeblocks(uint8_t *out,
                                 size_t nblocks,
                                 uint64_t s[25],
                                 unsigned int r)
{
  unsigned int i;

  while(nblocks) {
    KeccakF1600_StatePermute(s);
    for(i=0;i<r/8;i++)
      store64(out+8*i, s[i]);
    out += r;
    nblocks -= 1;
  }
}

/*************************************************
* Name:        shake128_init
*
* Description: Initilizes Keccak state for use as SHAKE128 XOF
*
* Arguments:   - keccak_state *state: pointer to (uninitialized) Keccak state
**************************************************/
void shake128_init(keccak_state *state)
{
  keccak_init(state->s);
  state->pos = 0;
}

/*************************************************
* Name:        shake128_absorb
*
* Description: Absorb step of the SHAKE128 XOF; incremental.
*
* Arguments:   - keccak_state *state: pointer to (initialized) output Keccak state
*              - const uint8_t *in: pointer to input to be absorbed into s
*              - size_t inlen: length of input in bytes
**************************************************/
void shake128_absorb(keccak_state *state, const uint8_t *in, size_t inlen)
{
  state->pos = keccak_absorb(state->s, state->pos, SHAKE128_RATE, in, inlen);
}
```

```
/*************************************************
* Name:        shake128_finalize
*
* Description: Finalize absorb step of the SHAKE128 XOF.
*
* Arguments:   - keccak_state *state: pointer to Keccak state
**************************************************/
void shake128_finalize(keccak_state *state)
{
  keccak_finalize(state->s, state->pos, SHAKE128_RATE, 0x1F);
  state->pos = SHAKE128_RATE;
}

/*************************************************
* Name:        shake128_squeeze
*
* Description: Squeeze step of SHAKE128 XOF. Squeezes arbitraily many
*              bytes. Can be called multiple times to keep squeezing.
*
* Arguments:   - uint8_t *out: pointer to output blocks
*              - size_t outlen : number of bytes to be squeezed (written to output)
*              - keccak_state *s: pointer to input/output Keccak state
**************************************************/
void shake128_squeeze(uint8_t *out, size_t outlen, keccak_state *state)
{
  state->pos = keccak_squeeze(out, outlen, state->s, state->pos, SHAKE128_RATE);
}

/*************************************************
* Name:        shake128_absorb_once
*
* Description: Initialize, absorb into and finalize SHAKE128 XOF; non-incremental.
*
* Arguments:   - keccak_state *state: pointer to (uninitialized) output Keccak state
*              - const uint8_t *in: pointer to input to be absorbed into s
*              - size_t inlen: length of input in bytes
**************************************************/
void shake128_absorb_once(keccak_state *state, const uint8_t *in, size_t inlen)
{
  keccak_absorb_once(state->s, SHAKE128_RATE, in, inlen, 0x1F);
  state->pos = SHAKE128_RATE;
}

/*************************************************
* Name:        shake128_squeezeblocks
*
* Description: Squeeze step of SHAKE128 XOF. Squeezes full blocks of
*              SHAKE128_RATE bytes each. Can be called multiple times
*              to keep squeezing. Assumes new block has not yet been
*              started (state->pos = SHAKE128_RATE).
*
* Arguments:   - uint8_t *out: pointer to output blocks
*              - size_t nblocks: number of blocks to be squeezed (written to output)
*              - keccak_state *s: pointer to input/output Keccak state
**************************************************/
void shake128_squeezeblocks(uint8_t *out, size_t nblocks, keccak_state *state)
{
  keccak_squeezeblocks(out, nblocks, state->s, SHAKE128_RATE);
}

/*************************************************
* Name:        shake256_init
*
* Description: Initilizes Keccak state for use as SHAKE256 XOF
*
* Arguments:   - keccak_state *state: pointer to (uninitialized) Keccak state
**************************************************/
void shake256_init(keccak_state *state)
```

```
{
  keccak_init(state->s);
  state->pos = 0;
}

/*************************************************
* Name:        shake256_absorb
*
* Description: Absorb step of the SHAKE256 XOF; incremental.
*
* Arguments:   - keccak_state *state: pointer to (initialized) output Keccak state
*              - const uint8_t *in: pointer to input to be absorbed into s
*              - size_t inlen: length of input in bytes
**************************************************/
void shake256_absorb(keccak_state *state, const uint8_t *in, size_t inlen)
{
  state->pos = keccak_absorb(state->s, state->pos, SHAKE256_RATE, in, inlen);
}

/*************************************************
* Name:        shake256_finalize
*
* Description: Finalize absorb step of the SHAKE256 XOF.
*
* Arguments:   - keccak_state *state: pointer to Keccak state
**************************************************/
void shake256_finalize(keccak_state *state)
{
  keccak_finalize(state->s, state->pos, SHAKE256_RATE, 0x1F);
  state->pos = SHAKE256_RATE;
}

/*************************************************
* Name:        shake256_squeeze
*
* Description: Squeeze step of SHAKE256 XOF. Squeezes arbitraily many
*              bytes. Can be called multiple times to keep squeezing.
*
* Arguments:   - uint8_t *out: pointer to output blocks
*              - size_t outlen : number of bytes to be squeezed (written to output)
*              - keccak_state *s: pointer to input/output Keccak state
**************************************************/
void shake256_squeeze(uint8_t *out, size_t outlen, keccak_state *state)
{
  state->pos = keccak_squeeze(out, outlen, state->s, state->pos, SHAKE256_RATE);
}

/*************************************************
* Name:        shake256_absorb_once
*
* Description: Initialize, absorb into and finalize SHAKE256 XOF; non-incremental.
*
* Arguments:   - keccak_state *state: pointer to (uninitialized) output Keccak state
*              - const uint8_t *in: pointer to input to be absorbed into s
*              - size_t inlen: length of input in bytes
**************************************************/
void shake256_absorb_once(keccak_state *state, const uint8_t *in, size_t inlen)
{
  keccak_absorb_once(state->s, SHAKE256_RATE, in, inlen, 0x1F);
  state->pos = SHAKE256_RATE;
}

/*************************************************
* Name:        shake256_squeezeblocks
*
* Description: Squeeze step of SHAKE256 XOF. Squeezes full blocks of
*              SHAKE256_RATE bytes each. Can be called multiple times
*              to keep squeezing. Assumes next block has not yet been
```

```
*               started (state->pos = SHAKE256_RATE).
*
* Arguments:    - uint8_t *out: pointer to output blocks
*               - size_t nblocks: number of blocks to be squeezed (written to output)
*               - keccak_state *s: pointer to input/output Keccak state
**************************************************/
void shake256_squeezeblocks(uint8_t *out, size_t nblocks, keccak_state *state)
{
  keccak_squeezeblocks(out, nblocks, state->s, SHAKE256_RATE);
}

/*************************************************
* Name:        shake128
*
* Description: SHAKE128 XOF with non-incremental API
*
* Arguments:   - uint8_t *out: pointer to output
*              - size_t outlen: requested output length in bytes
*              - const uint8_t *in: pointer to input
*              - size_t inlen: length of input in bytes
**************************************************/
void shake128(uint8_t *out, size_t outlen, const uint8_t *in, size_t inlen)
{
  size_t nblocks;
  keccak_state state;

  shake128_absorb_once(&state, in, inlen);
  nblocks = outlen/SHAKE128_RATE;
  shake128_squeezeblocks(out, nblocks, &state);
  outlen -= nblocks*SHAKE128_RATE;
  out += nblocks*SHAKE128_RATE;
  shake128_squeeze(out, outlen, &state);
}

/*************************************************
* Name:        shake256
*
* Description: SHAKE256 XOF with non-incremental API
*
* Arguments:   - uint8_t *out: pointer to output
*              - size_t outlen: requested output length in bytes
*              - const uint8_t *in: pointer to input
*              - size_t inlen: length of input in bytes
**************************************************/
void shake256(uint8_t *out, size_t outlen, const uint8_t *in, size_t inlen)
{
  size_t nblocks;
  keccak_state state;

  shake256_absorb_once(&state, in, inlen);
  nblocks = outlen/SHAKE256_RATE;
  shake256_squeezeblocks(out, nblocks, &state);
  outlen -= nblocks*SHAKE256_RATE;
  out += nblocks*SHAKE256_RATE;
  shake256_squeeze(out, outlen, &state);
}

/*************************************************
* Name:        sha3_256
*
* Description: SHA3-256 with non-incremental API
*
* Arguments:   - uint8_t *h: pointer to output (32 bytes)
*              - const uint8_t *in: pointer to input
*              - size_t inlen: length of input in bytes
**************************************************/
void sha3_256(uint8_t h[32], const uint8_t *in, size_t inlen)
{
```

```
  unsigned int i;
  uint64_t s[25];

  keccak_absorb_once(s, SHA3_256_RATE, in, inlen, 0x06);
  KeccakF1600_StatePermute(s);
  for(i=0;i<4;i++)
    store64(h+8*i,s[i]);
}

/*************************************************
* Name:        sha3_512
*
* Description: SHA3-512 with non-incremental API
*
* Arguments:   - uint8_t *h: pointer to output (64 bytes)
*              - const uint8_t *in: pointer to input
*              - size_t inlen: length of input in bytes
**************************************************/
void sha3_512(uint8_t h[64], const uint8_t *in, size_t inlen)
{
  unsigned int i;
  uint64_t s[25];

  keccak_absorb_once(s, SHA3_512_RATE, in, inlen, 0x06);
  KeccakF1600_StatePermute(s);
  for(i=0;i<8;i++)
    store64(h+8*i,s[i]);
}
```

```
#include <stddef.h>
#include <stdint.h>
#include "params.h"
#include "indcpa.h"
#include "polyvec.h"
#include "poly.h"
#include "ntt.h"
#include "symmetric.h"
#include "randombytes.h"

/*************************************************
* Name:        pack_pk
*
* Description: Serialize the public key as concatenation of the
*              serialized vector of polynomials pk
*              and the public seed used to generate the matrix A.
*
* Arguments:   uint8_t *r: pointer to the output serialized public key
*              polyvec *pk: pointer to the input public-key polyvec
*              const uint8_t *seed: pointer to the input public seed
**************************************************/
static void pack_pk(uint8_t r[KYBER_INDCPA_PUBLICKEYBYTES],
                    polyvec *pk,
                    const uint8_t seed[KYBER_SYMBYTES])
{
  size_t i;
  polyvec_tobytes(r, pk);
  for(i=0;i<KYBER_SYMBYTES;i++)
    r[i+KYBER_POLYVECBYTES] = seed[i];
}

/*************************************************
* Name:        unpack_pk
*
* Description: De-serialize public key from a byte array;
*              approximate inverse of pack_pk
*
* Arguments:   - polyvec *pk: pointer to output public-key polynomial vector
*              - uint8_t *seed: pointer to output seed to generate matrix A
*              - const uint8_t *packedpk: pointer to input serialized public key
**************************************************/
static void unpack_pk(polyvec *pk,
                      uint8_t seed[KYBER_SYMBYTES],
                      const uint8_t packedpk[KYBER_INDCPA_PUBLICKEYBYTES])
{
  size_t i;
  polyvec_frombytes(pk, packedpk);
  for(i=0;i<KYBER_SYMBYTES;i++)
    seed[i] = packedpk[i+KYBER_POLYVECBYTES];
}

/*************************************************
* Name:        pack_sk
*
* Description: Serialize the secret key
*
* Arguments:   - uint8_t *r: pointer to output serialized secret key
*              - polyvec *sk: pointer to input vector of polynomials (secret key)
**************************************************/
static void pack_sk(uint8_t r[KYBER_INDCPA_SECRETKEYBYTES], polyvec *sk)
{
  polyvec_tobytes(r, sk);
}

/*************************************************
* Name:        unpack_sk
*
* Description: De-serialize the secret key; inverse of pack_sk
```

```
*
* Arguments:   - polyvec *sk: pointer to output vector of polynomials (secret key)
*              - const uint8_t *packedsk: pointer to input serialized secret key
**************************************************/
static void unpack_sk(polyvec *sk, const uint8_t packedsk[KYBER_INDCPA_SECRETKEYBYTES])
{
  polyvec_frombytes(sk, packedsk);
}

/*************************************************
* Name:        pack_ciphertext
*
* Description: Serialize the ciphertext as concatenation of the
*              compressed and serialized vector of polynomials b
*              and the compressed and serialized polynomial v
*
* Arguments:   uint8_t *r: pointer to the output serialized ciphertext
*              poly *pk: pointer to the input vector of polynomials b
*              poly *v: pointer to the input polynomial v
**************************************************/
static void pack_ciphertext(uint8_t r[KYBER_INDCPA_BYTES], polyvec *b, poly *v)
{
  polyvec_compress(r, b);
  poly_compress(r+KYBER_POLYVECCOMPRESSEDBYTES, v);
}

/*************************************************
* Name:        unpack_ciphertext
*
* Description: De-serialize and decompress ciphertext from a byte array;
*              approximate inverse of pack_ciphertext
*
* Arguments:   - polyvec *b: pointer to the output vector of polynomials b
*              - poly *v: pointer to the output polynomial v
*              - const uint8_t *c: pointer to the input serialized ciphertext
**************************************************/
static void unpack_ciphertext(polyvec *b, poly *v, const uint8_t c[KYBER_INDCPA_BYTES])
{
  polyvec_decompress(b, c);
  poly_decompress(v, c+KYBER_POLYVECCOMPRESSEDBYTES);
}

/*************************************************
* Name:        rej_uniform
*
* Description: Run rejection sampling on uniform random bytes to generate
*              uniform random integers mod q
*
* Arguments:   - int16_t *r: pointer to output buffer
*              - unsigned int len: requested number of 16-bit integers (uniform mod q)
*              - const uint8_t *buf: pointer to input buffer (assumed to be uniformly rando
m bytes)
*              - unsigned int buflen: length of input buffer in bytes
*
* Returns number of sampled 16-bit integers (at most len)
**************************************************/
static unsigned int rej_uniform(int16_t *r,
                                unsigned int len,
                                const uint8_t *buf,
                                unsigned int buflen)
{
  unsigned int ctr, pos;
  uint16_t val0, val1;

  ctr = pos = 0;
  while(ctr < len && pos + 3 <= buflen) {
    val0 = ((buf[pos+0] >> 0) | ((uint16_t)buf[pos+1] << 8)) & 0xFFF;
    val1 = ((buf[pos+1] >> 4) | ((uint16_t)buf[pos+2] << 4)) & 0xFFF;
```

```
    pos += 3;

    if(val0 < KYBER_Q)
      r[ctr++] = val0;
    if(ctr < len && val1 < KYBER_Q)
      r[ctr++] = val1;
  }

  return ctr;
}

#define gen_a(A,B)  gen_matrix(A,B,0)
#define gen_at(A,B) gen_matrix(A,B,1)

/*************************************************
* Name:        gen_matrix
*
* Description: Deterministically generate matrix A (or the transpose of A)
*              from a seed. Entries of the matrix are polynomials that look
*              uniformly random. Performs rejection sampling on output of
*              a XOF
*
* Arguments:   - polyvec *a: pointer to ouptput matrix A
*              - const uint8_t *seed: pointer to input seed
*              - int transposed: boolean deciding whether A or A^T is generated
**************************************************/
#define GEN_MATRIX_NBLOCKS ((12*KYBER_N/8*(1 << 12)/KYBER_Q + XOF_BLOCKBYTES)/XOF_BLOCKBYTE
S)
// Not static for benchmarking
void gen_matrix(polyvec *a, const uint8_t seed[KYBER_SYMBYTES], int transposed)
{
  unsigned int ctr, i, j, k;
  unsigned int buflen, off;
  uint8_t buf[GEN_MATRIX_NBLOCKS*XOF_BLOCKBYTES+2];
  xof_state state;

  for(i=0;i<KYBER_K;i++) {
    for(j=0;j<KYBER_K;j++) {
      if(transposed)
        xof_absorb(&state, seed, i, j);
      else
        xof_absorb(&state, seed, j, i);

      xof_squeezeblocks(buf, GEN_MATRIX_NBLOCKS, &state);
      buflen = GEN_MATRIX_NBLOCKS*XOF_BLOCKBYTES;
      ctr = rej_uniform(a[i].vec[j].coeffs, KYBER_N, buf, buflen);

      while(ctr < KYBER_N) {
        off = buflen % 3;
        for(k = 0; k < off; k++)
          buf[k] = buf[buflen - off + k];
        xof_squeezeblocks(buf + off, 1, &state);
        buflen = off + XOF_BLOCKBYTES;
        ctr += rej_uniform(a[i].vec[j].coeffs + ctr, KYBER_N - ctr, buf, buflen);
      }
    }
  }
}

/*************************************************
* Name:        indcpa_keypair
*
* Description: Generates public and private key for the CPA-secure
*              public-key encryption scheme underlying Kyber
*
* Arguments:   - uint8_t *pk: pointer to output public key
*                             (of length KYBER_INDCPA_PUBLICKEYBYTES bytes)
*              - uint8_t *sk: pointer to output private key
```

```
                              (of length KYBER_INDCPA_SECRETKEYBYTES bytes)
**************************************************/
void indcpa_keypair(uint8_t pk[KYBER_INDCPA_PUBLICKEYBYTES],
                    uint8_t sk[KYBER_INDCPA_SECRETKEYBYTES])
{
  unsigned int i;
  uint8_t buf[2*KYBER_SYMBYTES];
  const uint8_t *publicseed = buf;
  const uint8_t *noiseseed = buf+KYBER_SYMBYTES;
  uint8_t nonce = 0;
  polyvec a[KYBER_K], e, pkpv, skpv;

  randombytes(buf, KYBER_SYMBYTES);
  hash_g(buf, buf, KYBER_SYMBYTES);

  gen_a(a, publicseed);

  for(i=0;i<KYBER_K;i++)
    poly_getnoise_eta1(&skpv.vec[i], noiseseed, nonce++);
  for(i=0;i<KYBER_K;i++)
    poly_getnoise_eta1(&e.vec[i], noiseseed, nonce++);

  polyvec_ntt(&skpv);
  polyvec_ntt(&e);

  // matrix-vector multiplication
  for(i=0;i<KYBER_K;i++) {
    polyvec_basemul_acc_montgomery(&pkpv.vec[i], &a[i], &skpv);
    poly_tomont(&pkpv.vec[i]);
  }

  polyvec_add(&pkpv, &pkpv, &e);
  polyvec_reduce(&pkpv);

  pack_sk(sk, &skpv);
  pack_pk(pk, &pkpv, publicseed);
}

/*************************************************
* Name:        indcpa_enc
*
* Description: Encryption function of the CPA-secure
*              public-key encryption scheme underlying Kyber.
*
* Arguments:   - uint8_t *c: pointer to output ciphertext
*                            (of length KYBER_INDCPA_BYTES bytes)
*              - const uint8_t *m: pointer to input message
*                                  (of length KYBER_INDCPA_MSGBYTES bytes)
*              - const uint8_t *pk: pointer to input public key
*                                   (of length KYBER_INDCPA_PUBLICKEYBYTES)
*              - const uint8_t *coins: pointer to input random coins used as seed
*                                      (of length KYBER_SYMBYTES) to deterministically
*                                      generate all randomness
**************************************************/
void indcpa_enc(uint8_t c[KYBER_INDCPA_BYTES],
                const uint8_t m[KYBER_INDCPA_MSGBYTES],
                const uint8_t pk[KYBER_INDCPA_PUBLICKEYBYTES],
                const uint8_t coins[KYBER_SYMBYTES])
{
  unsigned int i;
  uint8_t seed[KYBER_SYMBYTES];
  uint8_t nonce = 0;
  polyvec sp, pkpv, ep, at[KYBER_K], b;
  poly v, k, epp;

  unpack_pk(&pkpv, seed, pk);
  poly_frommsg(&k, m);
  gen_at(at, seed);
```

```c
  for(i=0;i<KYBER_K;i++)
    poly_getnoise_eta1(sp.vec+i, coins, nonce++);
  for(i=0;i<KYBER_K;i++)
    poly_getnoise_eta2(ep.vec+i, coins, nonce++);
  poly_getnoise_eta2(&epp, coins, nonce++);

  polyvec_ntt(&sp);

  // matrix-vector multiplication
  for(i=0;i<KYBER_K;i++)
    polyvec_basemul_acc_montgomery(&b.vec[i], &at[i], &sp);

  polyvec_basemul_acc_montgomery(&v, &pkpv, &sp);

  polyvec_invntt_tomont(&b);
  poly_invntt_tomont(&v);

  polyvec_add(&b, &b, &ep);
  poly_add(&v, &v, &epp);
  poly_add(&v, &v, &k);
  polyvec_reduce(&b);
  poly_reduce(&v);

  pack_ciphertext(c, &b, &v);
}

/*************************************************
* Name:        indcpa_dec
*
* Description: Decryption function of the CPA-secure
*              public-key encryption scheme underlying Kyber.
*
* Arguments:   - uint8_t *m: pointer to output decrypted message
*                            (of length KYBER_INDCPA_MSGBYTES)
*              - const uint8_t *c: pointer to input ciphertext
*                                  (of length KYBER_INDCPA_BYTES)
*              - const uint8_t *sk: pointer to input secret key
*                                   (of length KYBER_INDCPA_SECRETKEYBYTES)
**************************************************/
void indcpa_dec(uint8_t m[KYBER_INDCPA_MSGBYTES],
                const uint8_t c[KYBER_INDCPA_BYTES],
                const uint8_t sk[KYBER_INDCPA_SECRETKEYBYTES])
{
  polyvec b, skpv;
  poly v, mp;

  unpack_ciphertext(&b, &v, c);
  unpack_sk(&skpv, sk);

  polyvec_ntt(&b);
  polyvec_basemul_acc_montgomery(&mp, &skpv, &b);
  poly_invntt_tomont(&mp);

  poly_sub(&mp, &v, &mp);
  poly_reduce(&mp);

  poly_tomsg(m, &mp);
}
```

```c
#include <stddef.h>
#include <stdint.h>
#include "params.h"
#include "kem.h"
#include "indcpa.h"
#include "verify.h"
#include "symmetric.h"
#include "randombytes.h"

/*************************************************
* Name:        crypto_kem_keypair
*
* Description: Generates public and private key
*              for CCA-secure Kyber key encapsulation mechanism
*
* Arguments:   - uint8_t *pk: pointer to output public key
*                (an already allocated array of KYBER_PUBLICKEYBYTES bytes)
*              - uint8_t *sk: pointer to output private key
*                (an already allocated array of KYBER_SECRETKEYBYTES bytes)
*
* Returns 0 (success)
**************************************************/
int crypto_kem_keypair(uint8_t *pk,
                       uint8_t *sk)
{
  size_t i;
  indcpa_keypair(pk, sk);
  for(i=0;i<KYBER_INDCPA_PUBLICKEYBYTES;i++)
    sk[i+KYBER_INDCPA_SECRETKEYBYTES] = pk[i];
  hash_h(sk+KYBER_SECRETKEYBYTES-2*KYBER_SYMBYTES, pk, KYBER_PUBLICKEYBYTES);
  /* Value z for pseudo-random output on reject */
  randombytes(sk+KYBER_SECRETKEYBYTES-KYBER_SYMBYTES, KYBER_SYMBYTES);
  return 0;
}

/*************************************************
* Name:        crypto_kem_enc
*
* Description: Generates cipher text and shared
*              secret for given public key
*
* Arguments:   - uint8_t *ct: pointer to output cipher text
*                (an already allocated array of KYBER_CIPHERTEXTBYTES bytes)
*              - uint8_t *ss: pointer to output shared secret
*                (an already allocated array of KYBER_SSBYTES bytes)
*              - const uint8_t *pk: pointer to input public key
*                (an already allocated array of KYBER_PUBLICKEYBYTES bytes)
*
* Returns 0 (success)
**************************************************/
int crypto_kem_enc(uint8_t *ct,
                   uint8_t *ss,
                   const uint8_t *pk)
{
  uint8_t buf[2*KYBER_SYMBYTES];
  /* Will contain key, coins */
  uint8_t kr[2*KYBER_SYMBYTES];

  randombytes(buf, KYBER_SYMBYTES);
  /* Don't release system RNG output */
  hash_h(buf, buf, KYBER_SYMBYTES);

  /* Multitarget countermeasure for coins + contributory KEM */
  hash_h(buf+KYBER_SYMBYTES, pk, KYBER_PUBLICKEYBYTES);
  hash_g(kr, buf, 2*KYBER_SYMBYTES);

  /* coins are in kr+KYBER_SYMBYTES */
  indcpa_enc(ct, buf, pk, kr+KYBER_SYMBYTES);
```

```c
  /* overwrite coins in kr with H(c) */
  hash_h(kr+KYBER_SYMBYTES, ct, KYBER_CIPHERTEXTBYTES);
  /* hash concatenation of pre-k and H(c) to k */
  kdf(ss, kr, 2*KYBER_SYMBYTES);
  return 0;
}

/*************************************************
* Name:        crypto_kem_dec
*
* Description: Generates shared secret for given
*              cipher text and private key
*
* Arguments:   - uint8_t *ss: pointer to output shared secret
*                (an already allocated array of KYBER_SSBYTES bytes)
*              - const uint8_t *ct: pointer to input cipher text
*                (an already allocated array of KYBER_CIPHERTEXTBYTES bytes)
*              - const uint8_t *sk: pointer to input private key
*                (an already allocated array of KYBER_SECRETKEYBYTES bytes)
*
* Returns 0.
*
* On failure, ss will contain a pseudo-random value.
**************************************************/
int crypto_kem_dec(uint8_t *ss,
                   const uint8_t *ct,
                   const uint8_t *sk)
{
  size_t i;
  int fail;
  uint8_t buf[2*KYBER_SYMBYTES];
  /* Will contain key, coins */
  uint8_t kr[2*KYBER_SYMBYTES];
  uint8_t cmp[KYBER_CIPHERTEXTBYTES];
  const uint8_t *pk = sk+KYBER_INDCPA_SECRETKEYBYTES;

  indcpa_dec(buf, ct, sk);

  /* Multitarget countermeasure for coins + contributory KEM */
  for(i=0;i<KYBER_SYMBYTES;i++)
    buf[KYBER_SYMBYTES+i] = sk[KYBER_SECRETKEYBYTES-2*KYBER_SYMBYTES+i];
  hash_g(kr, buf, 2*KYBER_SYMBYTES);

  /* coins are in kr+KYBER_SYMBYTES */
  indcpa_enc(cmp, buf, pk, kr+KYBER_SYMBYTES);

  fail = verify(ct, cmp, KYBER_CIPHERTEXTBYTES);

  /* overwrite coins in kr with H(c) */
  hash_h(kr+KYBER_SYMBYTES, ct, KYBER_CIPHERTEXTBYTES);

  /* Overwrite pre-k with z on re-encryption failure */
  cmov(kr, sk+KYBER_SECRETKEYBYTES-KYBER_SYMBYTES, KYBER_SYMBYTES, fail);

  /* hash concatenation of pre-k and H(c) to k */
  kdf(ss, kr, 2*KYBER_SYMBYTES);
  return 0;
}
```

```
#include <stdint.h>
#include "kex.h"
#include "kem.h"
#include "symmetric.h"

void kex_uake_initA(uint8_t *send, uint8_t *tk, uint8_t *sk, const uint8_t *pkb)
{
  crypto_kem_keypair(send, sk);
  crypto_kem_enc(send+CRYPTO_PUBLICKEYBYTES, tk, pkb);
}

void kex_uake_sharedB(uint8_t *send, uint8_t *k, const uint8_t *recv, const uint8_t *skb)
{
  uint8_t buf[2*CRYPTO_BYTES];
  crypto_kem_enc(send, buf, recv);
  crypto_kem_dec(buf+CRYPTO_BYTES, recv+CRYPTO_PUBLICKEYBYTES, skb);
  kdf(k, buf, 2*CRYPTO_BYTES);
}

void kex_uake_sharedA(uint8_t *k, const uint8_t *recv, const uint8_t *tk, const uint8_t *sk
)
{
  unsigned int i;
  uint8_t buf[2*CRYPTO_BYTES];
  crypto_kem_dec(buf, recv, sk);
  for(i=0;i<CRYPTO_BYTES;i++)
    buf[i+CRYPTO_BYTES] = tk[i];
  kdf(k, buf, 2*CRYPTO_BYTES);
}

void kex_ake_initA(uint8_t *send, uint8_t *tk, uint8_t *sk, const uint8_t *pkb)
{
  crypto_kem_keypair(send, sk);
  crypto_kem_enc(send+CRYPTO_PUBLICKEYBYTES, tk, pkb);
}

void kex_ake_sharedB(uint8_t *send, uint8_t *k, const uint8_t* recv, const uint8_t *skb, co
nst uint8_t *pka)
{
  uint8_t buf[3*CRYPTO_BYTES];
  crypto_kem_enc(send, buf, recv);
  crypto_kem_enc(send+CRYPTO_CIPHERTEXTBYTES, buf+CRYPTO_BYTES, pka);
  crypto_kem_dec(buf+2*CRYPTO_BYTES, recv+CRYPTO_PUBLICKEYBYTES, skb);
  kdf(k, buf, 3*CRYPTO_BYTES);
}

void kex_ake_sharedA(uint8_t *k, const uint8_t *recv, const uint8_t *tk, const uint8_t *sk,
 const uint8_t *ska)
{
  unsigned int i;
  uint8_t buf[3*CRYPTO_BYTES];
  crypto_kem_dec(buf, recv, sk);
  crypto_kem_dec(buf+CRYPTO_BYTES, recv+CRYPTO_CIPHERTEXTBYTES, ska);
  for(i=0;i<CRYPTO_BYTES;i++)
    buf[i+2*CRYPTO_BYTES] = tk[i];
  kdf(k, buf, 3*CRYPTO_BYTES);
}
```

```c
#include <stdint.h>
#include "params.h"
#include "ntt.h"
#include "reduce.h"

/* Code to generate zetas and zetas_inv used in the number-theoretic transform:

#define KYBER_ROOT_OF_UNITY 17

static const uint8_t tree[128] = {
  0, 64, 32, 96, 16, 80, 48, 112, 8, 72, 40, 104, 24, 88, 56, 120,
  4, 68, 36, 100, 20, 84, 52, 116, 12, 76, 44, 108, 28, 92, 60, 124,
  2, 66, 34, 98, 18, 82, 50, 114, 10, 74, 42, 106, 26, 90, 58, 122,
  6, 70, 38, 102, 22, 86, 54, 118, 14, 78, 46, 110, 30, 94, 62, 126,
  1, 65, 33, 97, 17, 81, 49, 113, 9, 73, 41, 105, 25, 89, 57, 121,
  5, 69, 37, 101, 21, 85, 53, 117, 13, 77, 45, 109, 29, 93, 61, 125,
  3, 67, 35, 99, 19, 83, 51, 115, 11, 75, 43, 107, 27, 91, 59, 123,
  7, 71, 39, 103, 23, 87, 55, 119, 15, 79, 47, 111, 31, 95, 63, 127
};

void init_ntt() {
  unsigned int i;
  int16_t tmp[128];

  tmp[0] = MONT;
  for(i=1;i<128;i++)
    tmp[i] = fqmul(tmp[i-1],MONT*KYBER_ROOT_OF_UNITY % KYBER_Q);

  for(i=0;i<128;i++) {
    zetas[i] = tmp[tree[i]];
    if(zetas[i] > KYBER_Q/2)
      zetas[i] -= KYBER_Q;
    if(zetas[i] < -KYBER_Q/2)
      zetas[i] += KYBER_Q;
  }
}
*/

const int16_t zetas[128] = {
  -1044,  -758,  -359, -1517,  1493,  1422,   287,   202,
   -171,   622,  1577,   182,   962, -1202, -1474,  1468,
    573, -1325,   264,   383,  -829,  1458, -1602,  -130,
   -681,  1017,   732,   608, -1542,   411,  -205, -1571,
   1223,   652,  -552,  1015, -1293,  1491,  -282, -1544,
    516,    -8,  -320,  -666, -1618, -1162,   126,  1469,
   -853,   -90,  -271,   830,   107, -1421,  -247,  -951,
   -398,   961, -1508,  -725,   448, -1065,   677, -1275,
  -1103,   430,   555,   843, -1251,   871,  1550,   105,
    422,   587,   177,  -235,  -291,  -460,  1574,  1653,
   -246,   778,  1159,  -147,  -777,  1483,  -602,  1119,
  -1590,   644,  -872,   349,   418,   329,  -156,   -75,
    817,  1097,   603,   610,  1322, -1285, -1465,   384,
  -1215,  -136,  1218, -1335,  -874,   220, -1187, -1659,
  -1185, -1530, -1278,   794, -1510,  -854,  -870,   478,
   -108,  -308,   996,   991,   958, -1460,  1522,  1628
};

/*************************************************
* Name:        fqmul
*
* Description: Multiplication followed by Montgomery reduction
*
* Arguments:   - int16_t a: first factor
*              - int16_t b: second factor
*
* Returns 16-bit integer congruent to a*b*R^{-1} mod q
**************************************************/
static int16_t fqmul(int16_t a, int16_t b) {
```

```c
  return montgomery_reduce((int32_t)a*b);
}

/*************************************************
* Name:        ntt
*
* Description: Inplace number-theoretic transform (NTT) in Rq.
*              input is in standard order, output is in bitreversed order
*
* Arguments:   - int16_t r[256]: pointer to input/output vector of elements of Zq
**************************************************/
void ntt(int16_t r[256]) {
  unsigned int len, start, j, k;
  int16_t t, zeta;

  k = 1;
  for(len = 128; len >= 2; len >>= 1) {
    for(start = 0; start < 256; start = j + len) {
      zeta = zetas[k++];
      for(j = start; j < start + len; j++) {
        t = fqmul(zeta, r[j + len]);
        r[j + len] = r[j] - t;
        r[j] = r[j] + t;
      }
    }
  }
}

/*************************************************
* Name:        invntt_tomont
*
* Description: Inplace inverse number-theoretic transform in Rq and
*              multiplication by Montgomery factor 2^16.
*              Input is in bitreversed order, output is in standard order
*
* Arguments:   - int16_t r[256]: pointer to input/output vector of elements of Zq
**************************************************/
void invntt(int16_t r[256]) {
  unsigned int start, len, j, k;
  int16_t t, zeta;
  const int16_t f = 1441; // mont^2/128

  k = 127;
  for(len = 2; len <= 128; len <<= 1) {
    for(start = 0; start < 256; start = j + len) {
      zeta = zetas[k--];
      for(j = start; j < start + len; j++) {
        t = r[j];
        r[j] = barrett_reduce(t + r[j + len]);
        r[j + len] = r[j + len] - t;
        r[j + len] = fqmul(zeta, r[j + len]);
      }
    }
  }

  for(j = 0; j < 256; j++)
    r[j] = fqmul(r[j], f);
}

/*************************************************
* Name:        basemul
*
* Description: Multiplication of polynomials in Zq[X]/(X^2-zeta)
*              used for multiplication of elements in Rq in NTT domain
*
* Arguments:   - int16_t r[2]: pointer to the output polynomial
*              - const int16_t a[2]: pointer to the first factor
*              - const int16_t b[2]: pointer to the second factor
```

```
*               - int16_t zeta: integer defining the reduction polynomial
**************************************************/
void basemul(int16_t r[2], const int16_t a[2], const int16_t b[2], int16_t zeta)
{
  r[0]  = fqmul(a[1], b[1]);
  r[0]  = fqmul(r[0], zeta);
  r[0] += fqmul(a[0], b[0]);
  r[1]  = fqmul(a[0], b[1]);
  r[1] += fqmul(a[1], b[0]);
}
```

```c
#include <stdint.h>
#include "params.h"
#include "poly.h"
#include "ntt.h"
#include "reduce.h"
#include "cbd.h"
#include "symmetric.h"

/*************************************************
* Name:        poly_compress
*
* Description: Compression and subsequent serialization of a polynomial
*
* Arguments:   - uint8_t *r: pointer to output byte array
*                            (of length KYBER_POLYCOMPRESSEDBYTES)
*              - const poly *a: pointer to input polynomial
**************************************************/
void poly_compress(uint8_t r[KYBER_POLYCOMPRESSEDBYTES], const poly *a)
{
  unsigned int i,j;
  int16_t u;
  uint32_t d0;
  uint8_t t[8];

#if (KYBER_POLYCOMPRESSEDBYTES == 128)
  for(i=0;i<KYBER_N/8;i++) {
    for(j=0;j<8;j++) {
      // map to positive standard representatives
      u  = a->coeffs[8*i+j];
      u += (u >> 15) & KYBER_Q;
/*    t[j] = ((((uint16_t)u << 4) + KYBER_Q/2)/KYBER_Q) & 15; */
      d0 = u << 4;
      d0 += 1665;
      d0 *= 80635;
      d0 >>= 28;
      t[j] = d0 & 0xf;
    }

    r[0] = t[0] | (t[1] << 4);
    r[1] = t[2] | (t[3] << 4);
    r[2] = t[4] | (t[5] << 4);
    r[3] = t[6] | (t[7] << 4);
    r += 4;
  }
#elif (KYBER_POLYCOMPRESSEDBYTES == 160)
  for(i=0;i<KYBER_N/8;i++) {
    for(j=0;j<8;j++) {
      // map to positive standard representatives
      u  = a->coeffs[8*i+j];
      u += (u >> 15) & KYBER_Q;
/*      t[j] = ((((uint32_t)u << 5) + KYBER_Q/2)/KYBER_Q) & 31; */
      d0 = u << 5;
      d0 += 1664;
      d0 *= 40318;
      d0 >>= 27;
      t[j] = d0 & 0x1f;
    }

    r[0] = (t[0] >> 0) | (t[1] << 5);
    r[1] = (t[1] >> 3) | (t[2] << 2) | (t[3] << 7);
    r[2] = (t[3] >> 1) | (t[4] << 4);
    r[3] = (t[4] >> 4) | (t[5] << 1) | (t[6] << 6);
    r[4] = (t[6] >> 2) | (t[7] << 3);
    r += 5;
  }
#else
#error "KYBER_POLYCOMPRESSEDBYTES needs to be in {128, 160}"
#endif
```

```
}

/*************************************************
* Name:        poly_decompress
*
* Description: De-serialization and subsequent decompression of a polynomial;
*              approximate inverse of poly_compress
*
* Arguments:   - poly *r: pointer to output polynomial
*              - const uint8_t *a: pointer to input byte array
*                                  (of length KYBER_POLYCOMPRESSEDBYTES bytes)
**************************************************/
void poly_decompress(poly *r, const uint8_t a[KYBER_POLYCOMPRESSEDBYTES])
{
  unsigned int i;

#if (KYBER_POLYCOMPRESSEDBYTES == 128)
  for(i=0;i<KYBER_N/2;i++) {
    r->coeffs[2*i+0] = (((uint16_t)(a[0] & 15)*KYBER_Q) + 8) >> 4;
    r->coeffs[2*i+1] = (((uint16_t)(a[0] >> 4)*KYBER_Q) + 8) >> 4;
    a += 1;
  }
#elif (KYBER_POLYCOMPRESSEDBYTES == 160)
  unsigned int j;
  uint8_t t[8];
  for(i=0;i<KYBER_N/8;i++) {
    t[0] = (a[0] >> 0);
    t[1] = (a[0] >> 5) | (a[1] << 3);
    t[2] = (a[1] >> 2);
    t[3] = (a[1] >> 7) | (a[2] << 1);
    t[4] = (a[2] >> 4) | (a[3] << 4);
    t[5] = (a[3] >> 1);
    t[6] = (a[3] >> 6) | (a[4] << 2);
    t[7] = (a[4] >> 3);
    a += 5;

    for(j=0;j<8;j++)
      r->coeffs[8*i+j] = ((uint32_t)(t[j] & 31)*KYBER_Q + 16) >> 5;
  }
#else
#error "KYBER_POLYCOMPRESSEDBYTES needs to be in {128, 160}"
#endif
}

/*************************************************
* Name:        poly_tobytes
*
* Description: Serialization of a polynomial
*
* Arguments:   - uint8_t *r: pointer to output byte array
*                            (needs space for KYBER_POLYBYTES bytes)
*              - const poly *a: pointer to input polynomial
**************************************************/
void poly_tobytes(uint8_t r[KYBER_POLYBYTES], const poly *a)
{
  unsigned int i;
  uint16_t t0, t1;

  for(i=0;i<KYBER_N/2;i++) {
    // map to positive standard representatives
    t0  = a->coeffs[2*i];
    t0 += ((int16_t)t0 >> 15) & KYBER_Q;
    t1 = a->coeffs[2*i+1];
    t1 += ((int16_t)t1 >> 15) & KYBER_Q;
    r[3*i+0] = (t0 >> 0);
    r[3*i+1] = (t0 >> 8) | (t1 << 4);
    r[3*i+2] = (t1 >> 4);
  }
```

```
}

/*************************************************
* Name:        poly_frombytes
*
* Description: De-serialization of a polynomial;
*              inverse of poly_tobytes
*
* Arguments:   - poly *r: pointer to output polynomial
*              - const uint8_t *a: pointer to input byte array
*                                  (of KYBER_POLYBYTES bytes)
**************************************************/
void poly_frombytes(poly *r, const uint8_t a[KYBER_POLYBYTES])
{
  unsigned int i;
  for(i=0;i<KYBER_N/2;i++) {
    r->coeffs[2*i]   = ((a[3*i+0] >> 0) | ((uint16_t)a[3*i+1] << 8)) & 0xFFF;
    r->coeffs[2*i+1] = ((a[3*i+1] >> 4) | ((uint16_t)a[3*i+2] << 4)) & 0xFFF;
  }
}

/*************************************************
* Name:        poly_frommsg
*
* Description: Convert 32-byte message to polynomial
*
* Arguments:   - poly *r: pointer to output polynomial
*              - const uint8_t *msg: pointer to input message
**************************************************/
void poly_frommsg(poly *r, const uint8_t msg[KYBER_INDCPA_MSGBYTES])
{
  unsigned int i,j;
  int16_t mask;

#if (KYBER_INDCPA_MSGBYTES != KYBER_N/8)
#error "KYBER_INDCPA_MSGBYTES must be equal to KYBER_N/8 bytes!"
#endif

  for(i=0;i<KYBER_N/8;i++) {
    for(j=0;j<8;j++) {
      mask = -(int16_t)((msg[i] >> j)&1);
      r->coeffs[8*i+j] = mask & ((KYBER_Q+1)/2);
    }
  }
}

/*************************************************
* Name:        poly_tomsg
*
* Description: Convert polynomial to 32-byte message
*
* Arguments:   - uint8_t *msg: pointer to output message
*              - const poly *a: pointer to input polynomial
**************************************************/
void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const poly *a)
{
  unsigned int i,j;
  uint32_t t;

  for(i=0;i<KYBER_N/8;i++) {
    msg[i] = 0;
    for(j=0;j<8;j++) {
      t  = a->coeffs[8*i+j];
      // t += ((int16_t)t >> 15) & KYBER_Q;
      // t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
      t <<= 1;
      t += 1665;
      t *= 80635;
```

```
      t >>= 28;
      t &= 1;
      msg[i] |= t << j;
    }
  }
}

/*************************************************
* Name:        poly_getnoise_eta1
*
* Description: Sample a polynomial deterministically from a seed and a nonce,
*              with output polynomial close to centered binomial distribution
*              with parameter KYBER_ETA1
*
* Arguments:   - poly *r: pointer to output polynomial
*              - const uint8_t *seed: pointer to input seed
*                                     (of length KYBER_SYMBYTES bytes)
*              - uint8_t nonce: one-byte input nonce
**************************************************/
void poly_getnoise_eta1(poly *r, const uint8_t seed[KYBER_SYMBYTES], uint8_t nonce)
{
  uint8_t buf[KYBER_ETA1*KYBER_N/4];
  prf(buf, sizeof(buf), seed, nonce);
  poly_cbd_eta1(r, buf);
}

/*************************************************
* Name:        poly_getnoise_eta2
*
* Description: Sample a polynomial deterministically from a seed and a nonce,
*              with output polynomial close to centered binomial distribution
*              with parameter KYBER_ETA2
*
* Arguments:   - poly *r: pointer to output polynomial
*              - const uint8_t *seed: pointer to input seed
*                                     (of length KYBER_SYMBYTES bytes)
*              - uint8_t nonce: one-byte input nonce
**************************************************/
void poly_getnoise_eta2(poly *r, const uint8_t seed[KYBER_SYMBYTES], uint8_t nonce)
{
  uint8_t buf[KYBER_ETA2*KYBER_N/4];
  prf(buf, sizeof(buf), seed, nonce);
  poly_cbd_eta2(r, buf);
}


/*************************************************
* Name:        poly_ntt
*
* Description: Computes negacyclic number-theoretic transform (NTT) of
*              a polynomial in place;
*              inputs assumed to be in normal order, output in bitreversed order
*
* Arguments:   - uint16_t *r: pointer to in/output polynomial
**************************************************/
void poly_ntt(poly *r)
{
  ntt(r->coeffs);
  poly_reduce(r);
}

/*************************************************
* Name:        poly_invntt_tomont
*
* Description: Computes inverse of negacyclic number-theoretic transform (NTT)
*              of a polynomial in place;
*              inputs assumed to be in bitreversed order, output in normal order
*
```

```c
* Arguments:   - uint16_t *a: pointer to in/output polynomial
**************************************************/
void poly_invntt_tomont(poly *r)
{
  invntt(r->coeffs);
}

/*************************************************
* Name:        poly_basemul_montgomery
*
* Description: Multiplication of two polynomials in NTT domain
*
* Arguments:   - poly *r: pointer to output polynomial
*              - const poly *a: pointer to first input polynomial
*              - const poly *b: pointer to second input polynomial
**************************************************/
void poly_basemul_montgomery(poly *r, const poly *a, const poly *b)
{
  unsigned int i;
  for(i=0;i<KYBER_N/4;i++) {
    basemul(&r->coeffs[4*i], &a->coeffs[4*i], &b->coeffs[4*i], zetas[64+i]);
    basemul(&r->coeffs[4*i+2], &a->coeffs[4*i+2], &b->coeffs[4*i+2], -zetas[64+i]);
  }
}

/*************************************************
* Name:        poly_tomont
*
* Description: Inplace conversion of all coefficients of a polynomial
*              from normal domain to Montgomery domain
*
* Arguments:   - poly *r: pointer to input/output polynomial
**************************************************/
void poly_tomont(poly *r)
{
  unsigned int i;
  const int16_t f = (1ULL << 32) % KYBER_Q;
  for(i=0;i<KYBER_N;i++)
    r->coeffs[i] = montgomery_reduce((int32_t)r->coeffs[i]*f);
}

/*************************************************
* Name:        poly_reduce
*
* Description: Applies Barrett reduction to all coefficients of a polynomial
*              for details of the Barrett reduction see comments in reduce.c
*
* Arguments:   - poly *r: pointer to input/output polynomial
**************************************************/
void poly_reduce(poly *r)
{
  unsigned int i;
  for(i=0;i<KYBER_N;i++)
    r->coeffs[i] = barrett_reduce(r->coeffs[i]);
}

/*************************************************
* Name:        poly_add
*
* Description: Add two polynomials; no modular reduction is performed
*
* Arguments: - poly *r: pointer to output polynomial
*            - const poly *a: pointer to first input polynomial
*            - const poly *b: pointer to second input polynomial
**************************************************/
void poly_add(poly *r, const poly *a, const poly *b)
{
  unsigned int i;
```

```
  for(i=0;i<KYBER_N;i++)
    r->coeffs[i] = a->coeffs[i] + b->coeffs[i];
}

/*************************************************
* Name:        poly_sub
*
* Description: Subtract two polynomials; no modular reduction is performed
*
* Arguments: - poly *r:       pointer to output polynomial
*            - const poly *a: pointer to first input polynomial
*            - const poly *b: pointer to second input polynomial
**************************************************/
void poly_sub(poly *r, const poly *a, const poly *b)
{
  unsigned int i;
  for(i=0;i<KYBER_N;i++)
    r->coeffs[i] = a->coeffs[i] - b->coeffs[i];
}
```

```c
#include <stdint.h>
#include "params.h"
#include "poly.h"
#include "polyvec.h"

/*************************************************
* Name:        polyvec_compress
*
* Description: Compress and serialize vector of polynomials
*
* Arguments:   - uint8_t *r: pointer to output byte array
*                            (needs space for KYBER_POLYVECCOMPRESSEDBYTES)
*              - const polyvec *a: pointer to input vector of polynomials
**************************************************/
void polyvec_compress(uint8_t r[KYBER_POLYVECCOMPRESSEDBYTES], const polyvec *a)
{
  unsigned int i,j,k;
  uint64_t d0;

#if (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 352))
  uint16_t t[8];
  for(i=0;i<KYBER_K;i++) {
    for(j=0;j<KYBER_N/8;j++) {
      for(k=0;k<8;k++) {
        t[k]  = a->vec[i].coeffs[8*j+k];
        t[k] += ((int16_t)t[k] >> 15) & KYBER_Q;
/*      t[k]  = ((((uint32_t)t[k] << 11) + KYBER_Q/2)/KYBER_Q) & 0x7ff; */
        d0 = t[k];
        d0 <<= 11;
        d0 += 1664;
        d0 *= 645084;
        d0 >>= 31;
        t[k] = d0 & 0x7ff;
      }

      r[ 0] = (t[0] >>  0);
      r[ 1] = (t[0] >>  8) | (t[1] << 3);
      r[ 2] = (t[1] >>  5) | (t[2] << 6);
      r[ 3] = (t[2] >>  2);
      r[ 4] = (t[2] >> 10) | (t[3] << 1);
      r[ 5] = (t[3] >>  7) | (t[4] << 4);
      r[ 6] = (t[4] >>  4) | (t[5] << 7);
      r[ 7] = (t[5] >>  1);
      r[ 8] = (t[5] >>  9) | (t[6] << 2);
      r[ 9] = (t[6] >>  6) | (t[7] << 5);
      r[10] = (t[7] >>  3);
      r += 11;
    }
  }
#elif (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 320))
  uint16_t t[4];
  for(i=0;i<KYBER_K;i++) {
    for(j=0;j<KYBER_N/4;j++) {
      for(k=0;k<4;k++) {
        t[k]  = a->vec[i].coeffs[4*j+k];
        t[k] += ((int16_t)t[k] >> 15) & KYBER_Q;
/*      t[k]  = ((((uint32_t)t[k] << 10) + KYBER_Q/2)/ KYBER_Q) & 0x3ff; */
        d0 = t[k];
        d0 <<= 10;
        d0 += 1665;
        d0 *= 1290167;
        d0 >>= 32;
        t[k] = d0 & 0x3ff;
      }

      r[0] = (t[0] >> 0);
      r[1] = (t[0] >> 8) | (t[1] << 2);
      r[2] = (t[1] >> 6) | (t[2] << 4);
```

```
      r[3] = (t[2] >> 4) | (t[3] << 6);
      r[4] = (t[3] >> 2);
      r += 5;
    }
  }
#else
#error "KYBER_POLYVECCOMPRESSEDBYTES needs to be in {320*KYBER_K, 352*KYBER_K}"
#endif
}

/*************************************************
* Name:        polyvec_decompress
*
* Description: De-serialize and decompress vector of polynomials;
*              approximate inverse of polyvec_compress
*
* Arguments:   - polyvec *r:       pointer to output vector of polynomials
*              - const uint8_t *a: pointer to input byte array
*                                  (of length KYBER_POLYVECCOMPRESSEDBYTES)
**************************************************/
void polyvec_decompress(polyvec *r, const uint8_t a[KYBER_POLYVECCOMPRESSEDBYTES])
{
  unsigned int i,j,k;

#if (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 352))
  uint16_t t[8];
  for(i=0;i<KYBER_K;i++) {
    for(j=0;j<KYBER_N/8;j++) {
      t[0] = (a[0] >> 0) | ((uint16_t)a[ 1] << 8);
      t[1] = (a[1] >> 3) | ((uint16_t)a[ 2] << 5);
      t[2] = (a[2] >> 6) | ((uint16_t)a[ 3] << 2) | ((uint16_t)a[4] << 10);
      t[3] = (a[4] >> 1) | ((uint16_t)a[ 5] << 7);
      t[4] = (a[5] >> 4) | ((uint16_t)a[ 6] << 4);
      t[5] = (a[6] >> 7) | ((uint16_t)a[ 7] << 1) | ((uint16_t)a[8] << 9);
      t[6] = (a[8] >> 2) | ((uint16_t)a[ 9] << 6);
      t[7] = (a[9] >> 5) | ((uint16_t)a[10] << 3);
      a += 11;

      for(k=0;k<8;k++)
        r->vec[i].coeffs[8*j+k] = ((uint32_t)(t[k] & 0x7FF)*KYBER_Q + 1024) >> 11;
    }
  }
#elif (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 320))
  uint16_t t[4];
  for(i=0;i<KYBER_K;i++) {
    for(j=0;j<KYBER_N/4;j++) {
      t[0] = (a[0] >> 0) | ((uint16_t)a[1] << 8);
      t[1] = (a[1] >> 2) | ((uint16_t)a[2] << 6);
      t[2] = (a[2] >> 4) | ((uint16_t)a[3] << 4);
      t[3] = (a[3] >> 6) | ((uint16_t)a[4] << 2);
      a += 5;

      for(k=0;k<4;k++)
        r->vec[i].coeffs[4*j+k] = ((uint32_t)(t[k] & 0x3FF)*KYBER_Q + 512) >> 10;
    }
  }
#else
#error "KYBER_POLYVECCOMPRESSEDBYTES needs to be in {320*KYBER_K, 352*KYBER_K}"
#endif
}

/*************************************************
* Name:        polyvec_tobytes
*
* Description: Serialize vector of polynomials
*
* Arguments:   - uint8_t *r: pointer to output byte array
*                            (needs space for KYBER_POLYVECBYTES)
```

```
*               - const polyvec *a: pointer to input vector of polynomials
**************************************************/
void polyvec_tobytes(uint8_t r[KYBER_POLYVECBYTES], const polyvec *a)
{
  unsigned int i;
  for(i=0;i<KYBER_K;i++)
    poly_tobytes(r+i*KYBER_POLYBYTES, &a->vec[i]);
}

/*************************************************
* Name:        polyvec_frombytes
*
* Description: De-serialize vector of polynomials;
*              inverse of polyvec_tobytes
*
* Arguments:   - uint8_t *r:       pointer to output byte array
*              - const polyvec *a: pointer to input vector of polynomials
*                                  (of length KYBER_POLYVECBYTES)
**************************************************/
void polyvec_frombytes(polyvec *r, const uint8_t a[KYBER_POLYVECBYTES])
{
  unsigned int i;
  for(i=0;i<KYBER_K;i++)
    poly_frombytes(&r->vec[i], a+i*KYBER_POLYBYTES);
}

/*************************************************
* Name:        polyvec_ntt
*
* Description: Apply forward NTT to all elements of a vector of polynomials
*
* Arguments:   - polyvec *r: pointer to in/output vector of polynomials
**************************************************/
void polyvec_ntt(polyvec *r)
{
  unsigned int i;
  for(i=0;i<KYBER_K;i++)
    poly_ntt(&r->vec[i]);
}

/*************************************************
* Name:        polyvec_invntt_tomont
*
* Description: Apply inverse NTT to all elements of a vector of polynomials
*              and multiply by Montgomery factor 2^16
*
* Arguments:   - polyvec *r: pointer to in/output vector of polynomials
**************************************************/
void polyvec_invntt_tomont(polyvec *r)
{
  unsigned int i;
  for(i=0;i<KYBER_K;i++)
    poly_invntt_tomont(&r->vec[i]);
}

/*************************************************
* Name:        polyvec_basemul_acc_montgomery
*
* Description: Multiply elements of a and b in NTT domain, accumulate into r,
*              and multiply by 2^-16.
*
* Arguments: - poly *r: pointer to output polynomial
*            - const polyvec *a: pointer to first input vector of polynomials
*            - const polyvec *b: pointer to second input vector of polynomials
**************************************************/
void polyvec_basemul_acc_montgomery(poly *r, const polyvec *a, const polyvec *b)
{
  unsigned int i;
```

```
  poly t;

  poly_basemul_montgomery(r, &a->vec[0], &b->vec[0]);
  for(i=1;i<KYBER_K;i++) {
    poly_basemul_montgomery(&t, &a->vec[i], &b->vec[i]);
    poly_add(r, r, &t);
  }

  poly_reduce(r);
}

/*************************************************
* Name:        polyvec_reduce
*
* Description: Applies Barrett reduction to each coefficient
*              of each element of a vector of polynomials;
*              for details of the Barrett reduction see comments in reduce.c
*
* Arguments:   - polyvec *r: pointer to input/output polynomial
**************************************************/
void polyvec_reduce(polyvec *r)
{
  unsigned int i;
  for(i=0;i<KYBER_K;i++)
    poly_reduce(&r->vec[i]);
}

/*************************************************
* Name:        polyvec_add
*
* Description: Add vectors of polynomials
*
* Arguments: - polyvec *r: pointer to output vector of polynomials
*            - const polyvec *a: pointer to first input vector of polynomials
*            - const polyvec *b: pointer to second input vector of polynomials
**************************************************/
void polyvec_add(polyvec *r, const polyvec *a, const polyvec *b)
{
  unsigned int i;
  for(i=0;i<KYBER_K;i++)
    poly_add(&r->vec[i], &a->vec[i], &b->vec[i]);
}
```

```
//
//  PQCgenKAT_kem.c
//
//  Created by Bassham, Lawrence E (Fed) on 8/29/17.
//  Copyright Â© 2017 Bassham, Lawrence E (Fed). All rights reserved.
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "rng.h"
#include "kem.h"

#define MAX_MARKER_LEN          50
#define KAT_SUCCESS          0
#define KAT_FILE_OPEN_ERROR -1
#define KAT_DATA_ERROR      -3
#define KAT_CRYPTO_FAILURE  -4

int             FindMarker(FILE *infile, const char *marker);
int             ReadHex(FILE *infile, unsigned char *A, int Length, char *str);
void    fprintBstr(FILE *fp, char *S, unsigned char *A, unsigned long long L);

int
main()
{
    char                fn_req[32], fn_rsp[32];
    FILE                *fp_req, *fp_rsp;
    unsigned char       seed[48];
    unsigned char       entropy_input[48];
    unsigned char       ct[CRYPTO_CIPHERTEXTBYTES], ss[CRYPTO_BYTES], ss1[CRYPTO_BYTES];
    int                 count;
    int                 done;
    unsigned char       pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];
    int                 ret_val;

    // Create the REQUEST file
    sprintf(fn_req, "PQCkemKAT_%d.req", CRYPTO_SECRETKEYBYTES);
    if ( (fp_req = fopen(fn_req, "w")) == NULL ) {
        printf("Couldn't open <%s> for write\n", fn_req);
        return KAT_FILE_OPEN_ERROR;
    }
    sprintf(fn_rsp, "PQCkemKAT_%d.rsp", CRYPTO_SECRETKEYBYTES);
    if ( (fp_rsp = fopen(fn_rsp, "w")) == NULL ) {
        printf("Couldn't open <%s> for write\n", fn_rsp);
        return KAT_FILE_OPEN_ERROR;
    }

    for (int i=0; i<48; i++)
        entropy_input[i] = i;

    randombytes_init(entropy_input, NULL, 256);
    for (int i=0; i<100; i++) {
        fprintf(fp_req, "count = %d\n", i);
        randombytes(seed, 48);
        fprintBstr(fp_req, "seed = ", seed, 48);
        fprintf(fp_req, "pk =\n");
        fprintf(fp_req, "sk =\n");
        fprintf(fp_req, "ct =\n");
        fprintf(fp_req, "ss =\n\n");
    }
    fclose(fp_req);

    //Create the RESPONSE file based on what's in the REQUEST file
    if ( (fp_req = fopen(fn_req, "r")) == NULL ) {
        printf("Couldn't open <%s> for read\n", fn_req);
        return KAT_FILE_OPEN_ERROR;
```

```
    }

    fprintf(fp_rsp, "# %s\n\n", CRYPTO_ALGNAME);
    done = 0;
    do {
        if ( FindMarker(fp_req, "count = ") )
            fscanf(fp_req, "%d", &count);
        else {
            done = 1;
            break;
        }
        fprintf(fp_rsp, "count = %d\n", count);

        if ( !ReadHex(fp_req, seed, 48, "seed = ") ) {
            printf("ERROR: unable to read 'seed' from <%s>\n", fn_req);
            return KAT_DATA_ERROR;
        }
        fprintBstr(fp_rsp, "seed = ", seed, 48);

        randombytes_init(seed, NULL, 256);

        // Generate the public/private keypair
        if ( (ret_val = crypto_kem_keypair(pk, sk)) != 0) {
            printf("crypto_kem_keypair returned <%d>\n", ret_val);
            return KAT_CRYPTO_FAILURE;
        }
        fprintBstr(fp_rsp, "pk = ", pk, CRYPTO_PUBLICKEYBYTES);
        fprintBstr(fp_rsp, "sk = ", sk, CRYPTO_SECRETKEYBYTES);

        if ( (ret_val = crypto_kem_enc(ct, ss, pk)) != 0) {
            printf("crypto_kem_enc returned <%d>\n", ret_val);
            return KAT_CRYPTO_FAILURE;
        }
        fprintBstr(fp_rsp, "ct = ", ct, CRYPTO_CIPHERTEXTBYTES);
        fprintBstr(fp_rsp, "ss = ", ss, CRYPTO_BYTES);

        fprintf(fp_rsp, "\n");

        if ( (ret_val = crypto_kem_dec(ss1, ct, sk)) != 0) {
            printf("crypto_kem_dec returned <%d>\n", ret_val);
            return KAT_CRYPTO_FAILURE;
        }

        if ( memcmp(ss, ss1, CRYPTO_BYTES) ) {
            printf("crypto_kem_dec returned bad 'ss' value\n");
            return KAT_CRYPTO_FAILURE;
        }

    } while ( !done );

    fclose(fp_req);
    fclose(fp_rsp);

    return KAT_SUCCESS;
}



//
// ALLOW TO READ HEXADECIMAL ENTRY (KEYS, DATA, TEXT, etc.)
//
//
// ALLOW TO READ HEXADECIMAL ENTRY (KEYS, DATA, TEXT, etc.)
//
int
FindMarker(FILE *infile, const char *marker)
{
        char    line[MAX_MARKER_LEN];
```

```
        int             i, len;
        int curr_line;

        len = (int)strlen(marker);
        if ( len > MAX_MARKER_LEN-1 )
                len = MAX_MARKER_LEN-1;

        for ( i=0; i<len; i++ )
          {
            curr_line = fgetc(infile);
            line[i] = curr_line;
            if (curr_line == EOF )
              return 0;
          }
        line[len] = '\0';

        while ( 1 ) {
                if ( !strncmp(line, marker, len) )
                        return 1;

                for ( i=0; i<len-1; i++ )
                        line[i] = line[i+1];
                curr_line = fgetc(infile);
                line[len-1] = curr_line;
                if (curr_line == EOF )
                    return 0;
                line[len] = '\0';
        }

        // shouldn't get here
        return 0;
}

//
// ALLOW TO READ HEXADECIMAL ENTRY (KEYS, DATA, TEXT, etc.)
//
int
ReadHex(FILE *infile, unsigned char *A, int Length, char *str)
{
        int                     i, ch, started;
        unsigned char   ich;

        if ( Length == 0 ) {
                A[0] = 0x00;
                return 1;
        }
        memset(A, 0x00, Length);
        started = 0;
        if ( FindMarker(infile, str) )
                while ( (ch = fgetc(infile)) != EOF ) {
                        if ( !isxdigit(ch) ) {
                                if ( !started ) {
                                        if ( ch == '\n' )
                                                break;
                                        else
                                                continue;
                                }
                                else
                                        break;
                        }
                        started = 1;
                        if ( (ch >= '0') && (ch <= '9') )
                                ich = ch - '0';
                        else if ( (ch >= 'A') && (ch <= 'F') )
                                ich = ch - 'A' + 10;
                        else if ( (ch >= 'a') && (ch <= 'f') )
                                ich = ch - 'a' + 10;
            else // shouldn't ever get here
```

```
                ich = 0;

                        for ( i=0; i<Length-1; i++ )
                                A[i] = (A[i] << 4) | (A[i+1] >> 4);
                        A[Length-1] = (A[Length-1] << 4) | ich;
                }
        else
                return 0;

        return 1;
}

void
fprintBstr(FILE *fp, char *S, unsigned char *A, unsigned long long L)
{
        unsigned long long  i;

        fprintf(fp, "%s", S);

        for ( i=0; i<L; i++ )
                fprintf(fp, "%02X", A[i]);

        if ( L == 0 )
                fprintf(fp, "00");

        fprintf(fp, "\n");
}
```

```c
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include "randombytes.h"

#ifdef _WIN32
#include <windows.h>
#include <wincrypt.h>
#else
#include <fcntl.h>
#include <errno.h>
#ifdef __linux__
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#else
#include <unistd.h>
#endif
#endif

#ifdef _WIN32
void randombytes(uint8_t *out, size_t outlen) {
  HCRYPTPROV ctx;
  size_t len;

  if(!CryptAcquireContext(&ctx, NULL, NULL, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT))
    abort();

  while(outlen > 0) {
    len = (outlen > 1048576) ? 1048576 : outlen;
    if(!CryptGenRandom(ctx, len, (BYTE *)out))
      abort();

    out += len;
    outlen -= len;
  }

  if(!CryptReleaseContext(ctx, 0))
    abort();
}
#elif defined(__linux__) && defined(SYS_getrandom)
void randombytes(uint8_t *out, size_t outlen) {
  ssize_t ret;

  while(outlen > 0) {
    ret = syscall(SYS_getrandom, out, outlen, 0);
    if(ret == -1 && errno == EINTR)
      continue;
    else if(ret == -1)
      abort();

    out += ret;
    outlen -= ret;
  }
}
#else
void randombytes(uint8_t *out, size_t outlen) {
  static int fd = -1;
  ssize_t ret;

  while(fd == -1) {
    fd = open("/dev/urandom", O_RDONLY);
    if(fd == -1 && errno == EINTR)
      continue;
    else if(fd == -1)
      abort();
  }
```

```
  while(outlen > 0) {
    ret = read(fd, out, outlen);
    if(ret == -1 && errno == EINTR)
      continue;
    else if(ret == -1)
      abort();

    out += ret;
    outlen -= ret;
  }
}
#endif
```

```c
#include <stdint.h>
#include "params.h"
#include "reduce.h"

/*************************************************
* Name:        montgomery_reduce
*
* Description: Montgomery reduction; given a 32-bit integer a, computes
*              16-bit integer congruent to a * R^-1 mod q, where R=2^16
*
* Arguments:   - int32_t a: input integer to be reduced;
*                           has to be in {-q2^15,...,q2^15-1}
*
* Returns:     integer in {-q+1,...,q-1} congruent to a * R^-1 modulo q.
**************************************************/
int16_t montgomery_reduce(int32_t a)
{
  int16_t t;

  t = (int16_t)a*QINV;
  t = (a - (int32_t)t*KYBER_Q) >> 16;
  return t;
}

/*************************************************
* Name:        barrett_reduce
*
* Description: Barrett reduction; given a 16-bit integer a, computes
*              centered representative congruent to a mod q in {-(q-1)/2,...,(q-1)/2}
*
* Arguments:   - int16_t a: input integer to be reduced
*
* Returns:     integer in {-(q-1)/2,...,(q-1)/2} congruent to a modulo q.
**************************************************/
int16_t barrett_reduce(int16_t a) {
  int16_t t;
  const int16_t v = ((1<<26) + KYBER_Q/2)/KYBER_Q;

  t  = ((int32_t)v*a + (1<<25)) >> 26;
  t *= KYBER_Q;
  return a - t;
}
```

```
//
//  rng.c
//
//  Created by Bassham, Lawrence E (Fed) on 8/29/17.
//  Copyright Â© 2017 Bassham, Lawrence E (Fed). All rights reserved.
//

#include <string.h>
#include "rng.h"
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>

AES256_CTR_DRBG_struct  DRBG_ctx;

void    AES256_ECB(unsigned char *key, unsigned char *ctr, unsigned char *buffer);

/*
 seedexpander_init()
 ctx            - stores the current state of an instance of the seed expander
 seed           - a 32 byte random value
 diversifier    - an 8 byte diversifier
 maxlen         - maximum number of bytes (less than 2**32) generated under this seed and d
iversifier
 */
int
seedexpander_init(AES_XOF_struct *ctx,
                  unsigned char *seed,
                  unsigned char *diversifier,
                  unsigned long maxlen)
{
    if ( maxlen >= 0x100000000 )
        return RNG_BAD_MAXLEN;

    ctx->length_remaining = maxlen;

    memcpy(ctx->key, seed, 32);

    memcpy(ctx->ctr, diversifier, 8);
    ctx->ctr[11] = maxlen % 256;
    maxlen >>= 8;
    ctx->ctr[10] = maxlen % 256;
    maxlen >>= 8;
    ctx->ctr[9] = maxlen % 256;
    maxlen >>= 8;
    ctx->ctr[8] = maxlen % 256;
    memset(ctx->ctr+12, 0x00, 4);

    ctx->buffer_pos = 16;
    memset(ctx->buffer, 0x00, 16);

    return RNG_SUCCESS;
}

/*
 seedexpander()
    ctx  - stores the current state of an instance of the seed expander
    x    - returns the XOF data
    xlen - number of bytes to return
 */
int
seedexpander(AES_XOF_struct *ctx, unsigned char *x, unsigned long xlen)
{
    unsigned long   offset;

    if ( x == NULL )
        return RNG_BAD_OUTBUF;
    if ( xlen >= ctx->length_remaining )
```

```
        return RNG_BAD_REQ_LEN;

    ctx->length_remaining -= xlen;

    offset = 0;
    while ( xlen > 0 ) {
        if ( xlen <= (16-ctx->buffer_pos) ) { // buffer has what we need
            memcpy(x+offset, ctx->buffer+ctx->buffer_pos, xlen);
            ctx->buffer_pos += xlen;

            return RNG_SUCCESS;
        }

        // take what's in the buffer
        memcpy(x+offset, ctx->buffer+ctx->buffer_pos, 16-ctx->buffer_pos);
        xlen -= 16-ctx->buffer_pos;
        offset += 16-ctx->buffer_pos;

        AES256_ECB(ctx->key, ctx->ctr, ctx->buffer);
        ctx->buffer_pos = 0;

        //increment the counter
        for (int i=15; i>=12; i--) {
            if ( ctx->ctr[i] == 0xff )
                ctx->ctr[i] = 0x00;
            else {
                ctx->ctr[i]++;
                break;
            }
        }

    }

    return RNG_SUCCESS;
}


void handleErrors(void)
{
    ERR_print_errors_fp(stderr);
    abort();
}

// Use whatever AES implementation you have. This uses AES from openSSL library
//    key - 256-bit AES key
//    ctr - a 128-bit plaintext value
//    buffer - a 128-bit ciphertext value
void
AES256_ECB(unsigned char *key, unsigned char *ctr, unsigned char *buffer)
{
    EVP_CIPHER_CTX *ctx;

    int len;

    int ciphertext_len;

    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new())) handleErrors();

    if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_256_ecb(), NULL, key, NULL))
        handleErrors();

    if(1 != EVP_EncryptUpdate(ctx, buffer, &len, ctr, 16))
        handleErrors();
    ciphertext_len = len;

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);
```

```
}

void
randombytes_init(unsigned char *entropy_input,
                 unsigned char *personalization_string,
                 int security_strength)
{
    unsigned char   seed_material[48];

    memcpy(seed_material, entropy_input, 48);
    if (personalization_string)
        for (int i=0; i<48; i++)
            seed_material[i] ^= personalization_string[i];
    memset(DRBG_ctx.Key, 0x00, 32);
    memset(DRBG_ctx.V, 0x00, 16);
    AES256_CTR_DRBG_Update(seed_material, DRBG_ctx.Key, DRBG_ctx.V);
    DRBG_ctx.reseed_counter = 1;
}

int
randombytes(unsigned char *x, unsigned long long xlen)
{
    unsigned char   block[16];
    int             i = 0;

    while ( xlen > 0 ) {
        //increment V
        for (int j=15; j>=0; j--) {
            if ( DRBG_ctx.V[j] == 0xff )
                DRBG_ctx.V[j] = 0x00;
            else {
                DRBG_ctx.V[j]++;
                break;
            }
        }
        AES256_ECB(DRBG_ctx.Key, DRBG_ctx.V, block);
        if ( xlen > 15 ) {
            memcpy(x+i, block, 16);
            i += 16;
            xlen -= 16;
        }
        else {
            memcpy(x+i, block, xlen);
            xlen = 0;
        }
    }
    AES256_CTR_DRBG_Update(NULL, DRBG_ctx.Key, DRBG_ctx.V);
    DRBG_ctx.reseed_counter++;

    return RNG_SUCCESS;
}

void
AES256_CTR_DRBG_Update(unsigned char *provided_data,
                       unsigned char *Key,
                       unsigned char *V)
{
    unsigned char   temp[48];

    for (int i=0; i<3; i++) {
        //increment V
        for (int j=15; j>=0; j--) {
            if ( V[j] == 0xff )
                V[j] = 0x00;
            else {
                V[j]++;
                break;
            }
```

```
        }

        AES256_ECB(Key, V, temp+16*i);
    }
    if ( provided_data != NULL )
        for (int i=0; i<48; i++)
            temp[i] ^= provided_data[i];
    memcpy(Key, temp, 32);
    memcpy(V, temp+32, 16);
}
```

```c
/* Adapted from Public Domain code by D. J. Bernstein. */

#include <stddef.h>
#include <stdint.h>
#include "sha2.h"

static uint32_t load_bigendian(const uint8_t *x)
{
  return
      (uint32_t) (x[3]) \
    | (((uint32_t) (x[2])) << 8) \
    | (((uint32_t) (x[1])) << 16) \
    | (((uint32_t) (x[0])) << 24)
  ;
}

static void store_bigendian(uint8_t *x,uint32_t u)
{
  x[3] = u; u >>= 8;
  x[2] = u; u >>= 8;
  x[1] = u; u >>= 8;
  x[0] = u;
}

#define SHR(x,c) ((x) >> (c))
#define ROTR(x,c) (((x) >> (c)) | ((x) << (32 - (c))))

#define Ch(x,y,z) ((x & y) ^ (~x & z))
#define Maj(x,y,z) ((x & y) ^ (x & z) ^ (y & z))
#define Sigma0(x) (ROTR(x, 2) ^ ROTR(x,13) ^ ROTR(x,22))
#define Sigma1(x) (ROTR(x, 6) ^ ROTR(x,11) ^ ROTR(x,25))
#define sigma0(x) (ROTR(x, 7) ^ ROTR(x,18) ^ SHR(x, 3))
#define sigma1(x) (ROTR(x,17) ^ ROTR(x,19) ^ SHR(x,10))

#define M(w0,w14,w9,w1) w0 = sigma1(w14) + w9 + sigma0(w1) + w0;

#define EXPAND \
  M(w0 ,w14,w9 ,w1 ) \
  M(w1 ,w15,w10,w2 ) \
  M(w2 ,w0 ,w11,w3 ) \
  M(w3 ,w1 ,w12,w4 ) \
  M(w4 ,w2 ,w13,w5 ) \
  M(w5 ,w3 ,w14,w6 ) \
  M(w6 ,w4 ,w15,w7 ) \
  M(w7 ,w5 ,w0 ,w8 ) \
  M(w8 ,w6 ,w1 ,w9 ) \
  M(w9 ,w7 ,w2 ,w10) \
  M(w10,w8 ,w3 ,w11) \
  M(w11,w9 ,w4 ,w12) \
  M(w12,w10,w5 ,w13) \
  M(w13,w11,w6 ,w14) \
  M(w14,w12,w7 ,w15) \
  M(w15,w13,w8 ,w0 )

#define F(w,k) \
  T1 = h + Sigma1(e) + Ch(e,f,g) + k + w; \
  T2 = Sigma0(a) + Maj(a,b,c); \
  h = g; \
  g = f; \
  f = e; \
  e = d + T1; \
  d = c; \
  c = b; \
  b = a; \
  a = T1 + T2;

static int crypto_hashblocks_sha256(uint8_t *statebytes,const uint8_t *in,size_t inlen)
{
```

```
  uint32_t state[8];
  uint32_t a;
  uint32_t b;
  uint32_t c;
  uint32_t d;
  uint32_t e;
  uint32_t f;
  uint32_t g;
  uint32_t h;
  uint32_t T1;
  uint32_t T2;

  a = load_bigendian(statebytes +  0); state[0] = a;
  b = load_bigendian(statebytes +  4); state[1] = b;
  c = load_bigendian(statebytes +  8); state[2] = c;
  d = load_bigendian(statebytes + 12); state[3] = d;
  e = load_bigendian(statebytes + 16); state[4] = e;
  f = load_bigendian(statebytes + 20); state[5] = f;
  g = load_bigendian(statebytes + 24); state[6] = g;
  h = load_bigendian(statebytes + 28); state[7] = h;

  while (inlen >= 64) {
    uint32_t w0  = load_bigendian(in +  0);
    uint32_t w1  = load_bigendian(in +  4);
    uint32_t w2  = load_bigendian(in +  8);
    uint32_t w3  = load_bigendian(in + 12);
    uint32_t w4  = load_bigendian(in + 16);
    uint32_t w5  = load_bigendian(in + 20);
    uint32_t w6  = load_bigendian(in + 24);
    uint32_t w7  = load_bigendian(in + 28);
    uint32_t w8  = load_bigendian(in + 32);
    uint32_t w9  = load_bigendian(in + 36);
    uint32_t w10 = load_bigendian(in + 40);
    uint32_t w11 = load_bigendian(in + 44);
    uint32_t w12 = load_bigendian(in + 48);
    uint32_t w13 = load_bigendian(in + 52);
    uint32_t w14 = load_bigendian(in + 56);
    uint32_t w15 = load_bigendian(in + 60);

    F(w0 ,0x428a2f98)
    F(w1 ,0x71374491)
    F(w2 ,0xb5c0fbcf)
    F(w3 ,0xe9b5dba5)
    F(w4 ,0x3956c25b)
    F(w5 ,0x59f111f1)
    F(w6 ,0x923f82a4)
    F(w7 ,0xab1c5ed5)
    F(w8 ,0xd807aa98)
    F(w9 ,0x12835b01)
    F(w10,0x243185be)
    F(w11,0x550c7dc3)
    F(w12,0x72be5d74)
    F(w13,0x80deb1fe)
    F(w14,0x9bdc06a7)
    F(w15,0xc19bf174)

    EXPAND

    F(w0 ,0xe49b69c1)
    F(w1 ,0xefbe4786)
    F(w2 ,0x0fc19dc6)
    F(w3 ,0x240ca1cc)
    F(w4 ,0x2de92c6f)
    F(w5 ,0x4a7484aa)
    F(w6 ,0x5cb0a9dc)
    F(w7 ,0x76f988da)
    F(w8 ,0x983e5152)
    F(w9 ,0xa831c66d)
```

```
    F(w10,0xb00327c8)
    F(w11,0xbf597fc7)
    F(w12,0xc6e00bf3)
    F(w13,0xd5a79147)
    F(w14,0x06ca6351)
    F(w15,0x14292967)

    EXPAND

    F(w0 ,0x27b70a85)
    F(w1 ,0x2e1b2138)
    F(w2 ,0x4d2c6dfc)
    F(w3 ,0x53380d13)
    F(w4 ,0x650a7354)
    F(w5 ,0x766a0abb)
    F(w6 ,0x81c2c92e)
    F(w7 ,0x92722c85)
    F(w8 ,0xa2bfe8a1)
    F(w9 ,0xa81a664b)
    F(w10,0xc24b8b70)
    F(w11,0xc76c51a3)
    F(w12,0xd192e819)
    F(w13,0xd6990624)
    F(w14,0xf40e3585)
    F(w15,0x106aa070)

    EXPAND

    F(w0 ,0x19a4c116)
    F(w1 ,0x1e376c08)
    F(w2 ,0x2748774c)
    F(w3 ,0x34b0bcb5)
    F(w4 ,0x391c0cb3)
    F(w5 ,0x4ed8aa4a)
    F(w6 ,0x5b9cca4f)
    F(w7 ,0x682e6ff3)
    F(w8 ,0x748f82ee)
    F(w9 ,0x78a5636f)
    F(w10,0x84c87814)
    F(w11,0x8cc70208)
    F(w12,0x90befffa)
    F(w13,0xa4506ceb)
    F(w14,0xbef9a3f7)
    F(w15,0xc67178f2)

    a += state[0];
    b += state[1];
    c += state[2];
    d += state[3];
    e += state[4];
    f += state[5];
    g += state[6];
    h += state[7];

    state[0] = a;
    state[1] = b;
    state[2] = c;
    state[3] = d;
    state[4] = e;
    state[5] = f;
    state[6] = g;
    state[7] = h;

    in += 64;
    inlen -= 64;
  }

  store_bigendian(statebytes +  0,state[0]);
```

```
  store_bigendian(statebytes +  4,state[1]);
  store_bigendian(statebytes +  8,state[2]);
  store_bigendian(statebytes + 12,state[3]);
  store_bigendian(statebytes + 16,state[4]);
  store_bigendian(statebytes + 20,state[5]);
  store_bigendian(statebytes + 24,state[6]);
  store_bigendian(statebytes + 28,state[7]);

  return inlen;
}

#define blocks crypto_hashblocks_sha256

static const uint8_t iv[32] = {
  0x6a,0x09,0xe6,0x67,
  0xbb,0x67,0xae,0x85,
  0x3c,0x6e,0xf3,0x72,
  0xa5,0x4f,0xf5,0x3a,
  0x51,0x0e,0x52,0x7f,
  0x9b,0x05,0x68,0x8c,
  0x1f,0x83,0xd9,0xab,
  0x5b,0xe0,0xcd,0x19,
} ;

void sha256(uint8_t out[32],const uint8_t *in,size_t inlen)
{
  uint8_t h[32];
  uint8_t padded[128];
  unsigned int i;
  uint64_t bits = inlen << 3;

  for (i = 0;i < 32;++i) h[i] = iv[i];

  blocks(h,in,inlen);
  in += inlen;
  inlen &= 63;
  in -= inlen;

  for (i = 0;i < inlen;++i) padded[i] = in[i];
  padded[inlen] = 0x80;

  if (inlen < 56) {
    for (i = inlen + 1;i < 56;++i) padded[i] = 0;
    padded[56] = bits >> 56;
    padded[57] = bits >> 48;
    padded[58] = bits >> 40;
    padded[59] = bits >> 32;
    padded[60] = bits >> 24;
    padded[61] = bits >> 16;
    padded[62] = bits >> 8;
    padded[63] = bits;
    blocks(h,padded,64);
  } else {
    for (i = inlen + 1;i < 120;++i) padded[i] = 0;
    padded[120] = bits >> 56;
    padded[121] = bits >> 48;
    padded[122] = bits >> 40;
    padded[123] = bits >> 32;
    padded[124] = bits >> 24;
    padded[125] = bits >> 16;
    padded[126] = bits >> 8;
    padded[127] = bits;
    blocks(h,padded,128);
  }

  for (i = 0;i < 32;++i) out[i] = h[i];
}
```

```c
/*
Adapted from public domain code by D. J. Bernstein.
*/

#include <stddef.h>
#include <stdint.h>
#include "sha2.h"

static uint64_t load_bigendian(const uint8_t *x)
{
  return
      (uint64_t) (x[7]) \
    | (((uint64_t) (x[6])) << 8) \
    | (((uint64_t) (x[5])) << 16) \
    | (((uint64_t) (x[4])) << 24) \
    | (((uint64_t) (x[3])) << 32) \
    | (((uint64_t) (x[2])) << 40) \
    | (((uint64_t) (x[1])) << 48) \
    | (((uint64_t) (x[0])) << 56)
  ;
}

static void store_bigendian(uint8_t *x,uint64_t u)
{
  x[7] = u; u >>= 8;
  x[6] = u; u >>= 8;
  x[5] = u; u >>= 8;
  x[4] = u; u >>= 8;
  x[3] = u; u >>= 8;
  x[2] = u; u >>= 8;
  x[1] = u; u >>= 8;
  x[0] = u;
}

#define SHR(x,c) ((x) >> (c))
#define ROTR(x,c) (((x) >> (c)) | ((x) << (64 - (c))))

#define Ch(x,y,z) ((x & y) ^ (~x & z))
#define Maj(x,y,z) ((x & y) ^ (x & z) ^ (y & z))
#define Sigma0(x) (ROTR(x,28) ^ ROTR(x,34) ^ ROTR(x,39))
#define Sigma1(x) (ROTR(x,14) ^ ROTR(x,18) ^ ROTR(x,41))
#define sigma0(x) (ROTR(x, 1) ^ ROTR(x, 8) ^ SHR(x,7))
#define sigma1(x) (ROTR(x,19) ^ ROTR(x,61) ^ SHR(x,6))

#define M(w0,w14,w9,w1) w0 = sigma1(w14) + w9 + sigma0(w1) + w0;

#define EXPAND \
  M(w0 ,w14,w9 ,w1 ) \
  M(w1 ,w15,w10,w2 ) \
  M(w2 ,w0 ,w11,w3 ) \
  M(w3 ,w1 ,w12,w4 ) \
  M(w4 ,w2 ,w13,w5 ) \
  M(w5 ,w3 ,w14,w6 ) \
  M(w6 ,w4 ,w15,w7 ) \
  M(w7 ,w5 ,w0 ,w8 ) \
  M(w8 ,w6 ,w1 ,w9 ) \
  M(w9 ,w7 ,w2 ,w10) \
  M(w10,w8 ,w3 ,w11) \
  M(w11,w9 ,w4 ,w12) \
  M(w12,w10,w5 ,w13) \
  M(w13,w11,w6 ,w14) \
  M(w14,w12,w7 ,w15) \
  M(w15,w13,w8 ,w0 )

#define F(w,k) \
  T1 = h + Sigma1(e) + Ch(e,f,g) + k + w; \
  T2 = Sigma0(a) + Maj(a,b,c); \
  h = g; \
```

```c
  g = f; \
  f = e; \
  e = d + T1; \
  d = c; \
  c = b; \
  b = a; \
  a = T1 + T2;

static int crypto_hashblocks_sha512(uint8_t *statebytes,const uint8_t *in,size_t inlen)
{
  uint64_t state[8];
  uint64_t a;
  uint64_t b;
  uint64_t c;
  uint64_t d;
  uint64_t e;
  uint64_t f;
  uint64_t g;
  uint64_t h;
  uint64_t T1;
  uint64_t T2;

  a = load_bigendian(statebytes +  0); state[0] = a;
  b = load_bigendian(statebytes +  8); state[1] = b;
  c = load_bigendian(statebytes + 16); state[2] = c;
  d = load_bigendian(statebytes + 24); state[3] = d;
  e = load_bigendian(statebytes + 32); state[4] = e;
  f = load_bigendian(statebytes + 40); state[5] = f;
  g = load_bigendian(statebytes + 48); state[6] = g;
  h = load_bigendian(statebytes + 56); state[7] = h;

  while (inlen >= 128) {
    uint64_t w0  = load_bigendian(in +   0);
    uint64_t w1  = load_bigendian(in +   8);
    uint64_t w2  = load_bigendian(in +  16);
    uint64_t w3  = load_bigendian(in +  24);
    uint64_t w4  = load_bigendian(in +  32);
    uint64_t w5  = load_bigendian(in +  40);
    uint64_t w6  = load_bigendian(in +  48);
    uint64_t w7  = load_bigendian(in +  56);
    uint64_t w8  = load_bigendian(in +  64);
    uint64_t w9  = load_bigendian(in +  72);
    uint64_t w10 = load_bigendian(in +  80);
    uint64_t w11 = load_bigendian(in +  88);
    uint64_t w12 = load_bigendian(in +  96);
    uint64_t w13 = load_bigendian(in + 104);
    uint64_t w14 = load_bigendian(in + 112);
    uint64_t w15 = load_bigendian(in + 120);

    F(w0 ,0x428a2f98d728ae22ULL)
    F(w1 ,0x7137449123ef65cdULL)
    F(w2 ,0xb5c0fbcfec4d3b2fULL)
    F(w3 ,0xe9b5dba58189dbbcULL)
    F(w4 ,0x3956c25bf348b538ULL)
    F(w5 ,0x59f111f1b605d019ULL)
    F(w6 ,0x923f82a4af194f9bULL)
    F(w7 ,0xab1c5ed5da6d8118ULL)
    F(w8 ,0xd807aa98a3030242ULL)
    F(w9 ,0x12835b0145706fbeULL)
    F(w10,0x243185be4ee4b28cULL)
    F(w11,0x550c7dc3d5ffb4e2ULL)
    F(w12,0x72be5d74f27b896fULL)
    F(w13,0x80deb1fe3b1696b1ULL)
    F(w14,0x9bdc06a725c71235ULL)
    F(w15,0xc19bf174cf692694ULL)

    EXPAND
```

```
    F(w0 ,0xe49b69c19ef14ad2ULL)
    F(w1 ,0xefbe4786384f25e3ULL)
    F(w2 ,0x0fc19dc68b8cd5b5ULL)
    F(w3 ,0x240ca1cc77ac9c65ULL)
    F(w4 ,0x2de92c6f592b0275ULL)
    F(w5 ,0x4a7484aa6ea6e483ULL)
    F(w6 ,0x5cb0a9dcbd41fbd4ULL)
    F(w7 ,0x76f988da831153b5ULL)
    F(w8 ,0x983e5152ee66dfabULL)
    F(w9 ,0xa831c66d2db43210ULL)
    F(w10,0xb00327c898fb213fULL)
    F(w11,0xbf597fc7beef0ee4ULL)
    F(w12,0xc6e00bf33da88fc2ULL)
    F(w13,0xd5a79147930aa725ULL)
    F(w14,0x06ca6351e003826fULL)
    F(w15,0x142929670a0e6e70ULL)

    EXPAND

    F(w0 ,0x27b70a8546d22ffcULL)
    F(w1 ,0x2e1b21385c26c926ULL)
    F(w2 ,0x4d2c6dfc5ac42aedULL)
    F(w3 ,0x53380d139d95b3dfULL)
    F(w4 ,0x650a73548baf63deULL)
    F(w5 ,0x766a0abb3c77b2a8ULL)
    F(w6 ,0x81c2c92e47edaee6ULL)
    F(w7 ,0x92722c851482353bULL)
    F(w8 ,0xa2bfe8a14cf10364ULL)
    F(w9 ,0xa81a664bbc423001ULL)
    F(w10,0xc24b8b70d0f89791ULL)
    F(w11,0xc76c51a30654be30ULL)
    F(w12,0xd192e819d6ef5218ULL)
    F(w13,0xd69906245565a910ULL)
    F(w14,0xf40e35855771202aULL)
    F(w15,0x106aa07032bbd1b8ULL)

    EXPAND

    F(w0 ,0x19a4c116b8d2d0c8ULL)
    F(w1 ,0x1e376c085141ab53ULL)
    F(w2 ,0x2748774cdf8eeb99ULL)
    F(w3 ,0x34b0bcb5e19b48a8ULL)
    F(w4 ,0x391c0cb3c5c95a63ULL)
    F(w5 ,0x4ed8aa4ae3418acbULL)
    F(w6 ,0x5b9cca4f7763e373ULL)
    F(w7 ,0x682e6ff3d6b2b8a3ULL)
    F(w8 ,0x748f82ee5defb2fcULL)
    F(w9 ,0x78a5636f43172f60ULL)
    F(w10,0x84c87814a1f0ab72ULL)
    F(w11,0x8cc702081a6439ecULL)
    F(w12,0x90befffa23631e28ULL)
    F(w13,0xa4506cebde82bde9ULL)
    F(w14,0xbef9a3f7b2c67915ULL)
    F(w15,0xc67178f2e372532bULL)

    EXPAND

    F(w0 ,0xca273eceea26619cULL)
    F(w1 ,0xd186b8c721c0c207ULL)
    F(w2 ,0xeada7dd6cde0eb1eULL)
    F(w3 ,0xf57d4f7fee6ed178ULL)
    F(w4 ,0x06f067aa72176fbaULL)
    F(w5 ,0x0a637dc5a2c898a6ULL)
    F(w6 ,0x113f9804bef90daeULL)
    F(w7 ,0x1b710b35131c471bULL)
    F(w8 ,0x28db77f523047d84ULL)
    F(w9 ,0x32caab7b40c72493ULL)
    F(w10,0x3c9ebe0a15c9bebcULL)
```

```c
    F(w11,0x431d67c49c100d4cULL)
    F(w12,0x4cc5d4becb3e42b6ULL)
    F(w13,0x597f299cfc657e2aULL)
    F(w14,0x5fcb6fab3ad6faecULL)
    F(w15,0x6c44198c4a475817ULL)

    a += state[0];
    b += state[1];
    c += state[2];
    d += state[3];
    e += state[4];
    f += state[5];
    g += state[6];
    h += state[7];

    state[0] = a;
    state[1] = b;
    state[2] = c;
    state[3] = d;
    state[4] = e;
    state[5] = f;
    state[6] = g;
    state[7] = h;

    in += 128;
    inlen -= 128;
  }

  store_bigendian(statebytes +  0,state[0]);
  store_bigendian(statebytes +  8,state[1]);
  store_bigendian(statebytes + 16,state[2]);
  store_bigendian(statebytes + 24,state[3]);
  store_bigendian(statebytes + 32,state[4]);
  store_bigendian(statebytes + 40,state[5]);
  store_bigendian(statebytes + 48,state[6]);
  store_bigendian(statebytes + 56,state[7]);

  return inlen;
}

#define blocks crypto_hashblocks_sha512

static const uint8_t iv[64] = {
  0x6a,0x09,0xe6,0x67,0xf3,0xbc,0xc9,0x08,
  0xbb,0x67,0xae,0x85,0x84,0xca,0xa7,0x3b,
  0x3c,0x6e,0xf3,0x72,0xfe,0x94,0xf8,0x2b,
  0xa5,0x4f,0xf5,0x3a,0x5f,0x1d,0x36,0xf1,
  0x51,0x0e,0x52,0x7f,0xad,0xe6,0x82,0xd1,
  0x9b,0x05,0x68,0x8c,0x2b,0x3e,0x6c,0x1f,
  0x1f,0x83,0xd9,0xab,0xfb,0x41,0xbd,0x6b,
  0x5b,0xe0,0xcd,0x19,0x13,0x7e,0x21,0x79
} ;

void sha512(uint8_t out[64],const uint8_t *in,size_t inlen)
{
  uint8_t h[64];
  uint8_t padded[256];
  unsigned int i;
  uint64_t bytes = inlen;

  for (i = 0;i < 64;++i) h[i] = iv[i];

  blocks(h,in,inlen);
  in += inlen;
  inlen &= 127;
  in -= inlen;

  for (i = 0;i < inlen;++i) padded[i] = in[i];
```

```
  padded[inlen] = 0x80;

  if (inlen < 112) {
    for (i = inlen + 1;i < 119;++i) padded[i] = 0;
    padded[119] = bytes >> 61;
    padded[120] = bytes >> 53;
    padded[121] = bytes >> 45;
    padded[122] = bytes >> 37;
    padded[123] = bytes >> 29;
    padded[124] = bytes >> 21;
    padded[125] = bytes >> 13;
    padded[126] = bytes >> 5;
    padded[127] = bytes << 3;
    blocks(h,padded,128);
  } else {
    for (i = inlen + 1;i < 247;++i) padded[i] = 0;
    padded[247] = bytes >> 61;
    padded[248] = bytes >> 53;
    padded[249] = bytes >> 45;
    padded[250] = bytes >> 37;
    padded[251] = bytes >> 29;
    padded[252] = bytes >> 21;
    padded[253] = bytes >> 13;
    padded[254] = bytes >> 5;
    padded[255] = bytes << 3;
    blocks(h,padded,256);
  }

  for (i = 0;i < 64;++i) out[i] = h[i];
}
```

```c
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include "cpucycles.h"
#include "speed_print.h"

static int cmp_uint64(const void *a, const void *b) {
  if(*(uint64_t *)a < *(uint64_t *)b) return -1;
  if(*(uint64_t *)a > *(uint64_t *)b) return 1;
  return 0;
}

static uint64_t median(uint64_t *l, size_t llen) {
  qsort(l,llen,sizeof(uint64_t),cmp_uint64);

  if(llen%2) return l[llen/2];
  else return (l[llen/2-1]+l[llen/2])/2;
}

static uint64_t average(uint64_t *t, size_t tlen) {
  size_t i;
  uint64_t acc=0;

  for(i=0;i<tlen;i++)
    acc += t[i];

  return acc/tlen;
}

void print_results(const char *s, uint64_t *t, size_t tlen) {
  size_t i;
  static uint64_t overhead = -1;

  if(tlen < 2) {
    fprintf(stderr, "ERROR: Need a least two cycle counts!\n");
    return;
  }

  if(overhead  == (uint64_t)-1)
    overhead = cpucycles_overhead();

  tlen--;
  for(i=0;i<tlen;++i)
    t[i] = t[i+1] - t[i] - overhead;

  printf("%s\n", s);
  printf("median: %llu cycles/ticks\n", (unsigned long long)median(t, tlen));
  printf("average: %llu cycles/ticks\n", (unsigned long long)average(t, tlen));
  printf("\n");
}
```

```c
#include <stddef.h>
#include <stdint.h>
#include "params.h"
#include "symmetric.h"
#include "aes256ctr.h"

void kyber_aes256xof_absorb(aes256ctr_ctx *state, const uint8_t seed[32], uint8_t x, uint8_
t y)
{
  uint8_t expnonce[12] = {0};
  expnonce[0] = x;
  expnonce[1] = y;
  aes256ctr_init(state, seed, expnonce);
}

void kyber_aes256ctr_prf(uint8_t *out, size_t outlen, const uint8_t key[32], uint8_t nonce)
{
  uint8_t expnonce[12] = {0};
  expnonce[0] = nonce;
  aes256ctr_prf(out, outlen, key, expnonce);
}
```

```c
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include "params.h"
#include "symmetric.h"
#include "fips202.h"

/*************************************************
* Name:        kyber_shake128_absorb
*
* Description: Absorb step of the SHAKE128 specialized for the Kyber context.
*
* Arguments:   - keccak_state *state: pointer to (uninitialized) output Keccak state
*              - const uint8_t *seed: pointer to KYBER_SYMBYTES input to be absorbed into s
tate
*              - uint8_t i: additional byte of input
*              - uint8_t j: additional byte of input
**************************************************/
void kyber_shake128_absorb(keccak_state *state,
                           const uint8_t seed[KYBER_SYMBYTES],
                           uint8_t x,
                           uint8_t y)
{
  uint8_t extseed[KYBER_SYMBYTES+2];

  memcpy(extseed, seed, KYBER_SYMBYTES);
  extseed[KYBER_SYMBYTES+0] = x;
  extseed[KYBER_SYMBYTES+1] = y;

  shake128_absorb_once(state, extseed, sizeof(extseed));
}

/*************************************************
* Name:        kyber_shake256_prf
*
* Description: Usage of SHAKE256 as a PRF, concatenates secret and public input
*              and then generates outlen bytes of SHAKE256 output
*
* Arguments:   - uint8_t *out: pointer to output
*              - size_t outlen: number of requested output bytes
*              - const uint8_t *key: pointer to the key (of length KYBER_SYMBYTES)
*              - uint8_t nonce: single-byte nonce (public PRF input)
**************************************************/
void kyber_shake256_prf(uint8_t *out, size_t outlen, const uint8_t key[KYBER_SYMBYTES], uin
t8_t nonce)
{
  uint8_t extkey[KYBER_SYMBYTES+1];

  memcpy(extkey, key, KYBER_SYMBYTES);
  extkey[KYBER_SYMBYTES] = nonce;

  shake256(out, outlen, extkey, sizeof(extkey));
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include "kem.h"
#include "kex.h"

int main(void)
{
  uint8_t pkb[CRYPTO_PUBLICKEYBYTES];
  uint8_t skb[CRYPTO_SECRETKEYBYTES];

  uint8_t pka[CRYPTO_PUBLICKEYBYTES];
  uint8_t ska[CRYPTO_SECRETKEYBYTES];

  uint8_t eska[CRYPTO_SECRETKEYBYTES];

  uint8_t uake_senda[KEX_UAKE_SENDABYTES];
  uint8_t uake_sendb[KEX_UAKE_SENDBBYTES];

  uint8_t ake_senda[KEX_AKE_SENDABYTES];
  uint8_t ake_sendb[KEX_AKE_SENDBBYTES];

  uint8_t tk[KEX_SSBYTES];
  uint8_t ka[KEX_SSBYTES];
  uint8_t kb[KEX_SSBYTES];
  uint8_t zero[KEX_SSBYTES];
  int i;

  for(i=0;i<KEX_SSBYTES;i++)
    zero[i] = 0;

  crypto_kem_keypair(pkb, skb); // Generate static key for Bob

  crypto_kem_keypair(pka, ska); // Generate static key for Alice


  // Perform unilaterally authenticated key exchange

  kex_uake_initA(uake_senda, tk, eska, pkb); // Run by Alice

  kex_uake_sharedB(uake_sendb, kb, uake_senda, skb); // Run by Bob

  kex_uake_sharedA(ka, uake_sendb, tk, eska); // Run by Alice

  if(memcmp(ka,kb,KEX_SSBYTES))
    printf("Error in UAKE\n");

  if(!memcmp(ka,zero,KEX_SSBYTES))
    printf("Error: UAKE produces zero key\n");

  // Perform mutually authenticated key exchange

  kex_ake_initA(ake_senda, tk, eska, pkb); // Run by Alice

  kex_ake_sharedB(ake_sendb, kb, ake_senda, skb, pka); // Run by Bob

  kex_ake_sharedA(ka, ake_sendb, tk, eska, ska); // Run by Alice

  if(memcmp(ka,kb,KEX_SSBYTES))
    printf("Error in AKE\n");

  if(!memcmp(ka,zero,KEX_SSBYTES))
    printf("Error: AKE produces zero key\n");


  printf("KEX_UAKE_SENDABYTES: %d\n",KEX_UAKE_SENDABYTES);
  printf("KEX_UAKE_SENDBBYTES: %d\n",KEX_UAKE_SENDBBYTES);
```

```
  printf("KEX_AKE_SENDABYTES: %d\n",KEX_AKE_SENDABYTES);
  printf("KEX_AKE_SENDBBYTES: %d\n",KEX_AKE_SENDBBYTES);

  return 0;
}
```

```c
#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include "kem.h"
#include "randombytes.h"

#define NTESTS 1000

static int test_keys(void)
{
  uint8_t pk[CRYPTO_PUBLICKEYBYTES];
  uint8_t sk[CRYPTO_SECRETKEYBYTES];
  uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
  uint8_t key_a[CRYPTO_BYTES];
  uint8_t key_b[CRYPTO_BYTES];

  //Alice generates a public key
  crypto_kem_keypair(pk, sk);

  //Bob derives a secret key and creates a response
  crypto_kem_enc(ct, key_b, pk);

  //Alice uses Bobs response to get her shared key
  crypto_kem_dec(key_a, ct, sk);

  if(memcmp(key_a, key_b, CRYPTO_BYTES)) {
    printf("ERROR keys\n");
    return 1;
  }

  return 0;
}

static int test_invalid_sk_a(void)
{
  uint8_t pk[CRYPTO_PUBLICKEYBYTES];
  uint8_t sk[CRYPTO_SECRETKEYBYTES];
  uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
  uint8_t key_a[CRYPTO_BYTES];
  uint8_t key_b[CRYPTO_BYTES];

  //Alice generates a public key
  crypto_kem_keypair(pk, sk);

  //Bob derives a secret key and creates a response
  crypto_kem_enc(ct, key_b, pk);

  //Replace secret key with random values
  randombytes(sk, CRYPTO_SECRETKEYBYTES);

  //Alice uses Bobs response to get her shared key
  crypto_kem_dec(key_a, ct, sk);

  if(!memcmp(key_a, key_b, CRYPTO_BYTES)) {
    printf("ERROR invalid sk\n");
    return 1;
  }

  return 0;
}

static int test_invalid_ciphertext(void)
{
  uint8_t pk[CRYPTO_PUBLICKEYBYTES];
  uint8_t sk[CRYPTO_SECRETKEYBYTES];
  uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
  uint8_t key_a[CRYPTO_BYTES];
  uint8_t key_b[CRYPTO_BYTES];
```

```c
  uint8_t b;
  size_t pos;

  do {
    randombytes(&b, sizeof(uint8_t));
  } while(!b);
  randombytes((uint8_t *)&pos, sizeof(size_t));

  //Alice generates a public key
  crypto_kem_keypair(pk, sk);

  //Bob derives a secret key and creates a response
  crypto_kem_enc(ct, key_b, pk);

  //Change some byte in the ciphertext (i.e., encapsulated key)
  ct[pos % CRYPTO_CIPHERTEXTBYTES] ^= b;

  //Alice uses Bobs response to get her shared key
  crypto_kem_dec(key_a, ct, sk);

  if(!memcmp(key_a, key_b, CRYPTO_BYTES)) {
    printf("ERROR invalid ciphertext\n");
    return 1;
  }

  return 0;
}

int main(void)
{
  unsigned int i;
  int r;

  for(i=0;i<NTESTS;i++) {
    r  = test_keys();
    r |= test_invalid_sk_a();
    r |= test_invalid_ciphertext();
    if(r)
      return 1;
  }

  printf("CRYPTO_SECRETKEYBYTES:  %d\n",CRYPTO_SECRETKEYBYTES);
  printf("CRYPTO_PUBLICKEYBYTES:  %d\n",CRYPTO_PUBLICKEYBYTES);
  printf("CRYPTO_CIPHERTEXTBYTES: %d\n",CRYPTO_CIPHERTEXTBYTES);

  return 0;
}
```

```c
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include "kem.h"
#include "kex.h"
#include "params.h"
#include "indcpa.h"
#include "polyvec.h"
#include "poly.h"
#include "cpucycles.h"
#include "speed_print.h"

#define NTESTS 1000

uint64_t t[NTESTS];
uint8_t seed[KYBER_SYMBYTES] = {0};

int main(void)
{
  unsigned int i;
  uint8_t pk[CRYPTO_PUBLICKEYBYTES];
  uint8_t sk[CRYPTO_SECRETKEYBYTES];
  uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
  uint8_t key[CRYPTO_BYTES];
  uint8_t kexsenda[KEX_AKE_SENDABYTES];
  uint8_t kexsendb[KEX_AKE_SENDBBYTES];
  uint8_t kexkey[KEX_SSBYTES];
  polyvec matrix[KYBER_K];
  poly ap;

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    gen_matrix(matrix, seed, 0);
  }
  print_results("gen_a: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    poly_getnoise_eta1(&ap, seed, 0);
  }
  print_results("poly_getnoise_eta1: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    poly_getnoise_eta2(&ap, seed, 0);
  }
  print_results("poly_getnoise_eta2: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    poly_ntt(&ap);
  }
  print_results("NTT: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    poly_invntt_tomont(&ap);
  }
  print_results("INVNTT: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    polyvec_basemul_acc_montgomery(&ap, &matrix[0], &matrix[1]);
  }
  print_results("polyvec_basemul_acc_montgomery: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
```

```
    t[i] = cpucycles();
    poly_tomsg(ct,&ap);
  }
  print_results("poly_tomsg: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    poly_frommsg(&ap,ct);
  }
  print_results("poly_frommsg: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    poly_compress(ct,&ap);
  }
  print_results("poly_compress: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    poly_decompress(&ap,ct);
  }
  print_results("poly_decompress: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    polyvec_compress(ct,&matrix[0]);
  }
  print_results("polyvec_compress: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    polyvec_decompress(&matrix[0],ct);
  }
  print_results("polyvec_decompress: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    indcpa_keypair(pk, sk);
  }
  print_results("indcpa_keypair: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    indcpa_enc(ct, key, pk, seed);
  }
  print_results("indcpa_enc: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    indcpa_dec(key, ct, sk);
  }
  print_results("indcpa_dec: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    crypto_kem_keypair(pk, sk);
  }
  print_results("kyber_keypair: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    crypto_kem_enc(ct, key, pk);
  }
  print_results("kyber_encaps: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    crypto_kem_dec(key, ct, sk);
```

```
  }
  print_results("kyber_decaps: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    kex_uake_initA(kexsenda, key, sk, pk);
  }
  print_results("kex_uake_initA: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    kex_uake_sharedB(kexsendb, kexkey, kexsenda, sk);
  }
  print_results("kex_uake_sharedB: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    kex_uake_sharedA(kexkey, kexsendb, key, sk);
  }
  print_results("kex_uake_sharedA: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    kex_ake_initA(kexsenda, key, sk, pk);
  }
  print_results("kex_ake_initA: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    kex_ake_sharedB(kexsendb, kexkey, kexsenda, sk, pk);
  }
  print_results("kex_ake_sharedB: ", t, NTESTS);

  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    kex_ake_sharedA(kexkey, kexsendb, key, sk, sk);
  }
  print_results("kex_ake_sharedA: ", t, NTESTS);

  return 0;
}
```

```c
/* Deterministic randombytes by Daniel J. Bernstein */
/* taken from SUPERCOP (https://bench.cr.yp.to)     */

#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include "kem.h"
#include "randombytes.h"

#define NTESTS 10000

static uint32_t seed[32] = {
  3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,8,4,6,2,6,4,3,3,8,3,2,7,9,5
};
static uint32_t in[12];
static uint32_t out[8];
static int outleft = 0;

#define ROTATE(x,b) (((x) << (b)) | ((x) >> (32 - (b))))
#define MUSH(i,b) x = t[i] += (((x ^ seed[i]) + sum) ^ ROTATE(x,b));

static void surf(void)
{
  uint32_t t[12]; uint32_t x; uint32_t sum = 0;
  int r; int i; int loop;

  for (i = 0;i < 12;++i) t[i] = in[i] ^ seed[12 + i];
  for (i = 0;i < 8;++i) out[i] = seed[24 + i];
  x = t[11];
  for (loop = 0;loop < 2;++loop) {
    for (r = 0;r < 16;++r) {
      sum += 0x9e3779b9;
      MUSH(0,5) MUSH(1,7) MUSH(2,9) MUSH(3,13)
      MUSH(4,5) MUSH(5,7) MUSH(6,9) MUSH(7,13)
      MUSH(8,5) MUSH(9,7) MUSH(10,9) MUSH(11,13)
    }
    for (i = 0;i < 8;++i) out[i] ^= t[i + 4];
  }
}

void randombytes(uint8_t *x,size_t xlen)
{
  while (xlen > 0) {
    if (!outleft) {
      if (!++in[0]) if (!++in[1]) if (!++in[2]) ++in[3];
      surf();
      outleft = 8;
    }
    *x = out[--outleft];
    printf("%02x", *x);
    ++x;
    --xlen;
  }
  printf("\n");
}

int main(void)
{
  unsigned int i,j;
  uint8_t pk[CRYPTO_PUBLICKEYBYTES];
  uint8_t sk[CRYPTO_SECRETKEYBYTES];
  uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
  uint8_t key_a[CRYPTO_BYTES];
  uint8_t key_b[CRYPTO_BYTES];

  for(i=0;i<NTESTS;i++) {
    // Key-pair generation
    crypto_kem_keypair(pk, sk);
```

```c
    printf("Public Key: ");
    for(j=0;j<CRYPTO_PUBLICKEYBYTES;j++)
      printf("%02x",pk[j]);
    printf("\n");
    printf("Secret Key: ");
    for(j=0;j<CRYPTO_SECRETKEYBYTES;j++)
      printf("%02x",sk[j]);
    printf("\n");

    // Encapsulation
    crypto_kem_enc(ct, key_b, pk);
    printf("Ciphertext: ");
    for(j=0;j<CRYPTO_CIPHERTEXTBYTES;j++)
      printf("%02x",ct[j]);
    printf("\n");
    printf("Shared Secret B: ");
    for(j=0;j<CRYPTO_BYTES;j++)
      printf("%02x",key_b[j]);
    printf("\n");

    // Decapsulation
    crypto_kem_dec(key_a, ct, sk);
    printf("Shared Secret A: ");
    for(j=0;j<CRYPTO_BYTES;j++)
      printf("%02x",key_a[j]);
    printf("\n");

    for(j=0;j<CRYPTO_BYTES;j++) {
      if(key_a[j] != key_b[j]) {
        fprintf(stderr, "ERROR\n");
        return -1;
      }
    }
  }

  return 0;
}
```

```c
#include <stddef.h>
#include <stdint.h>
#include "verify.h"

/*************************************************
* Name:        verify
*
* Description: Compare two arrays for equality in constant time.
*
* Arguments:   const uint8_t *a: pointer to first byte array
*              const uint8_t *b: pointer to second byte array
*              size_t len:       length of the byte arrays
*
* Returns 0 if the byte arrays are equal, 1 otherwise
**************************************************/
int verify(const uint8_t *a, const uint8_t *b, size_t len)
{
  size_t i;
  uint8_t r = 0;

  for(i=0;i<len;i++)
    r |= a[i] ^ b[i];

  return (-(uint64_t)r) >> 63;
}

/*************************************************
* Name:        cmov
*
* Description: Copy len bytes from x to r if b is 1;
*              don't modify x if b is 0. Requires b to be in {0,1};
*              assumes two's complement representation of negative integers.
*              Runs in constant time.
*
* Arguments:   uint8_t *r:       pointer to output byte array
*              const uint8_t *x: pointer to input byte array
*              size_t len:       Amount of bytes to be copied
*              uint8_t b:        Condition bit; has to be in {0,1}
**************************************************/
void cmov(uint8_t *r, const uint8_t *x, size_t len, uint8_t b)
{
  size_t i;

#if defined(__GNUC__) || defined(__clang__)
  // Prevent the compiler from
  //    1) inferring that b is 0/1-valued, and
  //    2) handling the two cases with a branch.
  // This is not necessary when verify.c and kem.c are separate translation
  // units, but we expect that downstream consumers will copy this code and/or
  // change how it is built.
  __asm__("" : "+r"(b) : /* no inputs */);
#endif

  b = -b;
  for(i=0;i<len;i++)
    r[i] ^= b & (r[i] ^ x[i]);
}
```