A Cover Sheet

HILA5

Version 0.95 – Working Draft

Friday 20th October, 2017

HILA5 Key Encapsulation Mechanism (KEM)
and Public Key Encryption Algorithm

Author and Submitter

Markku-Juhani O. Saarinen

E-mail: mjos@iki.fi

(Independent Submission)

Poste Restante, City Post Office, 57-58 St Andrew's St, Cambridge CB2 3BZ, UK.
Tel: +1 (202) 559-0658

Executive Summary

Some classes of encrypted data must remain confidential for a long period of time – often at least few decades in national security applications. Therefore high-security cryptography should be resistant to attacks even with projected future technologies.

As there are no physical or theoretical barriers preventing progressive development of quantum computing technologies capable of breaking current RSA- and Elliptic Curve based cryptographic standards (using polynomial-time quantum algorithms already known [PZ03, Sho94]), a need for such quantum-resistant algorithms in national security applications has been identified [NSA16].

In December 2016 NIST issued a standardization call for quantum-resistant public key algorithms, together with requirements and evaluation criteria [NIS16]. This has made "Post-Quantum Cryptography" (PQC) central to cryptographic engineers who must now design concrete proposals for standardization. Practical issues such as performance, reliability, message and key sizes, implementation and side-channel security, and compatibility with existing and anticipated applications, protocols, and standards are as relevant as mere theoretical security and asymptotic feasibility when evaluating these proposals.

Ring-LWE lattice primitives offer some of the best performance and key size characteristics among quantum-resistant candidates [CJL+16]. These algorithms rely on "random noise" for security and always have some risk of decryption failure. This reliability issue can pose problems when used in non-interactive applications which are not designed to tolerate errors. The issue of decryption failure can be addressed via reconciliation methods, which is the focus of present work.

Our proposal, HILA5 [Saa17b] uses a new reconciliation method for Ring-LWE that has a significantly smaller failure rate than previous proposals while reducing ciphertext size and the amount of randomness required. It is based on a simple, deterministic variant of Peikert's reconciliation that works with our new "safe bits" selection and constant-time error correction techniques. The new method does not need randomized smoothing to achieve non-biased secrets.

When used with the very efficient "New Hope" Ring-LWE parametrization we achieve a decryption failure rate well below 2^{-128} – compared to $2^{-38.9}$ for recommended parameters of Frodo, 2^{-60} of New Hope, and even $2^{-71.9}$ of Kyber. This makes the scheme suitable for public key encryption in addition to key exchange protocols; the reconciliation approach saves about 40% in ciphertext size when compared to the common LP11 Ring-LWE encryption scheme.

We perform a combinatorial failure analysis using full probability convolutions, leading to a precise understanding of decryption failure conditions on bit level. Even with additional implementation security and safety measures the new scheme is still essentially as fast as the New Hope but has slightly shorter messages. The new techniques have been instantiated and implemented as a Key Encapsulation Mechanism (KEM) and public key encryption scheme designed to meet the requirements of NIST's Post-Quantum Cryptography effort at very high security level.

Acknowledgements. Much of HILA5 design and implementation work was performed during first half of 2017 while the author was employed by DARKMATTER LLC (UAE). The author therefore wishes to thank the DARKMATTER Crypto Team and Dr. Najwa Aaraj for providing feedback and for their support. However, this is an independent, individual submission. There are no patents, overly restrictive intellectual property claims, or other such corporate entanglements. All source code is released under "MIT" License.

Contents

Note. This is a submission document in response to the NIST call for quantum resistant algorithm proposals, and the structure of this document mostly follows their December 2016 call for proposals: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography

B Algorithm Specification and Supporting Documentation

B.1 Specification

The purpose of this section is to offer a clear functional description of the HILA5 algorithm in a way that is suitable for non-expert implementors as a compact starting point. For a more abstract treatment and a theoretical justification of HILA5, see Section B.4.2.

We are including snippets of C code from our unoptimized reference implementation, which is available (together with the latest version of this specification, optimized implementations, and full test data) at https://github.com/mjosaarinen/hila5

This reference implementation is not suitable for production use. Our optimized implementation has significantly better performance as it uses more advanced (and preferred) algorithmic techniques. The two implementations are fully compatible.

The HILA5 KEM can be adopted for Public Key Encryption in straightforward fashion. We recommend using the AES-256-GCM AEAD [FIP01, Dwo07] in conjunction with the KEM when public key encryption functionality is desired. If a suitable AEAD based a large permutation is standardized by NIST (e.g. Keyak [BDP+16] – based on SHA-3 Keccak) at some point in future, we suggest using it for increased security.

B.1.1 Rings and Number Theoretic Transforms

HILA5's ring arithmetic operates on polynomials of degree $n = 1024$. Polynomials are represented as 1024-element vectors of integers. Each coefficient is reduced mod q, where $q = 12289$. Reduction $x \mod q$ puts a number in non-negative range $0 \leq x < q$. Let $\mathcal{R}$ be a ring with elements $\mathbf{v} \in \mathbb{Z}_q^n$. Its coefficients $v_i \in [0, q-1]$ $(0 \leq i < n)$ can be interpreted as a polynomial via $v(x) = \sum_{i=0}^{n-1} v_i x^i$, or as a zero-indexed vector. This algebraic object $\mathcal{R}$ is a "ring" (and not a "field") since not all non-zero polynomials have unique inverses.

Adding and Scaling. Addition, subtraction, and scaling (scalar multiplication with c) follow the basic rules for polynomials or vectors.

```
#include <stdint.h>
#define HILA5_N         1024
#define HILA5_Q         12289

// Vector addition: d = a + b.

void slow_vadd(int32_t d[HILA5_N],
    const int32_t a[HILA5_N], const int32_t b[HILA5_N])
{
    for (int i = 0; i < HILA5_N; i++)
        d[i] = (a[i] + b[i]) % HILA5_Q;
}

// Scalar multiplication: v = c * v.

void slow_smul(int32_t v[HILA5_N], int32_t c)
{
    for (int i = 0; i < HILA5_N; i++)
        v[i] = (c * v[i]) % HILA5_Q;
}
```

Multiplication. For multiplication we use cyclotomic polynomial basis $\mathbb{Z}_q[x]/(x^n + 1)$. Products are reduced modulo q and $x^n + 1$ and results are therefore bound by degree $n - 1$ since $x^n \equiv q - 1$. We may write a direct "negative wrap-around" multiplication rule as:

$$\mathbf{h} = \mathbf{f} * \mathbf{g} \mod (x^n + 1) \iff h_i = \sum_{j=0}^{i} f_j g_{(i-j)} - \sum_{j=i+1}^{n-1} f_j g_{(n+i-j)}. \quad (1)$$

Algorithmically the multiplication rule of Equation 1 requires $O(n^2)$ elementary operations.

```
// Slow polynomial ring multiplication: d = a * b  (mod x^1024 + 1)

void slow_rmul(int32_t d[HILA5_N],
    const int32_t a[HILA5_N], const int32_t b[HILA5_N])
{
    int32_t x;

    for (int i = 0; i < HILA5_N; i++) {
        x = 0;
        for (int j = 0; j <= i; j++)            // positive side
            x = (x + a[j] * b[i - j]) % HILA5_Q;
        for (int j = i + 1; j < HILA5_N; j++)   // negative wraparound
            x = (x - a[j] * b[HILA5_N + i - j]) % HILA5_Q;
        // Force into positive [0, q-1] range
        d[i] = x + (-((x >> 31) & 1) & HILA5_Q);
    }
}
```

Number Theoretic Transforms. There is an $O(n \log n)$ multiplication method in this ring, originally due to Nussbaumer [Nus80], which uses the fast Number Theoretic Transform (NTT). However, we will only describe the "slow" NTT here and focus on encoding details.

Since some quantities are transmitted on the wire in the transformed domain, we must specify details of this domain even for the unoptimized implementation. We use generator $g = 1945$ which has multiplicative order of $2^{11} = 2048$ in $\mathbb{Z}^*_{12289}$. It follows that

$$g^n \equiv -1 \pmod{q}. \tag{2}$$

Condition of Equation 2 is necessary for the working of the negacyclic Nussbaumer transform. In our reference implementation we store powers of g in table `pow1945[2048]`.

```
static int32_t pow1945[2048];           // powers of g=1945 mod q
static int pow1945_ok = 0;              // true after initialization

// make sure that the pow1945[] table is initialized

void init_pow1945()
{
    if (pow1945_ok)                     // nothing to do then
        return;

    int x = 1;                          // 1945^0 = 1
    for (int i = 0; i < 2048; i++) {    // 1945^1024 = -1 (mod q)
        pow1945[i] = x;
        x = (1945 * x) % HILA5_Q;       // consecutive powers
    }
    pow1945_ok = !0;                    // table now ok
}
```

To be compatible with the bit-reversed fast transform in the optimized implementation, we need to specify a helper function

$$\mathsf{BitRev10}(x) = \sum_{i=0}^{9} 2^i \left(\left\lfloor \frac{x}{2^{9-i}} \right\rfloor \mod 2 \right) \tag{3}$$

```
// reverse order of ten bits i.e. 0x200 -> 0x001 and vice versa

int32_t bitrev10(int32_t x)
{
    int t;

    x &= 0x3FF;                         // 9876543210 original order
    x = (x << 5) | (x >> 5);            // 4321098765 5/5 bit swap
    t = (x ^ (x >> 4)) & 0x021;
    x ^= t ^ (t << 4);                  // 0321458769 outer bit swap
    t = (x ^ (x >> 2)) & 0x042;
    x ^= t ^ (t << 2);                  // 0123456789 inner bit swap

    return x & 0x3FF;
}
```

We may now define the equivalent transform as

$$\mathsf{NTT}(\mathbf{v}) = \hat{\mathbf{v}} \quad \text{with} \quad \hat{v}_i = \sum_{j=0}^{n-1} v_j g^{j*(2*\mathsf{BitRev10}(i)+1)} \quad \text{for each} \quad i \in [\,0, n-1\,]. \tag{4}$$

Our reference implementation uses this slow method (to avoid confusion with fast transforms, these functions are prefixed with `slow_`):

```
// Slow number theoretic transform and scaling: d = c * NTT(v).

void slow_ntt(int32_t d[HILA5_N], const int32_t v[HILA5_N], int32_t c)
{
    int k, r;
    int32_t x;

    for (int i = 0; i < HILA5_N; i++) {
        r = 2 * bitrev10(i) + 1;          // bit reverse index
        x = 0;
        k = 0;
        for (int j = 0; j < HILA5_N; j++) {
            x = (x + v[j] * pow1945[k]) % HILA5_Q;
            k = (k + r) & 0x7FF;          // k = (j * r) % 2048 next round
        }
        d[i] = (c * x) % HILA5_Q;         // multiply with scalar c
    }
}
```

We can also give the inverse transform that, if unscaled, satisfies $\mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{v})) = n\mathbf{v}$. Output (or input) must therefore be scaled back by $n^{-1} \equiv 12277 \mod q$.

```
// Slow inverse number theoretic transform: d = NTT^-1(v).

void slow_intt(int32_t d[HILA5_N], const int32_t v[HILA5_N])
{
    int k, r;

    for (int i = 0; i < HILA5_N; i++)     // zeroise d[]
        d[i] = 0;
    for (int i = 0; i < HILA5_N; i++) {
        r = 2 * bitrev10(i) + 1;          // reverse index
        k = 0;
        for (int j = 0; j < HILA5_N; j++) {
            d[j] = (d[j] + v[i] * pow1945[k]) % HILA5_Q;
            k = (k - r) & 0x7FF;          // inverses are negative
        }
    }
}
```

In the transformed domain multiplication no longer requires a full convolution; a simple pointwise multiplication suffices. This is analogous to multiplication of polynomials vs. multiplication of points on the polynomial curves; $(\mathbf{f} * \mathbf{g})(x) = f(x)g(x)$.

```
// Pointwise multiplication: d = a (*) b.

void slow_vmul(int32_t d[HILA5_N],
    const int32_t a[HILA5_N], const int32_t b[HILA5_N])
{
    for (int i = 0; i < HILA5_N; i++)
        d[i] = (a[i] * b[i]) % HILA5_Q;
}
```

Complexity. The method given above (Equation 4 or `slow_ntt()`) clearly has $O(n^2)$ complexity, but it produces numerically equivalent results to our fast transforms.

In our optimized implementation we use the $O(n \log n)$ Cooley-Tukey [CT65] algorithm, with the reduction tricks for this use case suggested recently by Longa and Naehrig [LN16]. The various scaling constants that are powers of 3 are artifacts caused by the specific reduction methods suggested in that work.

Examples. Consider a vector $\mathbf{v} = (F_0, F_1 \cdots F_{n-1})$ of Fibonacci numbers reduced mod q:

$$\mathbf{v} = (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \cdots, 4524, 8293, 528, 8821, 9349).$$

Applying the Number Theoretic Transform (Equation 4) we obtain $\hat{\mathbf{v}} = \mathsf{NTT}(\mathbf{v})$:

$$\hat{\mathbf{v}} = (10951, 5645, 3732, 4089, 442, \cdots, 10237, 754, 6341, 4211, 7921).$$

Applying the inverse transform on this result we obtain $\mathsf{NTT}^{-1}(\hat{\mathbf{v}}) = n\mathbf{v}$ or

$$\mathsf{NTT}^{-1}(\hat{\mathbf{v}}) = (0, 1024, 1024, 2048, 3072, \cdots, 11912, 333, 12245, 289, 245).$$

For randomized testing, one may perform convolution multiplication (Equation 1 and function `slow_rmul`) equivalently via Number Theoretic Transforms as follows:

```
// a[] and b[] should have the vectors to be multipliied
slow_rmul(x, a, b);                 // compute x = a * b directly

// compute same using NTT transforms and helper array t[]
init_pow1945();                     // make sure it's initialized
slow_ntt(t, a, 1);                  // t = NTT(a)
slow_ntt(y, b, 12277);              // y = NTT(b) / 1024
slow_vmul(t, t, y);                 // pointwise t = t (*) y
slow_intt(y, t);                    // y = NTT^-1(t) = a * b = x !!

// .. now verify that indeed the products match: x == y
```

B.1.2 Encoding and Decoding of Ring Polynomials

Even though we use the `int32_t` signed integer type in internal processing, we note that each ring coefficient fits into $\lceil \log_2 q \rceil = 14$ bits. We can therefore readily store 4 coefficients into $4 * 14 = 56$ bits or 7 bytes. For interoperability we specify a method of encoding en entire vector of n coefficients into $14 * 1024/8 = 1792$ bytes for transmission or storage.

In our implementation we simply concatenate each 14-bit segment into a continuous byte sequence in little-endian fashion. We view the least significant bit as bit zero and the most significant bit of the most significant byte as the last bit; bit i always has numerical value 2^i. Encoding can be accomplished via "packing" and the inverse operation is called "unpacking". Function prototypes:

```
#define HILA5_PACKED14  (14 * HILA5_N / 8)

// 14-bit packing; mod q integer vector v[1024] to byte sequence d[1792]
void hila5_pack14(uint8_t d[HILA5_PACKED14], const int32_t v[HILA5_N]);

// 14-bit unpacking; bytes in d[1792] to integer vector v[1024]
void hila5_unpack14(int32_t v[HILA5_N], const uint8_t d[HILA5_PACKED14]);
```

Examples. The packed increasing sequence of n integers $(0, 1, 2, 3, \cdots, 1023)$ has the following hexadecimal encoding into 1792 = `0x700` bytes:

```
[0000] : 00 40 00 20 00 0C 00 04 40 01 60 00 1C 00 08 40
[0010] : 02 A0 00 2C 00 0C 40 03 E0 00 3C 00 10 40 04 20
[0020] : 01 4C 00 14 40 05 60 01 5C 00 18 40 06 A0 01 6C
[0030] : 00 1C 40 07 E0 01 7C 00 20 40 08 20 02 8C 00 24
                              ....
[06C0] : 0F DC 43 F7 E0 3D 7C 0F E0 43 F8 20 3E 8C 0F E4
[06D0] : 43 F9 60 3E 9C 0F E8 43 FA A0 3E AC 0F EC 43 FB
[06E0] : E0 3E BC 0F F0 43 FC 20 3F CC 0F F4 43 FD 60 3F
[06F0] : DC 0F F8 43 FE A0 3F EC 0F FC 43 FF E0 3F FC 0F
```

Encoding is easiest to do in blocks of four coefficients; for example $(10951, 5645, 3732, 4089)$ corresponds to exactly seven bytes `{ 0xC7, 0x6A, 0x83, 0x45, 0xE9, 0xE4, 0x3F }`.

B.1.3 Random Samplers

HILA5 requires two kinds of random numbers; uniformly distributed in the range $[0, q-1]$ and from the binomial distribution Ψ_{16}.

Uniform expander. Sampler Parse(seed) deterministically maps a 256-bit seed value to a uniformly distributed ring polynomial using the SHAKE-256 XOF [FIP15]. As noted in [GS16], it is more efficient to do a rejection sampling on $5q = 61445$ (rejection rate 6.25%).

```
#define HILA5_SEED_LEN  32

// generate n uniform samples from the seed

void hila5_parse(int32_t v[HILA5_N], const uint8_t seed[HILA5_SEED_LEN])
{
    hila5_sha3_ctx_t sha3;              // init SHA3 state for SHAKE-256
    uint8_t buf[2];                     // two byte output buffer
    int32_t x;                          // random variable

    hila5_shake256_init(&sha3);         // initialize the context
    hila5_shake_update(&sha3, seed, HILA5_SEED_LEN);    // seed input
    hila5_shake_xof(&sha3);             // pad context to output mode

    // fill the vector with uniform samples
    for (int i = 0; i < HILA5_N; i++) {
        do {                            // rejection sampler
            hila5_shake_out(&sha3, buf, 2); // two bytes from SHAKE-256
            x = ((int32_t) buf[0]) + (((int32_t) buf[1]) << 8); // endianess
        } while (x >= 5 * HILA5_Q);     // reject
        v[i] = x;                       // reduction (mod q) unnecessary
    }
}
```

Example. Let `seed[32] = { 0, 1, 2, ... 31 }`. The output of $\mathbf{v} = \mathsf{Parse}(\text{seed})$ is

$$\mathbf{v} = (\ 34940, 52800, 640, 45901, 14601, \cdots, 46031, 8999, 56069, 2120, 49166\ ),$$

which is congruent and equivalent to the vector

$$\mathbf{v} \bmod q = (\ 10362, 3644, 640, 9034, 2312, \cdots, 9164, 8999, 6913, 2120, 10\ ).$$

Binomial distribution. Sampling from the binomial distribution Ψ_{16} basically involves a bit count of 32 random bits and subtracting 16 to put the random variable in range $[-16, 16]$. This distribution and its properties are analyzed in more detail in Section B.4.7.

$$\Psi_{16} = \sum_{i=0}^{16} b_i - b'_i \quad \text{where} \quad b_i, b'_i \xleftarrow{\$} \{0, 1\}. \tag{5}$$

```
// sample a vector of values from the psi16 distribution

void hila5_psi16(int32_t v[HILA5_N])
{
    uint32_t x = 0;                     // 32-bit variable

    for (int i = 0; i < HILA5_N; i++) {

        randombytes((unsigned char *) &x, sizeof(x));   // get 4 random bytes

        x -= (x >> 1) & 0x55555555;     // Hamming weight
        x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
        x = (x + (x >> 4)) & 0x0F0F0F0F;
        x += x >> 8;
        x = (x + (x >> 16)) & 0x3F;

        x -= 16;                        // Make signed in range [0, q-1]
        v[i] = x + (-((x >> 31) & 1) & HILA5_Q);    // "constant time"
    }
}
```

B.1.4 Error Correction Code

The error correction code XE5 is a key component of HILA5. It operates on blocks of 496 bits, of which 256 bits are the actual shared secret and 240 bits of redundancy is used to correct errors. Together the 256 + 240 = 496 bits are considered "payload". XE5 is always able to correct five bit flips anywhere in the payload, and more with high probability. Please see Section B.4.9 for more information and a theoretical treatment of XE5.

This implementation operates on unsigned 64-bit integers and assumes a little-endian platform. On big-endian systems all input and output words need to be flipped around. For initial computation of linear code $\mathbf{r} = \mathsf{XE5\_Cod}(\mathbf{d})$ for sending, zeroize array `r[4]` first. When receiving, use with the transmitted value of $\mathbf{r}$; this is what XE5_Fix() expects.

```
//  Field       subcodeword:     r0  r1  r2  r3  r4  r5  r6  r7  r8  r9 (end)
//  lengths.    bit offset:      0   16  32  49  80  99  128 151 176 203 240
static const int xe5_len[10] = { 16, 16, 17, 31, 19, 29, 23, 25, 27, 37 };

// Compute redundancy r[] (XOR over original) from data d[]

void xe5_cod(uint64_t r[4], const uint64_t d[4])
{
    int i, j, l;
    uint64_t x, t, ri[10];

    for (i = 0; i < 10; i++)            // initialize
        ri[i] = 0;

    for (i = 3; i >= 0; i--) {          // four words
        x = d[i];                       // payload
        for (j = 1; j < 10; j++) {
            l = xe5_len[j];             // length
            t = (ri[j] << (64 % l));    // rotate
            t ^= x;                     // payload
            if (l < 32)                 // extra fold
                t ^= t >> (2 * l);
            t ^= t >> l;                // fold
            ri[j] = t & ((1llu << l) - 1);  // mask
        }
        x ^= x >> 8;                    // parity of 16
        x ^= x >> 4;
        x ^= x >> 2;
        x ^= x >> 1;
        x &= 0x0001000100010001;        // four parallel
        x ^= (x >> (16 - 1)) ^ (x >> (32 - 2)) ^ (x >> (48 - 3));
        ri[0] |= (x & 0xF) << (4 * i);
    }
    // pack coefficients into 240 bits (note output the XOR)
    r[0] ^= ri[0] ^ (ri[1] << 16) ^ (ri[2] << 32) ^ (ri[3] << 49);
    r[1] ^= (ri[3] >> 15) ^ (ri[4] << 16) ^ (ri[5] << 35);
    r[2] ^= ri[6] ^ (ri[7] << 23) ^ (ri[8] << 48);
    r[3] ^= (ri[8] >> 16) ^ (ri[9] << 11);
}
```

Example. We will view the 256-bit data array `d` as a sequence of 32 bytes first:

```
uint8_t d[32] = { 0x00, 0x01, 0x01, 0x02, 0x03, 0x05, 0x08, 0x0D,
                  0x15, 0x22, 0x37, 0x59, 0x90, 0xE9, 0x79, 0x62,
                  0xDB, 0x3D, 0x18, 0x55, 0x6D, 0xC2, 0x2F, 0xF1,
                  0x20, 0x11, 0x31, 0x42, 0x73, 0xB5, 0x28, 0xDD };
```

When the same data `d` is interpreted as a little-endian 64-bit words, we have:

```
uint64_t_d[4] = { 0x0D08050302010100, 0x6279E99059372215,
                  0xF12FC26D55183DDB, 0xDD28B57342311120 };
```

The corresponding 240-bit redundancy code `r` is:

```
uint64_t r[4] = { 0x5D193C3A9B0A3171, 0xE439D357352B06CF,
                  0xDF517AD4F8F2DE07,     0x492E2AC7B92B };
```

Note that high 16 bits of `r[3]` are always missing as this array is 240 bits (not 256).

Fixing errors. Upon receiving payload ($\mathbf{d}, \mathbf{r}$), first call $\mathbf{r}' = \mathsf{XE5\_Cod}(\mathbf{d})$ to perform the linear operation. Then one can obtain "corrected" data via $\mathbf{d}' = \mathbf{d} \oplus \mathsf{XE5\_Fix}(\mathbf{r} \oplus \mathbf{r}')$. Our implementation performs many of these XORs in place.

```
// Fix errors in data d[] using redundancy in r[]

void xe5_fix(uint64_t d[4], const uint64_t r[4])
{
    int i, j, k, l;
    uint64_t x, t, ri[10];

    ri[0] = r[0];                          // unpack
    ri[1] = r[0] >> 16;
    ri[2] = r[0] >> 32;
    ri[3] = (r[0] >> 49) ^ (r[1] << 15);
    ri[4] = r[1] >> 16;
    ri[5] = r[1] >> 35;
    ri[6] = r[2];
    ri[7] = r[2] >> 23;
    ri[8] = (r[2] >> 48) ^ (r[3] << 16);
    ri[9] = r[3] >> 11;

    for (i = 0; i < 4; i++) {              // four words
        for (j = 1; j < 10; j++) {
            l = xe5_len[j];                // length
            x = ri[j] & ((1llu << l) - 1);  // mask
            x |= x << l;                   // expand
            if (l < 32)                    // extra unfold
                x |= (x << (2 * l));
            ri[j] = x;                     // store it
        }
        x = (ri[0] >> (4 * i)) & 0xF;      // parity mask for ri[0]
        x ^= (x << (16 - 1)) ^ (x << (32 - 2)) ^ (x << (48 - 3));
        x  = 0x0100010001000100 - (x & 0x0001000100010001);
        x &= 0x00FF00FF00FF00FF;
        x |= x << 8;

        for (j = 0; j < 4; j++) {          // threshold sum
            t = (x >> j) & 0x1111111111111111;
            for (k = 1; k < 10; k++)
                t += (ri[k] >> j) & 0x1111111111111111;
            // threshold 6 -- add 2 to weight and take bit number 3
            t = ((t + 0x2222222222222222) >> 3) & 0x1111111111111111;
            d[i] ^= t << j;                // fix bits
        }
        if (i < 3) {                       // rotate if not last
            for (j = 1; j < 10; j++)
                ri[j] >>= 64 % xe5_len[j];
        }
    }
}
```

Example. Let's flip bits $\{13, 123, 234\}$ in $\mathtt{d}$ and bits $\{89, 200\}$ in $\mathtt{r}$ in previous message.

$\mathtt{d} \oplus \mathtt{d}' =$ 0000000000002000 0800000000000000 0000000000000000 0000040000000000
$\mathtt{r} \oplus \mathtt{r}' =$ 0000000000000000 0000000002000000 0000000000000000 0000000000000100

```
uint64_t_d[4] = { 0x0D08050302012100, 0x6A79E99059372215,
                  0xF12FC26D55183DDB, 0xDD28B1 7342311120 };

uint64_t r[4] = { 0x5D193C3A9B0A3171, 0xE439D3573720B06CF,
                  0xDF517AD4F8F2DE07,     0x492E2AC7B82B };
```

Recomputing linear code difference via `xe5_cod(r, d)` we obtain $\mathtt{r}'' = \mathtt{r} \oplus \mathsf{XE5\_Cod}(\mathtt{d})$:

$\mathtt{r}'' =$ 400000102C004081 0001042020408004 A000401100002110 0000000001000104

We call the threshold fix function `xe5_fix(d, r)` and directly get $\mathtt{d}'' = \mathtt{d}' \oplus \mathsf{XE5\_Fix}(\mathtt{r}'')$:

$\mathtt{d}'' =$ 0D08050302010100 6279E99059372215 F12FC26D55183DDB DD28B57342311120.

B.1.5 Key Generation

We will now describe keypair generation for both KEM and Public Key Encryption usage. The secret key is a random variable $\mathbf{a} \xleftarrow{\$} \psi_{16}^n$, stored in NTT domain as $\hat{\mathbf{a}} = \mathsf{NTT}(\mathbf{a})$. Public value pk consists of a concatenation of a 256-bit random seed for uniform generator $\hat{\mathbf{g}} = \mathsf{Parse}(\text{seed})$ and the actual public key

$$\hat{\mathbf{A}} = 3^3\big(\hat{\mathbf{g}} \circledast \hat{\mathbf{a}} + \mathsf{NTT}(\mathbf{e})\big) \quad \text{with error} \quad \mathbf{e} \xleftarrow{\$} \psi_{16}^n. \tag{6}$$

Vectors in NTT domain are scaled by $3^3 = 27$ in the specification in order to facilitate compatibility with lazy reduction techniques of the optimized implementation.

```
#define HILA5_PUBKEY_LEN (HILA5_SEED_LEN + HILA5_PACKED14)
#define HILA5_PRIVKEY_LEN (HILA5_PACKED14 + 32)

// Generate a keypair

int crypto_kem_keypair( uint8_t *pk,    // HILA5_PUBKEY_LEN = 1824
                        uint8_t *sk)    // HILA5_PRIVKEY_LEN = 1824
{
    int32_t a[HILA5_N], e[HILA5_N], t[HILA5_N];

    init_pow1945();                     // make sure initialized

    // Create Secret Key
    hila5_psi16(t);                     // (t is a temporary variable)
    slow_ntt(a, t, 27);                 // a = 3**3 * NTT(Psi_16)

    // Public Key
    hila5_psi16(t);                     // t = Psi_16
    slow_ntt(e, t, 27);                 // e = 3**3 * NTT(Psi_16) -- noise
    randombytes(pk, HILA5_SEED_LEN);    // Random seed for g
    hila5_parse(t, pk);                 // (t =) g = parse(seed)
    slow_vmul(t, a, t);
    slow_vadd(t, t, e);                 // A = NTT(g * a + e)
    hila5_pack14(pk + HILA5_SEED_LEN, t);   // pk = seed | A

    hila5_pack14(sk, a);                // pack secret key
    // SHA3 hash of pubic key is stored with secret key due to API limitation
    hila5_sha3(pk, HILA5_PUBKEY_LEN, sk + HILA5_PACKED14, 32);

    return 0;                           // SUCCESS
}
```

Note that we must encode a SHA-3 hash of the public key with the secret key because the API does not make the public key available for decryption routines.

Example. instead of sampling from Ψ_{16}^n, we arbitrarily fix the (untransformed) secret key be a cycle-five sequence $\mathbf{a} \equiv (-1, +1, -2, -3, +5, -1, +1, -2, -3, +5, \cdots)$. We have

$$3^3\hat{\mathbf{a}} = (\ 11172, 5208, 9207, 8751, 251, \cdots, 7603, 3490, 9191, 8666, 8302\ ).$$

Furthermore we set error $\mathbf{e} \equiv (+2, +2, -4, +2, +2, -4, \cdots)$, a cycle of three. The seed consists of 32 zero bytes. The transformed quantities and the public key will then be

$$\begin{aligned} 3^3\hat{\mathbf{e}} &= (\ 8226, 10812, 6666, 1749, 2228, \cdots, 10169, 10648, 5731, 1585, 4171\ ) \\ \hat{\mathbf{g}} &\equiv (\ 2034, 8826, 9346, 872, 2929, \cdots, 2816, 441, 7160, 2952, 5275\ ) \\ \hat{\mathbf{A}} &= (\ 9713, 3471, 7710, 1152, 67, \cdots, 490, 1324, 5696, 10208, 11514\ ). \end{aligned}$$

The encoded byte vectors $\mathsf{pk} = (\ \text{seed} \mid \hat{\mathbf{A}}\ )$ and $\mathsf{sk} = \big(\ 3^3\hat{\mathbf{a}} \mid \mathsf{SHA3}(\mathsf{pk})\ \big)$ are

```
uint8_t pk[1824] = { 0x00, 0x00, ... 0x90, 0x05, 0x7E, 0xEA, 0xB3 };
uint8_t sk[1824] = { 0xA4, 0x2B, 0x16, 0x75, 0x3F, ... 0xE3, 0x3F };
```

B.1.6 Key Encapsulation

Following the NIST call [NIS16] and Peikert [Pei14], our scheme is formalized as a Key Encapsulation Mechanism (KEM), consisting of three algorithms:

$(\mathsf{PK}, \mathsf{SK})$	$\leftarrow$	$\mathsf{KeyGen}()$.	Generate a public key PK and a secret key SK.
$(\mathsf{CT}, \mathsf{K})$	$\leftarrow$	$\mathsf{Encaps}(\mathsf{PK})$.	Encapsulate a (random) key K in ciphertext CT.
K	$\leftarrow$	$\mathsf{Decaps}(\mathsf{SK}, \mathsf{CT})$.	Decapsulate shared key K from CT with SK.

In this model, reconciliation data is a part of ciphertext produced by Encaps. The three KEM algorithms constitute a natural single-roundtrip key exchange:

Alice		Bob
$(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{KeyGen}()$	$\xrightarrow{\mathsf{PK}}$	
	$\xleftarrow{\mathsf{CT}}$	$(\mathsf{CT}, \mathsf{K}) \leftarrow \mathsf{Encaps}(\mathsf{PK})$
$\mathsf{K} \leftarrow \mathsf{Decaps}(\mathsf{SK}, \mathsf{CT})$		

Even though a KEM cannot encrypt per se, a hybrid set-up that uses a KEM to determine random shared keys for message payload confidentiality (symmetric encryption) and integrity (via a message authentication code) is usually preferable to using asymmetric encryption directly on payload [CS03].

Reconciliation Data. HILA5 uses a novel reconciliation method based on "Safe Bits". Please see Section B.4.6 for a detailed description of this method and analysis of its parameters. Note that selector `sel`, reconciliation `rec`, and payload `pld` are all outputs.

```
#define HILA5_B             799
#define HILA5_PACKED1       (HILA5_N / 8)
#define HILA5_KEY_LEN       32
#define HILA5_ECC_LEN       30
#define HILA5_PAYLOAD_LEN   (HILA5_KEY_LEN + HILA5_ECC_LEN)

// Create a bit selector, reconciliation bits, and payload;
// return nonzero on failure.

int hila5_safebits(uint8_t sel[HILA5_PACKED1],
    uint8_t rec[HILA5_PAYLOAD_LEN],
    uint8_t pld[HILA5_PAYLOAD_LEN],
    const int32_t v[HILA5_N])
{
    int i, j, x;

    memset(sel, 0, HILA5_PACKED1);      // selector array
    memset(rec, 0, HILA5_PAYLOAD_LEN);  // reconciliation bits for payload
    memset(pld, 0, HILA5_PAYLOAD_LEN);  // the actual payload XOR mask

    j = 0;                              // reset the bit counter
    for (i = 0; i < HILA5_N; i++) {     // scan for "safe bits"
        // x in { [737, 2335] U [3809, 5407] U [6881, 8479] U [9953, 11551] }
        x = v[i] % (HILA5_Q / 4);
        if (x >= ((HILA5_Q / 8) - HILA5_B) &&
            x <= ((HILA5_Q / 8) + HILA5_B)) {
                                        // set selector bit
            sel[i >> 3] |= 1 << (i & 7);
            x = (4 * v[i]) / HILA5_Q;   // reconciliation bits
            rec[j >> 3] ^= (x & 1) << (j & 7);
            x >>= 1;                    // payload bits
            pld[j >> 3] ^= (x & 1) << (j & 7);
            j++;                        // payload bit count
            if (j >= 8 * HILA5_PAYLOAD_LEN)
                return 0;               // success: enough bits
        }
    }
    return j;                           // FAIL: not enough bits
}
```

Creating Ciphertext. The ciphertext in HILA5 consists of four different pieces of data concatenated together. The (scaled) representation of $\hat{\mathbf{B}}$ makes up first 1792 bytes:

$$\hat{\mathbf{B}} = \hat{\mathbf{g}} \circledast \hat{\mathbf{b}} + \mathsf{NTT}(\mathbf{e}'). \tag{7}$$

It is then followed by public selector `sel` (128 bytes), reconciliation data `rec` for payload ($32 + 30 = 62$ bytes), and encrypted error correction part (30 bytes). The encryption is a "one-time-pad" XOR with last 30 bytes of the payload. The first 32 bytes of the payload $\mathbf{z}$ is used to establish the shared secret. (See Algorithm 1).

```
#define HILA5_MAX_ITER  100             // Fail hard bound

// Encapsulate

int crypto_kem_enc( uint8_t *ct,        // HILA5_CIPHERTEXT_LEN = 2012
                    uint8_t *ss,        // HILA5_KEY_LEN = 32
                    const uint8_t *pk)  // HILA5_PUBKEY_LEN = 1824
{
    int i;
    int32_t a[HILA5_N], b[HILA5_N], e[HILA5_N], g[HILA5_N], t[HILA5_N];
    uint64_t z[8];
    uint8_t hash[32];
    hila5_sha3_ctx_t sha3;

    init_pow1945();                     // make sure initialized

    hila5_unpack14(a, pk + HILA5_SEED_LEN); // decode A = public key

    for (i = 0; i < HILA5_MAX_ITER; i++) {

        hila5_psi16(t);                 // recipients' ephemeral secret
        slow_ntt(b, t, 27);             // b = 3**3 NTT(Psi_16)
        slow_vmul(e, a, b);
        slow_intt(t, e);                // t = a * b  (approx. share "y")
        slow_smul(t, 1416);             // scale by 1416 = 1 / (3**6 * 1024)

        // Safe bits -- may fail (with about 1% probability);
        memset(z, 0, sizeof(z));        // ct = .. | sel | sec, z = payload
        if (hila5_safebits(ct + HILA5_PACKED14, //
            ct + HILA5_PACKED14 + HILA5_PACKED1, (uint8_t *) z, t) == 0)
            break;
    }
    if (i == HILA5_MAX_ITER)            // too many repeats -- fail hard
        return -1;

    xe5_cod(&z[4], z);                  // create linear ot
    memcpy(ct + HILA5_PACKED14 + HILA5_PACKED1 + HILA5_PAYLOAD_LEN,
        &z[4], HILA5_ECC_LEN);          // ct = .. | encrypted error cor. code

    // Construct ciphertext
    hila5_parse(g, pk);                 // g = Parse(seed)
    hila5_psi16(t);                     // noise error
    slow_ntt(e, t, 27);                 // e = 3**3 * NTT(Psi_16)
    slow_vmul(t, g, b);                 // t = NTT(g * b)
    slow_vadd(t, t, e);                 // t = NTT(g * b + e)
    hila5_pack14(ct, t);                // public value in ct

    hila5_sha3_init(&sha3, HILA5_KEY_LEN);          // final hash
    hila5_sha3_update(&sha3, "HILA5v10", 8);        // version ident
    hila5_sha3(pk, HILA5_PUBKEY_LEN, hash, 32);     // SHA3(pk)
    hila5_sha3_update(&sha3, hash, 32);
    hila5_sha3(ct, HILA5_CIPHERTEXT_LEN, hash, 32); // SHA3(ct)
    hila5_sha3_update(&sha3, hash, 32);
    hila5_sha3_update(&sha3, z, HILA5_KEY_LEN);     // actual shared secret z
    hila5_sha3_final(ss, &sha3);                    // hash out to ss

    return 0;                           // SUCCESS
}
```

Final Hashes. We see that the final shared secret `ss` is computed as

$$\mathtt{ss} = \mathsf{SHA3}(\text{ "HILA5v10" } \mid \mathsf{SHA3}(\mathtt{pk}) \mid \mathsf{SHA3}(\mathtt{ck}) \mid \mathbf{z}\ ). \tag{8}$$

All hashes are SHA3-256 [FIP15]. First 8 bytes is an ASCII version and format identifier.

Example. Let's use the public key from the key generation Example in Section B.1.5. We set the ephemeral secret to a cycle-7 sequence and compute its transform:

$$\mathbf{b} \equiv (\ 0, +1, +1, +2, -3, +4, -5, 0, +1, +1, +2, -3, +4, -5, \cdots\ )$$
$$3^3\hat{\mathbf{b}} = (\ 5361, 11011, 5111, 10968, 6240, \cdots, 1901, 10941, 7723, 10979, 9431\ )$$

Since the seed is a part of the public key, we end up at the same $\hat{\mathbf{g}}$ value. The scaled "approximate shared secret" $t = \mathbf{Ab}$, also known as $\mathbf{y}$ (Section B.4.4), has value

$$\mathbf{y} = (\ 11982, 1189, 1239, 8956, 11579, \cdots, 8947, 10863, 2725, 6368, 1295\ ).$$

Applying SafeBits, we obtain 1024-bit selector vector `sel`, which is placed in ciphertext after encoded $\hat{\mathbf{B}}$ (below), reconciliation bits for payload `rec`, which is the next part of ciphertext, and the actual payload `pld` which is which is cast as 64-bit words in $\mathbf{z}$. First 32 bytes (`z[0..3]`) of payload is used to create the actual shared secret, while the latter 30 bytes is used as a "one time pad" to encrypt XE5 error correcting code of that secret.

```
uint8_t sel[128] = { 0x26, 0x03, 0xF3, 0x56, 0x26, ... 0x00, 0x00 };
uint8_t rec[62]  = { 0xF8, 0x82, 0x56, 0x49, 0x9E, ... 0xB0, 0x33 };
uint8_t pld[62]  = { 0x70, 0xF1, 0x5B, 0xDD, 0x24, ... 0x1A, 0x5F };
```

When constructing ciphertext, we set error to cycle $\mathbf{e}' = (\ 0, +4, 0, -4, 0, +4, 0, -4, \cdots\ )$. After transformation and some arithmetic we obtain public value $\hat{\mathbf{B}} = 3^3(\hat{\mathbf{b}} \circledast \hat{\mathbf{g}} + \mathsf{NTT}(e'))$

$$t = \hat{\mathbf{B}} = (\ 9437, 8457, 4675, 10931, 3829, \cdots, 8113, 3081, 792, 10698, 8159\ ).$$

The ciphertext, and the shared secret (after all of the final hashing is computed) are:

```
uint8_t ct[2012] = { 0xDD, 0x64, 0x42, 0x38, 0x24, ... 0xED, 0x58 };
uint8_t ss[32]   = { 0xC2, 0x95, 0xA5, 0x2D, 0xBF, ... 0x72, 0x60 };
```

B.1.7 Key Decapsulation

Selection and Reconciliation. The inverse operation of SafeBits at the recipient side Select. It aims to arrive at the same secret payload data `pld`, given selector vector `sel`, reconciliation bits `rec`, and a vector $\mathbf{v} = \mathbf{x} \approx \mathbf{y}$ that is close the one given to SafeBits.

```
// decode selected key bits. return nonzero on failure

int hila5_select(uint8_t pld[HILA5_PAYLOAD_LEN],
    const uint8_t sel[HILA5_PACKED1],
    const uint8_t rec[HILA5_PAYLOAD_LEN],
    const int32_t v[HILA5_N])
{
    int i, j, x;

    memset(pld, 0x00, HILA5_PAYLOAD_LEN);

    j = 0;
    for (i = 0; i < HILA5_N; i++) {
        if ((sel[i >> 3] >> (i & 7)) & 1) {
            x = v[i] + HILA5_Q / 8;      // reconciliation
            x -= -((rec[j >> 3] >> (j & 7)) & 1) &
                    (HILA5_Q / 4);       // "90 degrees" if rec bit set
            x = ((2 * ((x + HILA5_Q) % HILA5_Q)) / HILA5_Q);
            pld[j >> 3] ^= (x & 1) << (j & 7);
            j++;
            if (j >= 8 * HILA5_PAYLOAD_LEN)
                return 0;                // got full payload
        }
    }

    return j;                            // FAIL: not enough bits
}
```

Decapsulating ciphertext. The function Decaps() takes "encapsulated" ciphertext `ct`, secret key `sk`, and arrives at the same shared secret `ss` as the encapsulation code.

```
// Decapsulate

int crypto_kem_dec( uint8_t *ss,        // HILA5_KEY_LEN = 32
                    const uint8_t *ct,  // HILA5_CIPHERTEXT_LEN = 2012
                    const uint8_t *sk)  // HILA5_PRIVKEY_LEN = 1824
{
    int32_t a[HILA5_N], b[HILA5_N];
    uint64_t z[8];
    uint8_t ct_hash[32];
    hila5_sha3_ctx_t sha3;

    init_pow1945();                     // make sure initialized

    hila5_unpack14(a, sk);              // unpack secret key
    hila5_unpack14(b, ct);              // get B from ciphertext
    slow_vmul(a, a, b);                 // a * B
    slow_intt(b, a);                    // shared secret ("x") in b
    slow_smul(b, 1416);                 // scale by 1416 = (3^6 * 1024)^-1

    memset(z, 0x00, sizeof(z));
    if (hila5_select((uint8_t *) z,     // reconciliation
        ct + HILA5_PACKED14, ct + HILA5_PACKED14 + HILA5_PACKED1, b))
        return -1;                      // FAIL: not enough bits

    // error correction -- decrypt with "one time pad" in payload
    for (int i = 0; i < HILA5_ECC_LEN; i++) {
        ((uint8_t *) &z[4])[i] ^=
            ct[HILA5_PACKED14 + HILA5_PACKED1 + HILA5_PAYLOAD_LEN + i];
    }
    xe5_cod(&z[4], z);                  // linear code
    xe5_fix(z, &z[4]);                  // fix possible errors

    hila5_sha3_init(&sha3, HILA5_KEY_LEN);              // final hash
    hila5_sha3_update(&sha3, "HILA5v10", 8);            // version identifier
    hila5_sha3_update(&sha3, sk + HILA5_PACKED14, 32);  // SHA3(pk)
    hila5_sha3(ct, HILA5_CIPHERTEXT_LEN, ct_hash, 32);  // hash the ciphertext
    hila5_sha3_update(&sha3, ct_hash, 32);              // SHA3(ct)
    hila5_sha3_update(&sha3, z, HILA5_KEY_LEN);         // shared secret
    hila5_sha3_final(ss, &sha3);

    return 0;                           // SUCCESS
}
```

Example. Given the ciphertext and secret key from previous examples,

```
uint8_t ct[2012] = { 0xDD, 0x64, 0x42, 0x38, 0x24, ... 0xED, 0x58 };
uint8_t sk[1824] = { 0xA4, 0x2B, 0x16, 0x75, 0x3F, ... 0xE3, 0x3F };
```

we arrive at the approximate shared secret $\mathbf{x} = \mathsf{NTT}^{-1}(\hat{\mathbf{B}} \circledast \hat{\mathbf{a}})$, which is set in variable `b`:

$$\mathbf{x} = (\ 11982, 1157, 1261, 8932, 11561, \cdots, 8967, 10861, 2727, 6374, 1259\ ).$$

The closeness if $\mathbf{x}$ to $\mathbf{y}$ (Section B.1.6) is demonstrated by

$$\mathbf{y} - \mathbf{x} = (\ 0, 32, -22, 24, 18, -56, -10, 40, \cdots, 42, 28, -16, -20, 2, -2, -6, 36\ ).$$

One should obviously also test that the shared secret `ss` fully matches.

```
uint8_t ss[32] = { 0xC2, 0x95, 0xA5, 0x2D, 0xBF, 0x0B, 0x86, 0x03,
                   0xAC, 0x49, 0xB4, 0x1A, 0x5B, 0xE1, 0xEE, 0xBD,
                   0x64, 0x0E, 0x34, 0x7D, 0x16, 0xC1, 0x58, 0xE1,
                   0xBD, 0xA0, 0x75, 0x96, 0x14, 0xB1, 0x72, 0x60 };
```

B.2 Performance Analysis

We chose to recycle "New Hope" [ADPS16b] ring (n, q) and sampler (q, Ψ_{16}) parameters as they have been extensively vetted for security against lattice attacks and originally selected for performance. The reconciliation and error correction methods are novel, but have a negligible effect on performance. Much of software and hardware implementation footprint is taken specifically by ring arithmetic and sampler components. Hance New Hope software and hardware performance numbers are largely applicable.

Software Optimizations. A significant effort has been dedicated (by several research groups) for the optimization these particular NTT and Sampler components. There already exists a number of permissively licensed open source implementations and a body of publications detailing specific optimizations for these NTT and sampler parameters.

There are at least two very fast AVX2 Intel optimized versions of the NTT core and Ψ_{16} sampler – the original [ADPS16b] and one by Longa and Naehrig [LN16]. Further sampler optimizations have been suggested in [GS16]. Implementations have also been reported for ARM Cortex-M MCUs [AJS16] and the ARM NEON instruction set [SS17].

New Hope has also been integrated in TLS stacks and cryptographic toolkits in 2016-17 by Google (BoringSSL), the Open Quantum Safe project, Microsoft (MS Lattice Library), ISARA Corporation, and possibly others. Many of these components and protocol integration techniques are recyclable for a HILA5 implementation.

Software Comparison. Our prototype implementation was integrated into a branch of the Open Quantum Safe (OQS) framework[1] where it was benchmarked against other quantum-resistant KEM schemes [SM16]. Table 1 summarizes the results. The slight (under 4%) performance between HILA5 and New Hope is principally due to our use of error correction and SHAKE-256 (rather than SHAKE-128). Note that HILA5 message size is slightly smaller and failure rate is significantly better.

Testing was performed on an Ubuntu 17.04 workstation with Core i7-6700 @ 3.40 GHz. We are also including RSA numbers with OpenSSL 1.0.2 (system default) on this target for reference and scale. A single Elliptic Curve DH operation requires 45.4μs for the NIST P-256 curve (highly optimized implementation), and 331.7μs for NIST P-521.

Hardware Implementations. Envieta [FNSW17] reports FPGA implementations on New Hope on Intel Arria 10 (266,240 bits of memory, 22 DSP, 6485 Registers, 300 MHz, 40,030 CLKs) and Xilinx Zynq (5 BRAM, 27 DSP, 6988 Registers, 180 Mhz, 40,030 CLKs). Kuo et al. [KLC+17] also report a New Hope implementation on Xilinx Zynq (13 BRAM, 32 DSP, 12,707 FFs, 19,781 LUTs, 13,024 slice registers, 114 MHz, 22,597 CLKs).

In all cases the key exchange required only a fraction of millisecond of computation for full key exchange; this is faster than any comparable classical alternative. NTT operations dominate the hardware implementation area and time.

B.3 Known Answer Test Values

Various intermediate values can be found in examples of Section B.1. Full 100-iteration KAT set is included in the submission:

```
KAT/PQCkemKAT_1824.req, 13590 bytes
SHA-256 = 36c27b6089b8910733a01fea1136469769b3ca3c35f2b375cfcc592f2112cfaa

KAT/PQCkemKAT_1824.rsp, 1152399 bytes
SHA-256 = 7d4336c35a0a5d3ed9be28aa2d812be03f6765572e788c7477a2a0839bb34e42
```

[1] Open Quantum Safe project home: https://openquantumsafe.org/

Table 1: Comparison of HILA5 to other Open Quantum Safe implementations [SM16].

Scheme	Init KeyGen()	Public Encaps()	Private Decaps()	KEX Total	Data Tot. xfer
New Hope [ADPS16b]	60.7μs	92.3μs	16.2μs	169.2μs	3,872 B
HILA5 [This work]	68.7μs	89.9μs	16.9μs	175.4μs	3,836 B
BCNS15 [BCNS15]	951.6μs	1546μs	196.9μs	2.694ms	8,320 B
LWE Frodo [BCD+16]	2.839ms	3.144ms	84.9μs	6.068ms	22,568 B
SIDH CLN16 [CLN16]	10.3ms	22.9ms	9.853ms	43.1ms	1,152 B
RSA-2048 [OpenSSL]	60ms	15.9μs	559.9μs	N/A	N/A
RSA-4096 [OpenSSL]	400ms	55.7μs	3.687ms	N/A	N/A

B.4 Design and Parameter Selection

This section contains reasoning for our design and parameter selection.

B.4.1 Expected Security Strength

Our design goal and security claim is that HILA5 meets NIST's "Category 5" post-quantum security requirement ([NIS16], Section 4.A.5): Compromising key K (shared secret `ss`) in a passive attack requires computational resources comparable to or greater than those required for key search on a block cipher with a 256-bit key (e.g. AES 256).

NIST requires at least IND-CPA [BDPR98] security from a KEM scheme (Section B.1.6). For a KEM without "plaintext", this essentially means that valid $(\mathsf{PK}, \mathsf{CT}, \mathsf{K})$ triplets are computationally indistinguishable from $(\mathsf{PK}, \mathsf{CT}, \mathsf{K}')$, where K' is random. We claim this to be true. The final hashing (that also includes full public key and ciphertext values) should also enable IND-CCA2 security against active attacks if an AEAD (Authenticated Encryption with Associated Data) [Rog02] is used to implement public key encryption.

B.4.2 Design Overview of HILA5

Our proposal – codenamed HILA5[2] – shares core Ring-LWE parameters with various "New Hope" variants, but uses an entirely different error management strategy. Algorithm 1 contains a pseudocode overview of the HILA5 Key Encapsulation Mechanism, using a number of auxiliary primitives and functions.

Notation and auxiliary functions. We represent elements of $\mathcal{R}$ in two different domains; the normal polynomial representation $\mathbf{v}$ and Number Theoretic Transform representation $\hat{\mathbf{v}}$. Convolution (polynomial multiplication) in the NTT domain is a linear-complexity operation, written $\hat{\mathbf{x}} \circledast \hat{\mathbf{y}}$. Addition and subtraction work as in normal representation. The transform and its inverse are denoted here by $\mathsf{NTT}(\mathbf{v}) = \hat{\mathbf{v}}$ and $\mathsf{NTT}^{-1}(\hat{\mathbf{v}}) = \mathbf{v}$, respectively. See Section B.1.1 for more information about these transforms.

The hash $h(x)$ is SHA3-256 [FIP15]. Function $\mathsf{Parse}()$ (Section B.1.3) deterministically samples a uniform $\hat{\mathbf{g}} \in \mathcal{R}$ based on arbitrary seed s using SHA3's XOF mode SHAKE-256 [FIP15]. While New Hope uses the slightly faster SHAKE-128 for this purpose, we consistently use SHAKE-256 or SHA3-256 in all parts of HILA5. For sampling modulo q we use the $5q$ trick suggested by Gueron and Schlieker in [GS16]. Binomial distribution values Ψ_{16} can be computed directly from 32 random bits (Section B.1.3, Definition 2).

[2] *Hila* is Finnish for a lattice. HILA5 – especially when written as "Hila V" – also refers to *hilavitkutin*, a nonsensical placeholder name usually meaning an unidentified, incomprehensibly complicated apparatus.

Algorithm 1 Semi-abstract protocol flow of the HILA5 KEM.

Alice		**Bob**
$(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{KeyGen}()$		
$\mathbf{s} \xleftarrow{\$} \{0,1\}^{256}$	*Public random seed.*	
$\hat{\mathbf{g}} \leftarrow \mathsf{Parse}(s)$	*Expand to "generator" in NTT domain.*	
$\mathbf{a} \xleftarrow{\$} \psi_{16}^n$	*Randomize Alice's secret key.*	
$\hat{\mathbf{a}} \leftarrow \mathsf{NTT}(\mathbf{a})$	*Transform it.*	
$\mathbf{e} \xleftarrow{\$} \psi_{16}^n$	*Generate masking noise.*	
$\hat{\mathbf{A}} \leftarrow \hat{\mathbf{g}} \circledast \hat{\mathbf{a}} + \mathsf{NTT}(\mathbf{e})$	*Compute Alice's public key in NTT domain.*	
Send $\mathsf{PK} = \mathbf{s} \mid \hat{\mathbf{A}}$	$\xrightarrow{\mathsf{PK}}$	
Keep $\mathsf{SK} = \hat{\mathbf{a}}$ and $h(\mathsf{PK})$.		$(\mathsf{CT}, \mathsf{K}) \leftarrow \mathsf{Encaps}(\mathsf{PK})$
	Randomize Bob's ephemeral secret key.	$\mathbf{b} \xleftarrow{\$} \psi_{16}^n$
	Transform it.	$\hat{\mathbf{b}} \leftarrow \mathsf{NTT}(\mathbf{b})$
	Bob's version of shared secret.	$\mathbf{y} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}} \circledast \hat{\mathbf{b}})$
	Get payload and reconciliation values.	$(\mathbf{d}, \mathbf{k}, \mathbf{c}) \leftarrow \mathsf{SafeBits}(\mathbf{y})$
	(Fail hard after more than a dozen restarts.)	If $\mathbf{k} = \mathsf{FAIL}$ restart $\mathsf{Encaps}()$
	Split to payload and redundancy "keystream".	$\mathbf{p} \mid \mathbf{z} = \mathbf{k}$
	Error correction code, encrypt it.	$\mathbf{r} \leftarrow \mathsf{XE5\_Cod}(\mathbf{p}) \oplus \mathbf{z}$
	Get "generator" from Alice's seed.	$\hat{\mathbf{g}} \leftarrow \mathsf{Parse}(s)$
	Generate masking noise.	$\mathbf{e}' \xleftarrow{\$} \psi_{16}^n$
	Compute Bob's one-time public value.	$\hat{\mathbf{B}} \leftarrow \hat{\mathbf{g}} \circledast \hat{\mathbf{b}} + \mathsf{NTT}(\mathbf{e}')$
	$\xleftarrow{\mathsf{CT}}$	Send $\mathsf{CT} = \hat{\mathbf{B}} \mid \mathbf{d} \mid \mathbf{c} \mid \mathbf{r}$
Hash the shared secret. V *is a version identifier.*		$\mathsf{K} = h(\mathsf{V} \mid h(\mathsf{PK}) \mid h(\mathsf{CT}) \mid \mathbf{p})$
$\mathsf{K} \leftarrow \mathsf{Decaps}(\mathsf{SK}, \mathsf{CT})$		
$\mathbf{x} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{B}} \circledast \hat{\mathbf{a}})$	*Alice's version of the shared secret.*	
$\mathbf{k}' \leftarrow \mathsf{Select}(\mathbf{x}, \mathbf{d}, \mathbf{c})$	*Get payload with the help of reconciliation.*	
$\mathbf{p}' \mid \mathbf{z}' = \mathbf{k}'$	*Split to payload and redundancy "keystream".*	
$\mathbf{r}' \leftarrow \mathsf{XE5\_Cod}(\mathbf{p}')$	*Get error correction code from Alice's version.*	
$\mathbf{p}'' \leftarrow \mathsf{XE5\_Fix}(\mathbf{r} \oplus \mathbf{z}' \oplus \mathbf{r}') \oplus \mathbf{p}'$	*Decrypt and apply Bob's error correction.*	
$\mathsf{K}' = h(\mathsf{V} \mid h(\mathsf{PK}) \mid h(\mathsf{CT}) \mid \mathbf{p}'')$	*Upon success shared secret* $\mathsf{K} = \mathsf{K}'$.	

Bob's reconciliation function SafeBits() (Section B.1.6) captures Equations 14 and 16 from Section B.4.6. Conversely, Alice's reconciliation function Select() (Section B.1.7) captures Equation 17.

The XE5 error correction functions $\mathbf{r} = \mathsf{XE5\_Cod}(\mathbf{p})$ and $\mathbf{p}' = \mathsf{XE5\_Fix}(\mathbf{r} \oplus \mathbf{r}') \oplus \mathbf{p}$ are defined in Sections B.1.4 and B.4.9. Here we have "error key" $\mathbf{k} = \mathbf{p} \mid \mathbf{r}$ with the payload key $\mathbf{p} \in \{0,1\}^{256}$ and redundancy $\mathbf{r} \in \{0,1\}^{240}$.

Encoding – shorter messages. Ring elements, whether or not in NTT domain, are encoded into $|\mathcal{R}| = \lceil \log_2 q \rceil n$ bits $= 1,792$ bytes. This is the private key size. Alice's public key PK with a 256-bit seed s and $\hat{\mathbf{A}}$ is $1,824$ bytes. Ciphertext CT is $|\mathcal{R}| + n + m + |\mathbf{r}|$ bits or $2,012$ bytes; 36 bytes less than New Hope [ADPS16b], 196 bytes less than the variant of [ADPS16a], and $1,572$ bytes less than LP11 [LP11].

Encryption: From noisy Diffie-Hellman to noisy ElGamal Modification of the scheme for public-key encryption is straightforward. Compared to the more usual "LP11" Ring-LWE Public Key Encryption construction [LP11] our reconciliation approach saves about 44 % in ciphertext size.

B.4.3 Hard Problem: Introduction to Ring-LWE

Notation. Let $\mathcal{R}$ be a ring with elements $\mathbf{v} \in \mathbb{Z}_q^n$. We use cyclotomic polynomial basis $\mathbb{Z}_q[x]/(x^n + 1)$. See Section B.1.1 for further information about arithmetic in this ring.

Definition 1 (Informal)**.** With all distributions and computations in ring $\mathcal{R}$, let $\mathbf{s}, \mathbf{e}$ be elements randomly chosen from some non-uniform distribution χ, and $\mathbf{g}$ be a uniformly random public value. Determining $\mathbf{s}$ from $(\mathbf{g}, \mathbf{g} * \mathbf{s} + \mathbf{e})$ in ring $\mathcal{R}$ is the (Normal Form Search) Ring Learning With Errors ($\mathrm{RLWE}_{\mathcal{R},\chi}$) problem.

Typically χ is chosen so that each coefficient is a Discrete Gaussian or from some other "Bell-Shaped" distribution that is relatively tightly concentrated around zero. The hardness of the problem is a function of n, q, and χ.

References and notes on RLWE problem. The Learning With Errors (LWE) problem in cryptography originates with Regev [Reg05] who showed its connection to fundamental lattice problems in a quantum setting. Regev also showed equivalence of search and decision variants [Reg09].

These ideas were extended to ring setting (RLWE) starting with [LPR10]. The connection between a uniform secret $\mathbf{s}$ and a secret chosen from χ is provided by Applebaum et al. [ACPS09] for LWE case, and for the ring setting in [LPR13].

Due to these reductions, the informal problem of Definition 1 can be understood to describe "RLWE". Best known methods for solving the problem expand an RLWE instance to the general (lattice) LWE, and therefore RLWE falls under "lattice cryptography" umbrella. For a recent review of its concrete hardness, see [APS15].

B.4.4 Noisy Diffie-Hellman in a Ring

A key exchange method analogous to Diffie-Hellman can be constructed in $\mathcal{R}$ in a straightforward manner, as first described in [AGL+10, Pei09]. Let $\mathbf{g} \xleftarrow{\$} \mathcal{R}$ be a uniformly random common parameter ("generator"), and χ a non-uniform distribution.

Alice		**Bob**
$\mathbf{a} \xleftarrow{\$} \chi$	*private keys*	$\mathbf{b} \xleftarrow{\$} \chi$
$\mathbf{e} \xleftarrow{\$} \chi$	*noise*	$\mathbf{e}' \xleftarrow{\$} \chi$
$\mathbf{A} = \mathbf{g} * \mathbf{a} + \mathbf{e}$	*public keys*	$\mathbf{B} = \mathbf{g} * \mathbf{b} + \mathbf{e}'$
	$\mathbf{A} \xrightarrow{}$	
	$\xleftarrow{\mathbf{B}}$	
$\mathbf{x} = \mathbf{B} * \mathbf{a}$	*shared secret*	$\mathbf{y} = \mathbf{A} * \mathbf{b}$

We see that that the way messages $\mathbf{A}, \mathbf{B}$ are generated makes the security of the scheme equivalent to Definition 1. This commutative scheme "almost" works like Diffie-Hellman because the shared secrets only approximately agree; $\mathbf{x} \approx \mathbf{y}$. Since the ring $\mathcal{R}$ is commutative, substituting $\mathbf{A}$ and $\mathbf{B}$ gives

$$\mathbf{x} = (\mathbf{g} * \mathbf{b} + \mathbf{e}') * \mathbf{a} = \mathbf{g} * \mathbf{a} * \mathbf{b} + \mathbf{e}' * \mathbf{a} \tag{9}$$

$$\mathbf{y} = (\mathbf{g} * \mathbf{a} + \mathbf{e}) * \mathbf{b} = \mathbf{g} * \mathbf{a} * \mathbf{b} + \mathbf{e} * \mathbf{b}. \tag{10}$$

The distance Δ therefore consists only of products of "noise" parameters:

$$\Delta = \mathbf{x} - \mathbf{y} = \mathbf{e}' * \mathbf{a} - \mathbf{e} * \mathbf{b}. \tag{11}$$

We observe that each of $\{\mathbf{a}, \mathbf{b}, \mathbf{e}, \mathbf{e}'\}$ in Δ are picked independently from χ, which should be relatively "small' and zero-centered. The coefficients of both $\mathbf{x}$ and $\mathbf{y}$ are dominated

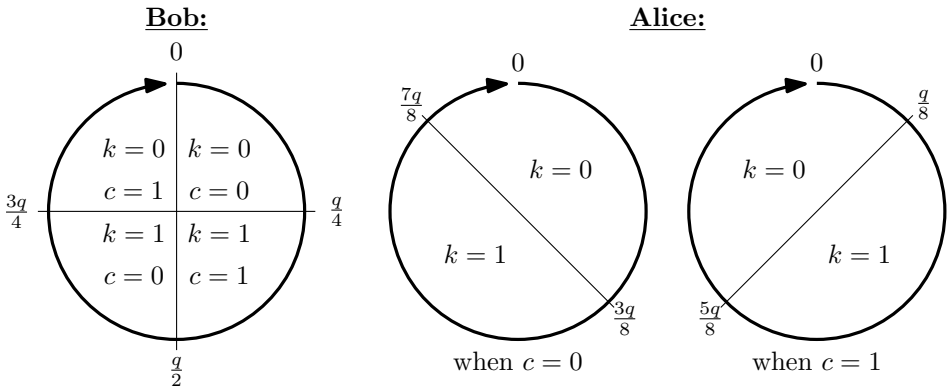

Figure 1: Simplified view of Peikert's original reconciliation mechanism [Pei14], ignoring randomized rounding. Alice and Bob have points $x \approx y \in \mathbb{Z}_q$ that are close to each other. Bob uses y to choose k and c as shown on left, and transmits c to Alice. Alice can use x, c to always arrive at the same shared bit k' if $|x - y| < \frac{q}{8}$, as shown on right. Without randomized smoothing the two halves $k = 0$ and $k = 1$ have an area of unequal size (when q is an odd prime) and the resulting key will be slightly biased.

by common, uniformly distributed factor $\mathbf{g} * \mathbf{a} * \mathbf{b} \approx \mathbf{x} \approx \mathbf{y}$. Up to n shared bits can be decoded from coefficients of $\mathbf{x}$ and $\mathbf{y}$ by a simple binary classifier such as $\lfloor \frac{2x_i}{q} \rceil \approx \lfloor \frac{2y_i}{q} \rceil$.

This type of generation will generate some disagreeing bits due to error Δ, however. Furthermore, the output of the classifier is slightly biased when q is odd. This is why additional steps are required.

B.4.5 Reconciliation

Let $\mathbf{x} \approx \mathbf{y}$ be two vectors in $\mathbb{Z}_q^n$ with a relatively small difference in each coefficient; the distribution of the distance $\delta_i = x_i - y_i$ is strongly centered around zero. In reconciliation, we wish the holders of $\mathbf{x}$ and $\mathbf{y}$ (Alice and Bob, respectively) to be able to arrive at exactly the same shared secret (key) $\mathbf{k}$ with a small amount of communication $\mathbf{c}$. However, single-message reconciliation can also be described simply as a part of an encryption algorithm (not a protocol).

Peikert's Reconciliation and BCNS Instantiation. In Peikert's reconciliation for odd modulus [Pei14], Bob first generates a randomization vector $\mathbf{r}$ such that each $r_i \in \{0, \pm 1\}$ is uniform modulo two. Bob can then determine the public reconciliation $\mathbf{c}$ and shared secret $\mathbf{k}$ via

$$c_i = \left\lfloor \frac{2(2y_i - r_i)}{q} \right\rceil \bmod 2 \qquad k_i = \left\lfloor \frac{2y_i - r_i}{q} \right\rceil \bmod 2. \tag{12}$$

We define disjoint helper sets $I_0 = [0, \lfloor \frac{q}{2} \rfloor]$ and $I_1 = [-\lfloor \frac{q}{2} \rfloor, -1]$ and $E = [-\frac{q}{4}, \frac{q}{4})$. Alice uses $\mathbf{x}$ to arrive at the shared secret $\mathbf{k}' = \mathbf{k}$ via

$$k'_i = \begin{cases} 0, & \text{if } 2x_i \in I_{c_i} + E \mod 2q \\ 1, & \text{otherwise.} \end{cases} \tag{13}$$

This mechanism is illustrated in Figure 1. Peikert's reconciliation was adopted for the Internet-oriented "BCNS" instantiation [BCNS15], which has a vanishingly small failure probability; $Pr(\mathbf{k}' \neq \mathbf{k}) < 2^{-16384}$.

New Hope Variants. "New Hope" is a prominent, more recent instantiation of Peikert's key exchange scheme [ADPS16b]. New Hope is parametrized at $n = 1024$, yet produces a 256-bit secret key k. This allowed the designers to develop a relatively complex reconciliation mechanism that uses $\frac{1024}{256} = 4$ coefficients of $\mathbf{x}$ and $2{*}4 = 8$ bits of reconciliation information to reach $< 2^{-60}$ failure rate.

In a follow-up paper [ADPS16a] the New Hope authors let Bob unilaterally choose the secret key, and significantly simplified their approach. This version also uses four coefficients, but requires $3 * 4 = 12$ bits of reconciliation (or "ciphertext") information per key bit. The total failure probability is the same $< 2^{-60}$.

Note that despite having a higher failure probability, the security level of New Hope (Section B.4.5) is higher than that of BCNS (Section B.4.5). Security of RLWE is closely related to the entropy and deviation of noise distribution χ in relation to modulus q. Higher noise ratio increases security against attacks, but also increases failure probability [APS15]. This is a fundamental trade-off in all Ring-LWE schemes.

References and Notes on Reconciliation. The term "reconciliation" comes from Quantum Cryptography. Standard Quantum Key Distribution (QKD) protocols such as BB84 [BB84] result in approximately agreeing shared secrets, which must be reconciled over a public channel with the help of classical information theory and cryptography [BBR88, BS93]. Ding et al. describe functionally similar (but mathematically very different) "Robust Extractors" in later versions of [DXL12] and patent [Din15, Din16].

B.4.6 SafeBits: New Reconciliation Method

We define a simpler, deterministic key and reconciliation bit generation rule from Bob's share $\mathbf{y}$ to be

$$k_i = \left\lfloor \frac{2y_i}{q} \right\rceil \quad \text{and} \quad c_i = \left\lfloor \frac{4y_i}{q} \right\rfloor \bmod 2. \tag{14}$$

Input y_i can be assumed to be uniform in range $[0, q-1]$. If taken in this plain form, the generator is slightly biased towards zero, since the interval for $k_i = 0$, $[0, \lfloor \frac{q}{2} \rfloor]$ is 1 larger than the interval $[\lceil \frac{q}{2} \rceil, q-1]$ for $k_i = 1$ when q is odd.

Intuition: Selecting safe bits (without reconciliation). Let's assume that we don't need all n bits given by the ring dimension. There is a straight-forward strategy for Bob to select m indexes in $\mathbf{y}$ that are most likely to agree. These safe coefficients are those that are closest to center points of $k = 0$ and $k = 1$ ranges, which in this case are $\frac{q}{4}$ and $\frac{3q}{4}$, respectively. Bob may choose a boundary window b, which defines shared bits to be used, and then communicate his binary selection vector $\mathbf{d}$ to Alice:

$$d_i = \begin{cases} 1 & \text{if } y_i \in \left[\lfloor \frac{q}{4} \rceil - b, \lfloor \frac{q}{4} \rceil + b \right] \quad \text{or} \quad y_i \in \left[\lfloor \frac{3q}{4} \rceil - b, \lfloor \frac{3q}{4} \rceil + b \right] \\ 0 & \text{otherwise.} \end{cases} \tag{15}$$

This simple case is illustrated on left side of Figure 2.

Since $\mathbf{y}$ is uniform in $\mathbb{Z}_q^n$, the Hamming weight of $\mathbf{d} = \mathsf{SafeBits}(\mathbf{y})$ satisfies $\mathsf{Wt}(\mathbf{d}) = \sum_{i=1}^{n-1} d_i \approx \frac{4b+2}{q} n$. Note that if not enough bits for the required payload can be obtained with bound b, Bob should re-randomize $\mathbf{y}$ rather than raising b as that can have an unexpected effect on failure rate. If there are too many selection bits for desired payload, one can just ignore them.

Importantly, both partitions are of equal size $2b + 1$ and therefore k is unbiased if there are no bit failures. If Alice also uses the simple rule $k'_i = \lfloor \frac{2x_i}{q} \rfloor$ to derive key bits (without c_i), the distance between shares must be at least $|x_i - y_i| > \frac{q}{4} - b$ for a bit error to occur.

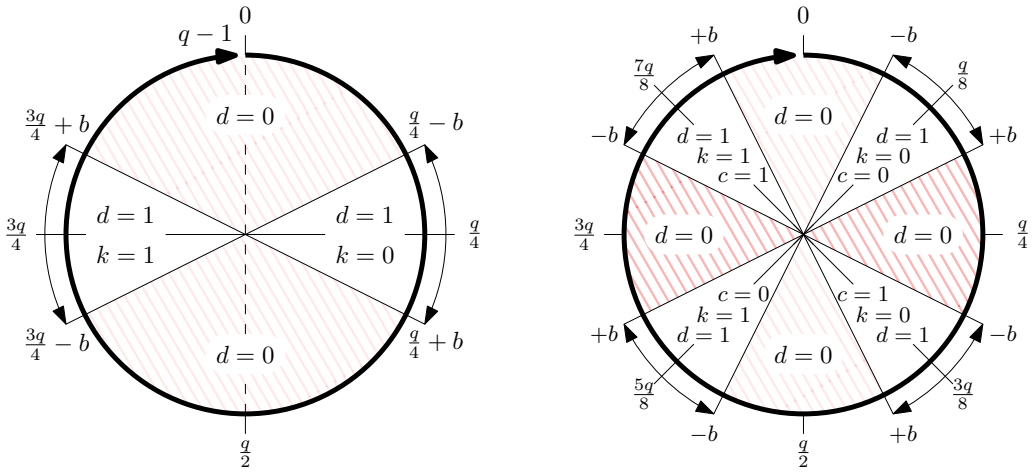

Figure 2: We use $k = \lfloor \frac{2y}{2} \rfloor$ ($k = 1$ on left half) instead of signed rounding $k = \lfloor \frac{2y}{2} + \epsilon \rceil$ ($k = 1$ in lower half) of Peikert (Figure 1). Illustration on the left gives intuition for the simple key bit selection and SafeBits without reconciliation. Bob uses window parameter b to select "safe" bits $d = 1$ which are farthest away from the negative ($k = 1$) / positive ($k = 0$) threshold. The bit selection d is sent to Alice, who then chooses the same bits as part of the shared secret k'. On right, safe bit selection when reconciliation bits c are used; this doubles the SafeBits "area". Each section constitutes a fraction $\frac{2b+1}{q}$, so bits are unbiased. The number of shared bits is not constant, however.

Even safer bits via Peikert's reconciliation. Let Bob use Equation 14 to determine his private key bits k_i and reconciliation bits c_i. Bob also uses a new $\mathbf{d} = \mathsf{SafeBits}(\mathbf{y}, b)$ function that accounts for Peikert-style reconciliation via

$$d_i = \begin{cases} 1 & \text{if } |(y_i \bmod \lfloor \frac{q}{4} \rceil) - \lfloor \frac{q}{8} \rfloor| \leq b \\ 0 & \text{otherwise.} \end{cases} \tag{16}$$

Note that there are now four "safe zones" (Figure 2, right side). Bob sends his bit selection vector $\mathbf{d}$ to Alice, along with reconciliation bits c_i at selected positions with $d_i = 1$. Alice can then get corresponding k'_i using c_i via

$$k'_i = \left\lfloor \frac{2}{q} \left(x_i - c_i \left\lfloor \frac{q}{4} \right\rceil + \left\lfloor \frac{q}{8} \right\rceil \mod q \right) \right\rceil . \tag{17}$$

Both parties derive a final key of length $m \leq \mathsf{Wt}(d)$ bits by concatenating the selected bits. Since $\mathbf{y}$ is uniform, each partition is still of size $2b + 1$, and the expected weight is now $\mathsf{Wt}(\mathbf{d}) = \sum_{i=1}^{n-1} d_i \approx \frac{8b+4}{q} n$, allowing the selection to be made essentially twice as tight while producing unbiased output.

Bob chooses key bits: Ding's Patents. Note that Bob is choosing the safe bits; he can use the direct rule of Equation 16, but really doesn't have to. In fact, such randomization may help security. With practical b boundaries there are typically many more bits with $d_i = 1$ than there are payload bits (Table 2); Bob can therefore directly choose much of the $\mathbf{k}$ secret, as in traditional public key encryption. Therefore patents [Din15, Din16] are not applicable as HILA5 does not perform reconciliation or joint-control key exchange as presented that work. This was also the rationale or "simple" New Hope variant [ADPS16a].

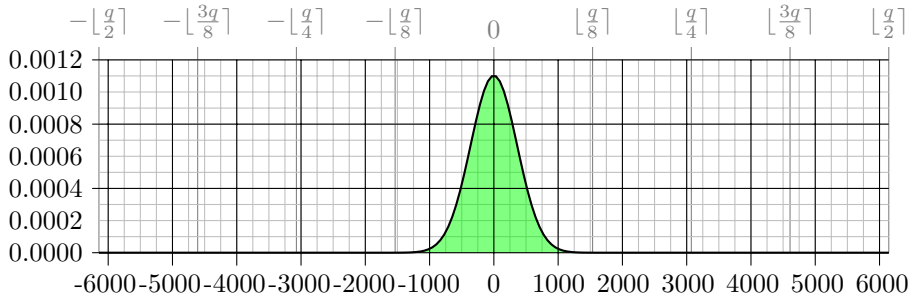

Figure 3: The error distribution E of $\delta = x_i - y_i$ (which we compute with high precision) is bell-shaped with variance $\sigma^2 = 2^{17}$. Its statistical distance to corresponding discrete Gaussian (with same σ) is $\approx 2^{-12.6}$, which has a significant effect on the bit failure rate. This is why we compute the discrete distributions numerically.

B.4.7 Analysis of Decryption Failure

Recall that we use the well-analyzed and optimized external ring parameters ($q = 12289$, $n = 1024$, and $\chi = \Psi_{16}$) from New Hope [ADPS16a, ADPS16b] in our proposal.

Definition 2. Let Ψ_k be a binomial distribution source

$$\Psi_k = \sum_{i=0}^{k} b_i - b'_i \quad \text{where} \quad b_i, b'_i \xleftarrow{\$} \{0, 1\}. \tag{18}$$

For random variable X from Ψ_k we have $P(X = i) = 2^{-2k}\binom{2k}{k+i}$. Furthermore, Ψ_k^n is a source of $\mathcal{R}$ elements where each one of n coefficients is independently chosen from Ψ_k. Since scheme is uses $k = 16$, a typical sampler implementation just computes the Hamming weight of a 32-bit random word and subtracts 16.

Lemma 1. *Let $\varepsilon, \varepsilon'$ be vectors of length $2n$ from Ψ_k^{2n}. Individual coefficients $\delta = \Delta_i$ of distance Equation 11 will have distribution equivalent to*

$$\delta = \sum_{i=1}^{2n} \varepsilon_i \varepsilon'_i. \tag{19}$$

Proof. When we investigate the multiplication rule of Equation 1, we see that each coefficient of independent polynomials $\{\mathbf{a}, \mathbf{b}, \mathbf{e}, \mathbf{e}'\}$ (or its inverse) in Δ is used in computation of each $\Delta_i = \delta$ exactly once. One may equivalently pick coefficients of $\varepsilon, \varepsilon'$ from $\{\pm\mathbf{e}, \pm\mathbf{e}', \pm\mathbf{s}_A, \pm\mathbf{s}_B\}$, without repetition. Therefore coefficients of $\varepsilon_i, \varepsilon'_i$ are independent and have distribution Ψ_k. □

Independence Assumption. Even though all of the variables in the sum of individual element $\delta = \Delta_i$ are independent in Equation 19, they are reused in other sums for $\Delta_j, i \neq j$. Therefore, while the average-case distribution of each one of the n coefficients of Δ is the same and precisely analyzable, they are not fully independent. In this work we perform error analysis on a single coefficient and then simply expand it to the whole vector. This independence assumption is analogous to our extension of LWE security properties to Ring-LWE with more structure and less independent variables.

The assumption is supported by our strictly bound error distribution Ψ_k and the structure of convolutions of signed random vectors (Equation 1). Our error estimate has a significant safety margin, however.

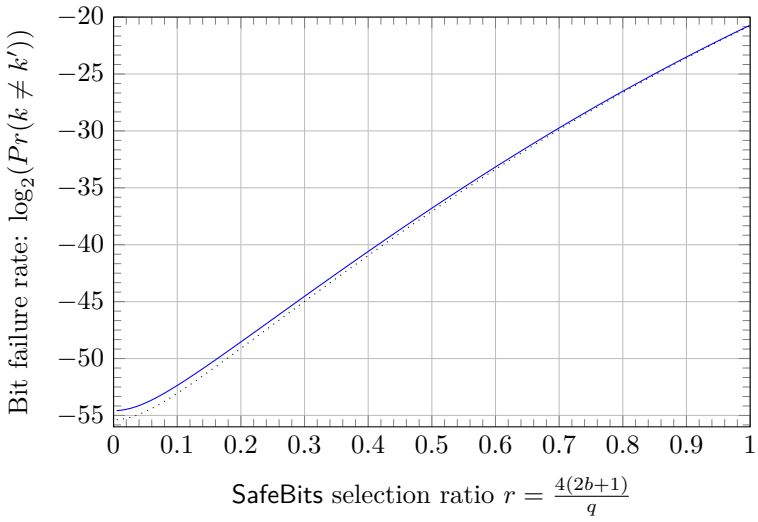

Figure 4: Relationship between individual bit failure rate and the selection window b. Dotted line is the rate derived from Gaussian approximation – it's up to 2× lower.

Computing the error distribution. The distribution of the product from two random variables from Ψ_k in Equation 19 is no longer binomial. Clearly its range is $[-k^2, k^2]$, but not all values are possible; for example, primes $p > k$ cannot occur in the product. However, it is easy to verify that the product is zero-centered and its standard deviation is exactly

$$\sigma = \sqrt{\sum_{i=-k}^{k} \sum_{j=-k}^{k} \frac{\binom{2k}{k+i}\binom{2k}{k+j}}{2^{4k}} (ij)^2} = \frac{k}{2}. \tag{20}$$

Hence, we may estimate δ of Equation 19 using the Central Limit Theorem as a Gaussian distribution with deviation

$$\sigma = \frac{k}{2}\sqrt{2n} \tag{21}$$

With our parameter selection this yields $\sigma \approx 362.0386$ (variance $\sigma^2 = 2^{17}$). However, the distribution of $X = \varepsilon_i \varepsilon'_i$ in Equation 19 is far from being "Bell-shaped" – its (total variation) statistical distance to a discrete Gaussian (with the same $\sigma = 8$) is ≈ 0.307988.

To calculate more accurate error distributions, we observe that since our domain $\mathbb{Z}_q$ is finite, we may always perform full convolutions between statistical distributions of independent random variables X and Y to arrive at the distribution of $X + Y$. The distributions can be represented as vectors of q real numbers. In order to get the exact shape of the error distribution we start with X, which is a "square" of Ψ_{16} and can be computed via binomial coefficients, as is done in Equation 20. The error distribution (Equation 19) is a sum $X + X + \cdots + X$ of $2n$ independent variables from that distribution. Using the convolution summing rule we can create a general "scalar multiplication algorithm" (analogous to square-and-multiply exponentiation) to quickly arrive at $E = 2048 \times X$.

We implemented finite distribution evaluation arithmetic in 256-bit floating point precision using the GNU MPFR library[3]. From these computations we know that the statistical distance of E to a discrete Gaussian with (same) $\sigma^2 = 2^{17}$ is approximately 0.0001603 or $2^{-12.6}$. Figure 3 illustrates this error distribution.

[3]The GNU MPFR is a widely available, free C library for multiple-precision floating-point computations with correct rounding: http://www.mpfr.org/

Proposition 1. *Bit selection mechanism of Section B.4.6 yields unbiased shared secret bits $k = k'$ if $\mathbf{y}$ is uniform. Discrete failure rate for individual bits $k \neq k'$ can be computed with high precision in our instance.*

Proof. Consider Bob's k value from in Equation 14, Bob's c and Alice's k' from Equation 17, and the four equivalently probable SafeBits ranges in Equation 16. With our $q = 12289$ instantiation the four possible $k \neq k'$ error conditions are:

Failure Case	Bob's y_i range for Y	Alice's Failing x_i
$k = 0, c = 0, k' = 1$	$[1536 - b, 1536 + b]$	$[4609, 10752]$
$k = 0, c = 1, k' = 1$	$[4608 - b, 4608 + b]$	$[0, 1535] \cup [7681, 12288]$
$k = 1, c = 0, k' = 0$	$[7680 - b, 7680 + b]$	$[0, 4608] \cup [10753, 12288]$
$k = 1, c = 1, k' = 0$	$[10752 - b, 10752 + b]$	$[1536, 7680]$

We examine each case separately (See Figure 2). Since the four non-overlapping y_i ranges are of the same size $2b+1$ and together constitute all selectable points $d_i = 1$ (Equation 16), the distribution of $k = k'$ is uniform. Furthermore, bit fail probability $k \neq k'$ is the average of these four cases. For each case, compute distribution Y which is uniform in the range of y_i. Then convolute it with error distribution to obtain $X = Y + E$, the distribution of x_i. The probability of failure is the sum of probabilities in X in the corresponding x_i failure range. □

B.4.8 Parameter Selection for Reconciliation

As can be seen in Figure 4, the relationship between window size b and bit failure rate is almost exponential. Some representative window sizes and payloads are given in Table 2, which also puts our selection $b = 799$ in context. Five-error correction (Section B.4.9) lowers the message failure probability to roughly $(2^{-27})^5 \approx 2^{-135}$ or even lower as 99% of six-bit errors are also corrected. We therefore meet the 2^{-128} message failure requirement with some safety margin.

Table 2: Potential window b sizes for safe bit selection (Equation 16) for different payload sizes. We target a payload of 496 bits, of which 256 are actual key bits and 240 bits are used to encrypt a five-error correcting code from XE5.

Payload bits* $m \approx r \times n$	Selection Window b	Selection Ratio $r = \frac{4(2b+1)}{q}$	Bit fail Probability p	Payload Failure $1 - (1-p)^m$
128	191	0.124664	$2^{-51.4715}$	$2^{-44.4715}$
256	383	0.249654	$2^{-46.5521}$	$2^{-38.5521}$
384	575	0.374644	$2^{-41.5811}$	$2^{-32.9962}$
496†	799	0.520465	$2^{-36.0359}$	$2^{-27.0818}$
512	767	0.499634	$2^{-36.8063}$	$2^{-27.8063}$
768	1151	0.749613	$2^{-28.1151}$	$2^{-18.5302}$
1024	1535	0.999593	$2^{-20.7259}$	$2^{-10.7263}$

* This is the minimum number of payload bits you get with 50% probability. The actual number is binomially distributed with density $f(k) = \binom{n}{k} r^k (1-r)^{n-k}$. Probability of at least m bits is therefore $\sum_{k=m}^{n} f(k)$.
† The payload could be 533 bits with 50% probability. We get 496 bits with 99% probability – this safety margin was chosen to minimize repetition rate (to $\approx \frac{1}{100}$).

B.4.9 Constant-Time Error Correction

We note that in our application the error correction mechanism operates on secret data. As with all other components of the scheme it is highly desirable that decoding can be implemented with an algorithm that requires constant processing time regardless of number of errors present. We are not aware of satisfactory constant-time decoding algorithms for BCH, Reed-Solomon, or other standard block multiple-error correcting codes [MS77, vL99].

We chose to design a linear block code specifically for our application. The design methodology is general, and a similar approach was used by the Author in the TRUNC8 Ring-LWE lightweight authentication scheme [Saa17c]. However, that work did not provide a detailed justification for the error correction code.

Definition 3. XE5 has a block size of 496 bits, out of which 256 bits are payload bits $\mathbf{p} = (p_0, p_1, \cdots, p_{255})$ and 240 provide redundancy $\mathbf{r}$. Redundancy is divided into ten subcodewords $r_0, r_1, \cdots, r_9$ of varying bit length $|r_i| = L_i$ with

$$(L_0, L_1, \cdots, L_9) = (16, 16, 17, 31, 19, 29, 23, 25, 27, 37). \tag{22}$$

Bits in each r_i are indexed $r_{(i,0)}, r_{(i,1)}, \cdots, r_{(i,L_i-1)}$. Each bit $k \in [0,\ L_0 - 1]$ in first subcodeword r_0 satisfies the parity equation

$$r_{0,k} = \sum_{j=0}^{15} p_{(16k+j)} \pmod 2 \tag{23}$$

and bits in $r_1, r_2, \cdots, r_9$ satisfy the parity congruence

$$r_{i,k} = \sum_{j-k \mid L_i} p_j \pmod 2. \tag{24}$$

We see that $r_{0,k}$ in Equation 23 is the parity of $k+1$:th block of 16 bits, while the $r_{i,k}$ in Equation 24 is parity of all p_j at congruent positions $j \equiv k \pmod{L_i}$.

Definition 4. For each payload bit position p_i we can assign corresponding integer "weight" $w_i \in [0, 10]$ as a sum

$$w_i = r_{(0,\lfloor i/16 \rfloor)} + \sum_{j=1}^{9} r_{(j, i \bmod L_j)}. \tag{25}$$

Lemma 2. *If message payload* $\mathbf{p}$ *only has a single nonzero bit* p_e*, then* $w_e = 10$ *and* $w_i \leq 1$ *for all* $i \neq e$*.*

Proof. Since each $L_i \geq \sqrt{|\mathbf{p}|}$ and all $L_{i\geq 1}$ are coprime (each is a prime power) it follows from the Chinese Remainder Theorem that any nonzero $i \neq j$ pair can satisfy both $r_{i, a \bmod L_i} = 1$ and $r_{j, a \bmod L_j} = 1$ only at $a = e$. Similar argument can be made for pairing $r_{0,a}$ with $r_{i\geq 1}$. Since the residues can be true pairwise only at e, weight w_a cannot be 2 or above when $a \neq e$. The $w_e = 10$ case follows directly from the Definition 3. □

Definition 5. Given XE5 input block $\mathbf{p} \mid \mathbf{r}$, we deliver a redundancy check $\mathbf{r}'$ from $\mathbf{p}$ via Equations 23 and 24. Furthermore we have distance $\mathbf{r}^\Delta = \mathbf{r} \oplus \mathbf{r}'$. Payload distance weight vector $\mathbf{w}^\Delta$ is derived from $\mathbf{r}^\Delta$ via Equation 25.

Since the code is entirely linear, Lemma 2 implies a direct way to correct a single error in $\mathbf{p}$ using Definition 5 – just flip bit p_x at position x where $w_x^\Delta = 10$. In fact any two redundancy subcodewords r_i and r_j would be sufficient to correct a single error in the payload; it's where $w_i^\Delta \geq 2$. It's easy to see if the single error would be in the redundancy part (r_i or r_j) instead of the payload, this is not an issue since in that case $w_x^\Delta \leq 1$ for all x. This type of reasoning leads to our main error correction strategy that is valid for up to five errors:

Theorem 1. *Let* $\mathbf{b} \mid \mathbf{r}$ *be an XE5 message block as in Definition 5. Changing each bit* p_i *when* $w_i^\Delta \geq 6$ *will correct a total of five bit errors in the block.*

Proof. We first note that if all five errors are in the redundancy part $\mathbf{r}$, then $w_i^\Delta \leq 5$ and no modifications in payload are done. If there are 4 errors in $\mathbf{r}$ and one in payload we still have $w_x^\Delta \geq 6$ at the payload error position p_x, etc. For each payload error p_x, each of ten subcodeword $\mathbf{r_i}$ will contribute one to weight w_x^Δ unless there is another congruent error p_y – i.e. we have $\lfloor x/16 \rfloor = \lfloor y/16 \rfloor$ for r_0 or $x \equiv y \pmod{L_i}$ for $r_{i \geq 1}$. Four errors cannot generate more than four such congruences (due to properties shown in the proof of Lemma 2), leaving fifth correctable via remaining six subcodewords ($w_i^\Delta \geq 6$). □

In order to verify the correctness of our implementation, we also performed a full exhaustive test (search space $\sum_{i=0}^{5} \frac{496!}{i!(496-i)!} \approx 2^{37.8}$). Experimentally XE5 corrects 99.4% of random 6-bit errors and 97.0% of random 7-bit errors.

Efficient constant-time implementation. The code generation and error correcting schemes can be implemented in bit-sliced fashion, without conditional clauses or table lookups on secret data. Please see listings in Section B.1.4 for an example implementation that runs in constant time.

The block is encoded simply as a 496-bit concatenation $\mathbf{p} \mid \mathbf{r}$. The reason for the ordering of L_i in Equation 22 is so that they can be packed into byte boundaries: $17 + 31 = 48$, $19 + 29 = 48$, $23 + 25 = 48$ and $27 + 37 = 64$.

B.5 Resistance to Known Attacks

Biases. Shared secret bits are unbiased. The shared key K also includes plaintext PT and ciphertext CT in the final hash to protect against a class of active attacks.

Quantum Attacks. Our new reconciliation mechanism has no effect on the security against (quantum) lattice attacks, so "New Hope" estimates [ADPS16b, AGVW17] are applicable. Currently this implies 2^{255} quantum security, with 2^{199} attacks plausible, which is well above the 2^{128} margin.

Pre-image security (but not collision resistance) is expected from SHA3 and SHAKE-256 in HILA5. Breaking the construction via these algorithms is expected to require approximately 2^{166} logical-qubit-cycles [AMG+16, CBHS17, Unr17].

Ring-LWE. Some researchers (notably authors of CRYSTALS - Kyber [BDK+17]) see risks in the algebraic structure of Ring-LWE and NTRU instances, and use that to motivate their use of Module-LWE. However, no actual attacks have been disclosed against our Ring-LWE parameters, and recent work such as [AD17, AGVW17] seems to reaffirm the original security estimates.

Timing and Side-Channel Attacks. The scheme has been designed from ground-up to be resistant against timing and side-channel attacks. The sampler Ψ_{16} is constant-time, as is our error correction code XE5. Ring arithmetic can also be implemented in constant time, but leakage can be further minimized via blinding [Saa17a] (Section 6).

B.6 Advantages and Limitations

Spec sheet: HILA5

Type:	Key Encapsulation and Public Key Encryption.
Underlying problem:	Ring-LWE (Learning With Errors in a Ring.)
Public key size:	1824 Bytes (+32 Byte private key hash.)
Private key size:	1792 Bytes (640 Bytes compressed.)
Ciphertext size:	2012 Byte expansion (KEM) + payload + MAC.
Failure rate:	$< 2^{-128}$, consistent with security level.
Classical security:	2^{256} (Category 5 – Equivalent to AES-256).
Quantum security:	2^{128} (Category 5 – Equivalent to AES-256).

B.6.1 Features

\+ **Very fast.** The HILA5 (ephemeral) key generation and decryption operations are tens (or even hundreds) times faster than with current RSA- or Elliptic Curve based cryptography at a commensurate level of classical security.

\+ **Drop-in compatible.** HILA5 is essentially drop-in compatible with current public key encryption applications. There are no practical usage restrictions. Key sizes and message expansion are of similar magnitude to current cryptographic standards.

\+ **Compact implementation.** HILA5 can be implemented on a wide range of target platforms, from most lightweight MCUs to high end vector architectures. No floating point arithmetic is required. Hardware implementations are straightforward.

\+ **Side-channel resistant.** HILA5 has been designed from ground up to be resistant against side-channel attacks such as timing attacks.

\+ **Well understood parameters.** Our Ring-LWE lattice parameters have attracted a lot of research and can be considered to be relatively conservative with a significant security margin. No vulnerabilities are known.

– **No signatures.** Just Key Encapsulation, Key Exchange, and Public Key Encryption. Algorithms such as BLISS [DDLL13, Saa17a] use very similar rings, however.

B.6.2 Compared to New Hope and other (R)LWE Proposals

\+ **HILA5 doesn't fail.** The algorithm has much lower failure probability, under 2^{-128} – compared to $2^{-38.9}$ for recommended parameters of Frodo [BCD+16], 2^{-60} for New Hope [ADPS16b], and even $2^{-71.9}$ for Kyber [BDK+17]. Non-negligible decryption failure rate is not acceptable in public key encryption applications.

\+ **Less randomness required.** Reconciliation method produces unbiased secrets without randomized smoothing; the system therefore requires less true randomness.

\+ **Non-malleable.** Computation of the final shared secret in HILA5 KEM uses the full public key and ciphertext messages, thereby reinforcing non-malleability and making a class of adaptive attacks infeasible.

\+ **Shorter messages.** Ciphertext messages are slightly smaller than New Hope's.

\+ **Patent free.** Since the sender can choose the message (see Section B.4.6), Ring-LWE key exchange patents [Din15, Din16] are even less applicable on this scheme.

– **Slightly slower.** Slight (< 5 %) performance penalty when compared to New Hope.

References

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 595–618. Springer, 2009. doi:10.1007/978-3-642-03356-8_35.

[AD17] Martin R. Albrecht and Amit Deo. Large modulus ring-lwe >= module-lwe. In *ASIACRYPT 2017*, 2017. URL: https://eprint.iacr.org/2017/612.

[ADPS16a] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Newhope without reconciliation. IACR ePrint 2016/1157, December 2016. URL: https://eprint.iacr.org/2016/1157.

[ADPS16b] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 16*, pages 327–343. USENIX Association, August 2016. Full version available as https://eprint.iacr.org/2015/1092. URL: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_alkim.pdf.

[AGL+10] Carlos Aguilar, Philippe Gaborit, Patrick Lacharme, Julien Schrek, and Gilles Zémor. Noisy Diffie-Hellman protocols, May 2010. Talk given by Philippe Gaborit at PQCrypto 2010 "Recent Results" session. URL: https://pqc2010.cased.de/rr/03.pdf.

[AGVW17] Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. Revisiting the expected cost of solving uSVP and applications to LWE. In *ASIACRYPT 2017*, 2017. URL: https://eprint.iacr.org/2017/815.

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. A new hope on ARM Cortex-M. IACR ePrint 2016/758, 2016. URL: https://eprint.iacr.org/2016/758.

[AMG+16] Matthew Amy, Olivia Di Matteo, Vlad Gheorghiu, Michele Mosca, Alex Parent, and John Schanck. Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. IACR ePrint 2016/992, 2016. To appear in Proc. SAC 2016. URL: http://eprint.iacr.org/2016/992.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, October 2015. URL: https://eprint.iacr.org/2015/046, doi:10.1515/jmc-2015-0016.

[BB84] Charles H. Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of IEEE International Conference on Computers, Systems and Signal Processing,*, pages 175–179. IEEE, December 1984. URL: http://researcher.watson.ibm.com/researcher/files/us-bennetc/BB84highest.pdf.

[BBR88] Charles H. Bennett, Gilles Brassard, and Jean-Marc Robert. Privacy amplification by public discussion. *Siam Journal on Computing*, 17(2):210–229, April 1988. doi:10.1137/0217014.

[BCD+16] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *ACM*

CCS 2016, pages 1006–1018. ACM, October 2016. Full version available as IACR ePrint 2016/659. URL: https://eprint.iacr.org/2016/659, doi:10.1145/2976749.2978425.

[BCNS15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *IEEE S & P 2015*, pages 553–570. IEEE Computer Society, 2015. Extended version available as IACR ePrint 2014/599. URL: https://eprint.iacr.org/2014/599, doi:10.1109/SP.2015.40.

[BDK+17] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. IACR ePrint 2016/634, 2017. URL: https://eprint.iacr.org/2017/634.

[BDP+16] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Caesar submission: Keyak v2, September 2016. CAESAR Candidate Specification. URL: http://keyak.noekeon.org/.

[BDPR98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *CRYPTO 1998*, volume 1462 of *LNCS*, pages 26–45. Springer, 1998. URL: https://www.di.ens.fr/~pointche/Documents/Papers/1998_crypto.pdf, doi:10.1007/BFb0055718.

[BS93] Gilles Brassard and Louis Salvail. Secret-key reconciliation by public discussion. In Tor Helleseth, editor, *EUROCRYPT 1993*, volume 765 of *LNCS*, pages 410–423. Springer, 1993. doi:10.1007/3-540-48285-7_35.

[CBHS17] Jan Czajkowski, Leon Groot Bruinderink, Andreas Hülsing, and Christian Schaffner. Quantum preimage, 2nd-preimage, and collision resistance of SHA3. IACR ePrint 2017/302, 2017. URL: https://eprint.iacr.org/2017/302.

[CJL+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. NISTIR 8105, April 2016. doi:10.6028/NIST.IR.8105.

[CLN16] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016*, volume 9814 of *LNCS*, pages 572–601. Springer, 2016. URL: https://eprint.iacr.org/2016/413, doi:10.1007/978-3-662-53018-4_21.

[CS03] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003. URL: http://www.shoup.net/papers/cca2.pdf, doi:10.1137/S0097539702403773.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, April 1965. doi:10.1090/S0025-5718-1965-0178586-1.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013*, pages 40–56. Springer, 2013. Extended version available as IACR ePrint 2013/383. URL: https://eprint.iacr.org/2013/383, doi:10.1007/978-3-642-40041-4_3.

[Din15] Jintai Ding. Improvements on cryptographic systems using pairing with errors, June 2015. Application PCT/CN2015/080697. URL: https://patents.google.com/patent/WO2015184991A1/en.

[Din16] Jintai Ding. New cryptographic systems using pairing with errors, January 2016. U.S. Patent US924667. URL: https://patents.google.com/patent/US9246675B2.

[Dwo07] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, November 2007. doi:10.6028/NIST.SP.800-38D.

[DXL12] Jintai Ding, Xiang Xie, and Xiaodong Lin. A simple provably secure key exchange scheme based on the learning with errors problem. IACR ePrint 2012/688, 2012. URL: https://eprint.iacr.org/2012/688.

[FIP01] FIPS. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, November 2001. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[FIP15] FIPS. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication 202, August 2015. doi:10.6028/NIST.FIPS.202.

[FNSW17] Roberta Faux, Karin Niles, Rino Sanchez, and John Wade. An FPGA study of lattice-based key exchanges, 2017. ETSI / IQC Quantum Safe Workshop, 13-15 September 2017, London, UK. URL: https://docbox.etsi.org/Workshop/2017/201709_ETSI_IQC_QUANTUMSAFE/TECHNICAL_TRACK/S04_SYSTEM_LEVEL_ISSUES/ENVIETA_FAUX.pdf.

[GS16] Shay Gueron and Fabian Schlieker. Speeding up R-LWE post-quantum key exchange. IACR ePrint 2016/467, 2016. URL: https://eprint.iacr.org/2016/467.

[KLC+17] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. Post-quantum key exchange on FPGAs. IACR ePrint 2017/690, 2017. URL: https://eprint.iacr.org/2017/690.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Sara Foresti and Giuseppe Persiano, editors, *CANS 2016*, volume 10052 of *LNCS*, pages 124–139. Springer, 2016. URL: https://eprint.iacr.org/2016/504, doi:10.1007/978-3-319-48965-0_8.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, 2011. doi:10.1007/978-3-642-19074-2_21.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, 2010. doi:10.1007/978-3-642-13190-5_1.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 35–54. Springer, 2013. Full version available as IACR ePrint 2013/293. URL: https://eprint.iacr.org/2013/293, doi:10.1007/978-3-642-38348-9_3.

[MS77] F. Jessie MacWilliams and Neil J.A. Sloane. *The theory of error-correcting codes*. North-Holland, 1977.

[NIS16] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Official Call for Proposals, National Institute for Standards and Technology, December 2016. URL: http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf.

[NSA16] NSA/CSS. Information assurance directorate: Commercial national security algorithm suite and quantum computing FAQ, January 2016. URL: https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm.

[Nus80] Henri J. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 28:205–215, 1980. doi:10.1109/TASSP.1980.1163372.

[Pei09] Chris Peikert. Some recent progress in lattice-based cryptography, March 2009. Invited Talk given at TCC 2009. URL: http://www.cc.gatech.edu/fac/cpeikert/pubs/slides-tcc09.pdf, doi:10.1007/978-3-642-00457-5_5.

[Pei14] Chris Peikert. Lattice cryptography for the internet. In Michele Mosca, editor, *PQCrypto 2014*, volume 8772 of *LNCS*, pages 197–219. Springer, 2014. URL: https://eprint.iacr.org/2014/070, doi:10.1007/978-3-319-11659-4_12.

[PZ03] John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Information & Computation*, 3(4):317–344, July 2003. Updated version available on arXiv. URL: https://arxiv.org/abs/quant-ph/9508027.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC '05*, pages 84–93. ACM, May 2005. doi:10.1145/1060590.1060603.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34:1–34:40, September 2009. doi:10.1145/1568318.1568324.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In *ACM CCS 2002*, pages 98–107. ACM Press, 2002. URL: http://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf, doi:10.1145/586110.586125.

[Saa17a] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures. *Journal of Cryptographic Engineering*, 2017. To appear. URL: http://rdcu.be/oHun, doi:10.1007/s13389-017-0149-6.

[Saa17b] Markku-Juhani O. Saarinen. HILA5: On reliability, reconciliation, and error correction for Ring-LWE encryption. In *Selected Areas in Cryptography – SAC 2017. 24th International Conference, Ottawa, ON, Canada, August 16 - 18, 2017*, volume 10640 of *LNCS*. Springer, August 2017. To Appear. Preprint available as IACR ePrint 2017/424. URL: https://eprint.iacr.org/2017/424.

[Saa17c] Markku-Juhani O. Saarinen. Ring-LWE ciphertext compression and error correction: Tools for lightweight post-quantum cryptography. In *Proceedings of the 3rd ACM International Workshop on IoT Privacy, Trust, and Security*, IoTPTS '17, pages 15–22. ACM, April 2017. doi:10.1145/3055245.3055254.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. FOCS '94*, pages 124–134. IEEE, 1994. Updated version available on arXiv. URL: https://arxiv.org/abs/quant-ph/9508027, doi:10.1109/SFCS.1994.365700.

[SM16] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. IACR ePrint 2016/1017, 2016. Based on the Stafford Tavares Invited Lecture at Selected Areas in Cryptography (SAC) 2016 by D. Stebila. URL: https://eprint.iacr.org/2016/1017.

[SS17] Silvan Streit and Fabrizio De Santis. Post-quantum key exchange on ARMv8-A – a new hope for NEON made simple. IACR ePrint 2017/388, 2017. URL: https://eprint.iacr.org/2017/388.

[Unr17] Dominique Unruh. Collapsing sponges: Post-quantum security of the sponge construction. IACR ePrint 2017/282, 2017. URL: https://eprint.iacr.org/2017/282.

[vL99] Jacobus H. van Lint. *Introduction to Coding Theory*, volume 86 of *Graduate Texts in Mathematics*. Springer, 3rd edition, 1999. doi:10.1007/978-3-642-58575-3.