```c
/* Based heavily on public-domain code by Romain Dolbeau
 * Different handling of nonce+counter than original version using
 * separated 64-bit nonce and internal 64-bit counter, starting from zero
 * Public Domain */

#include <stddef.h>
#include <stdint.h>
#include <immintrin.h>
#include "aes256ctr.h"

static inline void aesni_encrypt4(uint8_t out[64], __m128i *n, const __m128i rkeys[16])
{
  __m128i f,f0,f1,f2,f3;
  const __m128i idx = _mm_set_epi8(8,9,10,11,12,13,14,15,7,6,5,4,3,2,1,0);

  /* Load current counter value */
  f = _mm_load_si128(n);

  /* Increase counter in 4 consecutive blocks */
  f0 = _mm_shuffle_epi8(_mm_add_epi64(f,_mm_set_epi64x(0,0)),idx);
  f1 = _mm_shuffle_epi8(_mm_add_epi64(f,_mm_set_epi64x(1,0)),idx);
  f2 = _mm_shuffle_epi8(_mm_add_epi64(f,_mm_set_epi64x(2,0)),idx);
  f3 = _mm_shuffle_epi8(_mm_add_epi64(f,_mm_set_epi64x(3,0)),idx);

  /* Write counter for next iteration, increased by 4 */
  _mm_store_si128(n,_mm_add_epi64(f,_mm_set_epi64x(4,0)));

  /* Actual AES encryption, 4x interleaved */
  f  = _mm_load_si128(&rkeys[0]);
  f0 = _mm_xor_si128(f0,f);
  f1 = _mm_xor_si128(f1,f);
  f2 = _mm_xor_si128(f2,f);
  f3 = _mm_xor_si128(f3,f);

  for (int i = 1; i < 14; i++) {
    f  = _mm_load_si128(&rkeys[i]);
    f0 = _mm_aesenc_si128(f0,f);
    f1 = _mm_aesenc_si128(f1,f);
    f2 = _mm_aesenc_si128(f2,f);
    f3 = _mm_aesenc_si128(f3,f);
  }

  f  = _mm_load_si128(&rkeys[14]);
  f0 = _mm_aesenclast_si128(f0,f);
  f1 = _mm_aesenclast_si128(f1,f);
  f2 = _mm_aesenclast_si128(f2,f);
  f3 = _mm_aesenclast_si128(f3,f);

  /* Write results */
  _mm_storeu_si128((__m128i*)(out+ 0),f0);
  _mm_storeu_si128((__m128i*)(out+16),f1);
  _mm_storeu_si128((__m128i*)(out+32),f2);
  _mm_storeu_si128((__m128i*)(out+48),f3);
}

void aes256ctr_init(aes256ctr_ctx *state, const uint8_t key[32], uint64_t nonce)
{
  __m128i key0, key1, temp0, temp1, temp2, temp4;
  int idx = 0;

  key0 = _mm_loadu_si128((__m128i *)(key+ 0));
  key1 = _mm_loadu_si128((__m128i *)(key+16));
  state->n = _mm_loadl_epi64((__m128i *)&nonce);

  state->rkeys[idx++] = key0;
  temp0 = key0;
  temp2 = key1;
  temp4 = _mm_setzero_si128();
```

```c
#define BLOCK1(IMM)                                                      \
  temp1 = _mm_aeskeygenassist_si128(temp2, IMM);                         \
  state->rkeys[idx++] = temp2;                                           \
  temp4 = (__m128i)_mm_shuffle_ps((__m128)temp4, (__m128)temp0, 0x10);   \
  temp0 = _mm_xor_si128(temp0, temp4);                                   \
  temp4 = (__m128i)_mm_shuffle_ps((__m128)temp4, (__m128)temp0, 0x8c);   \
  temp0 = _mm_xor_si128(temp0, temp4);                                   \
  temp1 = (__m128i)_mm_shuffle_ps((__m128)temp1, (__m128)temp1, 0xff);   \
  temp0 = _mm_xor_si128(temp0, temp1)

#define BLOCK2(IMM)                                                      \
  temp1 = _mm_aeskeygenassist_si128(temp0, IMM);                         \
  state->rkeys[idx++] = temp0;                                           \
  temp4 = (__m128i)_mm_shuffle_ps((__m128)temp4, (__m128)temp2, 0x10);   \
  temp2 = _mm_xor_si128(temp2, temp4);                                   \
  temp4 = (__m128i)_mm_shuffle_ps((__m128)temp4, (__m128)temp2, 0x8c);   \
  temp2 = _mm_xor_si128(temp2, temp4);                                   \
  temp1 = (__m128i)_mm_shuffle_ps((__m128)temp1, (__m128)temp1, 0xaa);   \
  temp2 = _mm_xor_si128(temp2, temp1)

  BLOCK1(0x01);
  BLOCK2(0x01);

  BLOCK1(0x02);
  BLOCK2(0x02);

  BLOCK1(0x04);
  BLOCK2(0x04);

  BLOCK1(0x08);
  BLOCK2(0x08);

  BLOCK1(0x10);
  BLOCK2(0x10);

  BLOCK1(0x20);
  BLOCK2(0x20);

  BLOCK1(0x40);
  state->rkeys[idx++] = temp0;
}

void aes256ctr_squeezeblocks(uint8_t *out,
                             size_t nblocks,
                             aes256ctr_ctx *state)
{
  size_t i;
  for(i=0;i<nblocks;i++) {
    aesni_encrypt4(out, &state->n, state->rkeys);
    out += 64;
  }
}

void aes256ctr_prf(uint8_t *out,
                   size_t outlen,
                   const uint8_t seed[32],
                   uint64_t nonce)
{
  unsigned int i;
  uint8_t buf[64];
  aes256ctr_ctx state;

  aes256ctr_init(&state, seed, nonce);

  while(outlen >= 64) {
    aesni_encrypt4(out, &state.n, state.rkeys);
    outlen -= 64;
```

```
    out += 64;
  }

  if(outlen) {
    aesni_encrypt4(buf, &state.n, state.rkeys);
    for(i=0;i<outlen;i++)
      out[i] = buf[i];
  }
}
```

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

/*************************************************
* 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 __m256i *buf: pointer to aligned input byte array
**************************************************/
static void cbd2(poly * restrict r, const __m256i buf[2*KYBER_N/128])
{
  unsigned int i;
  __m256i f0, f1, f2, f3;
  const __m256i mask55 = _mm256_set1_epi32(0x55555555);
  const __m256i mask33 = _mm256_set1_epi32(0x33333333);
  const __m256i mask03 = _mm256_set1_epi32(0x03030303);
  const __m256i mask0F = _mm256_set1_epi32(0x0F0F0F0F);

  for(i = 0; i < KYBER_N/64; i++) {
    f0 = _mm256_load_si256(&buf[i]);

    f1 = _mm256_srli_epi16(f0, 1);
    f0 = _mm256_and_si256(mask55, f0);
    f1 = _mm256_and_si256(mask55, f1);
    f0 = _mm256_add_epi8(f0, f1);

    f1 = _mm256_srli_epi16(f0, 2);
    f0 = _mm256_and_si256(mask33, f0);
    f1 = _mm256_and_si256(mask33, f1);
    f0 = _mm256_add_epi8(f0, mask33);
    f0 = _mm256_sub_epi8(f0, f1);

    f1 = _mm256_srli_epi16(f0, 4);
    f0 = _mm256_and_si256(mask0F, f0);
    f1 = _mm256_and_si256(mask0F, f1);
    f0 = _mm256_sub_epi8(f0, mask03);
    f1 = _mm256_sub_epi8(f1, mask03);

    f2 = _mm256_unpacklo_epi8(f0, f1);
    f3 = _mm256_unpackhi_epi8(f0, f1);

    f0 = _mm256_cvtepi8_epi16(_mm256_castsi256_si128(f2));
    f1 = _mm256_cvtepi8_epi16(_mm256_extracti128_si256(f2,1));
    f2 = _mm256_cvtepi8_epi16(_mm256_castsi256_si128(f3));
    f3 = _mm256_cvtepi8_epi16(_mm256_extracti128_si256(f3,1));

    _mm256_store_si256(&r->vec[4*i+0], f0);
    _mm256_store_si256(&r->vec[4*i+1], f2);
    _mm256_store_si256(&r->vec[4*i+2], f1);
    _mm256_store_si256(&r->vec[4*i+3], f3);
  }
}

#if KYBER_ETA1 == 3
/*************************************************
* 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 __m256i *buf: pointer to aligned input byte array
**************************************************/
static void cbd3(poly * restrict r, const uint8_t buf[3*KYBER_N/4+8])
{
  unsigned int i;
  __m256i f0, f1, f2, f3;
  const __m256i mask249 = _mm256_set1_epi32(0x249249);
  const __m256i mask6DB = _mm256_set1_epi32(0x6DB6DB);
  const __m256i mask07 = _mm256_set1_epi32(7);
  const __m256i mask70 = _mm256_set1_epi32(7 << 16);
  const __m256i mask3 = _mm256_set1_epi16(3);
  const __m256i shufbidx = _mm256_set_epi8(-1,15,14,13,-1,12,11,10,-1, 9, 8, 7,-1, 6, 5, 4,
                                           -1,11,10, 9,-1, 8, 7, 6,-1, 5, 4, 3,-1, 2, 1, 0)
;

  for(i = 0; i < KYBER_N/32; i++) {
    f0 = _mm256_loadu_si256((__m256i *)&buf[24*i]);
    f0 = _mm256_permute4x64_epi64(f0,0x94);
    f0 = _mm256_shuffle_epi8(f0,shufbidx);

    f1 = _mm256_srli_epi32(f0,1);
    f2 = _mm256_srli_epi32(f0,2);
    f0 = _mm256_and_si256(mask249,f0);
    f1 = _mm256_and_si256(mask249,f1);
    f2 = _mm256_and_si256(mask249,f2);
    f0 = _mm256_add_epi32(f0,f1);
    f0 = _mm256_add_epi32(f0,f2);

    f1 = _mm256_srli_epi32(f0,3);
    f0 = _mm256_add_epi32(f0,mask6DB);
    f0 = _mm256_sub_epi32(f0,f1);

    f1 = _mm256_slli_epi32(f0,10);
    f2 = _mm256_srli_epi32(f0,12);
    f3 = _mm256_srli_epi32(f0, 2);
    f0 = _mm256_and_si256(f0,mask07);
    f1 = _mm256_and_si256(f1,mask70);
    f2 = _mm256_and_si256(f2,mask07);
    f3 = _mm256_and_si256(f3,mask70);
    f0 = _mm256_add_epi16(f0,f1);
    f1 = _mm256_add_epi16(f2,f3);
    f0 = _mm256_sub_epi16(f0,mask3);
    f1 = _mm256_sub_epi16(f1,mask3);

    f2 = _mm256_unpacklo_epi32(f0,f1);
    f3 = _mm256_unpackhi_epi32(f0,f1);

    f0 = _mm256_permute2x128_si256(f2,f3,0x20);
    f1 = _mm256_permute2x128_si256(f2,f3,0x31);

    _mm256_store_si256(&r->vec[2*i+0], f0);
    _mm256_store_si256(&r->vec[2*i+1], f1);
  }
}
#endif

/* buf 32 bytes longer for cbd3 */
void poly_cbd_eta1(poly *r, const __m256i buf[KYBER_ETA1*KYBER_N/128+1])
{
#if KYBER_ETA1 == 2
  cbd2(r, buf);
#elif KYBER_ETA1 == 3
  cbd3(r, (uint8_t *)buf);
#else
#error "This implementation requires eta1 in {2,3}"
#endif
}
```

```
void poly_cbd_eta2(poly *r, const __m256i buf[KYBER_ETA2*KYBER_N/128])
{
#if KYBER_ETA2 == 2
  cbd2(r, buf);
#else
#error "This implementation requires eta2 = 2"
#endif
}
```

```c
#include "align.h"
#include "params.h"
#include "consts.h"

#define Q KYBER_Q
#define MONT -1044 // 2^16 mod q
#define QINV -3327 // q^-1 mod 2^16
#define V 20159 // floor(2^26/q + 0.5)
#define FHI 1441 // mont^2/128
#define FLO -10079 // qinv*FHI
#define MONTSQHI 1353 // mont^2
#define MONTSQLO 20553 // qinv*MONTSQHI
#define MASK 4095
#define SHIFT 32

const qdata_t qdata = {{
#define _16XQ 0
  Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q,

#define _16XQINV 16
  QINV, QINV, QINV, QINV, QINV, QINV, QINV, QINV,
  QINV, QINV, QINV, QINV, QINV, QINV, QINV, QINV,

#define _16XV 32
  V, V, V, V, V, V, V, V, V, V, V, V, V, V, V, V,

#define _16XFLO 48
  FLO, FLO, FLO, FLO, FLO, FLO, FLO, FLO,
  FLO, FLO, FLO, FLO, FLO, FLO, FLO, FLO,

#define _16XFHI 64
  FHI, FHI, FHI, FHI, FHI, FHI, FHI, FHI,
  FHI, FHI, FHI, FHI, FHI, FHI, FHI, FHI,

#define _16XMONTSQLO 80
  MONTSQLO, MONTSQLO, MONTSQLO, MONTSQLO,
  MONTSQLO, MONTSQLO, MONTSQLO, MONTSQLO,
  MONTSQLO, MONTSQLO, MONTSQLO, MONTSQLO,
  MONTSQLO, MONTSQLO, MONTSQLO, MONTSQLO,

#define _16XMONTSQHI 96
  MONTSQHI, MONTSQHI, MONTSQHI, MONTSQHI,
  MONTSQHI, MONTSQHI, MONTSQHI, MONTSQHI,
  MONTSQHI, MONTSQHI, MONTSQHI, MONTSQHI,
  MONTSQHI, MONTSQHI, MONTSQHI, MONTSQHI,

#define _16XMASK 112
  MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK,
  MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK,

#define _REVIDXB 128
  3854, 3340, 2826, 2312, 1798, 1284, 770, 256,
  3854, 3340, 2826, 2312, 1798, 1284, 770, 256,

#define _REVIDXD 144
  7, 0, 6, 0, 5, 0, 4, 0, 3, 0, 2, 0, 1, 0, 0, 0,

#define _ZETAS_EXP 160
   31498,  31498,  31498,  31498,   -758,   -758,   -758,   -758,
    5237,   5237,   5237,   5237,   1397,   1397,   1397,   1397,
   14745,  14745,  14745,  14745,  14745,  14745,  14745,  14745,
   14745,  14745,  14745,  14745,  14745,  14745,  14745,  14745,
    -359,   -359,   -359,   -359,   -359,   -359,   -359,   -359,
    -359,   -359,   -359,   -359,   -359,   -359,   -359,   -359,
   13525,  13525,  13525,  13525,  13525,  13525,  13525,  13525,
  -12402, -12402, -12402, -12402, -12402, -12402, -12402, -12402,
    1493,   1493,   1493,   1493,   1493,   1493,   1493,   1493,
    1422,   1422,   1422,   1422,   1422,   1422,   1422,   1422,
```

```
  -20907, -20907, -20907, -20907,  27758,  27758,  27758,  27758,
   -3799,  -3799,  -3799,  -3799, -15690, -15690, -15690, -15690,
    -171,   -171,   -171,   -171,    622,    622,    622,    622,
    1577,   1577,   1577,   1577,    182,    182,    182,    182,
   -5827,  -5827,  17363,  17363, -26360, -26360, -29057, -29057,
    5571,   5571,  -1102,  -1102,  21438,  21438, -26242, -26242,
     573,    573,  -1325,  -1325,    264,    264,    383,    383,
    -829,   -829,   1458,   1458,  -1602,  -1602,   -130,   -130,
   -5689,  -6516,   1496,  30967, -23565,  20179,  20710,  25080,
  -12796,  26616,  16064, -12442,   9134,   -650, -25986,  27837,
    1223,    652,   -552,   1015,  -1293,   1491,   -282,  -1544,
     516,     -8,   -320,   -666,  -1618,  -1162,    126,   1469,
    -335, -11477, -32227,  20494, -27738,    945, -14883,   6182,
   32010,  10631,  29175, -28762, -18486,  17560, -14430,  -5276,
   -1103,    555,  -1251,   1550,    422,    177,   -291,   1574,
    -246,   1159,   -777,   -602,  -1590,   -872,    418,   -156,
   11182,  13387, -14233, -21655,  13131,  -4587,  23092,   5493,
  -32502,  30317, -18741,  12639,  20100,  18525,  19529, -12619,
     430,    843,    871,    105,    587,   -235,   -460,   1653,
     778,   -147,   1483,   1119,    644,    349,    329,    -75,
     787,    787,    787,    787,    787,    787,    787,    787,
     787,    787,    787,    787,    787,    787,    787,    787,
   -1517,  -1517,  -1517,  -1517,  -1517,  -1517,  -1517,  -1517,
   -1517,  -1517,  -1517,  -1517,  -1517,  -1517,  -1517,  -1517,
   28191,  28191,  28191,  28191,  28191,  28191,  28191,  28191,
  -16694, -16694, -16694, -16694, -16694, -16694, -16694, -16694,
     287,    287,    287,    287,    287,    287,    287,    287,
     202,    202,    202,    202,    202,    202,    202,    202,
   10690,  10690,  10690,  10690,   1358,   1358,   1358,   1358,
  -11202, -11202, -11202, -11202,  31164,  31164,  31164,  31164,
     962,    962,    962,    962,  -1202,  -1202,  -1202,  -1202,
   -1474,  -1474,  -1474,  -1474,   1468,   1468,   1468,   1468,
  -28073, -28073,  24313,  24313, -10532, -10532,   8800,   8800,
   18426,  18426,   8859,   8859,  26675,  26675, -16163, -16163,
    -681,   -681,   1017,   1017,    732,    732,    608,    608,
   -1542,  -1542,    411,    411,   -205,   -205,  -1571,  -1571,
   19883, -28250, -15887,  -8898, -28309,   9075, -30199,  18249,
   13426,  14017, -29156, -12757,  16832,   4311, -24155, -17915,
    -853,    -90,   -271,    830,    107,  -1421,   -247,   -951,
    -398,    961,  -1508,   -725,    448,  -1065,    677,  -1275,
  -31183,  25435,  -7382,  24391, -20927,  10946,  24214,  16989,
   10335,  -7934, -22502,  10906,  31636,  28644,  23998, -17422,
     817,    603,   1322,  -1465,  -1215,   1218,   -874,  -1187,
   -1185,  -1278,  -1510,   -870,   -108,    996,    958,   1522,
   20297,   2146,  15355, -32384,  -6280, -14903, -11044,  14469,
  -21498, -20198,  23210, -17442, -23860, -20257,   7756,  23132,
    1097,    610,  -1285,    384,   -136,  -1335,    220,  -1659,
   -1530,    794,   -854,    478,   -308,    991,  -1460,   1628,

#define _16XSHIFT 624
  SHIFT, SHIFT, SHIFT, SHIFT, SHIFT, SHIFT, SHIFT, SHIFT,
  SHIFT, SHIFT, SHIFT, SHIFT, SHIFT, SHIFT, SHIFT, SHIFT
}};
```

```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,
```

```
  (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]);
}
```

```c
#include <stddef.h>
#include <stdint.h>
#include <immintrin.h>
#include <string.h>
#include "fips202.h"
#include "fips202x4.h"

/* Use implementation from the Keccak Code Package */
#define KeccakF1600_StatePermute4x FIPS202X4_NAMESPACE(KeccakP1600times4_PermuteAll_24round
s)
extern void KeccakF1600_StatePermute4x(__m256i *s);

static void keccakx4_absorb_once(__m256i s[25],
                                 unsigned int r,
                                 const uint8_t *in0,
                                 const uint8_t *in1,
                                 const uint8_t *in2,
                                 const uint8_t *in3,
                                 size_t inlen,
                                 uint8_t p)
{
  size_t i;
  uint64_t pos = 0;
  __m256i t, idx;

  for(i = 0; i < 25; ++i)
    s[i] = _mm256_setzero_si256();

  idx = _mm256_set_epi64x((long long)in3, (long long)in2, (long long)in1, (long long)in0);
  while(inlen >= r) {
    for(i = 0; i < r/8; ++i) {
      t = _mm256_i64gather_epi64((long long *)pos, idx, 1);
      s[i] = _mm256_xor_si256(s[i], t);
      pos += 8;
    }
    inlen -= r;

    KeccakF1600_StatePermute4x(s);
  }

  for(i = 0; i < inlen/8; ++i) {
    t = _mm256_i64gather_epi64((long long *)pos, idx, 1);
    s[i] = _mm256_xor_si256(s[i], t);
    pos += 8;
  }
  inlen -= 8*i;

  if(inlen) {
    t = _mm256_i64gather_epi64((long long *)pos, idx, 1);
    idx = _mm256_set1_epi64x((1ULL << (8*inlen)) - 1);
    t = _mm256_and_si256(t, idx);
    s[i] = _mm256_xor_si256(s[i], t);
  }

  t = _mm256_set1_epi64x((uint64_t)p << 8*inlen);
  s[i] = _mm256_xor_si256(s[i], t);
  t = _mm256_set1_epi64x(1ULL << 63);
  s[r/8 - 1] = _mm256_xor_si256(s[r/8 - 1], t);
}

static void keccakx4_squeezeblocks(uint8_t *out0,
                                   uint8_t *out1,
                                   uint8_t *out2,
                                   uint8_t *out3,
                                   size_t nblocks,
                                   unsigned int r,
                                   __m256i s[25])
{
```

```
  unsigned int i;
  __m128d t;

  while(nblocks > 0) {
    KeccakF1600_StatePermute4x(s);
    for(i=0; i < r/8; ++i) {
      t = _mm_castsi128_pd(_mm256_castsi256_si128(s[i]));
      _mm_storel_pd((__attribute__((__may_alias__)) double *)&out0[8*i], t);
      _mm_storeh_pd((__attribute__((__may_alias__)) double *)&out1[8*i], t);
      t = _mm_castsi128_pd(_mm256_extracti128_si256(s[i],1));
      _mm_storel_pd((__attribute__((__may_alias__)) double *)&out2[8*i], t);
      _mm_storeh_pd((__attribute__((__may_alias__)) double *)&out3[8*i], t);
    }

    out0 += r;
    out1 += r;
    out2 += r;
    out3 += r;
    --nblocks;
  }
}

void shake128x4_absorb_once(keccakx4_state *state,
                            const uint8_t *in0,
                            const uint8_t *in1,
                            const uint8_t *in2,
                            const uint8_t *in3,
                            size_t inlen)
{
  keccakx4_absorb_once(state->s, SHAKE128_RATE, in0, in1, in2, in3, inlen, 0x1F);
}

void shake128x4_squeezeblocks(uint8_t *out0,
                              uint8_t *out1,
                              uint8_t *out2,
                              uint8_t *out3,
                              size_t nblocks,
                              keccakx4_state *state)
{
  keccakx4_squeezeblocks(out0, out1, out2, out3, nblocks, SHAKE128_RATE, state->s);
}

void shake256x4_absorb_once(keccakx4_state *state,
                            const uint8_t *in0,
                            const uint8_t *in1,
                            const uint8_t *in2,
                            const uint8_t *in3,
                            size_t inlen)
{
  keccakx4_absorb_once(state->s, SHAKE256_RATE, in0, in1, in2, in3, inlen, 0x1F);
}

void shake256x4_squeezeblocks(uint8_t *out0,
                              uint8_t *out1,
                              uint8_t *out2,
                              uint8_t *out3,
                              size_t nblocks,
                              keccakx4_state *state)
{
  keccakx4_squeezeblocks(out0, out1, out2, out3, nblocks, SHAKE256_RATE, state->s);
}

void shake128x4(uint8_t *out0,
                uint8_t *out1,
                uint8_t *out2,
                uint8_t *out3,
                size_t outlen,
                const uint8_t *in0,
```

```
                 const uint8_t *in1,
                 const uint8_t *in2,
                 const uint8_t *in3,
                 size_t inlen)
{
  unsigned int i;
  size_t nblocks = outlen/SHAKE128_RATE;
  uint8_t t[4][SHAKE128_RATE];
  keccakx4_state state;

  shake128x4_absorb_once(&state, in0, in1, in2, in3, inlen);
  shake128x4_squeezeblocks(out0, out1, out2, out3, nblocks, &state);

  out0 += nblocks*SHAKE128_RATE;
  out1 += nblocks*SHAKE128_RATE;
  out2 += nblocks*SHAKE128_RATE;
  out3 += nblocks*SHAKE128_RATE;
  outlen -= nblocks*SHAKE128_RATE;

  if(outlen) {
    shake128x4_squeezeblocks(t[0], t[1], t[2], t[3], 1, &state);
    for(i = 0; i < outlen; ++i) {
      out0[i] = t[0][i];
      out1[i] = t[1][i];
      out2[i] = t[2][i];
      out3[i] = t[3][i];
    }
  }
}

void shake256x4(uint8_t *out0,
                uint8_t *out1,
                uint8_t *out2,
                uint8_t *out3,
                size_t outlen,
                const uint8_t *in0,
                const uint8_t *in1,
                const uint8_t *in2,
                const uint8_t *in3,
                size_t inlen)
{
  unsigned int i;
  size_t nblocks = outlen/SHAKE256_RATE;
  uint8_t t[4][SHAKE256_RATE];
  keccakx4_state state;

  shake256x4_absorb_once(&state, in0, in1, in2, in3, inlen);
  shake256x4_squeezeblocks(out0, out1, out2, out3, nblocks, &state);

  out0 += nblocks*SHAKE256_RATE;
  out1 += nblocks*SHAKE256_RATE;
  out2 += nblocks*SHAKE256_RATE;
  out3 += nblocks*SHAKE256_RATE;
  outlen -= nblocks*SHAKE256_RATE;

  if(outlen) {
    shake256x4_squeezeblocks(t[0], t[1], t[2], t[3], 1, &state);
    for(i = 0; i < outlen; ++i) {
      out0[i] = t[0][i];
      out1[i] = t[1][i];
      out2[i] = t[2][i];
      out3[i] = t[3][i];
    }
  }
}
```

```c
#include <stddef.h>
#include <stdint.h>
#include <immintrin.h>
#include <string.h>
#include "align.h"
#include "params.h"
#include "indcpa.h"
#include "polyvec.h"
#include "poly.h"
#include "ntt.h"
#include "cbd.h"
#include "rejsample.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.
*              The polynomial coefficients in pk are assumed to
*              lie in the invertal [0,q], i.e. pk must be reduced
*              by polyvec_reduce().
*
* 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])
{
  polyvec_tobytes(r, pk);
  memcpy(r+KYBER_POLYVECBYTES, seed, KYBER_SYMBYTES);
}

/*************************************************
* 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])
{
  polyvec_frombytes(pk, packedpk);
  memcpy(seed, packedpk+KYBER_POLYVECBYTES, KYBER_SYMBYTES);
}

/*************************************************
* Name:        pack_sk
*
* Description: Serialize the secret key.
*              The polynomial coefficients in sk are assumed to
*              lie in the invertal [0,q], i.e. sk must be reduced
*              by polyvec_reduce().
*
* 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.
*              The polynomial coefficients in b and v are assumed to
*              lie in the invertal [0,q], i.e. b and v must be reduced
*              by polyvec_reduce() and poly_reduce(), respectively.
*
* 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 array
*              - 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 <= buflen - 3) {  // buflen is always at least 3
    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
**************************************************/
#ifdef KYBER_90S
void gen_matrix(polyvec *a, const uint8_t seed[32], int transposed)
{
  unsigned int ctr, i, j, k;
  unsigned int buflen, off;
  uint64_t nonce = 0;
  ALIGNED_UINT8(REJ_UNIFORM_AVX_NBLOCKS*AES256CTR_BLOCKBYTES) buf;
  aes256ctr_ctx state;

  aes256ctr_init(&state, seed, 0);

  for(i=0;i<KYBER_K;i++) {
    for(j=0;j<KYBER_K;j++) {
      if(transposed)
        nonce = (j << 8) | i;
      else
        nonce = (i << 8) | j;

      state.n = _mm_loadl_epi64((__m128i *)&nonce);
      aes256ctr_squeezeblocks(buf.coeffs, REJ_UNIFORM_AVX_NBLOCKS, &state);
      buflen = REJ_UNIFORM_AVX_NBLOCKS*AES256CTR_BLOCKBYTES;
      ctr = rej_uniform_avx(a[i].vec[j].coeffs, buf.coeffs);

      while(ctr < KYBER_N) {
        off = buflen % 3;
        for(k = 0; k < off; k++)
          buf.coeffs[k] = buf.coeffs[buflen - off + k];
        aes256ctr_squeezeblocks(buf.coeffs + off, 1, &state);
        buflen = off + AES256CTR_BLOCKBYTES;
        ctr += rej_uniform(a[i].vec[j].coeffs + ctr, KYBER_N - ctr, buf.coeffs, buflen);
      }
```

```c
      poly_nttunpack(&a[i].vec[j]);
    }
  }
}
#else
#if KYBER_K == 2
void gen_matrix(polyvec *a, const uint8_t seed[32], int transposed)
{
  unsigned int ctr0, ctr1, ctr2, ctr3;
  ALIGNED_UINT8(REJ_UNIFORM_AVX_NBLOCKS*SHAKE128_RATE) buf[4];
  __m256i f;
  keccakx4_state state;

  f = _mm256_loadu_si256((__m256i *)seed);
  _mm256_store_si256(buf[0].vec, f);
  _mm256_store_si256(buf[1].vec, f);
  _mm256_store_si256(buf[2].vec, f);
  _mm256_store_si256(buf[3].vec, f);

  if(transposed) {
    buf[0].coeffs[32] = 0;
    buf[0].coeffs[33] = 0;
    buf[1].coeffs[32] = 0;
    buf[1].coeffs[33] = 1;
    buf[2].coeffs[32] = 1;
    buf[2].coeffs[33] = 0;
    buf[3].coeffs[32] = 1;
    buf[3].coeffs[33] = 1;
  }
  else {
    buf[0].coeffs[32] = 0;
    buf[0].coeffs[33] = 0;
    buf[1].coeffs[32] = 1;
    buf[1].coeffs[33] = 0;
    buf[2].coeffs[32] = 0;
    buf[2].coeffs[33] = 1;
    buf[3].coeffs[32] = 1;
    buf[3].coeffs[33] = 1;
  }

  shake128x4_absorb_once(&state, buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs
, 34);
  shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, REJ_
UNIFORM_AVX_NBLOCKS, &state);

  ctr0 = rej_uniform_avx(a[0].vec[0].coeffs, buf[0].coeffs);
  ctr1 = rej_uniform_avx(a[0].vec[1].coeffs, buf[1].coeffs);
  ctr2 = rej_uniform_avx(a[1].vec[0].coeffs, buf[2].coeffs);
  ctr3 = rej_uniform_avx(a[1].vec[1].coeffs, buf[3].coeffs);

  while(ctr0 < KYBER_N || ctr1 < KYBER_N || ctr2 < KYBER_N || ctr3 < KYBER_N) {
    shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, 1,
 &state);

    ctr0 += rej_uniform(a[0].vec[0].coeffs + ctr0, KYBER_N - ctr0, buf[0].coeffs, SHAKE128_
RATE);
    ctr1 += rej_uniform(a[0].vec[1].coeffs + ctr1, KYBER_N - ctr1, buf[1].coeffs, SHAKE128_
RATE);
    ctr2 += rej_uniform(a[1].vec[0].coeffs + ctr2, KYBER_N - ctr2, buf[2].coeffs, SHAKE128_
RATE);
    ctr3 += rej_uniform(a[1].vec[1].coeffs + ctr3, KYBER_N - ctr3, buf[3].coeffs, SHAKE128_
RATE);
  }

  poly_nttunpack(&a[0].vec[0]);
  poly_nttunpack(&a[0].vec[1]);
  poly_nttunpack(&a[1].vec[0]);
  poly_nttunpack(&a[1].vec[1]);
```

```
}
#elif KYBER_K == 3
void gen_matrix(polyvec *a, const uint8_t seed[32], int transposed)
{
  unsigned int ctr0, ctr1, ctr2, ctr3;
  ALIGNED_UINT8(REJ_UNIFORM_AVX_NBLOCKS*SHAKE128_RATE) buf[4];
  __m256i f;
  keccakx4_state state;
  keccak_state state1x;

  f = _mm256_loadu_si256((__m256i *)seed);
  _mm256_store_si256(buf[0].vec, f);
  _mm256_store_si256(buf[1].vec, f);
  _mm256_store_si256(buf[2].vec, f);
  _mm256_store_si256(buf[3].vec, f);

  if(transposed) {
    buf[0].coeffs[32] = 0;
    buf[0].coeffs[33] = 0;
    buf[1].coeffs[32] = 0;
    buf[1].coeffs[33] = 1;
    buf[2].coeffs[32] = 0;
    buf[2].coeffs[33] = 2;
    buf[3].coeffs[32] = 1;
    buf[3].coeffs[33] = 0;
  }
  else {
    buf[0].coeffs[32] = 0;
    buf[0].coeffs[33] = 0;
    buf[1].coeffs[32] = 1;
    buf[1].coeffs[33] = 0;
    buf[2].coeffs[32] = 2;
    buf[2].coeffs[33] = 0;
    buf[3].coeffs[32] = 0;
    buf[3].coeffs[33] = 1;
  }

  shake128x4_absorb_once(&state, buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs
, 34);
  shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, REJ_
UNIFORM_AVX_NBLOCKS, &state);

  ctr0 = rej_uniform_avx(a[0].vec[0].coeffs, buf[0].coeffs);
  ctr1 = rej_uniform_avx(a[0].vec[1].coeffs, buf[1].coeffs);
  ctr2 = rej_uniform_avx(a[0].vec[2].coeffs, buf[2].coeffs);
  ctr3 = rej_uniform_avx(a[1].vec[0].coeffs, buf[3].coeffs);

  while(ctr0 < KYBER_N || ctr1 < KYBER_N || ctr2 < KYBER_N || ctr3 < KYBER_N) {
    shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, 1,
 &state);

    ctr0 += rej_uniform(a[0].vec[0].coeffs + ctr0, KYBER_N - ctr0, buf[0].coeffs, SHAKE128_
RATE);
    ctr1 += rej_uniform(a[0].vec[1].coeffs + ctr1, KYBER_N - ctr1, buf[1].coeffs, SHAKE128_
RATE);
    ctr2 += rej_uniform(a[0].vec[2].coeffs + ctr2, KYBER_N - ctr2, buf[2].coeffs, SHAKE128_
RATE);
    ctr3 += rej_uniform(a[1].vec[0].coeffs + ctr3, KYBER_N - ctr3, buf[3].coeffs, SHAKE128_
RATE);
  }

  poly_nttunpack(&a[0].vec[0]);
  poly_nttunpack(&a[0].vec[1]);
  poly_nttunpack(&a[0].vec[2]);
  poly_nttunpack(&a[1].vec[0]);

  f = _mm256_loadu_si256((__m256i *)seed);
  _mm256_store_si256(buf[0].vec, f);
```

```
  _mm256_store_si256(buf[1].vec, f);
  _mm256_store_si256(buf[2].vec, f);
  _mm256_store_si256(buf[3].vec, f);

  if(transposed) {
    buf[0].coeffs[32] = 1;
    buf[0].coeffs[33] = 1;
    buf[1].coeffs[32] = 1;
    buf[1].coeffs[33] = 2;
    buf[2].coeffs[32] = 2;
    buf[2].coeffs[33] = 0;
    buf[3].coeffs[32] = 2;
    buf[3].coeffs[33] = 1;
  }
  else {
    buf[0].coeffs[32] = 1;
    buf[0].coeffs[33] = 1;
    buf[1].coeffs[32] = 2;
    buf[1].coeffs[33] = 1;
    buf[2].coeffs[32] = 0;
    buf[2].coeffs[33] = 2;
    buf[3].coeffs[32] = 1;
    buf[3].coeffs[33] = 2;
  }

  shake128x4_absorb_once(&state, buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs
, 34);
  shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, REJ_
UNIFORM_AVX_NBLOCKS, &state);

  ctr0 = rej_uniform_avx(a[1].vec[1].coeffs, buf[0].coeffs);
  ctr1 = rej_uniform_avx(a[1].vec[2].coeffs, buf[1].coeffs);
  ctr2 = rej_uniform_avx(a[2].vec[0].coeffs, buf[2].coeffs);
  ctr3 = rej_uniform_avx(a[2].vec[1].coeffs, buf[3].coeffs);

  while(ctr0 < KYBER_N || ctr1 < KYBER_N || ctr2 < KYBER_N || ctr3 < KYBER_N) {
    shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, 1,
 &state);

    ctr0 += rej_uniform(a[1].vec[1].coeffs + ctr0, KYBER_N - ctr0, buf[0].coeffs, SHAKE128_
RATE);
    ctr1 += rej_uniform(a[1].vec[2].coeffs + ctr1, KYBER_N - ctr1, buf[1].coeffs, SHAKE128_
RATE);
    ctr2 += rej_uniform(a[2].vec[0].coeffs + ctr2, KYBER_N - ctr2, buf[2].coeffs, SHAKE128_
RATE);
    ctr3 += rej_uniform(a[2].vec[1].coeffs + ctr3, KYBER_N - ctr3, buf[3].coeffs, SHAKE128_
RATE);
  }

  poly_nttunpack(&a[1].vec[1]);
  poly_nttunpack(&a[1].vec[2]);
  poly_nttunpack(&a[2].vec[0]);
  poly_nttunpack(&a[2].vec[1]);

  f = _mm256_loadu_si256((__m256i *)seed);
  _mm256_store_si256(buf[0].vec, f);
  buf[0].coeffs[32] = 2;
  buf[0].coeffs[33] = 2;
  shake128_absorb_once(&state1x, buf[0].coeffs, 34);
  shake128_squeezeblocks(buf[0].coeffs, REJ_UNIFORM_AVX_NBLOCKS, &state1x);
  ctr0 = rej_uniform_avx(a[2].vec[2].coeffs, buf[0].coeffs);
  while(ctr0 < KYBER_N) {
    shake128_squeezeblocks(buf[0].coeffs, 1, &state1x);
    ctr0 += rej_uniform(a[2].vec[2].coeffs + ctr0, KYBER_N - ctr0, buf[0].coeffs, SHAKE128_
RATE);
  }

  poly_nttunpack(&a[2].vec[2]);
```

```
}
#elif KYBER_K == 4
void gen_matrix(polyvec *a, const uint8_t seed[32], int transposed)
{
  unsigned int i, ctr0, ctr1, ctr2, ctr3;
  ALIGNED_UINT8(REJ_UNIFORM_AVX_NBLOCKS*SHAKE128_RATE) buf[4];
  __m256i f;
  keccakx4_state state;

  for(i=0;i<4;i++) {
    f = _mm256_loadu_si256((__m256i *)seed);
    _mm256_store_si256(buf[0].vec, f);
    _mm256_store_si256(buf[1].vec, f);
    _mm256_store_si256(buf[2].vec, f);
    _mm256_store_si256(buf[3].vec, f);

    if(transposed) {
      buf[0].coeffs[32] = i;
      buf[0].coeffs[33] = 0;
      buf[1].coeffs[32] = i;
      buf[1].coeffs[33] = 1;
      buf[2].coeffs[32] = i;
      buf[2].coeffs[33] = 2;
      buf[3].coeffs[32] = i;
      buf[3].coeffs[33] = 3;
    }
    else {
      buf[0].coeffs[32] = 0;
      buf[0].coeffs[33] = i;
      buf[1].coeffs[32] = 1;
      buf[1].coeffs[33] = i;
      buf[2].coeffs[32] = 2;
      buf[2].coeffs[33] = i;
      buf[3].coeffs[32] = 3;
      buf[3].coeffs[33] = i;
    }

    shake128x4_absorb_once(&state, buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coef
fs, 34);
    shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, RE
J_UNIFORM_AVX_NBLOCKS, &state);

    ctr0 = rej_uniform_avx(a[i].vec[0].coeffs, buf[0].coeffs);
    ctr1 = rej_uniform_avx(a[i].vec[1].coeffs, buf[1].coeffs);
    ctr2 = rej_uniform_avx(a[i].vec[2].coeffs, buf[2].coeffs);
    ctr3 = rej_uniform_avx(a[i].vec[3].coeffs, buf[3].coeffs);

    while(ctr0 < KYBER_N || ctr1 < KYBER_N || ctr2 < KYBER_N || ctr3 < KYBER_N) {
      shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs,
1, &state);

      ctr0 += rej_uniform(a[i].vec[0].coeffs + ctr0, KYBER_N - ctr0, buf[0].coeffs, SHAKE12
8_RATE);
      ctr1 += rej_uniform(a[i].vec[1].coeffs + ctr1, KYBER_N - ctr1, buf[1].coeffs, SHAKE12
8_RATE);
      ctr2 += rej_uniform(a[i].vec[2].coeffs + ctr2, KYBER_N - ctr2, buf[2].coeffs, SHAKE12
8_RATE);
      ctr3 += rej_uniform(a[i].vec[3].coeffs + ctr3, KYBER_N - ctr3, buf[3].coeffs, SHAKE12
8_RATE);
    }

    poly_nttunpack(&a[i].vec[0]);
    poly_nttunpack(&a[i].vec[1]);
    poly_nttunpack(&a[i].vec[2]);
    poly_nttunpack(&a[i].vec[3]);
  }
}
#endif
```

```c
#endif

/*************************************************
* 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;
  polyvec a[KYBER_K], e, pkpv, skpv;

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

  gen_a(a, publicseed);

#ifdef KYBER_90S
#define NOISE_NBLOCKS ((KYBER_ETA1*KYBER_N/4)/AES256CTR_BLOCKBYTES) /* Assumes divisibility
 */
  uint64_t nonce = 0;
  ALIGNED_UINT8(NOISE_NBLOCKS*AES256CTR_BLOCKBYTES+32) coins; // +32 bytes as required by p
oly_cbd_eta1
  aes256ctr_ctx state;
  aes256ctr_init(&state, noiseseed, nonce++);
  for(i=0;i<KYBER_K;i++) {
    aes256ctr_squeezeblocks(coins.coeffs, NOISE_NBLOCKS, &state);
    state.n = _mm_loadl_epi64((__m128i *)&nonce);
    nonce += 1;
    poly_cbd_eta1(&skpv.vec[i], coins.vec);
  }
  for(i=0;i<KYBER_K;i++) {
    aes256ctr_squeezeblocks(coins.coeffs, NOISE_NBLOCKS, &state);
    state.n = _mm_loadl_epi64((__m128i *)&nonce);
    nonce += 1;
    poly_cbd_eta1(&e.vec[i], coins.vec);
  }
#else
#if KYBER_K == 2
  poly_getnoise_eta1_4x(skpv.vec+0, skpv.vec+1, e.vec+0, e.vec+1, noiseseed, 0, 1, 2, 3);
#elif KYBER_K == 3
  poly_getnoise_eta1_4x(skpv.vec+0, skpv.vec+1, skpv.vec+2, e.vec+0, noiseseed, 0, 1, 2, 3)
;
  poly_getnoise_eta1_4x(e.vec+1, e.vec+2, pkpv.vec+0, pkpv.vec+1, noiseseed, 4, 5, 6, 7);
#elif KYBER_K == 4
  poly_getnoise_eta1_4x(skpv.vec+0, skpv.vec+1, skpv.vec+2, skpv.vec+3, noiseseed,  0, 1, 2
, 3);
  poly_getnoise_eta1_4x(e.vec+0, e.vec+1, e.vec+2, e.vec+3, noiseseed, 4, 5, 6, 7);
#endif
#endif

  polyvec_ntt(&skpv);
  polyvec_reduce(&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];
  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);

#ifdef KYBER_90S
#define NOISE_NBLOCKS ((KYBER_ETA1*KYBER_N/4)/AES256CTR_BLOCKBYTES) /* Assumes divisibility
 */
#define CIPHERTEXTNOISE_NBLOCKS ((KYBER_ETA2*KYBER_N/4)/AES256CTR_BLOCKBYTES) /* Assumes di
visibility */
  uint64_t nonce = 0;
  ALIGNED_UINT8(NOISE_NBLOCKS*AES256CTR_BLOCKBYTES+32) buf; /* +32 bytes as required by pol
y_cbd_eta1 */
  aes256ctr_ctx state;
  aes256ctr_init(&state, coins, nonce++);
  for(i=0;i<KYBER_K;i++) {
    aes256ctr_squeezeblocks(buf.coeffs, NOISE_NBLOCKS, &state);
    state.n = _mm_loadl_epi64((__m128i *)&nonce);
    nonce += 1;
    poly_cbd_eta1(&sp.vec[i], buf.vec);
  }
  for(i=0;i<KYBER_K;i++) {
    aes256ctr_squeezeblocks(buf.coeffs, CIPHERTEXTNOISE_NBLOCKS, &state);
    state.n = _mm_loadl_epi64((__m128i *)&nonce);
    nonce += 1;
    poly_cbd_eta2(&ep.vec[i], buf.vec);
  }
  aes256ctr_squeezeblocks(buf.coeffs, CIPHERTEXTNOISE_NBLOCKS, &state);
  poly_cbd_eta2(&epp, buf.vec);
#else
#if KYBER_K == 2
  poly_getnoise_eta1122_4x(sp.vec+0, sp.vec+1, ep.vec+0, ep.vec+1, coins, 0, 1, 2, 3);
  poly_getnoise_eta2(&epp, coins, 4);
```

```c
#elif KYBER_K == 3
  poly_getnoise_eta1_4x(sp.vec+0, sp.vec+1, sp.vec+2, ep.vec+0, coins, 0, 1, 2 ,3);
  poly_getnoise_eta1_4x(ep.vec+1, ep.vec+2, &epp, b.vec+0, coins,  4, 5, 6, 7);
#elif KYBER_K == 4
  poly_getnoise_eta1_4x(sp.vec+0, sp.vec+1, sp.vec+2, sp.vec+3, coins, 0, 1, 2, 3);
  poly_getnoise_eta1_4x(ep.vec+0, ep.vec+1, ep.vec+2, ep.vec+3, coins, 4, 5, 6, 7);
  poly_getnoise_eta2(&epp, coins, 8);
#endif
#endif

  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 <string.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)
{
  indcpa_keypair(pk, sk);
  memcpy(sk+KYBER_INDCPA_SECRETKEYBYTES, pk, KYBER_INDCPA_PUBLICKEYBYTES);
  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)
{
  int fail;
  uint8_t buf[2*KYBER_SYMBYTES];
  /* Will contain key, coins */
  uint8_t kr[2*KYBER_SYMBYTES];
  ALIGNED_UINT8(KYBER_CIPHERTEXTBYTES) cmp;
  const uint8_t *pk = sk+KYBER_INDCPA_SECRETKEYBYTES;

  indcpa_dec(buf, ct, sk);

  /* Multitarget countermeasure for coins + contributory KEM */
  memcpy(buf+KYBER_SYMBYTES, sk+KYBER_SECRETKEYBYTES-2*KYBER_SYMBYTES, KYBER_SYMBYTES);
  hash_g(kr, buf, 2*KYBER_SYMBYTES);

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

  fail = verify(ct, cmp.coeffs, 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;
}
```

```c
#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 <immintrin.h>
#include <string.h>
#include "align.h"
#include "params.h"
#include "poly.h"
#include "ntt.h"
#include "consts.h"
#include "reduce.h"
#include "cbd.h"
#include "symmetric.h"

/*************************************************
* Name:        poly_compress
*
* Description: Compression and subsequent serialization of a polynomial.
*              The coefficients of the input polynomial are assumed to
*              lie in the invertal [0,q], i.e. the polynomial must be reduced
*              by poly_reduce().
*
* Arguments:   - uint8_t *r: pointer to output byte array
*                            (of length KYBER_POLYCOMPRESSEDBYTES)
*              - const poly *a: pointer to input polynomial
**************************************************/
#if (KYBER_POLYCOMPRESSEDBYTES == 96)
void poly_compress(uint8_t r[96], const poly * restrict a)
{
  unsigned int i;
  __m256i f0, f1, f2, f3;
  __m128i t0, t1;
  const __m256i v = _mm256_load_si256(&qdata.vec[_16XV/16]);
  const __m256i shift1 = _mm256_set1_epi16(1 << 8);
  const __m256i mask = _mm256_set1_epi16(7);
  const __m256i shift2 = _mm256_set1_epi16((8 << 8) + 1);
  const __m256i shift3 = _mm256_set1_epi32((64 << 16) + 1);
  const __m256i sllvdidx = _mm256_set1_epi64x(12LL << 32);
  const __m256i shufbidx = _mm256_set_epi8( 8, 2, 1, 0,-1,-1,-1,-1,14,13,12, 6, 5, 4,10, 9,
                                           -1,-1,-1,-1,14,13,12, 6, 5, 4,10, 9, 8, 2, 1, 0)
;

  for(i=0;i<KYBER_N/64;i++) {
    f0 = _mm256_load_si256(&a->vec[4*i+0]);
    f1 = _mm256_load_si256(&a->vec[4*i+1]);
    f2 = _mm256_load_si256(&a->vec[4*i+2]);
    f3 = _mm256_load_si256(&a->vec[4*i+3]);
    f0 = _mm256_mulhi_epi16(f0,v);
    f1 = _mm256_mulhi_epi16(f1,v);
    f2 = _mm256_mulhi_epi16(f2,v);
    f3 = _mm256_mulhi_epi16(f3,v);
    f0 = _mm256_mulhrs_epi16(f0,shift1);
    f1 = _mm256_mulhrs_epi16(f1,shift1);
    f2 = _mm256_mulhrs_epi16(f2,shift1);
    f3 = _mm256_mulhrs_epi16(f3,shift1);
    f0 = _mm256_and_si256(f0,mask);
    f1 = _mm256_and_si256(f1,mask);
    f2 = _mm256_and_si256(f2,mask);
    f3 = _mm256_and_si256(f3,mask);
    f0 = _mm256_packus_epi16(f0,f1);
    f2 = _mm256_packus_epi16(f2,f3);
    f0 = _mm256_maddubs_epi16(f0,shift2);       // a0 a1 a2 a3 b0 b1 b2 b3 a4 a5 a6 a7 b4 b
5 b6 b7
    f2 = _mm256_maddubs_epi16(f2,shift2);       // c0 c1 c2 c3 d0 d1 d2 d3 c4 c5 c6 c7 d4 d
5 d6 d7
    f0 = _mm256_madd_epi16(f0,shift3);          // a0 a1 b0 b1 a2 a3 b2 b3
    f2 = _mm256_madd_epi16(f2,shift3);          // c0 c1 d0 d1 c2 c3 d2 d3
    f0 = _mm256_sllv_epi32(f0,sllvdidx);
    f2 = _mm256_sllv_epi32(f2,sllvdidx);
    f0 = _mm256_hadd_epi32(f0,f2);              // a0 c0 c0 d0 a1 b1 c1 d1
```

```c
    f0 = _mm256_permute4x64_epi64(f0,0xD8);      // a0 b0 a1 b1 c0 d0 c1 d1
    f0 = _mm256_shuffle_epi8(f0,shufbidx);
    t0 = _mm256_castsi256_si128(f0);
    t1 = _mm256_extracti128_si256(f0,1);
    t0 = _mm_blend_epi32(t0,t1,0x08);
    _mm_storeu_si128((__m128i *)&r[24*i+ 0],t0);
    _mm_storel_epi64((__m128i *)&r[24*i+16],t1);
  }
}

/*************************************************
* 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 * restrict r, const uint8_t a[96])
{
  unsigned int i;
  __m128i t;
  __m256i f;
  const __m256i q = _mm256_load_si256(&qdata.vec[_16XQ/16]);
  const __m256i shufbidx = _mm256_set_epi8(5,5,5,5,5,4,4,4,4,4,4,3,3,3,3,3,
                                           2,2,2,2,2,1,1,1,1,1,1,0,0,0,0,0);
  const __m256i mask = _mm256_set_epi16(224,28,896,112,14,448,56,7,
                                        224,28,896,112,14,448,56,7);
  const __m256i shift = _mm256_set_epi16(128,1024,32,256,2048,64,512,4096,
                                         128,1024,32,256,2048,64,512,4096);

  for(i=0;i<KYBER_N/16;i++) {
    t = _mm_castps_si128(_mm_load_ss((float *)&a[6*i+0])));
    t = _mm_insert_epi16(t,*(int16_t *)&a[6*i+4],2);
    f = _mm256_broadcastsi128_si256(t);
    f = _mm256_blend_epi16(f,g,0x);
    f = _mm256_shuffle_epi8(f,shufbidx);
    f = _mm256_and_si256(f,mask);
    f = _mm256_mullo_epi16(f,shift);
    f = _mm256_mulhrs_epi16(f,q);
    _mm256_store_si256(&r->vec[i],f);
  }
}

#elif (KYBER_POLYCOMPRESSEDBYTES == 128)
void poly_compress(uint8_t r[128], const poly * restrict a)
{
  unsigned int i;
  __m256i f0, f1, f2, f3;
  const __m256i v = _mm256_load_si256(&qdata.vec[_16XV/16]);
  const __m256i shift1 = _mm256_set1_epi16(1 << 9);
  const __m256i mask = _mm256_set1_epi16(15);
  const __m256i shift2 = _mm256_set1_epi16((16 << 8) + 1);
  const __m256i permdidx = _mm256_set_epi32(7,3,6,2,5,1,4,0);

  for(i=0;i<KYBER_N/64;i++) {
    f0 = _mm256_load_si256(&a->vec[4*i+0]);
    f1 = _mm256_load_si256(&a->vec[4*i+1]);
    f2 = _mm256_load_si256(&a->vec[4*i+2]);
    f3 = _mm256_load_si256(&a->vec[4*i+3]);
    f0 = _mm256_mulhi_epi16(f0,v);
    f1 = _mm256_mulhi_epi16(f1,v);
    f2 = _mm256_mulhi_epi16(f2,v);
    f3 = _mm256_mulhi_epi16(f3,v);
    f0 = _mm256_mulhrs_epi16(f0,shift1);
    f1 = _mm256_mulhrs_epi16(f1,shift1);
```

```c
    f2 = _mm256_mulhrs_epi16(f2,shift1);
    f3 = _mm256_mulhrs_epi16(f3,shift1);
    f0 = _mm256_and_si256(f0,mask);
    f1 = _mm256_and_si256(f1,mask);
    f2 = _mm256_and_si256(f2,mask);
    f3 = _mm256_and_si256(f3,mask);
    f0 = _mm256_packus_epi16(f0,f1);
    f2 = _mm256_packus_epi16(f2,f3);
    f0 = _mm256_maddubs_epi16(f0,shift2);
    f2 = _mm256_maddubs_epi16(f2,shift2);
    f0 = _mm256_packus_epi16(f0,f2);
    f0 = _mm256_permutevar8x32_epi32(f0,permdidx);
    _mm256_storeu_si256((__m256i *)&r[32*i],f0);
  }
}

void poly_decompress(poly * restrict r, const uint8_t a[128])
{
  unsigned int i;
  __m128i t;
  __m256i f;
  const __m256i q = _mm256_load_si256(&qdata.vec[_16XQ/16]);
  const __m256i shufbidx = _mm256_set_epi8(7,7,7,7,6,6,6,6,5,5,5,5,4,4,4,4,
                                            3,3,3,3,2,2,2,2,1,1,1,1,0,0,0,0);
  const __m256i mask = _mm256_set1_epi32(0x00F0000F);
  const __m256i shift = _mm256_set1_epi32((128 << 16) + 2048);

  for(i=0;i<KYBER_N/16;i++) {
    t = _mm_loadl_epi64((__m128i *)&a[8*i]);
    f = _mm256_broadcastsi128_si256(t);
    f = _mm256_shuffle_epi8(f,shufbidx);
    f = _mm256_and_si256(f,mask);
    f = _mm256_mullo_epi16(f,shift);
    f = _mm256_mulhrs_epi16(f,q);
    _mm256_store_si256(&r->vec[i],f);
  }
}

#elif (KYBER_POLYCOMPRESSEDBYTES == 160)
void poly_compress(uint8_t r[160], const poly * restrict a)
{
  unsigned int i;
  __m256i f0, f1;
  __m128i t0, t1;
  const __m256i v = _mm256_load_si256(&qdata.vec[_16XV/16]);
  const __m256i shift1 = _mm256_set1_epi16(1 << 10);
  const __m256i mask = _mm256_set1_epi16(31);
  const __m256i shift2 = _mm256_set1_epi16((32 << 8) + 1);
  const __m256i shift3 = _mm256_set1_epi32((1024 << 16) + 1);
  const __m256i sllvdidx = _mm256_set1_epi64x(12);
  const __m256i shufbidx = _mm256_set_epi8( 8,-1,-1,-1,-1,-1, 4, 3, 2, 1, 0,-1,12,11,10, 9,
                                            -1,12,11,10, 9, 8,-1,-1,-1,-1,-1 ,4, 3, 2, 1, 0)
;

  for(i=0;i<KYBER_N/32;i++) {
    f0 = _mm256_load_si256(&a->vec[2*i+0]);
    f1 = _mm256_load_si256(&a->vec[2*i+1]);
    f0 = _mm256_mulhi_epi16(f0,v);
    f1 = _mm256_mulhi_epi16(f1,v);
    f0 = _mm256_mulhrs_epi16(f0,shift1);
    f1 = _mm256_mulhrs_epi16(f1,shift1);
    f0 = _mm256_and_si256(f0,mask);
    f1 = _mm256_and_si256(f1,mask);
    f0 = _mm256_packus_epi16(f0,f1);
    f0 = _mm256_maddubs_epi16(f0,shift2);       // a0 a1 a2 a3 b0 b1 b2 b3 a4 a5 a6 a7 b4 b
5 b6 b7
    f0 = _mm256_madd_epi16(f0,shift3);          // a0 a1 b0 b1 a2 a3 b2 b3
    f0 = _mm256_sllv_epi32(f0,sllvdidx);
```

```
    f0 = _mm256_srlv_epi64(f0,sllvdidx);
    f0 = _mm256_shuffle_epi8(f0,shufbidx);
    t0 = _mm256_castsi256_si128(f0);
    t1 = _mm256_extracti128_si256(f0,1);
    t0 = _mm_blendv_epi8(t0,t1,_mm256_castsi256_si128(shufbidx));
    _mm_storeu_si128((__m128i *)&r[20*i+ 0],t0);
    memcpy(&r[20*i+16],&t1,4);
  }
}

void poly_decompress(poly * restrict r, const uint8_t a[160])
{
  unsigned int i;
  __m128i t;
  __m256i f;
  int16_t ti;
  const __m256i q = _mm256_load_si256(&qdata.vec[_16XQ/16]);
  const __m256i shufbidx = _mm256_set_epi8(9,9,9,8,8,8,8,7,7,6,6,6,6,5,5,5,
                                           4,4,4,3,3,3,3,2,2,1,1,1,1,0,0,0);
  const __m256i mask = _mm256_set_epi16(248,1984,62,496,3968,124,992,31,
                                        248,1984,62,496,3968,124,992,31);
  const __m256i shift = _mm256_set_epi16(128,16,512,64,8,256,32,1024,
                                         128,16,512,64,8,256,32,1024);

  for(i=0;i<KYBER_N/16;i++) {
    t = _mm_loadl_epi64((__m128i *)&a[10*i+0]);
    memcpy(&ti,&a[10*i+8],2);
    t = _mm_insert_epi16(t,ti,4);
    f = _mm256_broadcastsi128_si256(t);
    f = _mm256_shuffle_epi8(f,shufbidx);
    f = _mm256_and_si256(f,mask);
    f = _mm256_mullo_epi16(f,shift);
    f = _mm256_mulhrs_epi16(f,q);
    _mm256_store_si256(&r->vec[i],f);
  }
}

#endif

/*************************************************
* Name:        poly_tobytes
*
* Description: Serialization of a polynomial in NTT representation.
*              The coefficients of the input polynomial are assumed to
*              lie in the invertal [0,q], i.e. the polynomial must be reduced
*              by poly_reduce(). The coefficients are orderd as output by
*              poly_ntt(); the serialized output coefficients are in bitreversed
*              order.
*
* Arguments:   - uint8_t *r: pointer to output byte array
*                            (needs space for KYBER_POLYBYTES bytes)
*              - poly *a: pointer to input polynomial
**************************************************/
void poly_tobytes(uint8_t r[KYBER_POLYBYTES], const poly *a)
{
  ntttobytes_avx(r, a->vec, qdata.vec);
}

/*************************************************
* 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)
**************************************************/
```

```c
void poly_frombytes(poly *r, const uint8_t a[KYBER_POLYBYTES])
{
  nttfrombytes_avx(r->vec, a, qdata.vec);
}

/*************************************************
* 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 * restrict r, const uint8_t msg[KYBER_INDCPA_MSGBYTES])
{
#if (KYBER_INDCPA_MSGBYTES != 32)
#error "KYBER_INDCPA_MSGBYTES must be equal to 32!"
#endif
  __m256i f, g0, g1, g2, g3, h0, h1, h2, h3;
  const __m256i shift = _mm256_broadcastsi128_si256(_mm_set_epi32(0,1,2,3));
  const __m256i idx = _mm256_broadcastsi128_si256(_mm_set_epi8(15,14,11,10,7,6,3,2,13,12,9,
8,5,4,1,0));
  const __m256i hqs = _mm256_set1_epi16((KYBER_Q+1)/2);

#define FROMMSG64(i)                                                    \
  g3 = _mm256_shuffle_epi32(f,0x55*i);                                  \
  g3 = _mm256_sllv_epi32(g3,shift);                                     \
  g3 = _mm256_shuffle_epi8(g3,idx);                                     \
  g0 = _mm256_slli_epi16(g3,12);                                        \
  g1 = _mm256_slli_epi16(g3,8);                                         \
  g2 = _mm256_slli_epi16(g3,4);                                         \
  g0 = _mm256_srai_epi16(g0,15);                                        \
  g1 = _mm256_srai_epi16(g1,15);                                        \
  g2 = _mm256_srai_epi16(g2,15);                                        \
  g3 = _mm256_srai_epi16(g3,15);                                        \
  g0 = _mm256_and_si256(g0,hqs);  /* 19 18 17 16  3  2  1  0 */         \
  g1 = _mm256_and_si256(g1,hqs);  /* 23 22 21 20  7  6  5  4 */         \
  g2 = _mm256_and_si256(g2,hqs);  /* 27 26 25 24 11 10  9  8 */         \
  g3 = _mm256_and_si256(g3,hqs);  /* 31 30 29 28 15 14 13 12 */         \
  h0 = _mm256_unpacklo_epi64(g0,g1);                                    \
  h2 = _mm256_unpackhi_epi64(g0,g1);                                    \
  h1 = _mm256_unpacklo_epi64(g2,g3);                                    \
  h3 = _mm256_unpackhi_epi64(g2,g3);                                    \
  g0 = _mm256_permute2x128_si256(h0,h1,0x20);                           \
  g2 = _mm256_permute2x128_si256(h0,h1,0x31);                           \
  g1 = _mm256_permute2x128_si256(h2,h3,0x20);                           \
  g3 = _mm256_permute2x128_si256(h2,h3,0x31);                           \
  _mm256_store_si256(&r->vec[0+2*i+0],g0);       \
  _mm256_store_si256(&r->vec[0+2*i+1],g1);       \
  _mm256_store_si256(&r->vec[8+2*i+0],g2);       \
  _mm256_store_si256(&r->vec[8+2*i+1],g3)

  f = _mm256_loadu_si256((__m256i *)msg);
  FROMMSG64(0);
  FROMMSG64(1);
  FROMMSG64(2);
  FROMMSG64(3);
}

/*************************************************
* Name:        poly_tomsg
*
* Description: Convert polynomial to 32-byte message.
*              The coefficients of the input polynomial are assumed to
*              lie in the invertal [0,q], i.e. the polynomial must be reduced
*              by poly_reduce().
*
* Arguments:   - uint8_t *msg: pointer to output message
```

```c
*               - poly *a: pointer to input polynomial
**************************************************/
void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const poly * restrict a)
{
  unsigned int i;
  uint32_t small;
  __m256i f0, f1, g0, g1;
  const __m256i hq = _mm256_set1_epi16((KYBER_Q - 1)/2);
  const __m256i hhq = _mm256_set1_epi16((KYBER_Q - 1)/4);

  for(i=0;i<KYBER_N/32;i++) {
    f0 = _mm256_load_si256(&a->vec[2*i+0]);
    f1 = _mm256_load_si256(&a->vec[2*i+1]);
    f0 = _mm256_sub_epi16(hq, f0);
    f1 = _mm256_sub_epi16(hq, f1);
    g0 = _mm256_srai_epi16(f0, 15);
    g1 = _mm256_srai_epi16(f1, 15);
    f0 = _mm256_xor_si256(f0, g0);
    f1 = _mm256_xor_si256(f1, g1);
    f0 = _mm256_sub_epi16(f0, hhq);
    f1 = _mm256_sub_epi16(f1, hhq);
    f0 = _mm256_packs_epi16(f0, f1);
    f0 = _mm256_permute4x64_epi64(f0, 0xD8);
    small = _mm256_movemask_epi8(f0);
    memcpy(&msg[4*i], &small, 4);
  }
}

/*************************************************
* 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)
{
  ALIGNED_UINT8(KYBER_ETA1*KYBER_N/4+32) buf; // +32 bytes as required by poly_cbd_eta1
  prf(buf.coeffs, KYBER_ETA1*KYBER_N/4, seed, nonce);
  poly_cbd_eta1(r, buf.vec);
}

/*************************************************
* 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)
{
  ALIGNED_UINT8(KYBER_ETA2*KYBER_N/4) buf;
  prf(buf.coeffs, KYBER_ETA2*KYBER_N/4, seed, nonce);
  poly_cbd_eta2(r, buf.vec);
}

#ifndef KYBER_90S
#define NOISE_NBLOCKS ((KYBER_ETA1*KYBER_N/4+SHAKE256_RATE-1)/SHAKE256_RATE)
```

```
void poly_getnoise_eta1_4x(poly *r0,
                           poly *r1,
                           poly *r2,
                           poly *r3,
                           const uint8_t seed[32],
                           uint8_t nonce0,
                           uint8_t nonce1,
                           uint8_t nonce2,
                           uint8_t nonce3)
{
  ALIGNED_UINT8(NOISE_NBLOCKS*SHAKE256_RATE) buf[4];
  __m256i f;
  keccakx4_state state;

  f = _mm256_loadu_si256((__m256i *)seed);
  _mm256_store_si256(buf[0].vec, f);
  _mm256_store_si256(buf[1].vec, f);
  _mm256_store_si256(buf[2].vec, f);
  _mm256_store_si256(buf[3].vec, f);

  buf[0].coeffs[32] = nonce0;
  buf[1].coeffs[32] = nonce1;
  buf[2].coeffs[32] = nonce2;
  buf[3].coeffs[32] = nonce3;

  shake256x4_absorb_once(&state, buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs
, 33);
  shake256x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, NOIS
E_NBLOCKS, &state);

  poly_cbd_eta1(r0, buf[0].vec);
  poly_cbd_eta1(r1, buf[1].vec);
  poly_cbd_eta1(r2, buf[2].vec);
  poly_cbd_eta1(r3, buf[3].vec);
}

#if KYBER_K == 2
void poly_getnoise_eta1122_4x(poly *r0,
                              poly *r1,
                              poly *r2,
                              poly *r3,
                              const uint8_t seed[32],
                              uint8_t nonce0,
                              uint8_t nonce1,
                              uint8_t nonce2,
                              uint8_t nonce3)
{
  ALIGNED_UINT8(NOISE_NBLOCKS*SHAKE256_RATE) buf[4];
  __m256i f;
  keccakx4_state state;

  f = _mm256_loadu_si256((__m256i *)seed);
  _mm256_store_si256(buf[0].vec, f);
  _mm256_store_si256(buf[1].vec, f);
  _mm256_store_si256(buf[2].vec, f);
  _mm256_store_si256(buf[3].vec, f);

  buf[0].coeffs[32] = nonce0;
  buf[1].coeffs[32] = nonce1;
  buf[2].coeffs[32] = nonce2;
  buf[3].coeffs[32] = nonce3;

  shake256x4_absorb_once(&state, buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs
, 33);
  shake256x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, NOIS
E_NBLOCKS, &state);

  poly_cbd_eta1(r0, buf[0].vec);
```

```c
  poly_cbd_eta1(r1, buf[1].vec);
  poly_cbd_eta2(r2, buf[2].vec);
  poly_cbd_eta2(r3, buf[3].vec);
}
#endif
#endif

/*************************************************
* Name:        poly_ntt
*
* Description: Computes negacyclic number-theoretic transform (NTT) of
*              a polynomial in place.
*              Input coefficients assumed to be in normal order,
*              output coefficients are in special order that is natural
*              for the vectorization. Input coefficients are assumed to be
*              bounded by q in absolute value, output coefficients are bounded
*              by 16118 in absolute value.
*
* Arguments:   - poly *r: pointer to in/output polynomial
**************************************************/
void poly_ntt(poly *r)
{
  ntt_avx(r->vec, qdata.vec);
}

/*************************************************
* Name:        poly_invntt_tomont
*
* Description: Computes inverse of negacyclic number-theoretic transform (NTT)
*              of a polynomial in place;
*              Input coefficients assumed to be in special order from vectorized
*              forward ntt, output in normal order. Input coefficients can be
*              arbitrary 16-bit integers, output coefficients are bounded by 14870
*              in absolute value.
*
* Arguments:   - poly *a: pointer to in/output polynomial
**************************************************/
void poly_invntt_tomont(poly *r)
{
  invntt_avx(r->vec, qdata.vec);
}

void poly_nttunpack(poly *r)
{
  nttunpack_avx(r->vec, qdata.vec);
}

/*************************************************
* Name:        poly_basemul_montgomery
*
* Description: Multiplication of two polynomials in NTT domain.
*              One of the input polynomials needs to have coefficients
*              bounded by q, the other polynomial can have arbitrary
*              coefficients. Output coefficients are bounded by 6656.
*
* 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)
{
  basemul_avx(r->vec, a->vec, b->vec, qdata.vec);
}

/*************************************************
* 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)
{
  tomont_avx(r->vec, qdata.vec);
}

/*************************************************
* 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)
{
  reduce_avx(r->vec, qdata.vec);
}

/*************************************************
* 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;
  __m256i f0, f1;

  for(i=0;i<KYBER_N/16;i++) {
    f0 = _mm256_load_si256(&a->vec[i]);
    f1 = _mm256_load_si256(&b->vec[i]);
    f0 = _mm256_add_epi16(f0, f1);
    _mm256_store_si256(&r->vec[i], f0);
  }
}

/*************************************************
* 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;
  __m256i f0, f1;

  for(i=0;i<KYBER_N/16;i++) {
    f0 = _mm256_load_si256(&a->vec[i]);
    f1 = _mm256_load_si256(&b->vec[i]);
    f0 = _mm256_sub_epi16(f0, f1);
    _mm256_store_si256(&r->vec[i], f0);
  }
}
```

```
#include <stdint.h>
#include <immintrin.h>
#include <string.h>
#include "params.h"
#include "polyvec.h"
#include "poly.h"
#include "ntt.h"
#include "consts.h"

#if (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 320))
static void poly_compress10(uint8_t r[320], const poly * restrict a)
{
  unsigned int i;
  __m256i f0, f1, f2;
  __m128i t0, t1;
  const __m256i v = _mm256_load_si256(&qdata.vec[_16XV/16]);
  const __m256i v8 = _mm256_slli_epi16(v,3);
  const __m256i off = _mm256_set1_epi16(15);
  const __m256i shift1 = _mm256_set1_epi16(1 << 12);
  const __m256i mask = _mm256_set1_epi16(1023);
  const __m256i shift2 = _mm256_set1_epi64x((1024LL << 48) + (1LL << 32) + (1024 << 16) + 1
);
  const __m256i sllvdidx = _mm256_set1_epi64x(12);
  const __m256i shufbidx = _mm256_set_epi8( 8, 4, 3, 2, 1, 0,-1,-1,-1,-1,-1,-1,12,11,10, 9,
                                           -1,-1,-1,-1,-1,-1,12,11,10, 9, 8, 4, 3, 2, 1, 0)
;

  for(i=0;i<KYBER_N/16;i++) {
    f0 = _mm256_load_si256(&a->vec[i]);
    f1 = _mm256_mullo_epi16(f0,v8);
    f2 = _mm256_add_epi16(f0,off);
    f0 = _mm256_slli_epi16(f0,3);
    f0 = _mm256_mulhi_epi16(f0,v);
    f2 = _mm256_sub_epi16(f1,f2);
    f1 = _mm256_andnot_si256(f1,f2);
    f1 = _mm256_srli_epi16(f1,15);
    f0 = _mm256_sub_epi16(f0,f1);
    f0 = _mm256_mulhrs_epi16(f0,shift1);
    f0 = _mm256_and_si256(f0,mask);
    f0 = _mm256_madd_epi16(f0,shift2);
    f0 = _mm256_sllv_epi32(f0,sllvdidx);
    f0 = _mm256_srli_epi64(f0,12);
    f0 = _mm256_shuffle_epi8(f0,shufbidx);
    t0 = _mm256_castsi256_si128(f0);
    t1 = _mm256_extracti128_si256(f0,1);
    t0 = _mm_blend_epi16(t0,t1,0xE0);
    _mm_storeu_si128((__m128i *)&r[20*i+ 0],t0);
    memcpy(&r[20*i+16],&t1,4);
  }
}

static void poly_decompress10(poly * restrict r, const uint8_t a[320+12])
{
  unsigned int i;
  __m256i f;
  const __m256i q = _mm256_set1_epi32((KYBER_Q << 16) + 4*KYBER_Q);
  const __m256i shufbidx = _mm256_set_epi8(11,10,10, 9, 9, 8, 8, 7,
                                            6, 5, 5, 4, 4, 3, 3, 2,
                                            9, 8, 8, 7, 7, 6, 6, 5,
                                            4, 3, 3, 2, 2, 1, 1, 0);
  const __m256i sllvdidx = _mm256_set1_epi64x(4);
  const __m256i mask = _mm256_set1_epi32((32736 << 16) + 8184);

  for(i=0;i<KYBER_N/16;i++) {
    f = _mm256_loadu_si256((__m256i *)&a[20*i]);
    f = _mm256_permute4x64_epi64(f,0x94);
    f = _mm256_shuffle_epi8(f,shufbidx);
    f = _mm256_sllv_epi32(f,sllvdidx);
```

```
    f = _mm256_srli_epi16(f,1);
    f = _mm256_and_si256(f,mask);
    f = _mm256_mulhrs_epi16(f,q);
    _mm256_store_si256(&r->vec[i],f);
  }
}

#elif (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 352))
static void poly_compress11(uint8_t r[352+2], const poly * restrict a)
{
  unsigned int i;
  __m256i f0, f1, f2;
  __m128i t0, t1;
  const __m256i v = _mm256_load_si256(&qdata.vec[_16XV/16]);
  const __m256i v8 = _mm256_slli_epi16(v,3);
  const __m256i off = _mm256_set1_epi16(36);
  const __m256i shift1 = _mm256_set1_epi16(1 << 13);
  const __m256i mask = _mm256_set1_epi16(2047);
  const __m256i shift2 = _mm256_set1_epi64x((2048LL << 48) + (1LL << 32) + (2048 << 16) + 1
);
  const __m256i sllvdidx = _mm256_set1_epi64x(10);
  const __m256i srlvqidx = _mm256_set_epi64x(30,10,30,10);
  const __m256i shufbidx = _mm256_set_epi8( 4, 3, 2, 1, 0, 0,-1,-1,-1,-1,10, 9, 8, 7, 6, 5,
                                           -1,-1,-1,-1,-1,10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
;

  for(i=0;i<KYBER_N/16;i++) {
    f0 = _mm256_load_si256(&a->vec[i]);
    f1 = _mm256_mullo_epi16(f0,v8);
    f2 = _mm256_add_epi16(f0,off);
    f0 = _mm256_slli_epi16(f0,3);
    f0 = _mm256_mulhi_epi16(f0,v);
    f2 = _mm256_sub_epi16(f1,f2);
    f1 = _mm256_andnot_si256(f1,f2);
    f1 = _mm256_srli_epi16(f1,15);
    f0 = _mm256_sub_epi16(f0,f1);
    f0 = _mm256_mulhrs_epi16(f0,shift1);
    f0 = _mm256_and_si256(f0,mask);
    f0 = _mm256_madd_epi16(f0,shift2);
    f0 = _mm256_sllv_epi32(f0,sllvdidx);
    f1 = _mm256_bsrli_epi128(f0,8);
    f0 = _mm256_srlv_epi64(f0,srlvqidx);
    f1 = _mm256_slli_epi64(f1,34);
    f0 = _mm256_add_epi64(f0,f1);
    f0 = _mm256_shuffle_epi8(f0,shufbidx);
    t0 = _mm256_castsi256_si128(f0);
    t1 = _mm256_extracti128_si256(f0,1);
    t0 = _mm_blendv_epi8(t0,t1,_mm256_castsi256_si128(shufbidx));
    _mm_storeu_si128((__m128i *)&r[22*i+ 0],t0);
    _mm_storel_epi64((__m128i *)&r[22*i+16],t1);
  }
}

static void poly_decompress11(poly * restrict r, const uint8_t a[352+10])
{
  unsigned int i;
  __m256i f;
  const __m256i q = _mm256_load_si256(&qdata.vec[_16XQ/16]);
  const __m256i shufbidx = _mm256_set_epi8(13,12,12,11,10, 9, 9, 8,
                                            8, 7, 6, 5, 5, 4, 4, 3,
                                           10, 9, 9, 8, 7, 6, 6, 5,
                                            5, 4, 3, 2, 2, 1, 1, 0);
  const __m256i srlvdidx = _mm256_set_epi32(0,0,1,0,0,0,1,0);
  const __m256i srlvqidx = _mm256_set_epi64x(2,0,2,0);
  const __m256i shift = _mm256_set_epi16(4,32,1,8,32,1,4,32,4,32,1,8,32,1,4,32);
  const __m256i mask = _mm256_set1_epi16(32752);

  for(i=0;i<KYBER_N/16;i++) {
```

```c
    f = _mm256_loadu_si256((__m256i *)&a[22*i]);
    f = _mm256_permute4x64_epi64(f,0x94);
    f = _mm256_shuffle_epi8(f,shufbidx);
    f = _mm256_srlv_epi32(f,srlvdidx);
    f = _mm256_srlv_epi64(f,srlvqidx);
    f = _mm256_mullo_epi16(f,shift);
    f = _mm256_srli_epi16(f,1);
    f = _mm256_and_si256(f,mask);
    f = _mm256_mulhrs_epi16(f,q);
    _mm256_store_si256(&r->vec[i],f);
  }
}

#endif

/*************************************************
* Name:        polyvec_compress
*
* Description: Compress and serialize vector of polynomials
*
* Arguments:   - uint8_t *r: pointer to output byte array
*                            (needs space for KYBER_POLYVECCOMPRESSEDBYTES)
*              - polyvec *a: pointer to input vector of polynomials
**************************************************/
void polyvec_compress(uint8_t r[KYBER_POLYVECCOMPRESSEDBYTES+2], const polyvec *a)
{
  unsigned int i;

#if (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 320))
  for(i=0;i<KYBER_K;i++)
    poly_compress10(&r[320*i],&a->vec[i]);
#elif (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 352))
  for(i=0;i<KYBER_K;i++)
    poly_compress11(&r[352*i],&a->vec[i]);
#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+12])
{
  unsigned int i;

#if (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 320))
  for(i=0;i<KYBER_K;i++)
    poly_decompress10(&r->vec[i],&a[320*i]);
#elif (KYBER_POLYVECCOMPRESSEDBYTES == (KYBER_K * 352))
  for(i=0;i<KYBER_K;i++)
    poly_decompress11(&r->vec[i],&a[352*i]);
#endif
}

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

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

/*************************************************
* 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;
```

```c
    }

    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 <immintrin.h>
#include <string.h>
#include "params.h"
#include "consts.h"
#include "rejsample.h"

//#define BMI

#ifndef BMI
static const uint8_t idx[256][8] = {
  {-1, -1, -1, -1, -1, -1, -1, -1},
  { 0, -1, -1, -1, -1, -1, -1, -1},
  { 2, -1, -1, -1, -1, -1, -1, -1},
  { 0,  2, -1, -1, -1, -1, -1, -1},
  { 4, -1, -1, -1, -1, -1, -1, -1},
  { 0,  4, -1, -1, -1, -1, -1, -1},
  { 2,  4, -1, -1, -1, -1, -1, -1},
  { 0,  2,  4, -1, -1, -1, -1, -1},
  { 6, -1, -1, -1, -1, -1, -1, -1},
  { 0,  6, -1, -1, -1, -1, -1, -1},
  { 2,  6, -1, -1, -1, -1, -1, -1},
  { 0,  2,  6, -1, -1, -1, -1, -1},
  { 4,  6, -1, -1, -1, -1, -1, -1},
  { 0,  4,  6, -1, -1, -1, -1, -1},
  { 2,  4,  6, -1, -1, -1, -1, -1},
  { 0,  2,  4,  6, -1, -1, -1, -1},
  { 8, -1, -1, -1, -1, -1, -1, -1},
  { 0,  8, -1, -1, -1, -1, -1, -1},
  { 2,  8, -1, -1, -1, -1, -1, -1},
  { 0,  2,  8, -1, -1, -1, -1, -1},
  { 4,  8, -1, -1, -1, -1, -1, -1},
  { 0,  4,  8, -1, -1, -1, -1, -1},
  { 2,  4,  8, -1, -1, -1, -1, -1},
  { 0,  2,  4,  8, -1, -1, -1, -1},
  { 6,  8, -1, -1, -1, -1, -1, -1},
  { 0,  6,  8, -1, -1, -1, -1, -1},
  { 2,  6,  8, -1, -1, -1, -1, -1},
  { 0,  2,  6,  8, -1, -1, -1, -1},
  { 4,  6,  8, -1, -1, -1, -1, -1},
  { 0,  4,  6,  8, -1, -1, -1, -1},
  { 2,  4,  6,  8, -1, -1, -1, -1},
  { 0,  2,  4,  6,  8, -1, -1, -1},
  {10, -1, -1, -1, -1, -1, -1, -1},
  { 0, 10, -1, -1, -1, -1, -1, -1},
  { 2, 10, -1, -1, -1, -1, -1, -1},
  { 0,  2, 10, -1, -1, -1, -1, -1},
  { 4, 10, -1, -1, -1, -1, -1, -1},
  { 0,  4, 10, -1, -1, -1, -1, -1},
  { 2,  4, 10, -1, -1, -1, -1, -1},
  { 0,  2,  4, 10, -1, -1, -1, -1},
  { 6, 10, -1, -1, -1, -1, -1, -1},
  { 0,  6, 10, -1, -1, -1, -1, -1},
  { 2,  6, 10, -1, -1, -1, -1, -1},
  { 0,  2,  6, 10, -1, -1, -1, -1},
  { 4,  6, 10, -1, -1, -1, -1, -1},
  { 0,  4,  6, 10, -1, -1, -1, -1},
  { 2,  4,  6, 10, -1, -1, -1, -1},
  { 0,  2,  4,  6, 10, -1, -1, -1},
  { 8, 10, -1, -1, -1, -1, -1, -1},
  { 0,  8, 10, -1, -1, -1, -1, -1},
  { 2,  8, 10, -1, -1, -1, -1, -1},
  { 0,  2,  8, 10, -1, -1, -1, -1},
  { 4,  8, 10, -1, -1, -1, -1, -1},
  { 0,  4,  8, 10, -1, -1, -1, -1},
  { 2,  4,  8, 10, -1, -1, -1, -1},
  { 0,  2,  4,  8, 10, -1, -1, -1},
  { 6,  8, 10, -1, -1, -1, -1, -1},
```

```
  { 0,  6,  8, 10, -1, -1, -1, -1},
  { 2,  6,  8, 10, -1, -1, -1, -1},
  { 0,  2,  6,  8, 10, -1, -1, -1},
  { 4,  6,  8, 10, -1, -1, -1, -1},
  { 0,  4,  6,  8, 10, -1, -1, -1},
  { 2,  4,  6,  8, 10, -1, -1, -1},
  { 0,  2,  4,  6,  8, 10, -1, -1},
  {12, -1, -1, -1, -1, -1, -1, -1},
  { 0, 12, -1, -1, -1, -1, -1, -1},
  { 2, 12, -1, -1, -1, -1, -1, -1},
  { 0,  2, 12, -1, -1, -1, -1, -1},
  { 4, 12, -1, -1, -1, -1, -1, -1},
  { 0,  4, 12, -1, -1, -1, -1, -1},
  { 2,  4, 12, -1, -1, -1, -1, -1},
  { 0,  2,  4, 12, -1, -1, -1, -1},
  { 6, 12, -1, -1, -1, -1, -1, -1},
  { 0,  6, 12, -1, -1, -1, -1, -1},
  { 2,  6, 12, -1, -1, -1, -1, -1},
  { 0,  2,  6, 12, -1, -1, -1, -1},
  { 4,  6, 12, -1, -1, -1, -1, -1},
  { 0,  4,  6, 12, -1, -1, -1, -1},
  { 2,  4,  6, 12, -1, -1, -1, -1},
  { 0,  2,  4,  6, 12, -1, -1, -1},
  { 8, 12, -1, -1, -1, -1, -1, -1},
  { 0,  8, 12, -1, -1, -1, -1, -1},
  { 2,  8, 12, -1, -1, -1, -1, -1},
  { 0,  2,  8, 12, -1, -1, -1, -1},
  { 4,  8, 12, -1, -1, -1, -1, -1},
  { 0,  4,  8, 12, -1, -1, -1, -1},
  { 2,  4,  8, 12, -1, -1, -1, -1},
  { 0,  2,  4,  8, 12, -1, -1, -1},
  { 6,  8, 12, -1, -1, -1, -1, -1},
  { 0,  6,  8, 12, -1, -1, -1, -1},
  { 2,  6,  8, 12, -1, -1, -1, -1},
  { 0,  2,  6,  8, 12, -1, -1, -1},
  { 4,  6,  8, 12, -1, -1, -1, -1},
  { 0,  4,  6,  8, 12, -1, -1, -1},
  { 2,  4,  6,  8, 12, -1, -1, -1},
  { 0,  2,  4,  6,  8, 12, -1, -1},
  {10, 12, -1, -1, -1, -1, -1, -1},
  { 0, 10, 12, -1, -1, -1, -1, -1},
  { 2, 10, 12, -1, -1, -1, -1, -1},
  { 0,  2, 10, 12, -1, -1, -1, -1},
  { 4, 10, 12, -1, -1, -1, -1, -1},
  { 0,  4, 10, 12, -1, -1, -1, -1},
  { 2,  4, 10, 12, -1, -1, -1, -1},
  { 0,  2,  4, 10, 12, -1, -1, -1},
  { 6, 10, 12, -1, -1, -1, -1, -1},
  { 0,  6, 10, 12, -1, -1, -1, -1},
  { 2,  6, 10, 12, -1, -1, -1, -1},
  { 0,  2,  6, 10, 12, -1, -1, -1},
  { 4,  6, 10, 12, -1, -1, -1, -1},
  { 0,  4,  6, 10, 12, -1, -1, -1},
  { 2,  4,  6, 10, 12, -1, -1, -1},
  { 0,  2,  4,  6, 10, 12, -1, -1},
  { 8, 10, 12, -1, -1, -1, -1, -1},
  { 0,  8, 10, 12, -1, -1, -1, -1},
  { 2,  8, 10, 12, -1, -1, -1, -1},
  { 0,  2,  8, 10, 12, -1, -1, -1},
  { 4,  8, 10, 12, -1, -1, -1, -1},
  { 0,  4,  8, 10, 12, -1, -1, -1},
  { 2,  4,  8, 10, 12, -1, -1, -1},
  { 0,  2,  4,  8, 10, 12, -1, -1},
  { 6,  8, 10, 12, -1, -1, -1, -1},
  { 0,  6,  8, 10, 12, -1, -1, -1},
  { 2,  6,  8, 10, 12, -1, -1, -1},
  { 0,  2,  6,  8, 10, 12, -1, -1},
  { 4,  6,  8, 10, 12, -1, -1, -1},
```

```
  { 0,  4,  6,  8, 10, 12, -1, -1},
  { 2,  4,  6,  8, 10, 12, -1, -1},
  { 0,  2,  4,  6,  8, 10, 12, -1},
  {14, -1, -1, -1, -1, -1, -1, -1},
  { 0, 14, -1, -1, -1, -1, -1, -1},
  { 2, 14, -1, -1, -1, -1, -1, -1},
  { 0,  2, 14, -1, -1, -1, -1, -1},
  { 4, 14, -1, -1, -1, -1, -1, -1},
  { 0,  4, 14, -1, -1, -1, -1, -1},
  { 2,  4, 14, -1, -1, -1, -1, -1},
  { 0,  2,  4, 14, -1, -1, -1, -1},
  { 6, 14, -1, -1, -1, -1, -1, -1},
  { 0,  6, 14, -1, -1, -1, -1, -1},
  { 2,  6, 14, -1, -1, -1, -1, -1},
  { 0,  2,  6, 14, -1, -1, -1, -1},
  { 4,  6, 14, -1, -1, -1, -1, -1},
  { 0,  4,  6, 14, -1, -1, -1, -1},
  { 2,  4,  6, 14, -1, -1, -1, -1},
  { 0,  2,  4,  6, 14, -1, -1, -1},
  { 8, 14, -1, -1, -1, -1, -1, -1},
  { 0,  8, 14, -1, -1, -1, -1, -1},
  { 2,  8, 14, -1, -1, -1, -1, -1},
  { 0,  2,  8, 14, -1, -1, -1, -1},
  { 4,  8, 14, -1, -1, -1, -1, -1},
  { 0,  4,  8, 14, -1, -1, -1, -1},
  { 2,  4,  8, 14, -1, -1, -1, -1},
  { 0,  2,  4,  8, 14, -1, -1, -1},
  { 6,  8, 14, -1, -1, -1, -1, -1},
  { 0,  6,  8, 14, -1, -1, -1, -1},
  { 2,  6,  8, 14, -1, -1, -1, -1},
  { 0,  2,  6,  8, 14, -1, -1, -1},
  { 4,  6,  8, 14, -1, -1, -1, -1},
  { 0,  4,  6,  8, 14, -1, -1, -1},
  { 2,  4,  6,  8, 14, -1, -1, -1},
  { 0,  2,  4,  6,  8, 14, -1, -1},
  {10, 14, -1, -1, -1, -1, -1, -1},
  { 0, 10, 14, -1, -1, -1, -1, -1},
  { 2, 10, 14, -1, -1, -1, -1, -1},
  { 0,  2, 10, 14, -1, -1, -1, -1},
  { 4, 10, 14, -1, -1, -1, -1, -1},
  { 0,  4, 10, 14, -1, -1, -1, -1},
  { 2,  4, 10, 14, -1, -1, -1, -1},
  { 0,  2,  4, 10, 14, -1, -1, -1},
  { 6, 10, 14, -1, -1, -1, -1, -1},
  { 0,  6, 10, 14, -1, -1, -1, -1},
  { 2,  6, 10, 14, -1, -1, -1, -1},
  { 0,  2,  6, 10, 14, -1, -1, -1},
  { 4,  6, 10, 14, -1, -1, -1, -1},
  { 0,  4,  6, 10, 14, -1, -1, -1},
  { 2,  4,  6, 10, 14, -1, -1, -1},
  { 0,  2,  4,  6, 10, 14, -1, -1},
  { 8, 10, 14, -1, -1, -1, -1, -1},
  { 0,  8, 10, 14, -1, -1, -1, -1},
  { 2,  8, 10, 14, -1, -1, -1, -1},
  { 0,  2,  8, 10, 14, -1, -1, -1},
  { 4,  8, 10, 14, -1, -1, -1, -1},
  { 0,  4,  8, 10, 14, -1, -1, -1},
  { 2,  4,  8, 10, 14, -1, -1, -1},
  { 0,  2,  4,  8, 10, 14, -1, -1},
  { 6,  8, 10, 14, -1, -1, -1, -1},
  { 0,  6,  8, 10, 14, -1, -1, -1},
  { 2,  6,  8, 10, 14, -1, -1, -1},
  { 0,  2,  6,  8, 10, 14, -1, -1},
  { 4,  6,  8, 10, 14, -1, -1, -1},
  { 0,  4,  6,  8, 10, 14, -1, -1},
  { 2,  4,  6,  8, 10, 14, -1, -1},
  { 0,  2,  4,  6,  8, 10, 14, -1},
  {12, 14, -1, -1, -1, -1, -1, -1},
```

```
  { 0, 12, 14, -1, -1, -1, -1, -1},
  { 2, 12, 14, -1, -1, -1, -1, -1},
  { 0,  2, 12, 14, -1, -1, -1, -1},
  { 4, 12, 14, -1, -1, -1, -1, -1},
  { 0,  4, 12, 14, -1, -1, -1, -1},
  { 2,  4, 12, 14, -1, -1, -1, -1},
  { 0,  2,  4, 12, 14, -1, -1, -1},
  { 6, 12, 14, -1, -1, -1, -1, -1},
  { 0,  6, 12, 14, -1, -1, -1, -1},
  { 2,  6, 12, 14, -1, -1, -1, -1},
  { 0,  2,  6, 12, 14, -1, -1, -1},
  { 4,  6, 12, 14, -1, -1, -1, -1},
  { 0,  4,  6, 12, 14, -1, -1, -1},
  { 2,  4,  6, 12, 14, -1, -1, -1},
  { 0,  2,  4,  6, 12, 14, -1, -1},
  { 8, 12, 14, -1, -1, -1, -1, -1},
  { 0,  8, 12, 14, -1, -1, -1, -1},
  { 2,  8, 12, 14, -1, -1, -1, -1},
  { 0,  2,  8, 12, 14, -1, -1, -1},
  { 4,  8, 12, 14, -1, -1, -1, -1},
  { 0,  4,  8, 12, 14, -1, -1, -1},
  { 2,  4,  8, 12, 14, -1, -1, -1},
  { 0,  2,  4,  8, 12, 14, -1, -1},
  { 6,  8, 12, 14, -1, -1, -1, -1},
  { 0,  6,  8, 12, 14, -1, -1, -1},
  { 2,  6,  8, 12, 14, -1, -1, -1},
  { 0,  2,  6,  8, 12, 14, -1, -1},
  { 4,  6,  8, 12, 14, -1, -1, -1},
  { 0,  4,  6,  8, 12, 14, -1, -1},
  { 2,  4,  6,  8, 12, 14, -1, -1},
  { 0,  2,  4,  6,  8, 12, 14, -1},
  {10, 12, 14, -1, -1, -1, -1, -1},
  { 0, 10, 12, 14, -1, -1, -1, -1},
  { 2, 10, 12, 14, -1, -1, -1, -1},
  { 0,  2, 10, 12, 14, -1, -1, -1},
  { 4, 10, 12, 14, -1, -1, -1, -1},
  { 0,  4, 10, 12, 14, -1, -1, -1},
  { 2,  4, 10, 12, 14, -1, -1, -1},
  { 0,  2,  4, 10, 12, 14, -1, -1},
  { 6, 10, 12, 14, -1, -1, -1, -1},
  { 0,  6, 10, 12, 14, -1, -1, -1},
  { 2,  6, 10, 12, 14, -1, -1, -1},
  { 0,  2,  6, 10, 12, 14, -1, -1},
  { 4,  6, 10, 12, 14, -1, -1, -1},
  { 0,  4,  6, 10, 12, 14, -1, -1},
  { 2,  4,  6, 10, 12, 14, -1, -1},
  { 0,  2,  4,  6, 10, 12, 14, -1},
  { 8, 10, 12, 14, -1, -1, -1, -1},
  { 0,  8, 10, 12, 14, -1, -1, -1},
  { 2,  8, 10, 12, 14, -1, -1, -1},
  { 0,  2,  8, 10, 12, 14, -1, -1},
  { 4,  8, 10, 12, 14, -1, -1, -1},
  { 0,  4,  8, 10, 12, 14, -1, -1},
  { 2,  4,  8, 10, 12, 14, -1, -1},
  { 0,  2,  4,  8, 10, 12, 14, -1},
  { 6,  8, 10, 12, 14, -1, -1, -1},
  { 0,  6,  8, 10, 12, 14, -1, -1},
  { 2,  6,  8, 10, 12, 14, -1, -1},
  { 0,  2,  6,  8, 10, 12, 14, -1},
  { 4,  6,  8, 10, 12, 14, -1, -1},
  { 0,  4,  6,  8, 10, 12, 14, -1},
  { 2,  4,  6,  8, 10, 12, 14, -1},
  { 0,  2,  4,  6,  8, 10, 12, 14}
};
#endif

#define _mm256_cmpge_epu16(a, b) _mm256_cmpeq_epi16(_mm256_max_epu16(a, b), a)
#define _mm_cmpge_epu16(a, b) _mm_cmpeq_epi16(_mm_max_epu16(a, b), a)
```

```
unsigned int rej_uniform_avx(int16_t * restrict r, const uint8_t *buf)
{
  unsigned int ctr, pos;
  uint16_t val0, val1;
  uint32_t good;
#ifdef BMI
  uint64_t idx0, idx1, idx2, idx3;
#endif
  const __m256i bound  = _mm256_load_si256(&qdata.vec[_16XQ/16]);
  const __m256i ones   = _mm256_set1_epi8(1);
  const __m256i mask  = _mm256_set1_epi16(0xFFF);
  const __m256i idx8  = _mm256_set_epi8(15,14,14,13,12,11,11,10,
                                         9, 8, 8, 7, 6, 5, 5, 4,
                                        11,10,10, 9, 8, 7, 7, 6,
                                         5, 4, 4, 3, 2, 1, 1, 0);
  __m256i f0, f1, g0, g1, g2, g3;
  __m128i f, t, pilo, pihi;

  ctr = pos = 0;
  while(ctr <= KYBER_N - 32 && pos <= REJ_UNIFORM_AVX_BUFLEN - 48) {
    f0 = _mm256_loadu_si256((__m256i *)&buf[pos]);
    f1 = _mm256_loadu_si256((__m256i *)&buf[pos+24]);
    f0 = _mm256_permute4x64_epi64(f0, 0x94);
    f1 = _mm256_permute4x64_epi64(f1, 0x94);
    f0 = _mm256_shuffle_epi8(f0, idx8);
    f1 = _mm256_shuffle_epi8(f1, idx8);
    g0 = _mm256_srli_epi16(f0, 4);
    g1 = _mm256_srli_epi16(f1, 4);
    f0 = _mm256_blend_epi16(f0, g0, 0xAA);
    f1 = _mm256_blend_epi16(f1, g1, 0xAA);
    f0 = _mm256_and_si256(f0, mask);
    f1 = _mm256_and_si256(f1, mask);
    pos += 48;

    g0 = _mm256_cmpgt_epi16(bound, f0);
    g1 = _mm256_cmpgt_epi16(bound, f1);

    g0 = _mm256_packs_epi16(g0, g1);
    good = _mm256_movemask_epi8(g0);

#ifdef BMI
    idx0 = _pdep_u64(good >>  0, 0x0101010101010101);
    idx1 = _pdep_u64(good >>  8, 0x0101010101010101);
    idx2 = _pdep_u64(good >> 16, 0x0101010101010101);
    idx3 = _pdep_u64(good >> 24, 0x0101010101010101);
    idx0 = (idx0 << 8) - idx0;
    idx0  = _pext_u64(0x0E0C0A0806040200, idx0);
    idx1 = (idx1 << 8) - idx1;
    idx1  = _pext_u64(0x0E0C0A0806040200, idx1);
    idx2 = (idx2 << 8) - idx2;
    idx2  = _pext_u64(0x0E0C0A0806040200, idx2);
    idx3 = (idx3 << 8) - idx3;
    idx3  = _pext_u64(0x0E0C0A0806040200, idx3);

    g0 = _mm256_castsi128_si256(_mm_cvtsi64_si128(idx0));
    g1 = _mm256_castsi128_si256(_mm_cvtsi64_si128(idx1));
    g0 = _mm256_inserti128_si256(g0, _mm_cvtsi64_si128(idx2), 1);
    g1 = _mm256_inserti128_si256(g1, _mm_cvtsi64_si128(idx3), 1);
#else
    g0 = _mm256_castsi128_si256(_mm_loadl_epi64((__m128i *)&idx[(good >>  0) & 0xFF]));
    g1 = _mm256_castsi128_si256(_mm_loadl_epi64((__m128i *)&idx[(good >>  8) & 0xFF]));
    g0 = _mm256_inserti128_si256(g0, _mm_loadl_epi64((__m128i *)&idx[(good >> 16) & 0xFF]),
 1);
    g1 = _mm256_inserti128_si256(g1, _mm_loadl_epi64((__m128i *)&idx[(good >> 24) & 0xFF]),
 1);
#endif
```

```
    g2 = _mm256_add_epi8(g0, ones);
    g3 = _mm256_add_epi8(g1, ones);
    g0 = _mm256_unpacklo_epi8(g0, g2);
    g1 = _mm256_unpacklo_epi8(g1, g3);

    f0 = _mm256_shuffle_epi8(f0, g0);
    f1 = _mm256_shuffle_epi8(f1, g1);

    _mm_storeu_si128((__m128i *)&r[ctr], _mm256_castsi256_si128(f0));
    ctr += _mm_popcnt_u32((good >>  0) & 0xFF);
    _mm_storeu_si128((__m128i *)&r[ctr], _mm256_extracti128_si256(f0, 1));
    ctr += _mm_popcnt_u32((good >> 16) & 0xFF);
    _mm_storeu_si128((__m128i *)&r[ctr], _mm256_castsi256_si128(f1));
    ctr += _mm_popcnt_u32((good >>  8) & 0xFF);
    _mm_storeu_si128((__m128i *)&r[ctr], _mm256_extracti128_si256(f1, 1));
    ctr += _mm_popcnt_u32((good >> 24) & 0xFF);
  }

  while(ctr <= KYBER_N - 8 && pos <= REJ_UNIFORM_AVX_BUFLEN - 12) {
    f = _mm_loadu_si128((__m128i *)&buf[pos]);
    f = _mm_shuffle_epi8(f, _mm256_castsi256_si128(idx8));
    t = _mm_srli_epi16(f, 4);
    f = _mm_blend_epi16(f, t, 0xAA);
    f = _mm_and_si128(f, _mm256_castsi256_si128(mask));
    pos += 12;

    t = _mm_cmpgt_epi16(_mm256_castsi256_si128(bound), f);
    good = _mm_movemask_epi8(t);

#ifdef BMI
    good &= 0x5555;
    idx0 = _pdep_u64(good, 0x1111111111111111);
    idx0 = (idx0 << 8) - idx0;
    idx0 = _pext_u64(0x0E0C0A0806040200, idx0);
    pilo = _mm_cvtsi64_si128(idx0);
#else
    good = _pext_u32(good, 0x5555);
    pilo = _mm_loadl_epi64((__m128i *)&idx[good]);
#endif

    pihi = _mm_add_epi8(pilo, _mm256_castsi256_si128(ones));
    pilo = _mm_unpacklo_epi8(pilo, pihi);
    f = _mm_shuffle_epi8(f, pilo);
    _mm_storeu_si128((__m128i *)&r[ctr], f);
    ctr += _mm_popcnt_u32(good);
  }

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

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

  return ctr;
}
```

```
//
//  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);
```

```c
}

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
#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");
}
```

```
#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];
```

```
  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 "randombytes.h"
#include "cpucycles.h"
#include "speed_print.h"

#define NTESTS 1000

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

/* Dummy randombytes for speed tests that simulates a fast randombytes implementation
 * as in SUPERCOP so that we get comparable cycle counts */
void randombytes(__attribute__((unused)) uint8_t *r, __attribute__((unused)) size_t len) {
  return;
}

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;
#ifndef KYBER_90S
  poly bp, cp, dp;
#endif

  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);

#ifndef KYBER_90S
  for(i=0;i<NTESTS;i++) {
    t[i] = cpucycles();
    poly_getnoise_eta1_4x(&ap, &bp, &cp, &dp, seed, 0, 1, 2, 3);
  }
  print_results("poly_getnoise_eta1_4x: ", t, NTESTS);
#endif

  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);
```

```
    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 <stdlib.h>
#include <stdint.h>
#include <immintrin.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;
  uint64_t r;
  __m256i f, g, h;

  h = _mm256_setzero_si256();
  for(i=0;i<len/32;i++) {
    f = _mm256_loadu_si256((__m256i *)&a[32*i]);
    g = _mm256_loadu_si256((__m256i *)&b[32*i]);
    f = _mm256_xor_si256(f,g);
    h = _mm256_or_si256(h,f);
  }
  r = 1 - _mm256_testz_si256(h,h);

  a += 32*i;
  b += 32*i;
  len -= 32*i;
  for(i=0;i<len;i++)
    r |= a[i] ^ b[i];

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

/*************************************************
* 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 * restrict r, const uint8_t *x, size_t len, uint8_t b)
{
  size_t i;
  __m256i xvec, rvec, bvec;

#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
```

```c
  bvec = _mm256_set1_epi64x(-(uint64_t)b);
  for(i=0;i<len/32;i++) {
    rvec = _mm256_loadu_si256((__m256i *)&r[32*i]);
    xvec = _mm256_loadu_si256((__m256i *)&x[32*i]);
    rvec = _mm256_blendv_epi8(rvec,xvec,bvec);
    _mm256_storeu_si256((__m256i *)&r[32*i],rvec);
  }

  r += 32*i;
  x += 32*i;
  len -= 32*i;
  for(i=0;i<len;i++)
    r[i] ^= -b & (x[i] ^ r[i]);
}
```