Meeting 25/3/7

What I did

- Finished reading FIPS 203
- Read some related math papers
  - (Skipped most of them)

FIPS 203

Federal Information Processing Standards Publication

Module-Lattice-Based Key-Encapsulation Mechanism Standard

Category: Computer Security **Subcategory: Cryptography**

Learning with error

LWE. The LWE problem asks to recover a secret $\mathbf{s} \in \mathbb{Z}_q^n$ given a sequence of 'approximate' random linear equations on $\mathbf{s}$. For instance, the input might be

$$f \text{ (mod 17)} = A \cdot s + error = \begin{aligned} 14s_1 + 15s_2 + 5s_3 + 2s_4 &\approx 8 \pmod{17} \\ 13s_1 + 14s_2 + 14s_3 + 6s_4 &\approx 16 \pmod{17} \\ 6s_1 + 10s_2 + 13s_3 + 1s_4 &\approx 3 \pmod{17} \\ 10s_1 + 4s_2 + 12s_3 + 16s_4 &\approx 12 \pmod{17} \\ 9s_1 + 5s_2 + 9s_3 + 6s_4 &\approx 9 \pmod{17} \\ 3s_1 + 6s_2 + 4s_3 + 5s_4 &\approx 16 \pmod{17} \\ &\vdots \\ 6s_1 + 7s_2 + 16s_3 + 2s_4 &\approx 3 \pmod{17} \end{aligned}$$

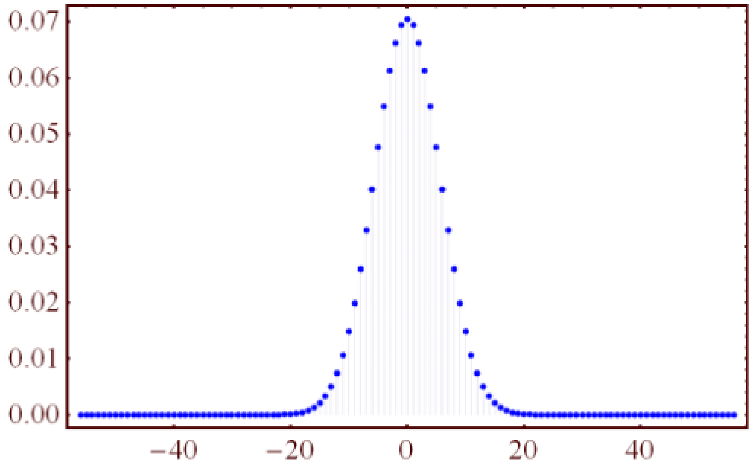


Figure 1: The error distribution with $q = 113$ and $\alpha = 0.05$.

→ Finding **s** after number of **f** → ∞

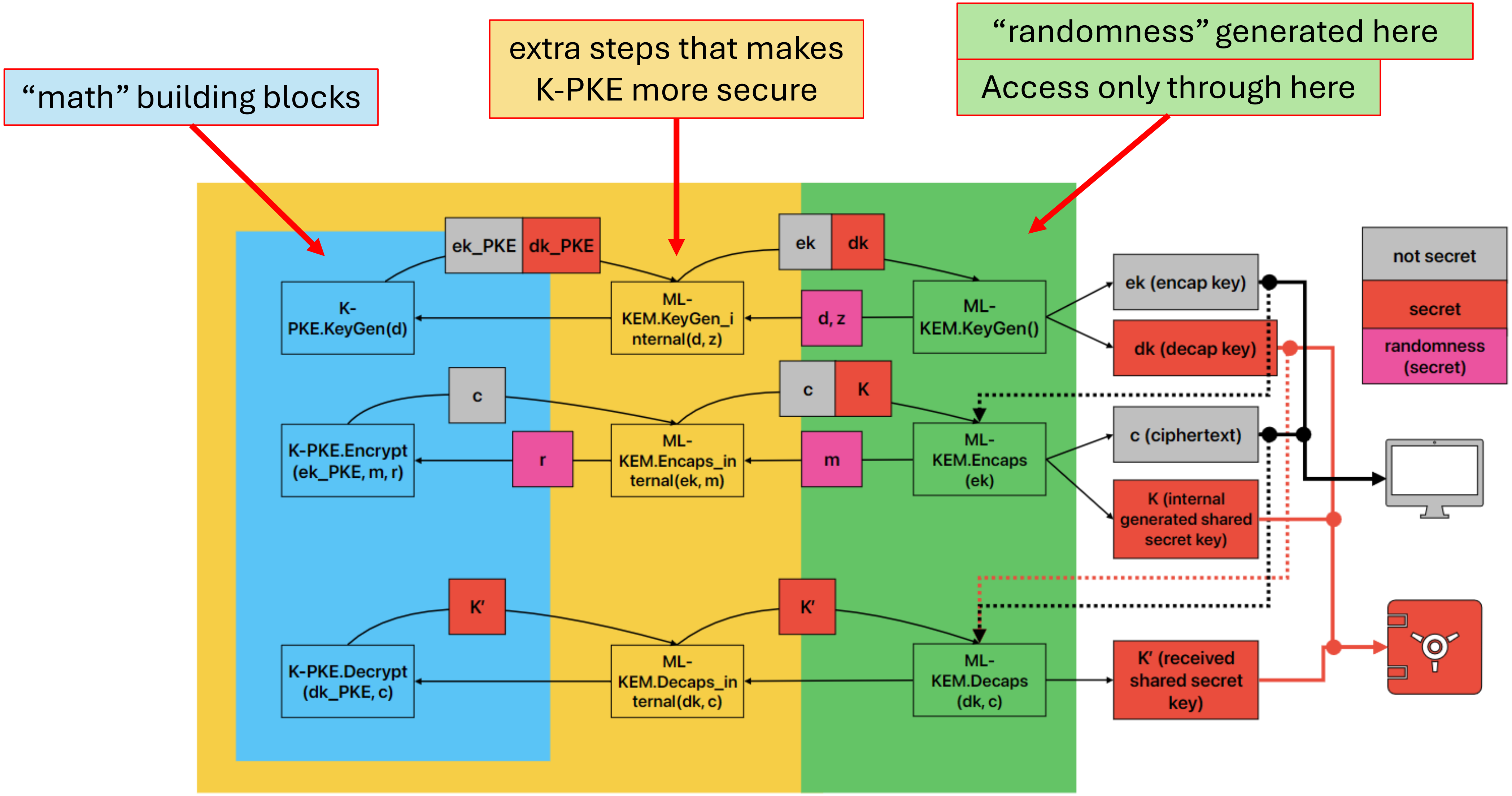
Kyber (learning-with error problem)

- $t = A \cdot s + e \ (\mathrm{mod}\ q)$
  - let $s \in \mathrm{k\ vector}$
  - let $t \in \mathrm{k\ vector}, A \in \mathrm{k} \times \mathrm{k\ matrix}, \mathrm{e} \in \mathrm{k\ vector}, q \in \mathrm{prime\ number}$
  - "**e**" (the error) is chosen from a probability distribution
- The problem:
  - Find **s** given access to **A** and **q**, and as many samples of **t**
- When samples → ∞, one can solve the LWE problem
  - (outputs **s** with high probability)
  - **Worst-case hardness** (all "**e**" must be found before **s** is found)
  - = very hard to find **s**

Kyber (public key crypto in its purest form)

- **Public key:** (A, t)
  - $t = A \cdot s + e$
- **Private key:** (s)
- **Encrypt:**
  - ciphertext = (u, v)
  - Randomness generated: (y, e1, e2)
  - $u = A^T \cdot y + e_1 \in \text{k vector}$
  - $v = t^T \cdot y + e_2 + \text{message}' \in \text{"number"}$
    - message' is an upscaled bit array of *message* ex. (1,0,1) → (1664,0,1664)
- **Decrypt:**
  - $v' = s^T \cdot u = s^T \cdot A^T \cdot y + e_1 \cdot s^T$
  - $\text{message}'' = v - v' = \underbrace{ey + e_2 - e_1 \cdot s^T}_{\text{small}} + \underbrace{\text{message}'}_{\text{large}}$ → *message*
  - Ex. (1665,123,1700) → (1,0,1)

Module-Lattice Key Encapsulation Mechanism (ML-KEM)



K-PKE Key Generation (Kyber-Public Key Encryption)

Algorithm 13 K-PKE.KeyGen(d)

Uses randomness to generate an encryption key and a corresponding decryption key.

Input: randomness $d \in \mathbb{B}^{32}$.
Output: encryption key $\mathsf{ek}_{\mathsf{PKE}} \in \mathbb{B}^{384k+32}$. ρ seed
Output: decryption key $\mathsf{dk}_{\mathsf{PKE}} \in \mathbb{B}^{384k}$.

$(\rho, \sigma) \leftarrow \boxed{\mathsf{G}(d \| k)}$ ▷ expand 32+1 bytes to two pseudorandom 32-byte seeds[1]
$N \leftarrow 0$
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$ global constant
  for $(j \leftarrow 0;\ j < k;\ j{+}{+})$
    $\hat{\mathbf{A}}[i,j] \leftarrow \boxed{\mathsf{SampleNTT}(\rho \| j \| i)}$ ▷ j and i are bytes 33 and 34 of the input
  end for
end for
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{s} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{s}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\underbrace{\mathsf{PRF}_{\eta_1}(\sigma, N)}_{\in \mathbb{B}^{64\eta_1}})$ ▷ $\mathbf{s}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{e} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{e}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(\sigma, N))$ ▷ $\mathbf{e}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$ ▷ run NTT k times (once for each coordinate of $\mathbf{s}$)
$\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$ ▷ run NTT k times
$\boxed{\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}}$ ▷ noisy linear system in NTT domain
$\mathsf{ek}_{\mathsf{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\mathbf{t}}) \| \rho$ ▷ run $\mathsf{ByteEncode}_{12}$ k times, then append $\hat{\mathbf{A}}$-seed
$\mathsf{dk}_{\mathsf{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\mathbf{s}})$ ▷ run $\mathsf{ByteEncode}_{12}$ k times
return $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}})$

- Seed (ρ, σ) generated via hash function (SHA3) from randomness **d**
- Seed (ρ, σ) expanded via hash functions to **(A, s, e)**
- Public key calculation

K-PKE Encryption (Kyber-Public Key Encryption)

Algorithm 14 K-PKE.Encrypt($\mathsf{ek}_{\mathsf{PKE}}, m, r$)

Uses the encryption key to encrypt a plaintext message using the randomness r.

Input: encryption key $\mathsf{ek}_{\mathsf{PKE}} \in \mathbb{B}^{384k+32}$. $\hat{A}$ stored in the form of ρ-seed
Input: message $m \in \mathbb{B}^{32}$. AES [illegible] max
Input: randomness $r \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

$N \leftarrow 0$
$\hat{\mathbf{t}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{ek}_{\mathsf{PKE}}[0:384k])$ ▷ run $\mathsf{ByteDecode}_{12}$ k times to decode $\hat{\mathbf{t}} \in (\mathbb{Z}_q^{256})^k$
$\rho \leftarrow \mathsf{ek}_{\mathsf{PKE}}[384k : 384k+32]$ ▷ extract 32-byte seed from $\mathsf{ek}_{\mathsf{PKE}}$
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ re-generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$ sampled in Alg. 13
  for $(j \leftarrow 0;\ j < k;\ j{+}{+})$
    $\hat{\mathbf{A}}[i,j] \leftarrow \boxed{\mathsf{SampleNTT}(\rho \| j \| i)}$ ▷ j and i are bytes 33 and 34 of the input
  end for
end for
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{y} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{y}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(r, N))$ ▷ $\mathbf{y}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{e_1} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{e_1}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$ ▷ $\mathbf{e_1}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
$e_2 \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$ ▷ sample $e_2 \in \mathbb{Z}_q^{256}$ from CBD
$\hat{\mathbf{y}} \leftarrow \mathsf{NTT}(\mathbf{y})$ ▷ run NTT k times
$\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^{\top} \circ \hat{\mathbf{y}}) + \mathbf{e_1}$ $\rightarrow u \in \mathbb{Z}_q^{256 \times k}$ ▷ run NTT^{-1} k times
$\mu \leftarrow \boxed{\mathsf{Decompress}_1}(\mathsf{ByteDecode}_1(m))$ $\rightarrow$ upscale "m" by $\frac{q}{2}$ ex. $(1, 0, 1, 1) \rightarrow (\frac{q}{2}, 0, \frac{q}{2}, \frac{q}{2})$
add "gap" $v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^{\top} \circ \hat{\mathbf{y}}) + e_2 + \mu$ $\rightarrow v \in \mathbb{Z}_q^{256}$ ▷ encode plaintext m into polynomial v
$c_1 \leftarrow \mathsf{ByteEncode}_{d_u}(\boxed{\mathsf{Compress}_{d_u}(\mathbf{u})})$ ▷ run $\mathsf{ByteEncode}_{d_u}$ and $\underbrace{\mathsf{Compress}_{d_u}}_{\text{drop bits}}$ k times
$c_2 \leftarrow \mathsf{ByteEncode}_{d_v}(\boxed{\mathsf{Compress}_{d_v}(v)})$ $v' \in \mathbb{Z}_{2^{d_v}}^{256} \rightarrow \mathbb{B}^{(256/8) \cdot d_v}$
return $c \leftarrow (c_1 \| c_2)$

- Regenerate matrix **A** via seed ρ in encryption key
- Randomness **r**
  - Generates (**y, e1, e2**)
  - Using hash functions (SHA3)
- Compress/decompress
  - Descale/upscale numbers
  - Reduce ciphertext size

K-PKE Decryption (Kyber-Public Key Encryption)

Algorithm 15 K-PKE.Decrypt($\mathsf{dk}_{\mathsf{PKE}}, c$)

Uses the decryption key to decrypt a ciphertext.

Input: decryption key $\mathsf{dk}_{\mathsf{PKE}} \in \mathbb{B}^{384k}$.
Input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Output: message $m \in \mathbb{B}^{32}$.

$c_1 \leftarrow c[0 : 32d_u k]$

$c_2 \leftarrow c[32d_u k : 32(d_u k + d_v)]$ $(A^T \cdot y) + e_1$

$\mathbf{u}' \leftarrow \mathsf{Decompress}_{d_u}(\mathsf{ByteDecode}_{d_u}(c_1))$ ▷ run $\mathsf{Decompress}_{d_u}$ and $\mathsf{ByteDecode}_{d_u}$ k times

$v' \leftarrow \mathsf{Decompress}_{d_v}(\mathsf{ByteDecode}_{d_v}(c_2))$ $(\hat{t}^T \circ \hat{y}) + e_2 + \mu$

$\hat{\mathbf{s}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{dk}_{\mathsf{PKE}})$ ▷ run $\mathsf{ByteDecode}_{12}$ k times

$w \leftarrow v' - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^{\top} \circ \mathsf{NTT}(\mathbf{u}'))$ ▷ run NTT k times; run NTT^{-1} once

$m \leftarrow \mathsf{ByteEncode}_1(\boxed{\mathsf{Compress}_1(w)})$ ▷ decode plaintext m from polynomial v

return m

- Compress to 2^1
  - Large number → 1
  - Small number → 0

ML-KEM internal (1/2)

Algorithm 16 ML-KEM.KeyGen_internal(d, z)

Uses randomness to generate an encapsulation key and a corresponding decapsulation key.

Input: randomness $d \in \mathbb{B}^{32}$.
Input: randomness $z \in \mathbb{B}^{32}$.
Output: encapsulation key $\mathsf{ek} \in \mathbb{B}^{384k+32}$.
Output: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.

1: $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}}) \leftarrow$ K-PKE.KeyGen(d) ▷ run key generation for K-PKE
2: $\mathsf{ek} \leftarrow \mathsf{ek}_{\mathsf{PKE}}$ ▷ KEM encaps key is just the PKE encryption key
3: $\mathsf{dk} \leftarrow \boxed{(\mathsf{dk}_{\mathsf{PKE}} \| \mathsf{ek} \| \mathsf{H}(\mathsf{ek}) \| z)}$ ▷ KEM decaps key includes PKE decryption key
4: **return** $(\mathsf{ek}, \mathsf{dk})$

$(\hat{t} \| \rho)$
(A matrix seed)

- Decryption key contains the hash of encryption key

Algorithm 17 ML-KEM.Encaps_internal(ek, m)

Uses the encapsulation key and randomness to generate a key and an associated ciphertext.

Input: encapsulation key $\mathsf{ek} \in \mathbb{B}^{384k+32}$.
Input: randomness $m \in \mathbb{B}^{32}$.
Output: shared secret key $K \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

1: $(K, r) \leftarrow \mathsf{G}(m \| \mathsf{H}(\mathsf{ek}))$ SHA3-512 ▷ derive shared secret key K and randomness r
2: $c \leftarrow$ K-PKE.Encrypt$\boxed{(\mathsf{ek}, m, r)}$ ▷ encrypt m using K-PKE with randomness r
3: **return** (K, c)

- Encapsulates the “seed” (m) of the shared secret key

ML-KEM internal (2/2)

Algorithm 18 ML-KEM.Decaps_internal(dk, c)

Uses the decapsulation key to produce a shared secret key from a ciphertext.

Input: decapsulation key dk $\in \mathbb{B}^{768k+96}$.
Input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Output: shared secret key $K \in \mathbb{B}^{32}$.

$\mathsf{dk}_{\mathsf{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$ ▷ extract (from KEM decaps key) the PKE decryption key
$\mathsf{ek}_{\mathsf{PKE}} \leftarrow \mathsf{dk}[384k : 768k + 32]$ ▷ extract PKE encryption key
$h \leftarrow \mathsf{dk}[768k + 32 : 768k + 64]$ ▷ extract hash of PKE encryption key
$z \leftarrow \mathsf{dk}[768k + 64 : 768k + 96]$ ▷ extract implicit rejection value
$m' \leftarrow$ K-PKE.Decrypt$(\mathsf{dk}_{\mathsf{PKE}}, c)$ ▷ decrypt ciphertext
$(K', r') \leftarrow \mathsf{G}(m' \| h)$
$\bar{K} \leftarrow \mathsf{J}(z \| c)$
$c' \leftarrow$ K-PKE.Encrypt$(\mathsf{ek}_{\mathsf{PKE}}, m', r')$ ▷ re-encrypt using the derived randomness r'
if $c \neq c'$ **then**
    $K' \leftarrow \bar{K}$ ▷ if ciphertexts do not match, "implicitly reject"
end if
return K'

- Re-encrypt the derived message (m') to check for validity
  - Reject invalid ciphertext implicitly
  - to prevent *chosen-ciphertext attack*
  - (the why is explained in a math paper I skipped reading)

ML-KEM

Algorithm 19 ML-KEM.KeyGen()

Generates an encapsulation key and a corresponding decapsulation key.

Output: encapsulation key $\mathsf{ek} \in \mathbb{B}^{384k+32}$.
Output: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.

```
d ←$− B^32                                  ▷ d is 32 random bytes (see Section 3.3)
z ←$− B^32                                  ▷ z is 32 random bytes (see Section 3.3)
if d == NULL or z == NULL then
    return ⊥                                ▷ return an error indication if random bit generation failed
end if
(ek, dk) ← ML-KEM.KeyGen_internal(d, z)     ▷ run internal key generation algorithm
return (ek, dk)
```

Algorithm 20 ML-KEM.Encaps(ek)

Uses the encapsulation key to generate a shared secret key and an associated ciphertext.

Checked input: encapsulation key $\mathsf{ek} \in \mathbb{B}^{384k+32}$.
Output: shared secret key $K \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

```
m ←$− B^32                                  ▷ m is 32 random bytes (see Section 3.3)
if m == NULL then
    return ⊥                                ▷ return an error indication if random bit generation failed
end if
(K, c) ← ML-KEM.Encaps_internal(ek, m)      ▷ run internal encapsulation algorithm
return (K, c)
```

Algorithm 21 ML-KEM.Decaps(dk, c)

Uses the decapsulation key to produce a shared secret key from a ciphertext.

Checked input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Checked input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Output: shared secret key $K \in \mathbb{B}^{32}$.

```
K' ← ML-KEM.Decaps_internal(dk, c)          ▷ run internal decapsulation algorithm
return K'
```

- “API”
- Generates the randomness (not deterministic hashes)

Compression/decompression

- In short...
  - Compress: Map q → 2^d
  - Decompress: Map 2^d → q
- $d \leq \text{ceil}(\log_2(q))$
- q = 3329 = 2^8 *13+1

$$\begin{aligned} \text{Compress}_d : \mathbb{Z}_q &\longrightarrow \mathbb{Z}_{2^d} \\ x &\longmapsto \lceil (2^d/q) \cdot x \rfloor \bmod 2^d . \end{aligned}$$

$$\begin{aligned} \text{Decompress}_d : \mathbb{Z}_{2^d} &\longrightarrow \mathbb{Z}_q \\ y &\longmapsto \lceil (q/2^d) \cdot y \rfloor . \end{aligned}$$

K-PKE.Encrypt

20: $\mu \leftarrow \text{Decompress}_1(\text{ByteDecode}_1(m))$

22: $c_1 \leftarrow \text{ByteEncode}_{d_u}(\text{Compress}_{d_u}(\mathbf{u}))$

23: $c_2 \leftarrow \text{ByteEncode}_{d_v}(\text{Compress}_{d_v}(v))$

K-PKE.Decrypt

7: $m \leftarrow \text{ByteEncode}_1(\text{Compress}_1(w))$

Number theoretic transform (NTT)

- aka “Discrete Fourier transform over a ring”
- “number” = a polynomial (多項式)
- Makes polynomial multiplication faster
- Polynomial “number”/vector/matrix
  - → transform polynomial to NTT domain
  - → direct multiplication
  - → transform back to polynomials
- Addition still must be done in polynomial domain

K-PKE.KeyGen

16: $\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$

17: $\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$

18: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$

K-PKE.Encrypt

21: $v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^{\top} \circ \hat{\mathbf{y}}) + e_2 + \mu$.

Next steps…

- Check the C code https://github.com/pq-crystals/kyber
  - The inventors of Kyber's code, still actively maintained
- Build the major components first:
  - Key calculations/encrypt/decrypt→ deterministic sampling functions → hash functions?
- Learn similar building blocks for calculations in HDL
- Check known Kyber implementations
- Learn why and how ML-KEM is secure
  - Math paper readings I skipped
- Learn what secure storage is (for the decryption keys)

Thanks for listening

:P