CRYSTALS-kyber 算法梳理

CRYSTALS-kyber 算法梳理

'''
@Descripttion: CRYSTALS-kyber 算法梳理
@version: V1.0
@Author: HZW
@Date: 2025-03-07 19:21:00
'''

简介

- ref：doc/ National Institute of Standards and Technology (US) - 2024 - Module-lattice-based key-encapsulation mechanism standard.pdf
- 本文档旨在梳理ML_KEM算法的数据流变化;
- 阅读时，简要阅读1.辅助函数章节，重点关注3.K_PKE;
- 如果只关注算法流程，可以跳过2.快速数论变换（NTT）的原理推导部分;
- 在阅读3.K_PKE时，可以不用关注NTT,INTT以及辅助函数的具体实现;
- 重点关注数据位宽的变化;

1.辅助函数

(1)BitsToBytes(b)

Algorithm 3 BitsToBytes(b)

Converts a bit array (of a length that is a multiple of eight) into an array of bytes.

Input: bit array $b \in \{0,1\}^{8\cdot\ell}$.
Output: byte array $B \in \mathbb{B}^{\ell}$.

```
1: B ← (0,...,0)
2: for (i ← 0; i < 8ℓ; i++)
3:     B[⌊i/8⌋] ← B[⌊i/8⌋] + b[i] · 2^(i mod 8)
4: end for
5: return B
```

- 功能：比特数组转字节数组
- b这里为小端存储，比如 $11010001 = 2^0 + 2^1 + 2^3 + 2^7 = 139$

(2)BytesToBits(B)

Algorithm 4 BytesToBits(B)

Performs the inverse of BitsToBytes*, converting a byte array into a bit array.*

Input: byte array $B \in \mathbb{B}^{\ell}$.
Output: bit array $b \in \{0,1\}^{8\cdot\ell}$.

```
1: C ← B                                    ▷ copy B into array C ∈ 𝔹^ℓ
2: for (i ← 0; i < ℓ; i++)
3:     for (j ← 0; j < 8; j++)
4:         b[8i + j] ← C[i] mod 2
5:         C[i] ← ⌊C[i]/2⌋
6:     end for
7: end for
8: return b
```

- 功能:字节数组转比特数组
- b这里为小端存储，比如 $11010001 = 2^0 + 2^1 + 2^3 + 2^7 = 139$

(3)sampleNTT(B)

Algorithm 7 SampleNTT(B)

Takes a 32-byte seed and two indices as input and outputs a pseudorandom element of T_q.

Input: byte array $B \in \mathbb{B}^{34}$. ▷ a 32-byte seed along with two indices
Output: array $\hat{a} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the NTT of a polynomial

ctx ← XOF.Init()
ctx ← XOF.Absorb(ctx, B) ▷ input the given byte array into XOF
$j \leftarrow 0$
while $j < 256$ **do**
  (ctx, C) ← XOF.Squeeze(ctx, 3) ▷ get a fresh 3-byte array C from XOF
  $d_1 \leftarrow C[0] + 256 \cdot (C[1] \bmod 16)$ ▷ $0 \le d_1 < 2^{12}$
  $d_2 \leftarrow \lfloor C[1]/16 \rfloor + 16 \cdot C[2]$ ▷ $0 \le d_2 < 2^{12}$
  if $d_1 < q$ **then**
    $\hat{a}[j] \leftarrow d_1$ ▷ $\hat{a} \in \mathbb{Z}_q^{256}$
    $j \leftarrow j+1$
  end if
  if $d_2 < q$ and $j < 256$ **then**
    $\hat{a}[j] \leftarrow d_2$
    $j \leftarrow j+1$
  end if
end while
return $\hat{a}$

- 功能：均匀采样，在其他版本中为Parse
- 输入：B，34字节，其中32字节的seed加两个字节的索引
- 输出: $\hat{a} \in Z_q^{256}$

(4)sampleCBD(B)

Algorithm 8 SamplePolyCBD$_\eta(B)$

Takes a seed as input and outputs a pseudorandom sample from the distribution $\mathcal{D}_\eta(R_q)$.

Input: byte array $B \in \mathbb{B}^{64\eta}$.
Output: array $f \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the sampled polynomial

1: $b \leftarrow$ BytesToBits(B)
2: **for** $(i \leftarrow 0; i < 256; i{+}{+})$
3:   $x \leftarrow \sum_{j \leftarrow 0}^{\eta-1} b[2i\eta + j]$ ▷ $0 \le x \le \eta$
4:   $y \leftarrow \sum_{j \leftarrow 0}^{\eta-1} b[2i\eta + \eta + j]$ ▷ $0 \le y \le \eta$
5:   $f[i] \leftarrow x - y \bmod q$ ▷ $0 \le f[i] \le \eta$ or $q - \eta \le f[i] \le q-1$
6: **end for**
7: **return** f

- 功能：中心二项分布采样，在其他版本中为CBD
- 输入：$B^{64\eta}$，64η字节的种子，其中$\eta \in \{2,3\}$
- 输出: $f \in Z_q^{256}$
- 1： b是一个512η长度的bit数组

(5)ByteEncode(B)（编码函数）

Algorithm 5 ByteEncode$_d(F)$

Encodes an array of d-bit integers into a byte array for $1 \le d \le 12$.

Input: integer array $F \in \mathbb{Z}_m^{256}$, where $m = 2^d$ if $d < 12$, and $m = q$ if $d = 12$.
Output: byte array $B \in \mathbb{B}^{32d}$.

```
for (i ← 0; i < 256; i++)
    a ← F[i]                                  ▷ a ∈ Z_m
    for (j ← 0; j < d; j++)
        b[i·d+j] ← a mod 2                    ▷ b ∈ {0,1}^{256·d}
        a ← (a − b[i·d+j])/2                  ▷ note a − b[i·d+j] is always even
    end for
end for
B ← BitsToBytes(b)
return B
```

- 功能：将一个长度为256的整数数组转换为长度32d的字节数组，其中整数数组的元素为d bit大小
- 输入：$\boldsymbol{F} \in \mathbb{Z}_m^{256}$，其中$m = 2^d, 1 \le d < 12$,and $m = q$ if $d = 12$。
- 输出: B为长度32d的字节数组
- 长度变换为：$256 \times d = 32 \times d \times 8$

(6)ByteDecode(B)（解码函数，序列反序化）

Algorithm 6 ByteDecode$_d(B)$

Decodes a byte array into an array of d-bit integers for $1 \le d \le 12$.

Input: byte array $B \in \mathbb{B}^{32d}$.
Output: integer array $F \in \mathbb{Z}_m^{256}$, where $m = 2^d$ if $d < 12$ and $m = q$ if $d = 12$.

```
b ← BytesToBits(B)
for (i ← 0; i < 256; i++)
    F[i] ← Σ_{j←0}^{d−1} b[i·d+j]·2^j mod m
end for
return F
```

- 功能：将一个长度为32d的字节数组转换为整数数组，其中整数数组的元素为d bit大小
- 长度变换为：$32 \times d \times 8 = 256 \times d$

(7)compress and Decompress

$$\mathsf{Compress}_d : \mathbb{Z}_q \longrightarrow \mathbb{Z}_{2^d} \tag{4.7}$$
$$x \longmapsto \lceil (2^d/q) \cdot x \rfloor \bmod 2^d .$$
$$\mathsf{Decompress}_d : \mathbb{Z}_{2^d} \longrightarrow \mathbb{Z}_q \tag{4.8}$$
$$y \longmapsto \lceil (q/2^d) \cdot y \rfloor .$$

- 功能：compress用于舍弃密文中一些对解密正确性影响不大的低比特位，从而减小密文大小。Decompress为其逆过程。

2.快速数论变换(NTT)

(1)NTT的数学结构

1）NTT的原理

对于$n=256$，$q=3329$时，在Z_q上只有256次单位根，即$\zeta^{256} \bmod q = \zeta^{256} \bmod 3329 = 1$，其中$\zeta=17$时256次单位根。因此有

$$\zeta^{256} = (\zeta^{128})^2 \equiv 1 \bmod q \Rightarrow \zeta^{128} \equiv -1 \bmod q \tag{1}$$

则:

$$\begin{aligned} X^{256}+1 &= (X^{256}-\zeta^{128}) \\ &= \prod_{i=0}^{127}(X^2-\zeta^{2i+1}) \\ &= \prod_{i=0}^{127}(X^2-\zeta^{2\mathrm{BitRev}_7(i)+1}) \bmod q \end{aligned} \tag{2}$$

其中$\mathrm{BitRev}_7(r)$作用是将7bit无符号数的bit位顺序反转，即
$\mathrm{BitRev}_7(r) = \mathrm{BitRev}_7(r_0 2^0 + r_1 2^1 + \dots r_6 2^6) = r_6 2^0 + r_5 2^1 + \dots r_0 2^6$。因此，多项式环$R_q = \mathbb{Z}_q[X]/(X^{256}+1)$同构于128个2次扩展的直和，即$T_q = \bigoplus_{i=1}^{n} \mathbb{Z}_q[X]/(X^2-\zeta^{2\mathrm{BitRev}_7(\mathrm{i})+1})$.

多项式$f=\sum_{i=0}^{255} f_i x^i$的NTT形式为:

$$\hat{f} = \mathrm{NTT}(f) = \hat{f}_0 + \hat{f}_1 X + \cdots \hat{f}_{255} X^{255} \tag{3}$$

注意上述代数结构$\mathrm{NTT}(f)$不具有任何数学意义。基于环上中国剩余定理，即多项式环$R_q \to T_q$的同构映射。实际$\mathrm{NTT}(f)$的各个系数可以表示为以下128个1次剩余多项式组成的向量:

$$\begin{aligned} \hat{f} &= \mathrm{NTT}(f) \\ &= (f \bmod X^2-\zeta^{2\mathrm{BitRev}_7(0)+1}, \dots, f \bmod X^2-\zeta^{2\mathrm{BitRev}_7(127)+1}) \\ &= (\hat{f}_0+\hat{f}_1 X, \hat{f}_2+\hat{f}_3 X, \dots, \hat{f}_{254}+\hat{f}_{255} X) \\ &= (F_0, F_1, \dots, F_{127}) \end{aligned} \tag{4}$$

证明以下过程：$f \bmod X^2-\zeta^{2\mathrm{BitRev}_7(i)+1} \Rightarrow \hat{f}_{2i}+\hat{f}_{2i+1}X$
因为：$X^2 \equiv \zeta^{2\mathrm{BitRev}_7(i)+1} \bmod X^2-\zeta^{2\mathrm{BitRev}_7(i)+1}$

$$\begin{aligned} f \bmod X^2-\zeta^{2\mathrm{BitRev}_7(i)+1} &= \sum_{i=0}^{255} f_i x^i \bmod X^2-\zeta^{2\mathrm{BitRev}_7(i)+1} \\ &= (\sum_{j=0}^{127} f_{2j}x^{2j} + \sum_{j=0}^{127} f_{2j+1}x^{2j+1}) \bmod X^2-\zeta^{2\mathrm{BitRev}_7(i)+1} \\ &= (\sum_{j=0}^{127} f_{2j}\zeta^{(2\mathrm{BitRev}_7(i)+1)j} + X\sum_{j=0}^{127} f_{2j+1}\zeta^{(2\mathrm{BitRev}_7(i)+1)j}) \bmod X^2-\zeta^{2\mathrm{BitRev}_7(i)+1} \end{aligned} \tag{5}$$

令:

$$\begin{aligned} \hat{f}_{2i} &= \sum_{j=0}^{127} f_{2j}\zeta^{(2\mathrm{BitRev}_7(i)+1)j} \\ \hat{f}_{2i+1} &= \sum_{j=0}^{127} f_{2j+1}\zeta^{(2\mathrm{BitRev}_7(i)+1)j} \end{aligned} \tag{6}$$

因此，$f \bmod X^2-\zeta^{2\mathrm{BitRev}_7(i)+1} = \hat{f}_{2i}+\hat{f}_{2i+1}X$
实际上NTT算法的核心就是利用单位根的对称性加速式（6）的计算

2）PWM逐点乘法，符号记作○

对于多项式乘法$h(x) = f(x) \cdot g(x) \bmod x^n + 1$，其中$h(x)$的NTT向量形式为$(H_0, H_1, \ldots, H_{127})$，$g(x)$的NTT向量形式为$(G_0, G_1, \ldots, G_{127})$
则多项式乘积的NTT系数向量为：

$$\begin{aligned}(H_0, H_1, \ldots, H_{127}) &= (F_0, F_1, \ldots, F_{127}) \circ (G_0, G_1, \ldots, G_{127}) \\ &= (F_0 \cdot G_0 \bmod X^2 - \zeta^{2\mathrm{BitRev}_7(0)+1}, \ldots, F_{127} \cdot G_{127} \bmod X^2 - \zeta^{2\mathrm{BitRev}_7(127)+1})\end{aligned} \tag{7}$$

对于H_i有：

$$\begin{aligned}H_i &= F_i \cdot G_i \bmod X^2 - \zeta^{2\mathrm{BitRev}_7(i)+1} \\ &= (\hat{f}_{2i} + X\hat{f}_{2i+1}) \cdot (\hat{g}_{2i} + X\hat{g}_{2i+1}) \bmod X^2 - \zeta^{2\mathrm{BitRev}_7(i)+1} \\ &= (\hat{f}_{2i}\hat{g}_{2i} + X^2\hat{f}_{2i+1}\hat{g}_{2i+1} + X\hat{f}_{2i}\hat{g}_{2i+1} + X\hat{f}_{2i+1}\hat{g}_{2i}) \bmod X^2 - \zeta^{2\mathrm{BitRev}_7(i)+1} \\ &= (\hat{f}_{2i}\hat{g}_{2i} + \hat{f}_{2i+1}\hat{g}_{2i+1}\zeta^{2\mathrm{BitRev}_7(i)+1} + X(\hat{f}_{2i}\hat{g}_{2i+1} + \hat{f}_{2i+1}\hat{g}_{2i})) \bmod X^2 - \zeta^{2\mathrm{BitRev}_7(i)+1}\end{aligned} \tag{8}$$

---

(2) 伪代码

1）NTT

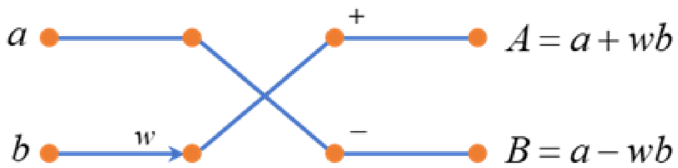
Algorithm 9 NTT(f)

Computes the NTT representation $\hat{f}$ of the given polynomial $f \in R_q$.

Input: array $f \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the input polynomial
Output: array $\hat{f} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the NTT of the input polynomial

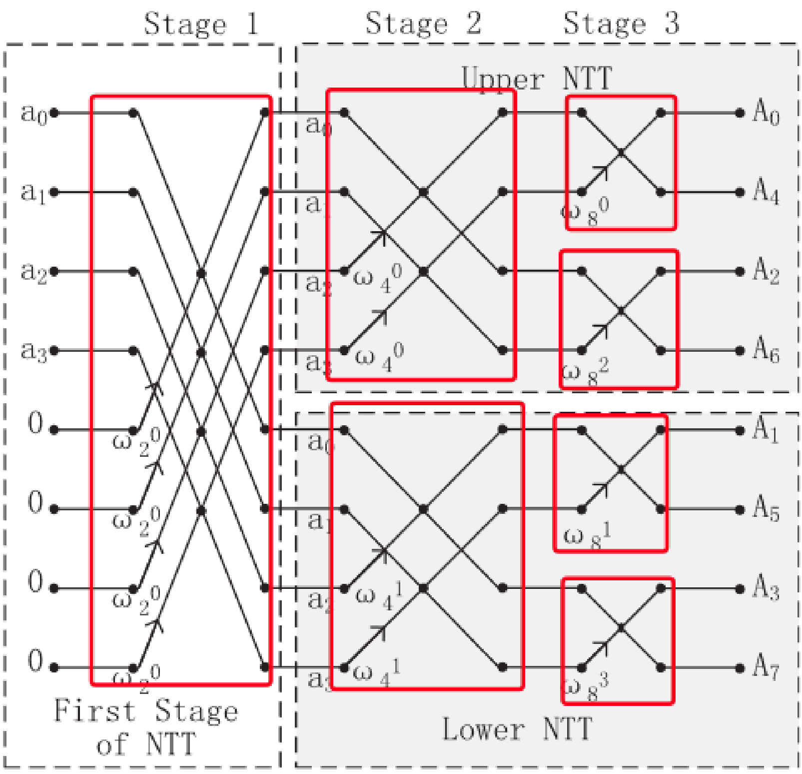
1: $\hat{f} \leftarrow f$ ▷ will compute in place on a copy of input array
2: $i \leftarrow 1$
3: **for** ($len \leftarrow 128$; $len \geq 2$; $len \leftarrow len/2$)
4:     **for** ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
5:         $zeta \leftarrow \zeta^{\mathrm{BitRev}_7(i)} \bmod q$
6:         $i \leftarrow i + 1$
7:         **for** ($j \leftarrow start$; $j < start + len$; j++)
8:             $t \leftarrow zeta \cdot \hat{f}[j + len]$ ▷ steps 8-10 done modulo q
9:             $\hat{f}[j + len] \leftarrow \hat{f}[j] - t$
10:             $\hat{f}[j] \leftarrow \hat{f}[j] + t$
11:         **end for**
12:     **end for**
13: **end for**
14: **return** $\hat{f}$

- 功能：将R_q上的多项式系数转换为T_q上的多项式系数
- 输入：长度为256的系数向量，元素大小为12bit(在q=3329时)
- 输出：长度为256的系数向量，元素大小为12bit(在q=3329时)
- 核心功能：
  - 这例算法给出的是原位NTT，即本地读写方式，不会产生额外的内存。
  - 并且核心过程采用的cooley-Tukey(CT)蝶形运算，又称为时域抽取（DIT），如下图所示。对于应8-10行。



(a) Cooley-Tukey Butterfly

- 以16点NTT为例解释伪代码



- 第一个for循环控制stage数，伪代码中需要7次
- 第二个for循环根据旋转因子进行分区，如图中红色框标记。
- 第三个for循环对块内执行时域抽取

---

2) INTT

Algorithm 10 $\mathsf{NTT}^{-1}(\hat{f})$

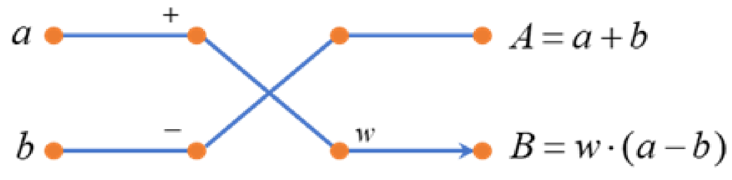
Computes the polynomial $f \in R_q$ *that corresponds to the given NTT representation* $\hat{f} \in T_q$.

Input: array $\hat{f} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of input NTT representation
Output: array $f \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the inverse NTT of the input

1: $f \leftarrow \hat{f}$ ▷ will compute in place on a copy of input array
2: $i \leftarrow 127$
3: **for** ($len \leftarrow 2$; $len \leq 128$; $len \leftarrow 2 \cdot len$)
4: **for** ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
5: $zeta \leftarrow \zeta^{\mathsf{BitRev}_7(i)} \bmod q$
6: $i \leftarrow i - 1$
7: **for** ($j \leftarrow start$; $j < start + len$; j++)
8: $t \leftarrow f[j]$
9: $f[j] \leftarrow t + f[j + len]$ ▷ steps 9-10 done modulo q
10: $f[j + len] \leftarrow zeta \cdot (f[j + len] - t)$
11: **end for**
12: **end for**
13: **end for**
14: $f \leftarrow f \cdot 3303 \bmod q$ ▷ multiply every entry by $3303 \equiv 128^{-1} \bmod q$
15: **return** f

- 功能：该过程为NTT的逆过程
- 核心功能：
  - 这例算法给出的是原位INTT，即本地读写方式，不会产生额外的内存。

- 并且核心过程采用的Gentleman-Sande(GS)蝶形运算，又称为频域抽取（DIF），如下图所示。对于应8-10行。



(b) Gentleman-Sande Butterfly

3）PWM

Algorithm 12 BaseCaseMultiply$(a_0, a_1, b_0, b_1, \gamma)$

Computes the product of two degree-one polynomials with respect to a quadratic modulus.

Input: $a_0, a_1, b_0, b_1 \in \mathbb{Z}_q$. ▷ the coefficients of $a_0 + a_1X$ and $b_0 + b_1X$
Input: $\gamma \in \mathbb{Z}_q$. ▷ the modulus is $X^2 - \gamma$
Output: $c_0, c_1 \in \mathbb{Z}_q$. ▷ the coefficients of the product of the two polynomials

```
1: c_0 ← a_0 · b_0 + a_1 · b_1 · γ          ▷ steps 1-2 done modulo q
2: c_1 ← a_0 · b_1 + a_1 · b_0
3: return (c_0, c_1)
```

- 功能:计算式(8)
- 输入:12bit的a_0, a_1, b_0, b_1，以及本源根γ
- 输出:12bit的c_0, c_1

Algorithm 11 MultiplyNTTs$(\hat{f}, \hat{g})$

Computes the product (in the ring T_q) of two NTT representations.

Input: Two arrays $\hat{f} \in \mathbb{Z}_q^{256}$ and $\hat{g} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of two NTT representations
Output: An array $\hat{h} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the product of the inputs

```
for (i ← 0; i < 128; i++)
    (ĥ[2i], ĥ[2i+1]) ← BaseCaseMultiply(f̂[2i], f̂[2i+1], ĝ[2i], ĝ[2i+1], ζ^(2BitRev_7(i)+1))
end for
return ĥ
```

- 功能：执行式（7）的逐点乘

3.K_PKE组件方案

(1)k-PKE.KeyGen密钥生成算法

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 $\mathsf{ek}_{\mathsf{PKE}} \in \mathbb{B}^{384k+32}$.
Output: decryption key $\mathsf{dk}_{\mathsf{PKE}} \in \mathbb{B}^{384k}$.

1: $(\rho, \sigma) \leftarrow \mathsf{G}(d \| k)$ ▷ expand 32+1 bytes to two pseudorandom 32-byte seeds[1]
2: $N \leftarrow 0$
3: **for** $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$
4:     **for** $(j \leftarrow 0;\ j < k;\ j{+}{+})$
5:         $\hat{\mathbf{A}}[i,j] \leftarrow \mathsf{SampleNTT}(\rho \| j \| i)$ ▷ j and i are bytes 33 and 34 of the input
6:     **end for**
7: **end for**
8: **for** $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{s} \in (\mathbb{Z}_q^{256})^k$
9:     $\mathbf{s}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(\sigma, N))$ ▷ $\mathbf{s}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
10:     $N \leftarrow N + 1$
11: **end for**
12: **for** $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{e} \in (\mathbb{Z}_q^{256})^k$
13:     $\mathbf{e}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(\sigma, N))$ ▷ $\mathbf{e}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
14:     $N \leftarrow N + 1$
15: **end for**
16: $\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$ ▷ run NTT k times (once for each coordinate of $\mathbf{s}$)
17: $\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$ ▷ run NTT k times
18: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ noisy linear system in NTT domain
19: $\mathsf{ek}_{\mathsf{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\mathbf{t}}) \| \rho$ ▷ run ByteEncode$_{12}$ k times, then append $\hat{\mathbf{A}}$-seed
20: $\mathsf{dk}_{\mathsf{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\mathbf{s}})$ ▷ run ByteEncode$_{12}$ k times
21: **return** $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}})$

- 输入：d,32字节
- 输出: 加密密钥$ek_{\mathrm{PKE}} \in B^{384k+32}$，长度为384k+32字节
- 输出: 解密密钥$dk_{\mathrm{PKE}} \in B^{384k}$，长度为384k字节
- 1:(a,b)=G(d||k)=SHA3-512(d||k)，即将32+1个字节扩展为两个32字节的随机种子，其中SHA3-512为哈希标准函数
- 2-7：重复调用sampleNTT生成多项式矩阵的NTT形式$\hat{\boldsymbol{A}} \in (Z_q^{256})^{k \times k}$，其中$\rho \parallel j \parallel i$将$\rho$，j和i拼接成34字节
- 8-15: 重复调用$\mathrm{sampleCBD}_{\eta_1}$生成多项式向量$\boldsymbol{s}, \boldsymbol{e} \in (Z_q^{256})^k$。其中PRF通过调用$\mathrm{SHAKE256}(\sigma \parallel N, 8 \cdot 64 \cdot \eta_1)$生成$64\eta_1$字节长度的种子。然后灌入$\mathrm{sampleCBD}_{\eta_1}$生成多项式系数。其中SHAKE256为kaccak-f标准函数
- 16-18: 对多项式向量$\boldsymbol{s}, \boldsymbol{e}$执行NTT变换，以及执行NTT形式的多项式点乘（需要注意的是这里元素间的乘法是逐点乘法，比如，$\sum_{j=0}^{k-1} \hat{\boldsymbol{A}}[i,j] \circ \hat{\boldsymbol{s}}[j]$,其中$\hat{\boldsymbol{A}}[i,j] \circ \hat{\boldsymbol{s}}[j]$为逐点乘法，PWM）
- 19：调用k次$\mathrm{ByteEncode}_{12}$, 将k个NTT形式的多项式系数编码并拼接，长度变换为$256 \times 12 \times k = 32 \times 12 \times k \times 8$，则$ek_{\mathrm{PKE}}$的长度为384k+32字节
- 20：同理

(2)k-PKE.Encrypt加密算法

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

1: $N \leftarrow 0$
2: $\hat{\mathbf{t}} \leftarrow$ ByteDecode$_{12}(\mathsf{ek}_{\mathsf{PKE}}[0 : 384k])$ ▷ run ByteDecode$_{12}$ k times to decode $\hat{\mathbf{t}} \in (\mathbb{Z}_q^{256})^k$
3: $\rho \leftarrow \mathsf{ek}_{\mathsf{PKE}}[384k : 384k + 32]$ ▷ extract 32-byte seed from $\mathsf{ek}_{\mathsf{PKE}}$
4: **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
5: **for** $(j \leftarrow 0;\ j < k;\ j{+}{+})$
6: $\hat{\mathbf{A}}[i, j] \leftarrow$ SampleNTT$(\rho \| j \| i)$ ▷ j and i are bytes 33 and 34 of the input
7: **end for**
8: **end for**
9: **for** $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{y} \in (\mathbb{Z}_q^{256})^k$
10: $\mathbf{y}[i] \leftarrow$ SamplePolyCBD$_{\eta_1}$(PRF$_{\eta_1}(r, N)$) ▷ $\mathbf{y}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
11: $N \leftarrow N + 1$
12: **end for**
13: **for** $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{e_1} \in (\mathbb{Z}_q^{256})^k$
14: $\mathbf{e_1}[i] \leftarrow$ SamplePolyCBD$_{\eta_2}$(PRF$_{\eta_2}(r, N)$) ▷ $\mathbf{e_1}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
15: $N \leftarrow N + 1$
16: **end for**
17: $e_2 \leftarrow$ SamplePolyCBD$_{\eta_2}$(PRF$_{\eta_2}(r, N)$) ▷ sample $e_2 \in \mathbb{Z}_q^{256}$ from CBD
18: $\hat{\mathbf{y}} \leftarrow$ NTT$(\mathbf{y})$ ▷ run NTT k times
19: $\mathbf{u} \leftarrow$ NTT$^{-1}(\hat{\mathbf{A}}^\top \circ \hat{\mathbf{y}}) + \mathbf{e_1}$ ▷ run NTT^{-1} k times
20: $\mu \leftarrow$ Decompress$_1$(ByteDecode$_1(m)$)
21: $v \leftarrow$ NTT$^{-1}(\hat{\mathbf{t}}^\top \circ \hat{\mathbf{y}}) + e_2 + \mu$ ▷ encode plaintext m into polynomial v
22: $c_1 \leftarrow$ ByteEncode$_{d_u}$(Compress$_{d_u}(\mathbf{u})$) ▷ run ByteEncode$_{d_u}$ and Compress$_{d_u}$ k times
23: $c_2 \leftarrow$ ByteEncode$_{d_v}$(Compress$_{d_v}(v)$)
24: **return** $c \leftarrow (c_1 \| c_2)$

- 功能：k-PKE.Encrypt以加密密钥ek_{PKE}、32字节明文m和随机数r作为输入，产生单个输出：密文c。
- 输入：ek_{PKE}的长度为384k+32字节，m为32字节，r为32字节
- 输出：密文c为$32(d_u k + d_v)$字节
- 2-3：k-PKE.Encrypt从加密密钥ek_{PKE}中提取向量$\hat{\boldsymbol{t}}$和ρ。
- 4-8：然后以K-PKE.Gen中相同的方式扩展种子重新生成矩阵$\hat{\boldsymbol{A}}$。
- 9-12: 重复k次调用$\mathrm{sampleCBD}_{\eta_1}$生成多项式向量$\boldsymbol{y} \in (Z_q^{256})^k$。其中PRF通过调用$\mathrm{SHAKE256}(r \parallel N, 8 \cdot 64 \cdot \eta_1)$生成$64\eta_1$字节长度的种子。然后灌入$\mathrm{sampleCBD}_{\eta_1}$生成多项式系数。其中SHAKE256为kaccak-f标准函数
- 13-16: 重复k次调用$\mathrm{sampleCBD}_{\eta_2}$生成多项式向量$\boldsymbol{e}_1 \in (Z_q^{256})^k$。其中PRF通过调用$\mathrm{SHAKE256}(r \parallel N, 8 \cdot 64 \cdot \eta_2)$生成$64\eta_2$字节长度的种子。然后灌入$\mathrm{sampleCBD}_{\eta_2}$生成多项式系数。其中SHAKE256为kaccak-f标准函数
- 17: 调用$\mathrm{sampleCBD}_{\eta_2}$生成多项式$e_2 \in Z_q^{256}$。
- 18-19:计算噪声方程$\boldsymbol{c}_1 = \mathbf{A}^T \boldsymbol{y} + \boldsymbol{e}_1$，其中$\boldsymbol{c}_1 \in (Z_q^{256})^k$
- 20：调用$\mathrm{ByteDecode}_1$将明文32字节的明文m解码成元素大小为1 bit 整数数组，长度为256。然后调用解压缩函数$\mathrm{Decompress}_1$ 即将整数数组的每个元素都扩展为12bit。这一步实质是把明文m的每一个bit编码到多项式系数中。
- 21：计算噪声方程$c_2 = \boldsymbol{t}^T \boldsymbol{y} + \boldsymbol{e}_2 + m$，其中$c_2 \in Z_q^{256}$
- 22: 分别调用k次$\mathrm{ByteEncode}_{\mathrm{d_u}}$和$\mathrm{compress}_{\mathrm{d_u}}$,其中$\boldsymbol{u} \in (Z_q^{256})^k$。对于多项式$\boldsymbol{u}[i] \in Z_q^{256}$，即将其每一个系数都进行压缩，将q bit转换为$d_u$ bit大小。然后对$temp = \mathrm{compress}_{\mathrm{d_u}}(\mathbf{u}[i]) \in \mathrm{Z}_{\mathrm{q}}^{256}$执行编码，得到字节数组$tempB \in Z_{d_u}^{256}$。整个过程的长度转换为：$256 \times 12 \times k\ bit \Rightarrow 256 \times d_u \times k\ bit = 32 \times d_u \times k\ Byte$。返回$32 d_u k$的字节数组$c_1$
- 23:同理，返回$32 d_v$的字节数组c_2
- 24:最后将c_1, c_2拼接后输出c

(3)k-PKE.Decrypt解密算法

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

Uses the decryption key to decrypt a ciphertext.

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

1: $c_1 \leftarrow c[0 : 32 d_u k]$
2: $c_2 \leftarrow c[32 d_u k : 32(d_u k + d_v)]$
3: $\mathbf{u}' \leftarrow \text{Decompress}_{d_u}(\text{ByteDecode}_{d_u}(c_1))$ ▷ run Decompress_{d_u} and ByteDecode_{d_u} k times
4: $v' \leftarrow \text{Decompress}_{d_v}(\text{ByteDecode}_{d_v}(c_2))$
5: $\hat{\mathbf{s}} \leftarrow \text{ByteDecode}_{12}(\text{dk}_{\text{PKE}})$ ▷ run ByteDecode_{12} k times
6: $w \leftarrow v' - \text{NTT}^{-1}(\hat{\mathbf{s}}^{\top} \circ \text{NTT}(\mathbf{u}'))$ ▷ run NTT k times; run NTT^{-1} once
7: $m \leftarrow \text{ByteEncode}_1(\text{Compress}_1(w))$ ▷ decode plaintext m from polynomial v
8: **return** m

- 功能:k-PKE.Decrypt以解密密钥dk_{PKE}、密文c作为输入，产生单个输出：明文m。
- 输入：解密密钥dk_{PKE}长度为384k字节的字节数组，$32(d_u k + d_v k)$字节的密文
- 输出：32字节的明文m
- 1-2:分离c_1, c_2,注意着$B[k:m] = (B[k], B[k+1], \ldots, B[m-1])$
- 3:c_1经过$\text{ByteDecode}_{\text{d}_\text{u}}$和$\text{Decompress}_{\text{d}_\text{u}}$的长度变换为，
  $32 \times d_u \times k\ Byte = 256 \times d_u \times k\ bit \Rightarrow 256 \times 12 \times k\ bit$
- 4:c_2经过$\text{ByteDecode}_{\text{d}_\text{v}}$和$\text{Decompress}_{\text{d}_\text{v}}$的长度变换为，$32 \times d_v\ Byte = 256 \times d_v\ bit \Rightarrow 256 \times 12\ bit$
- 5:将dk_{PKE}解码，长度变换为，$384k\ Byte = 256 \times 12 \times k\ bit \Rightarrow 256 \times 12\ bit$
- 6:核心过程，即计算
  $m = c_2 - \boldsymbol{s}^T \boldsymbol{c}_1 = \boldsymbol{t}^T \boldsymbol{y} + \boldsymbol{e}_2 + m - \boldsymbol{s}^T(\mathbf{A}^T \boldsymbol{y} + \boldsymbol{e}_1) = (\boldsymbol{A}\boldsymbol{s} + \boldsymbol{e})^T \boldsymbol{y} + \boldsymbol{e}_2 + m - \boldsymbol{s}^T(\mathbf{A}^T \boldsymbol{y} + \boldsymbol{e}_1) = m + \boldsymbol{e}^T \boldsymbol{y} + e_2 - \boldsymbol{s}^T \boldsymbol{e}_1$
  ，该代码行中$w \in Z_q^{256}$
- 7:对多项式w的每个系数执行压缩，长度变换$256 \times 12\ bit \Rightarrow 256\ bit$,最后编码成32字节的明文m.

4.内部算法ML_KEM_internal

- 本节指定了三种算法：
  - ML-KEM.KeyGen_internal
  - ML-KEM.Encaps_internal
  - ML-KEM.Decaps_internal
- 这三种算法都是确定性的，这意味着它们的输出完全由它们的输入决定。在这些算法中没有随机抽样。这三种算法将用于构造ML-KEM。本节中的算法使用了参数n, q, k, d_u, d_v。
  它们调用的子程序还使用了参数η_1, η_2。当n=256，q=3329时，其余参数在可能的参数集之间变化。本节中指定的接口将用于通过加密算法验证程序（CAVP）测试ML-KEM实现。本节中的密钥生成函数也可用于从种子重新扩展密钥，本节中指定的接口**不应提供给除测试目的**以外的应用程序，以及随机种子（如ML-KEM中指定的）。

(1)ML-KEM.KeyGen_internal

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: $\mathsf{dk} \leftarrow (\mathsf{dk}_{\mathsf{PKE}} \| \mathsf{ek} \| \mathsf{H}(\mathsf{ek}) \| z)$ ▷ KEM decaps key includes PKE decryption key
4: **return** $(\mathsf{ek}, \mathsf{dk})$

- 输入：随机种子d、z,长度为32的字节数组
- 输出：封装密钥ek为384k+32长度的字节数组，解封装密钥dk为768k+96长度的字节数组
- 1:调用k-PKE.KeyGen(d),生成长度为384k+32字节的加密密钥 ek_{PKE}和长度为384k字节解密密钥$dk_{\mathrm{PKE}} \in B^{384k}$
- 2：封装密钥ek就是加密密钥
- 3：首先计算封装密钥的哈希值，H(ek)=SHA3-256(ek)，H(ek)输出为32字节，SHA3-256为哈希标准函数。然后将dk_{PKE}，ek，$H(ek), z$拼接，所以长度为：$384k + 32 + 384k + 32 + 32 = 768k + 96$

(2)ML-KEM.Encaps_internal

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

- 输入：封装密钥ek,以及32字节随机数m
- 输出：32字节共享密钥K，$32(d_u k + d_v)$字节密文c
- 1：m||H(ek)为33字节，(K,r)=G(m||H(ek))=SHA3-512(m||H(ek)),即将32+1个字节扩展为两个32字节的随机种子,其中SHA3-512为哈希标准函数
- 2:调用k-PKE.Encrypt

(3)ML-KEM.Decaps_internal

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 $K \in \mathbb{B}^{32}$.

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

- 输入：768k+96字节的解封装密钥dk,以及$32(d_u k + d_v)$字节密文c
- 输出：32字节共享密钥K，
- 1-4：从dk中恢复将dk_{PKE}，ek，$H(ek), z$
- 5: 对c进行解密得m'
- 6-7:计算检查失败时用于隐式拒绝的共享密钥$\overline{K}$
- 8:对m'再次加密得c'
- 9-12:执行检查并返回共享密钥

5.ML-KEM密钥封装机制

(1)ML-KEM参数说明

- ML-KEM包含以下3个子算法：
  - ML-KEM.KeyGen()
  - ML-KEM.Encaps()
  - ML-KEM.Decaps()
- ML-KEM有3组不同的参数取值，k, η_1, η_2，d_u, d_v，n=256和q=3329
  - k决定k-PKE中的多项式矩阵$\hat{\boldsymbol{A}}$维数，多项式向量$\boldsymbol{s}, \boldsymbol{e}, \boldsymbol{y}, \boldsymbol{e}_1$的维数
  - η_1决定多项式向量$\boldsymbol{s}, \boldsymbol{e}, \boldsymbol{y}$的分布
  - η_2决定多项式向量$\boldsymbol{e}_1$和多项式e_2的分布
  - d_u, d_v决定压缩compress，解压缩Decompress,编码ByteEncode和解码ByteDecode的压缩尺寸

- 标准参数如下表2所示，表3相应给出了每个参数集的ML-KEM密钥和密文的大小

Table 2. Approved parameter sets for ML-KEM

	n	q	k	η_1	η_2	d_u	d_v	required RBG strength (bits)
ML-KEM-512	256	3329	2	3	2	10	4	128
ML-KEM-768	256	3329	3	2	2	10	4	192
ML-KEM-1024	256	3329	4	2	2	11	5	256

Table 3. Sizes (in bytes) of keys and ciphertexts of ML-KEM

	encapsulation key	decapsulation key	ciphertext	shared secret key
ML-KEM-512	800	1632	768	32
ML-KEM-768	1184	2400	1088	32
ML-KEM-1024	1568	3168	1568	32

(2)ML-KEM.KeyGen

Algorithm 19 ML-KEM.KeyGen()

Generates an encapsulation key and a corresponding decapsulation key.

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

1: $d \xleftarrow{\$} \mathbb{B}^{32}$ ▷ d is 32 random bytes (see Section 3.3)
2: $z \xleftarrow{\$} \mathbb{B}^{32}$ ▷ z is 32 random bytes (see Section 3.3)
3: **if** $d ==$ NULL **or** $z ==$ NULL **then**
4:     **return** $\perp$ ▷ return an error indication if random bit generation failed
5: **end if**
6: $(\text{ek}, \text{dk}) \leftarrow$ ML-KEM.KeyGen_internal(d, z) ▷ run internal key generation algorithm
7: **return** (ek, dk)

- ML-KEM不接受任何输入；
- 输出：封装密钥ek和解封装密钥dk
- 1-2:生成随机种子d,z
- 3-5:执行随机生成检查
- 6：调用内部ML-KEM.KeyGen_internal
- 如果用户不是自己生成的密钥，而是从第3方收到一对密钥，用户可以进行如下检查，确保密钥对的合法性
  - 种子一致性检查：如果有种子(d,z)，执行ML-KEM.KeyGen_internal，验证输出是否与收到的密钥对一致
  - 封装密钥检查：见ML-KEM.Encaps()
  - 解封装密钥检查：见ML-KEM.Decaps()
  - 密钥对一致性：随机生成一个消息m;利用ML-KEM.Encaps_internal生成（K,c）;利用ML-KEM.Decaps_internal生成K';检查K'和K是否一致。不一致，则拒绝

(3)ML-KEM.Encaps

Algorithm 20 ML-KEM.Encaps(ek)

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

Checked input: encapsulation key 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)
```

- 在运行上述算法前需要先检查封装密钥ek的合法性
    - 类型检查：如果ek不是一个长度为384k+32的字节数组，则检查失败
    - 模数检查：计算$test = ByteEncode_{12}(ByteDecode_{12}(ek[0:384k]))$,如果test==ek[0:384k],则通过，反之则失败
- 输入：384k字节的封装密钥ek
- 输出：共享密钥K和密文c

---

(4)ML-KEM.Decaps

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

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

Checked input: decapsulation key 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'
```

- 在运行上述算法前需要先检查解封装密钥(dk,c)的合法性
    - 密钥类型检查：如果c不是一个长度为$32(d_u k + d_v)$的字节数组，则检查失败
    - 解封装密钥类型检查：如果dk不是一个长度为768k+96的字节数组，则检查失败
    - 哈希检查：计算test=H(dk[384k:768k+32]),如果test！=dk[768k+32:768k+64],则检查失败
- 输入：768k+96字节的解封装密钥dk和密文c
- 输出：共享密钥K