CRYSTALS-Dilithium 算法梳理

CRYSTALS-Dilithium 算法梳理

'''
@Descripttion: CRYSTALS-Dilithium 算法梳理
@version: V1.0
@Author: HZW
@Date: 2025-03-10 16:51:00
'''

简介

- ref:doc/NIST.FIPS.204.pdf
- 本文档旨在梳理ML_DSA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2.ML_DSA内部组件;
- 如果只关注算法流程，可以跳过1.5.快速数论变换（NTT）的原理推导部分;
- 在阅读2.ML_DSA内部组件时，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

1.1 数据类型转换

(1)IntegerToBits(x,a)

将整数转化为 α 长度的二进制串

Algorithm 9 IntegerToBits(x, α)

Computes a base-2 representation of x mod 2^α using little-endian order.

Input: A nonnegative integer x and a positive integer α.
Output: A bit string y of length α.

```
x' ← x
for i from 0 to α − 1 do
    y[i] ← x' mod 2
    x' ← ⌊x'/2⌋
end for
return y
```

- 功能：计算整数$y = x \bmod 2^\alpha$,并转为bit数组
- y这里为小端存储，比如 $11010001 = 2^0 + 2^1 + 2^3 + 2^7 = 139$
- 硬件实现时直接截取低α位

---

(2)BitsToInteger(y,a)

长度为 α 的二进制串转化为整数

Algorithm 10 BitsToInteger(y, α)

Computes the integer value expressed by a bit string using little-endian order.

Input: A positive integer α and a bit string y of length α.
Output: A nonnegative integer x.

```
x ← 0
for i from 1 to α do
    x ← 2x + y[α − i]
end for
return x
```

- 功能：将长度为αbit的比特数组y转换为整数x
- y这里为小端存储，比如 $11010001 = 2^0 + 2^1 + 2^3 + 2^7 = 139$

(3)IntegerToBytes(x,a)

整数转化为字节数组。一个字节是8bit。即2^8=256

Algorithm 11 IntegerToBytes(x, α)

Computes a base-256 representation of $x \bmod 256^\alpha$ using little-endian order.

Input: A nonnegative integer x and a positive integer α.
Output: A byte string y of length α.

```
x′ ← x
for i from 0 to α − 1 do
    y[i] ← x′ mod 256
    x′ ← ⌊x′/256⌋
end for
return y
```

- 功能：计算整数$y = x \bmod 256^\alpha$,并转为字节数组
- y这里为小端存储，以字节为单位的小端序

bit转为字节数组，长度大为缩短。字节数组采用十进制表示，或者十六进制表示。

(4)BitsToBytes(y)

Algorithm 12 BitsToBytes(y)

Converts a bit string into a byte string using little-endian order.

Input: A bit string y of length α.
Output: A byte string z of length $\lceil \alpha/8 \rceil$.

```
z ∈ B^⌈α/8⌉ ← 0^⌈α/8⌉
for i from 0 to α − 1 do
    z[⌊i/8⌋] ← z[⌊i/8⌋] + y[i] · 2^(i mod 8)
end for
return z
```

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

(5)BytesToBits(Z)

Algorithm 13 BytesToBits(z)

Converts a byte string into a bit string using little-endian order.

Input: A byte string z of length α.
Output: A bit string y of length 8α.

$z' \leftarrow z$
for i **from** 0 **to** $\alpha - 1$ **do**
  for j **from** 0 **to** 7 **do** ▷ convert the byte $z[i]$ into 8 bits
    $y[8i+j] \leftarrow z'[i] \bmod 2$
    $z'[i] \leftarrow \lfloor z'[i]/2 \rfloor$
  end for
end for
return y

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

(6)CoeffFromThreeBytes(b0,b1,b2)

取三个字节，转化为模q的整数。理论最大2^24。实际最大为q

Algorithm 14 CoeffFromThreeBytes(b_0, b_1, b_2)

Generates an element of $\{0, 1, 2, \ldots, q-1\} \cup \{\perp\}$.

Input: Bytes b_0, b_1, b_2.
Output: An integer modulo q or $\perp$.

$b_2' \leftarrow b_2$
if $b_2' > 127$ **then**
  $b_2' \leftarrow b_2' - 128$ ▷ set the top bit of b_2' to zero
end if
$z \leftarrow 2^{16} \cdot b_2' + 2^8 \cdot b_1 + b_0$ ▷ $0 \le z \le 2^{23} - 1$
if $z < q$ **then return** z ▷ rejection sampling
else return $\perp$
end if

- 功能:将3个字节转换模q的整数，用于生成多项式系数（拒绝采样）

(7)CoeffFromHalfBytes(b)

将b进行指定范围内的归一化

拒绝采样

Algorithm 15 CoeffFromHalfByte(b)

Let $\eta \in \{2, 4\}$. *Generates an element of* $\{-\eta, -\eta+1, \ldots, \eta\} \cup \{\perp\}$.

Input: Integer $b \in \{0, 1, \ldots, 15\}$.
Output: An integer between $-\eta$ and η, or $\perp$.

if $\eta = 2$ **and** $b < 15$ **then return** $2 - (b \bmod 5)$ ▷ rejection sampling from $\{-2, \ldots, 2\}$
else
  if $\eta = 4$ **and** $b < 9$ **then return** $4 - b$ ▷ rejection sampling from $\{-4, \ldots, 4\}$
  else return $\perp$
  end if
end if

- 功能:将元素b转换到$[-\eta, \eta]$,用于生成多项式系数(拒绝采样)

将多项式系数数组编码为字节为单位的数组

(8)SimpleBitPack(w)

Algorithm 16 SimpleBitPack(w, b)

Encodes a polynomial w into a byte string.

Input: $b \in \mathbb{N}$ and $w \in R$ such that the coefficients of w are all in $[0, b]$.
Output: A byte string of length $32 \cdot \text{bitlen } b$.

$z \leftarrow ()$ ▷ set z to the empty bit string
for i **from** 0 **to** 255 **do**
  $z \leftarrow z \| \text{IntegerToBits}(w_i, \text{bitlen } b)$
end for
return BitsToBytes(z)

设d=bitlen(b)，则代表以精度d量化多项式系数数组。

- 功能:将多项式系数数组w编码为字节数组
- 输入：w为多项式系数数组
- 输出：为字节数组
- 设d=bitlen(b),其中d为w中元素w[i]的位宽，长度变换过程为：$256 \cdot d = 32 \cdot d \cdot 8$
- 同kyber中的编码函数,ByteEncode

b表示多项式的系数最大为b。（正整数范围）

(9)BitPack(w)

Algorithm 17 BitPack(w, a, b)

引入负的系数。
允许多项式系数存在负数，更加科学了。

Encodes a polynomial w into a byte string.

Input: $a, b \in \mathbb{N}$ and $w \in R$ such that the coefficients of w are all in $[-a, b]$.
Output: A byte string of length $32 \cdot \text{bitlen } (a + b)$.

$z \leftarrow ()$ ▷ set z to the empty bit string
for i **from** 0 **to** 255 **do**
  $z \leftarrow z \| \text{IntegerToBits}(b - w_i, \text{bitlen } (a + b))$
end for
return BitsToBytes(z)

- 功能:将多项式系数数组w编码为字节数组，设d=bitlen(a+b),其中d为w中元素w[i]的位宽，长度变换过程为：$256 \cdot d = 32 \cdot d \cdot 8$

(10)SimpleBitUnPack(v,b)

Algorithm 18 SimpleBitUnpack(v, b)

Reverses the procedure SimpleBitPack.

Input: $b \in \mathbb{N}$ and a byte string v of length $32 \cdot \text{bitlen } b$.
Output: A polynomial $w \in R$ with coefficients in $[0, 2^c - 1]$, where $c = \text{bitlen } b$.
When $b + 1$ is a power of 2, the coefficients are in $[0, b]$.

$c \leftarrow \text{bitlen } b$
$z \leftarrow \text{BytesToBits}(v)$
for i **from** 0 **to** 255 **do**
  $w_i \leftarrow \text{BitsToInteger}((z[ic], z[ic + 1], \ldots z[ic + c - 1]), c)$
end for
return w

- 功能:SimpleBitPack的逆过程
- 同kyber中的解码函数,ByteDecode

(11)BitUnPack(v,b)

Algorithm 19 BitUnpack(v, a, b)

Reverses the procedure BitPack*.*

Input: $a, b \in \mathbb{N}$ and a byte string v of length $32 \cdot \text{bitlen}\ (a+b)$.
Output: A polynomial $w \in R$ with coefficients in $[b - 2^c + 1, b]$, where $c = \text{bitlen}\ (a+b)$.
When $a + b + 1$ is a power of 2, the coefficients are in $[-a, b]$.

$c \leftarrow \text{bitlen}\ (a+b)$
$z \leftarrow$ BytesToBits(v)
for i **from** 0 **to** 255 **do**
  $w_i \leftarrow b -$ BitsToInteger$((z[ic], z[ic+1], \ldots z[ic+c-1]), c)$
end for
return w

这个算法特别适用于多项式向量中非零系数数量较少的情况，即稀疏多项式向量。
只记录非零系数的位置和值，大大减少了存储空间的需求

- 功能:BitPack的逆过程

类似bitpack，不过这次的多项式系数是直接二进制表示的，需要转化为字节数组

(12)HintBitpack(h)

Algorithm 20 HintBitPack$(\mathbf{h})$

Encodes a polynomial vector $\mathbf{h}$ *with binary coefficients into a byte string.*

Input: A polynomial vector $\mathbf{h} \in R_2^k$ such that the polynomials $\mathbf{h}[0], \mathbf{h}[1], \ldots, \mathbf{h}[k-1]$ have collectively at most ω nonzero coefficients.
Output: A byte string y of length $\omega + k$ that encodes $\mathbf{h}$ as described above.

$y \in \mathbb{B}^{\omega+k} \leftarrow 0^{\omega+k}$
Index $\leftarrow 0$ ▷ Index for writing the first ω bytes of y
for i **from** 0 **to** $k-1$ **do** ▷ look at $\mathbf{h}[i]$
  for j **from** 0 **to** 255 **do**
    if $\mathbf{h}[i]_j \neq 0$ **then**
      $y[\text{Index}] \leftarrow j$ ▷ store the locations of the nonzero coefficients in $\mathbf{h}[i]$
      Index $\leftarrow$ Index $+ 1$
    end if
  end for
  $y[\omega + i] \leftarrow$ Index ▷ after processing $\mathbf{h}[i]$, store the value of Index
end for
return y

- 功能:将一个具有二进制系数的多项式向量编码为字节数组
- 输入：长度为k的多项式向量系数数组$(h[0], \ldots, h[k-1])$,其中h[i]为长度为256的多项式系数数组。其中多项式向量系数数组至多有w个非零系数。
- 输出：长度为w+k的字节数组y,前w的元素记录非零位置，后k个字节用于记录每个h[i]中多少个非零bit

(13)HintBitUnPack(y)

Algorithm 21 HintBitUnpack(y)

Reverses the procedure HintBitPack.

Input: A byte string y of length $\omega + k$ that encodes $\mathbf{h}$ as described above.
Output: A polynomial vector $\mathbf{h} \in R_2^k$ or $\perp$.

```
h ∈ R_2^k ← 0^k
Index ← 0                                          ▷ Index for reading the first ω bytes of y
for i from 0 to k − 1 do                           ▷ reconstruct h[i]
    if y[ω + i] < Index or y[ω + i] > ω then return ⊥     ▷ malformed input
    end if
    First ← Index
    while Index < y[ω + i] do                      ▷ y[ω + i] says how far one can advance Index
        if Index > First then
            if y[Index − 1] ≥ y[Index] then return ⊥      ▷ malformed input
            end if
        end if
        h[i]_{y[Index]} ← 1                        ▷ y[Index] says which coefficient in h[i] should be 1
        Index ← Index + 1
    end while
end for
for i from Index to ω − 1 do                       ▷ read any leftover bytes in the first ω bytes of y
    if y[i] ≠ 0 then return ⊥                      ▷ malformed input
    end if
end for
return h
```

- 功能:HintBitpack的逆过程
- 输入：长度为k的多项式向量系数数组$(h[0], \ldots, h[k-1])$,其中h[i]为长度为256的多项式系数数组。其中多项式向量系数数组至多有w个非零系数。
- 输出：长度为w+k的字节数组y,前w的元素记录非零位置，后k个字节用于记录每个h[i]中多少个非零bit

1.2 ML_DSA密钥和签名的编码

t1：多项式向量，共有k维度

(14)pkEncode

Algorithm 22 pkEncode($\rho, \mathbf{t}_1$)

Encodes a public key for ML-DSA into a byte string.

Input:$\rho \in \mathbb{B}^{32}$, $\mathbf{t}_1 \in R^k$ with coefficients in $[0, 2^{\text{bitlen}\,(q-1)-d} - 1]$.
Output: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}\,(q-1)-d)}$.

```
pk ← ρ
for i from 0 to k − 1 do
    pk ← pk || SimpleBitPack (t_1[i], 2^{bitlen (q−1)−d} − 1)
end for
return pk
```

- 功能:实际就是编码函数的扩展

将多项式向量和ρ都编码为字节串，作为公钥

(15)pkDecode

Algorithm 23 pkDecode(pk)

Reverses the procedure pkEncode.

Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$.
Output: $\rho \in \mathbb{B}^{32}, \mathbf{t}_1 \in R^k$ with coefficients in $[0, 2^{\text{bitlen }(q-1)-d} - 1]$.

1: $(\rho, z_0, \ldots, z_{k-1}) \in \mathbb{B}^{32} \times \left(\mathbb{B}^{32(\text{bitlen }(q-1)-d)}\right)^k \leftarrow pk$
2: **for** i **from** 0 **to** $k-1$ **do**
3: $\quad \mathbf{t}_1[i] \leftarrow$ SimpleBitUnpack$(z_i, 2^{\text{bitlen }(q-1)-d} - 1)$ $\quad \triangleright$ This is always in the correct range
4: **end for**
5: **return** $(\rho, \mathbf{t}_1)$

- 功能:实际就是解码的扩展

将这一大堆东西编码相连。得到字节串表示的私钥

(16)skEncode

Algorithm 24 skEncode($\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0$)

Encodes a secret key for ML-DSA into a byte string.

Input: $\rho \in \mathbb{B}^{32}$, $K \in \mathbb{B}^{32}$, $tr \in \mathbb{B}^{64}$, $\mathbf{s}_1 \in R^\ell$ with coefficients in $[-\eta, \eta]$, $\mathbf{s}_2 \in R^k$ with coefficients in $[-\eta, \eta]$, $\mathbf{t}_0 \in R^k$ with coefficients in $[-2^{d-1}+1, 2^{d-1}]$.
Output: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((k+\ell)\cdot\text{bitlen }(2\eta)+dk)}$.

1: $sk \leftarrow \rho || K || tr$
2: **for** i **from** 0 **to** $\ell-1$ **do**
3: $\quad sk \leftarrow sk$ || BitPack $(\mathbf{s}_1[i], \eta, \eta)$
4: **end for**
5: **for** i **from** 0 **to** $k-1$ **do**
6: $\quad sk \leftarrow sk$ || BitPack $(\mathbf{s}_2[i], \eta, \eta)$
7: **end for**
8: **for** i **from** 0 **to** $k-1$ **do**
9: $\quad sk \leftarrow sk$ || BitPack $(\mathbf{t}_0[i], 2^{d-1}-1, 2^{d-1})$
10: **end for**
11: **return** sk

显然可以进行硬件的并行优化

- 功能:实际就是编码函数的扩展

(17)skDecode

Algorithm 25 skDecode(sk)

Reverses the procedure skEncode.

Input: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen}\,(2\eta)+dk)}$.
Output: $\rho \in \mathbb{B}^{32}$, $K \in \mathbb{B}^{32}$, $tr \in \mathbb{B}^{64}$,
$\mathbf{s}_1 \in R^{\ell}$, $\mathbf{s}_2 \in R^{k}$, $\mathbf{t}_0 \in R^{k}$ with coefficients in $[-2^{d-1}+1, 2^{d-1}]$.

$(\rho, K, tr, y_0, \ldots, y_{\ell-1}, z_0, \ldots, z_{k-1}, w_0, \ldots, w_{k-1}) \in \mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{64} \times \left(\mathbb{B}^{32\cdot\text{bitlen}\,(2\eta)}\right)^{\ell} \times \left(\mathbb{B}^{32\cdot\text{bitlen}\,(2\eta)}\right)^{k} \times \left(\mathbb{B}^{32d}\right)^{k} \leftarrow sk$
for i **from** 0 **to** $\ell-1$ **do**
  $\mathbf{s}_1[i] \leftarrow$ BitUnpack(y_i, η, η) ▷ this may lie outside $[-\eta, \eta]$ if input is malformed
end for
for i **from** 0 **to** $k-1$ **do**
  $\mathbf{s}_2[i] \leftarrow$ BitUnpack(z_i, η, η) ▷ this may lie outside $[-\eta, \eta]$ if input is malformed
end for
for i **from** 0 **to** $k-1$ **do**
  $\mathbf{t}_0[i] \leftarrow$ BitUnpack$(w_i, 2^{d-1}-1, 2^{d-1})$ ▷ this is always in the correct range
end for
return $(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

- 功能:实际就是解码的扩展

签名编码转化为字节串

(18)sigEncode

Algorithm 26 sigEncode$(\tilde{c}, \mathbf{z}, \mathbf{h})$

Encodes a signature into a byte string.

Input: $\tilde{c} \in \mathbb{B}^{\lambda/4}$, $\mathbf{z} \in R^{\ell}$ with coefficients in $[-\gamma_1+1, \gamma_1]$, $\mathbf{h} \in R_2^k$.
Output: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}\,(\gamma_1-1))+\omega+k}$.

$\sigma \leftarrow \tilde{c}$
for i **from** 0 **to** $\ell-1$ **do**
  $\sigma \leftarrow \sigma \,||\,$ BitPack $(\mathbf{z}[i], \gamma_1-1, \gamma_1)$
end for
$\sigma \leftarrow \sigma \,||\,$ HintBitPack $(\mathbf{h})$
return σ

- 功能:实际就是编码的扩展

(19)sigDecode

Algorithm 27 sigDecode(σ)

Reverses the procedure sigEncode.

Input: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}\,(\gamma_1-1))+\omega+k}$.
Output: $\tilde{c} \in \mathbb{B}^{\lambda/4}$, $\mathbf{z} \in R^{\ell}$ with coefficients in $[-\gamma_1+1, \gamma_1]$, $\mathbf{h} \in R_2^k$, or $\perp$.

$(\tilde{c}, x_0, \ldots, x_{\ell-1}, y) \in \mathbb{B}^{\lambda/4} \times \left(\mathbb{B}^{32\cdot(1+\text{bitlen}\,(\gamma_1-1))}\right)^{\ell} \times \mathbb{B}^{\omega+k} \leftarrow \sigma$
for i **from** 0 **to** $\ell-1$ **do**
  $\mathbf{z}[i] \leftarrow$ BitUnpack$(x_i, \gamma_1-1, \gamma_1)$ ▷ this is in the correct range, as γ_1 is a power of 2
end for
$\mathbf{h} \leftarrow$ HintBitUnpack(y)
return $(\tilde{c}, \mathbf{z}, \mathbf{h})$

- 功能:实际就是解码函数的扩展

多项式向量编码为字节串

(20)w1Encode

Algorithm 28 w1Encode($\mathbf{w}_1$)

Encodes a polynomial vector $\mathbf{w}_1$ into a byte string.

Input: $\mathbf{w}_1 \in R^k$ whose polynomial coordinates have coefficients in $[0, (q-1)/(2\gamma_2) - 1]$.
Output: A byte string representation $\tilde{\mathbf{w}}_1 \in \mathbb{B}^{32k \cdot \mathrm{bitlen}\,((q-1)/(2\gamma_2)-1)}$.

$\tilde{\mathbf{w}}_1 \leftarrow ()$
for i **from** 0 **to** $k-1$ **do**
  $\tilde{\mathbf{w}}_1 \leftarrow \tilde{\mathbf{w}}_1 \,||\,$ SimpleBitPack $(\mathbf{w}_1[i], (q-1)/(2\gamma_2) - 1)$
end for
return $\tilde{\mathbf{w}}_1$

- 功能:实际就是编码的扩展

1.3 伪随机采样

又称为碰撞强度

(21)SampleInBall

Algorithm 29 SampleInBall(ρ)

Samples a polynomial $c \in R$ with coefficients from $\{-1, 0, 1\}$ and Hamming weight $\tau \leq 64$.

Input: A seed $\rho \in \mathbb{B}^{\lambda/4}$
Output: A polynomial c in R.

哈希函数自身生成上下文ctx

从哈希函数中提取8个字节，并且转化为比特串h

$c \leftarrow 0$
ctx $\leftarrow$ H.Init()
ctx $\leftarrow$ H.Absorb(ctx, ρ)
(ctx, s) $\leftarrow$ H.Squeeze(ctx, 8)
$h \leftarrow$ BytesToBits(s) ▷ h is a bit string of length 64
for i **from** $256 - \tau$ **to** 255 **do**
  (ctx, j) $\leftarrow$ H.Squeeze(ctx, 1)
  while $j > i$ **do** ▷ rejection sampling in $\{0, \ldots, i\}$
    (ctx, j) $\leftarrow$ H.Squeeze(ctx, 1)
  end while ▷ j is a pseudorandom byte that is $\leq i$
  $c_i \leftarrow c_j$
  $c_j \leftarrow (-1)^{h[i+\tau-256]}$
end for
return c

从哈希函数提取1个字节，存储在j中。当j>i时，一直提取，直到<=。这是拒绝采样过程，确保j在0~i中均匀分布。

- 功能：通过H函数从一个长度为$\lambda/4$的字节数组种子中扩展，然后通过拒绝采样生成一个长度为256的多项式，其中多项式的系数属于{-1,0,1}

拒绝采样技术是一种在密码学中常用的方法，它用于从一个源概率分布中采样以获得目标概率分布的样本。这种技术由 Neumann 提出，并且已经在多个密码学应用中得到使用，包括基于格的身份识别协议和数字签名算法。拒绝采样的关键在于通过拒绝采样降低参数的规模，从而优化系统参数，减少方案公钥和密文规模，有效改善密文运算的计算复杂性，提升同态加密方案的性能。

在拒绝采样过程中，首先从易于采样的提议分布中生成样本，然后根据一定的接受概率来决定是否接受这些样本。具体来说，拒绝采样的基本思想是利用一个辅助分布来生成样本，并按照一定的概率接受或拒绝样本。这个辅助分布通常是一个简单的分布，比如均匀分布或高斯分布。在拒绝采样中，接受样本的概率与目标分布的概率成比例。如果生成的样本符合目标分布，那么就接受这个样本；否则就拒绝这个样本，并重新生成一个新的样本。这种方法可以用于生成符合目标分布的样本，增加训练数据的多样性，从而提高模型的泛化能力。

拒绝采样技术在密码学中的应用包括构造基于格的 BLISS数字签名算法等，NIST 征集的多数后量子数字签名候选算法也采用了拒绝采样技术。此外，拒绝采样引理在密码学中是一种重要的技术，用于证明密码学方案的安全性，它通常用于证明一个随机变量的分布在某些条件下满足特定的安全性要求。

由G函数生成的哈希上下文中提取出属于环Tq的多项式系数。
是NTT的

(22)RejNTTPoly

Algorithm 30 RejNTTPoly(ρ)

Samples a polynomial $\in T_q$.

Input: A seed $\rho \in \mathbb{B}^{34}$.
Output: An element $\hat{a} \in T_q$.

```
j ← 0
ctx ← G.Init()
ctx ← G.Absorb(ctx, ρ)
while j < 256 do
    (ctx, s) ← G.Squeeze(ctx, 3)
    â[j] ← CoeffFromThreeBytes(s[0], s[1], s[2])
    if â[j] ≠ ⊥ then
        j ← j + 1
    end if
end while
return â
```

- 功能:通过G函数从一个长度为34的字节数组种子中扩展，生成多项式的NTT形式系数表示

(23)RejBoundedPoly

Algorithm 31 RejBoundedPoly(ρ)

Samples an element $a \in R$ *with coefficients in* $[-\eta, \eta]$ *computed via rejection sampling from* ρ.

Input: A seed $\rho \in \mathbb{B}^{66}$.
Output: A polynomial $a \in R$.

```
j ← 0
ctx ← H.Init()
ctx ← H.Absorb(ctx, ρ)
while j < 256 do
    z ← H.Squeeze(ctx, 1)
    z0 ← CoeffFromHalfByte(z mod 16)
    z1 ← CoeffFromHalfByte(⌊z/16⌋)
    if z0 ≠ ⊥ then
        a_j ← z0
        j ← j + 1
    end if
    if z1 ≠ ⊥ and j < 256 then
        a_j ← z1
        j ← j + 1
    end if
end while
return a
```

通过拒绝采样生成多项式。范围是-yita~+yita

- 功能:通过H函数从一个长度为64的字节数组种子中扩展。然后通过拒绝采样生成多项式

对种子不断更新，重复调用RejNTTPoly生成多项式矩阵A_hat。
种子的更新和矩阵的索引相关。

(24)ExpandA

Algorithm 32 ExpandA(ρ)

Samples a $k \times \ell$ *matrix* $\hat{\mathbf{A}}$ *of elements of* T_q.

Input: A seed $\rho \in \mathbb{B}^{32}$.
Output: Matrix $\hat{\mathbf{A}} \in (T_q)^{k \times \ell}$.

```
for r from 0 to k − 1 do
    for s from 0 to ℓ − 1 do
        ρ′ ← ρ||IntegerToBytes(s, 1)||IntegerToBytes(r, 1)
        Â[r, s] ← RejNTTPoly(ρ′)                    ▷ seed ρ′ depends on s and r
    end for
end for
return Â
```

- 功能:重复调用RejNTTPoly生成多项式矩阵$\hat{A}$

重复调用RejBoundedPoly生成多项式向量s1，s2.两个向量的维度分别为l和k。种子的更新与索引相关。

(25)ExpandS

Algorithm 33 ExpandS(ρ)

Samples vectors $\mathbf{s}_1 \in R^\ell$ *and* $\mathbf{s}_2 \in R^k$*, each with polynomial coordinates whose coefficients are in the interval* $[-\eta, \eta]$.

Input: A seed $\rho \in \mathbb{B}^{64}$.
Output: Vectors $\mathbf{s}_1, \mathbf{s}_2$ of polynomials in R.

```
for r from 0 to ℓ − 1 do
    s1[r] ← RejBoundedPoly(ρ||IntegerToBytes(r, 2))          ▷ seed depends on r
end for
for r from 0 to k − 1 do
    s2[r] ← RejBoundedPoly(ρ||IntegerToBytes(r + ℓ, 2))      ▷ seed depends on r + ℓ
end for
return (s1, s2)
```

- 功能:重复调用RejBoundedPoly生成多项式向量s_1, s_2

从种子和参数中采样得到多项式向量y，维度是L

(26)ExpandMask

Algorithm 34 ExpandMask(ρ, μ)

Samples a vector $\mathbf{y} \in R^\ell$ *such that each polynomial* $\mathbf{y}[r]$ *has coefficients between* $-\gamma_1 + 1$ *and* γ_1.

Input: A seed $\rho \in \mathbb{B}^{64}$ and a nonnegative integer μ.
Output: Vector $\mathbf{y} \in R^\ell$.

```
c ← 1 + bitlen (γ1 − 1)                           ▷ γ1 is always a power of 2
for r from 0 to ℓ − 1 do
    ρ′ ← ρ||IntegerToBytes(μ + r, 2)
    v ← H(ρ′, 32c)                                ▷ seed depends on μ + r
    y[r] ← BitUnpack(v, γ1 − 1, γ1)
end for
return y
```

- 功能:

1.4 高阶位和低阶位提示

- **Power2Round,Decompose,HighBits,LowBits,Makehint,UseHint**全部都是用于多项式的每个系数
- Power2Round:分解$r \bmod q = r_1 \cdot 2^d + r_0$的高低位比特，其中$r \in \mathbb{Z}_q$。

$$r_0 = (r \bmod q) \bmod^{\pm} 2^d$$
$$r_1 = (r \bmod q - r_0)/2^d$$

这个意思是r mod q后等于那一些

(27)Power2Round

Algorithm 35 Power2Round(r)

Decomposes r into (r_1, r_0) such that $r \equiv r_1 2^d + r_0$ mod q.

Input: $r \in \mathbb{Z}_q$.
Output: Integers (r_1, r_0).

算法中的正负号的意思是结果可正可负

```
r⁺ ← r mod q
r₀ ← r⁺ mod± 2^d
return ((r⁺ − r₀)/2^d, r₀)
```

- 功能:将整数环上的数分解为两部分

另一种分解方式

(28)Decompose

Algorithm 36 Decompose(r)

Decomposes r into (r_1, r_0) such that $r \equiv r_1(2\gamma_2) + r_0$ mod q.

Input: $r \in \mathbb{Z}_q$.
Output: Integers (r_1, r_0).

里面的gamma_2来自于既定的密码参数

```
r⁺ ← r mod q
r₀ ← r⁺ mod±(2γ₂)
if r⁺ − r₀ = q − 1 then
    r₁ ← 0
    r₀ ← r₀ − 1
else r₁ ← (r⁺ − r₀)/(2γ₂)
end if
return (r₁, r₀)
```

- 功能:将整数环上的数分解为两部分,用于与提示相关的计算

(29)HighBits

提取decompose的r1

Algorithm 37 HighBits(r)

Returns r_1 from the output of Decompose (r).

Input: $r \in \mathbb{Z}_q$.
Output: Integer r_1.

```
(r₁, r₀) ← Decompose(r)
return r₁
```

- 功能:从Decompose中返回高位

(30)LowBits

Algorithm 38 LowBits(r)

Returns r_0 from the output of Decompose (r).

Input: $r \in \mathbb{Z}_q$.
Output: Integer r_0.

$(r_1, r_0) \leftarrow$ Decompose(r)
return r_0

- 功能:从Decompose中返回低位

看看向r中添加z是
否会影响r1.
作用是

(31)MakeHint

Algorithm 39 MakeHint(z, r)

Computes hint bit indicating whether adding z to r alters the high bits of r.

Input: $z, r \in \mathbb{Z}_q$.
Output: Boolean.

$r_1 \leftarrow$ HighBits(r)
$v_1 \leftarrow$ HighBits$(r + z)$
return $[[r_1 \neq v_1]]$

保持高位比特对于某些计算的正确
性以及效率很重要

- 功能:用于判断向r中添加z是否改变r的高位

你说这个高位比特不变是不是可以用
来做一些优化之类的？不动点。。

(32)UseHint

Algorithm 40 UseHint(h, r)

Returns the high bits of r adjusted according to hint h.

Input: Boolean h, $r \in \mathbb{Z}_q$.
Output: $r_1 \in \mathbb{Z}$ with $0 \leq r_1 \leq \frac{q-1}{2\gamma_2}$.

根据h调整r的高位比特。
用于精确控制高位比特

$m \leftarrow (q-1)/(2\gamma_2)$
$(r_1, r_0) \leftarrow$ Decompose(r)
if $h = 1$ and $r_0 > 0$ **return** $(r_1 + 1) \bmod m$
if $h = 1$ and $r_0 \leq 0$ **return** $(r_1 - 1) \bmod m$
return r_1

- 功能:返回根据提示调整的后的r的高位

1.5 快速数论变换

(1) NTT的原理

- 由于Dilithium中采用$q = 838047, n = 256$,因此存在512次本源根，能够将多项式环$\mathbb{Z}_q[X]/(X^{256} + 1)$完全分解为256个一次多项式环，其中$\zeta = 1753$是512次单位根

$$\zeta^{512} = (\zeta^{256})^2 \equiv 1 \bmod q \Rightarrow \zeta^{256} \equiv -1 \bmod q$$

则:

$$
\begin{aligned}
X^{256}+1 &= (X^{256}-\zeta^{256}) \\
&= \prod_{i=0}^{255}(X-\zeta^{i}) \\
&= \prod_{i=0}^{255}(X-\zeta^{\mathrm{BitRev}_8(i)}) \bmod q
\end{aligned}
$$

其中$\mathrm{BitRev}_8(r)$作用是将8bit无符号数的bit位顺序反转，即
$\mathrm{BitRev}_8(r)=\mathrm{BitRev}_8(r_0 2^0+r_1 2^1+\dots r_7 2^7)=r_7 2^0+r_5 2^1+\dots r_0 2^7$。因此，多项式环$R_q=\mathbb{Z}_q[X]/(X^{256}+1)$同构于256个一次扩展的直和，即$T_q=\bigoplus_{i=1}^{n}\mathbb{Z}_q[X]/(X-\zeta^{\mathrm{BitRev}_8(\mathrm{i})})$.

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

$$
\hat{f}=\mathrm{NTT}(f)=\hat{f}_0+\hat{f}_1X+\cdots\hat{f}_{255}X^{255}
$$

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256个0次剩余多项式组成的向量:

$$
\begin{aligned}
\hat{f} &= \mathrm{NTT}(f) \\
&= (f \bmod X-\zeta^{\mathrm{BitRev}_8(0)},\dots,f \bmod X-\zeta^{\mathrm{BitRev}_8(255)}) \\
&= (\hat{f}_0,\hat{f}_1,\dots,\hat{f}_{254},\hat{f}_{255})
\end{aligned}
$$

证明以下过程：$f \bmod X-\zeta^{\mathrm{BitRev}_8(i)}\Rightarrow\hat{f}_i$
因为：$X\equiv\zeta^{\mathrm{BitRev}_8(i)} \bmod X-\zeta^{\mathrm{BitRev}_8(i)}$

$$
\begin{aligned}
f \bmod X-\zeta^{\mathrm{BitRev}_8(i)} &= \sum_{j=0}^{255}f_j x^j \bmod X-\zeta^{\mathrm{BitRev}_8(i)} \\
&= (\sum_{j=0}^{255}f_j x^j) \bmod X-\zeta^{\mathrm{BitRev}_8(i)} \\
&= (\sum_{j=0}^{255}f_j\zeta^{\mathrm{BitRev}_8(i)j}) \bmod X-\zeta^{\mathrm{BitRev}_8(i)}
\end{aligned}
$$

令:

$$
\hat{f}_i=\sum_{j=0}^{255}f_j\zeta^{(\mathrm{BitRev}_8(i))j}
$$

因此，$f \bmod X-\zeta^{\mathrm{BitRev}_8(i)}=\hat{f}_i$
实际上NTT算法的核心就是利用单位根的对称性加速式（7）的计算

（2）PWM逐点乘法，符号记作∘

计算多项式乘法→转换为NTT域计算→NTT进行256次（乘法+模+减去幂运算）→进行INTT运算。即可得到多项式乘法结果。

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$(\hat{f}_0,\hat{f}_1,\dots,\hat{f}_{255})$，$g(x)$的NTT向量形式为$(\hat{g}_0,\hat{g}_1,\dots,\hat{g}_{255})$
则多项式乘积的NTT系数向量为:

$$
\begin{aligned}
(\hat{h}_0,\hat{h}_1,\dots,\hat{h}_{255}) &= (\hat{f}_0,\hat{f}_1,\dots,\hat{f}_{255})\circ(\hat{g}_0,\hat{g}_1,\dots,\hat{g}_{255}) \\
&= (F_0\cdot G_0 \bmod X-\zeta^{\mathrm{BitRev}_8(0)},\dots,F_{255}\cdot G_{255} \bmod X-\zeta^{\mathrm{BitRev}_8(255)})
\end{aligned}
$$

---

(3) 伪代码

(33)NTT

Algorithm 41 NTT(w)

Computes the NTT.

Input: Polynomial $w(X) = \sum_{j=0}^{255} w_j X^j \in R_q$.

Output: $\hat{w} = (\hat{w}[0], \ldots, \hat{w}[255]) \in T_q$.

老生常谈了

1: **for** j **from** 0 **to** 255 **do**
2: $\quad \hat{w}[j] \leftarrow w_j$
3: **end for**
4: $m \leftarrow 0$
5: $len \leftarrow 128$
6: **while** $len \geq 1$ **do**
7: $\quad start \leftarrow 0$
8: $\quad$ **while** $start < 256$ **do**
9: $\quad\quad m \leftarrow m + 1$
10: $\quad\quad z \leftarrow \mathsf{zetas}[m]$ $\quad \triangleright\ z \leftarrow \zeta^{\mathsf{BitRev}_8(m)} \bmod q$
11: $\quad\quad$ **for** j **from** $start$ **to** $start + len - 1$ **do**
12: $\quad\quad\quad t \leftarrow (z \cdot \hat{w}[j + len]) \bmod q$
13: $\quad\quad\quad \hat{w}[j + len] \leftarrow (\hat{w}[j] - t) \bmod q$
14: $\quad\quad\quad \hat{w}[j] \leftarrow (\hat{w}[j] + t) \bmod q$
15: $\quad\quad$ **end for**
16: $\quad\quad start \leftarrow start + 2 \cdot len$
17: $\quad$ **end while**
18: $\quad len \leftarrow \lfloor len/2 \rfloor$
19: **end while**
20: **return** $\hat{w}$

- 功能:核心过程采用的cooley-Tukey(CT)蝶形运算，又称为时域抽取（DIT）。对于应12-14行。

(34)INTT

Algorithm 42 $\mathsf{NTT}^{-1}(\hat{w})$

Computes the inverse of the NTT.

Input: $\hat{w} = (\hat{w}[0], \ldots, \hat{w}[255]) \in T_q$.

Output: Polynomial $w(X) = \sum_{j=0}^{255} w_j X^j \in R_q$.

也是老生常谈了

1: **for** j **from** 0 **to** 255 **do**
2: $\quad w_j \leftarrow \hat{w}[j]$
3: **end for**
4: $m \leftarrow 256$
5: $len \leftarrow 1$
6: **while** $len < 256$ **do**
7: $\quad start \leftarrow 0$
8: $\quad$ **while** $start < 256$ **do**
9: $\quad\quad m \leftarrow m - 1$
10: $\quad\quad z \leftarrow -zetas[m]$ $\qquad \triangleright\ z \leftarrow -\zeta^{\mathsf{BitRev}_8(m)} \bmod q$
11: $\quad\quad$ **for** j **from** $start$ **to** $start + len - 1$ **do**
12: $\quad\quad\quad t \leftarrow w_j$
13: $\quad\quad\quad w_j \leftarrow (t + w_{j+len}) \bmod q$
14: $\quad\quad\quad w_{j+len} \leftarrow (t - w_{j+len}) \bmod q$
15: $\quad\quad\quad w_{j+len} \leftarrow (z \cdot w_{j+len}) \bmod q$
16: $\quad\quad$ **end for**
17: $\quad\quad start \leftarrow start + 2 \cdot len$
18: $\quad$ **end while**
19: $\quad len \leftarrow 2 \cdot len$
20: **end while**
21: $f \leftarrow 8347681$ $\qquad \triangleright\ f = 256^{-1} \bmod q$
22: **for** j **from** 0 **to** 255 **do**
23: $\quad w_j \leftarrow (f \cdot w_j) \bmod q$
24: **end for**
25: **return** w

- 功能: 核心过程采用的Gentleman-Sande(GS)蝶形运算，又称为频域抽取（DIF）对于应13-15行。

(35)BitRev8

Algorithm 43 $\mathsf{BitRev}_8(m)$

Transforms a byte by reversing the order of bits in its 8-bit binary expansion.

Input: A byte $m \in [0, 255]$.

Output: A byte $r \in [0, 255]$.

$b \leftarrow \mathsf{IntegerToBits}(m, 8)$ $\qquad$ bit逆序
$b_{\mathsf{rev}} \in \{0,1\}^8 \leftarrow (0, \ldots, 0)$
for i **from** 0 **to** 7 **do**
$\quad b_{\mathsf{rev}}[i] \leftarrow b[7 - i]$
end for
$r \leftarrow \mathsf{BitsToInteger}(b_{\mathsf{rev}}, 8)$
return r

NTT域中的加法，
注意是在环上的

(36)AddNTT

Algorithm 44 AddNTT$(\hat{a}, \hat{b})$

Computes the sum $\hat{a} + \hat{b}$ of two elements $\hat{a}, \hat{b} \in T_q$.

Input: $\hat{a}, \hat{b} \in T_q$.
Output: $\hat{c} \in T_q$.

1: **for** i **from** 0 **to** 255 **do**
2:     $\hat{c}[i] \leftarrow \hat{a}[i] + \hat{b}[i]$
3: **end for**
4: **return** $\hat{c}$

- 功能：实现多项式系数NTT表示的逐点加法

NTT域中的乘法，
注意是在环上的

(37)MultiplyNTT（PWM）

Algorithm 45 MultiplyNTT$(\hat{a}, \hat{b})$

Computes the product $\hat{a} \circ \hat{b}$ of two elements $\hat{a}, \hat{b} \in T_q$.

Input: $\hat{a}, \hat{b} \in T_q$.
Output: $\hat{c} \in T_q$.

1: **for** i **from** 0 **to** 255 **do**
2:     $\hat{c}[i] \leftarrow \hat{a}[i] \cdot \hat{b}[i]$
3: **end for**
4: **return** $\hat{c}$

- 功能：实现多项式系数NTT表示的逐点乘法

多项式向量的系数NTT加法。
两个长度相等的向量相同位置对应相加

(38)AddVectorNTT

Algorithm 46 AddVectorNTT$(\hat{\mathbf{v}}, \hat{\mathbf{w}})$

Computes the sum $\hat{\mathbf{v}} + \hat{\mathbf{w}}$ of two vectors $\hat{\mathbf{v}}, \hat{\mathbf{w}}$ over T_q.

Input: $\ell \in \mathbb{N}, \hat{\mathbf{v}} \in T_q^\ell, \hat{\mathbf{w}} \in T_q^\ell$.
Output: $\hat{\mathbf{u}} \in T_q^\ell$.

1: **for** i **from** 0 **to** $\ell - 1$ **do**
2:     $\hat{\mathbf{u}}[i] \leftarrow$ AddNTT$(\hat{\mathbf{v}}[i], \hat{\mathbf{w}}[i])$
3: **end for**
4: **return** $\hat{\mathbf{u}}$

- 功能：实现多项式向量的系数NTT表示的逐点加法

对多项式向量中的每个元素都添加一
个常数。

(39)ScalarVectorNTT

Algorithm 47 ScalarVectorNTT$(\hat{c}, \hat{\mathbf{v}})$

Computes the product $\hat{c} \circ \hat{\mathbf{v}}$ of a scalar $\hat{c}$ and a vector $\hat{\mathbf{v}}$ over T_q.

Input: $\hat{c} \in T_q, \ell \in \mathbb{N}, \hat{\mathbf{v}} \in T_q^\ell$.
Output: $\hat{\mathbf{w}} \in T_q^\ell$.

1: **for** i **from** 0 **to** $\ell - 1$ **do**
2:     $\hat{\mathbf{w}}[i] \leftarrow$ MultiplyNTT$(\hat{c}, \hat{\mathbf{v}}[i])$
3: **end for**
4: **return** $\hat{\mathbf{w}}$

- 功能：依次对多项式向量中的每个多项式的NTT表示进行标量的逐点乘法

矩阵乘以向量，最后得到一条向量。

(40)MatrixVectorNTT

Algorithm 48 MatrixVectorNTT$(\hat{\mathbf{M}}, \hat{\mathbf{v}})$

Computes the product $\hat{\mathbf{M}} \circ \hat{\mathbf{v}}$ *of a matrix* $\hat{\mathbf{M}}$ *and a vector* $\hat{\mathbf{v}}$ *over* T_q.

Input: $k, \ell \in \mathbb{N}$, $\hat{\mathbf{M}} \in T_q^{k\times\ell}$, $\hat{\mathbf{v}} \in T_q^{\ell}$.
Output: $\hat{\mathbf{w}} \in T_q^{k}$.

```
1: ŵ ← 0^k
2: for i from 0 to k − 1 do
3:     for j from 0 to ℓ − 1 do
4:         ŵ[i] ← AddNTT(ŵ[i], MultiplyNTT(M̂[i, j], v̂[j]))
5:     end for
6: end for
7: return ŵ
```

- 功能：NTT形式的多项式矩阵乘法
- **上述过程除了PWM以外，和kyber的NTT完全一致**

2.ML_DSA内部组件方案（internal)

- **除了测试目的外，本部分规定的密钥生成和签名生成的接口不应提供给应用程序，因为密钥生成和签名生成所需的任何随机值都应由密码模块生成**

2.1 参数说明

Table 1. ML-DSA parameter sets

Parameters (see Sections 6.1 and 6.2 of this document)	Values assigned by each parameter set ML-DSA-44	ML-DSA-65	ML-DSA-87
q - modulus [see §6.1]	8380417	8380417	8380417
ζ - a 512th root of unity in $\mathbb{Z}_q$ [see §7.5]	1753	1753	1753
d - # of dropped bits from $\mathbf{t}$ [see §6.1]	13	13	13
τ - # of ± 1's in polynomial c [see §6.2]	39	49	60
λ - collision strength of $\tilde{c}$ [see §6.2]	128	192	256
γ_1 - coefficient range of $\mathbf{y}$ [see §6.2]	2^{17}	2^{19}	2^{19}
γ_2 - low-order rounding range [see §6.2]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k,ℓ) - dimensions of $\mathbf{A}$ [see §6.1]	(4,4)	(6,5)	(8,7)
η - private key range [see §6.1]	2	4	2
$\beta = \tau \cdot \eta$ [see §6.2]	78	196	120
ω - max # of 1's in the hint $\mathbf{h}$ [see §6.2]	80	55	75
Challenge entropy $\log_2 \binom{256}{\tau} + \tau$ [see §6.2]	192	225	257
Repetitions (see explanation below)	4.25	5.1	3.85
Claimed security strength	Category 2	Category 3	Category 5

- q:模数
- ζ:512次单位根
- d:公钥多项式$\boldsymbol{t}$每个系数需要丢弃的bit数
- τ:验证者挑战的多项式c的汉明权重
- λ:碰撞强度
- γ_1:多项式向量$\boldsymbol{y}$的系数范围
- γ_2:低位舍入范围

- η:私钥的系数范围
- w:提示向量$\boldsymbol{h}$中1的最大个数

Table 2. Sizes (in bytes) of keys and signatures of ML-DSA

	Private Key	Public Key	Signature Size
ML-DSA-44	2560	1312	2420
ML-DSA-65	4032	1952	3309
ML-DSA-87	4896	2592	4627

2.2 ML_DSA内部组件方案

(1)ML_DSA Key Generation密钥生成算法

k和l由安全标准参数决定

Algorithm 6 ML-DSA.KeyGen_internal(ξ)

Generates a public-private key pair from a seed.

对t进行压缩得到两部分。显然一个生成公钥，一个生成私钥

Input: Seed $\xi \in \mathbb{B}^{32}$
Output: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}\,(q-1)-d)}$
and private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen}\,(2\eta)+dk)}$.

1: $(\rho, \rho', K) \in \mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \leftarrow \text{H}(\xi\|\text{IntegerToBytes}(k, 1)\|\text{IntegerToBytes}(\ell, 1), 128)$
2: ▷ expand seed
3: $\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
4: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \text{ExpandS}(\rho')$
5: $\mathbf{t} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$ ▷ compute $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \text{Power2Round}(\mathbf{t})$ ▷ compress $\mathbf{t}$
7: ▷ PowerTwoRound is applied componentwise (see explanatory text in Section 7.4)
8: $pk \leftarrow \text{pkEncode}(\rho, \mathbf{t}_1)$
9: $tr \leftarrow \text{H}(pk, 64)$
10: $sk \leftarrow \text{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$ ▷ K and tr are for use in signing
11: **return** (pk, sk)

K和tr用于签名操作，在后续的签名算法中会被提取出来。

- 输入：通过SHAKE256生成的种子ξ,长度为32的字节数组
- 输出：公钥pk,长度为$32 + 32k(\text{bitlen}(q-1) - d)$的字节数组；私钥sk,长度为$32 + 32 + 64 + 32((l+k)\text{bitlen}(2\eta) + dk)$的字节数组
- 1：将种子ξ与k, l的1字节形式拼接，然后送入H函数（实际上为SHAKE256）,扩展出长度为32字节的公共随机种子ρ，以及64字节和32字节的私钥随机种子ρ'和K
- 3:通过32字节的公共随机种子ρ以及ExpandA获取公钥矩阵$\hat{\boldsymbol{A}}$，该过程类似于kyber,但采样细节有所区别
- 4：通过64字节的公共随机种子ρ以及ExpandS获取多项式向量$\boldsymbol{s}_1, \boldsymbol{s}_2$，其中多项式的系数在$(-\eta, \eta)$范围内
- 5：计算公钥多项式向量$\boldsymbol{t}$
- 6：采用Power2Round公钥多项式向量$\boldsymbol{t}$中每个多项形式系数高低为比特分离，从而实现压缩，降低计算量
- 8：将公钥多项式向量$\boldsymbol{t}1$和ρ进行编码，过程和kyber完全一致
- 9：计算公钥pk的哈希，输出长度为64的字节数组tr
- 10:对私钥随机种子$\rho, K, tr, \boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{t}_0$进行编码

格式化的消息M

(2)ML_DSA Signing签名算法

Algorithm 7 ML-DSA.Sign_internal(sk, M', rnd)

Deterministic algorithm to generate a signature for a formatted message M'.

Input: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen}(2\eta)+dk)}$, formatted message $M' \in \{0,1\}^*$, and per message randomness or dummy variable $rnd \in \mathbb{B}^{32}$.

Output: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}(\gamma_1-1))+\omega+k}$.

增强签名随机性的变量rnd

$(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0) \leftarrow$ skDecode(sk)

$\hat{\mathbf{s}}_1 \leftarrow$ NTT$(\mathbf{s}_1)$

$\hat{\mathbf{s}}_2 \leftarrow$ NTT$(\mathbf{s}_2)$

$\hat{\mathbf{t}}_0 \leftarrow$ NTT$(\mathbf{t}_0)$

消息生成哈希

$\hat{\mathbf{A}} \leftarrow$ ExpandA(ρ) ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$

$\mu \leftarrow$ H(BytesToBits$(tr)||M', 64)$ ▷ message representative that may optionally be computed in a different cryptographic module

拒绝采样循环

$\rho'' \leftarrow$ H$(K||rnd||\mu, 64)$ ▷ compute private random seed

进一步哈希

$\kappa \leftarrow 0$ ▷ initialize counter κ

$(\mathbf{z}, \mathbf{h}) \leftarrow \perp$

while $(\mathbf{z}, \mathbf{h}) = \perp$ **do** ▷ rejection sampling loop

  $\mathbf{y} \in R_q^\ell \leftarrow$ ExpandMask(ρ'', κ)

  $\mathbf{w} \leftarrow$ NTT$^{-1}(\hat{\mathbf{A}} \circ$ NTT$(\mathbf{y}))$

  $\mathbf{w}_1 \leftarrow$ HighBits$(\mathbf{w})$ ▷ signer's commitment

  ▷ HighBits is applied componentwise (see explanatory text in Section 7.4)

  $\tilde{c} \leftarrow$ H$(\mu||$w1Encode$(\mathbf{w}_1), \lambda/4)$ ▷ commitment hash

计算承诺哈希

  $c \in R_q \leftarrow$ SampleInBall$(\tilde{c})$ ▷ verifier's challenge

  $\hat{c} \leftarrow$ NTT(c)

  $\langle\langle c\mathbf{s}_1\rangle\rangle \leftarrow$ NTT$^{-1}(\hat{c} \circ \hat{\mathbf{s}}_1)$

  $\langle\langle c\mathbf{s}_2\rangle\rangle \leftarrow$ NTT$^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2)$

有效性检查

  $\mathbf{z} \leftarrow \mathbf{y} + \langle\langle c\mathbf{s}_1\rangle\rangle$ ▷ signer's response

  $\mathbf{r}_0 \leftarrow$ LowBits$(\mathbf{w} - \langle\langle c\mathbf{s}_2\rangle\rangle)$

  ▷ LowBits is applied componentwise (see explanatory text in Section 7.4)

  if $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ **or** $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$ **then** $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$ ▷ validity checks

  else

    $\langle\langle c\mathbf{t}_0\rangle\rangle \leftarrow$ NTT$^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)$

z的范数

    $\mathbf{h} \leftarrow$ MakeHint$(-\langle\langle c\mathbf{t}_0\rangle\rangle, \mathbf{w} - \langle\langle c\mathbf{s}_2\rangle\rangle + \langle\langle c\mathbf{t}_0\rangle\rangle)$ ▷ Signer's hint

    ▷ MakeHint is applied componentwise (see explanatory text in Section 7.4)

    if $||\langle\langle c\mathbf{t}_0\rangle\rangle||_\infty \geq \gamma_2$ **or** the number of 1's in $\mathbf{h}$ is greater than ω, **then** $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$

    end if

计算提示位h

  end if

  $\kappa \leftarrow \kappa + \ell$ ▷ increment counter

end while

$\sigma \leftarrow$ sigEncode$(\tilde{c}, \mathbf{z} \bmod^{\pm} q, \mathbf{h})$

return σ

- 功能：以编码为字节数组的私钥sk,编码为bit数组的格式化消息M'以及32字节的随机数rnd作为输入，输出编码为字节数组的签名。
- ML_DSA Signing有对冲和确定性变体两种。对冲使用的是新的随机值，而确定性变体使用的是常量字节数组（{0}32）
- 输入:
  - 私钥sk,长度为$32+32+64+32((l+k)\text{bitlen}(2\eta)+dk)$的字节数组
  - 格式化消息M':比特数组
  - 随机变量rnd:32长度的字节数组
- 输出：签名σ,长度为$\lambda/4+32l(1+bitlen(\gamma_1-1))+w+k$,其中$\lambda/4$为碰撞强度，$\gamma_1$为多项式向量$\boldsymbol{y}$的系数范围，$k,l$为多项式矩阵的维度，$w$为提示$\boldsymbol{h}$的中1bit的个数

- 1：签名者通过skDecode从私钥sk中提取以下信息：公共随机种子ρ，32字节的私有随机种子K，64字节的公有密钥pk的哈希值，私钥多项式向量$\boldsymbol{s}_1$和$\boldsymbol{s}_2$，以及编码未压缩的公有密钥多项式$\boldsymbol{t}$的每个系数的d个最低有效位的多项式向量$\boldsymbol{t}_0$。
- 2-4:分别对私钥多项式向量$\boldsymbol{s}_1$和$\boldsymbol{s}_2$以及多项式向量$\boldsymbol{t}_0$执行NTT
- 5:通过ρ恢复与密钥生成相同的矩阵$\hat{\boldsymbol{A}}$
- 6：在消息M'进行签名之前，将其与公钥哈希tr进行级联，并使用H将其哈希到一个64字节的消息代表μ
- 7:在每次签名过程中，签名者需要产生一个额外的64字节的种子ρ''，用于产生私钥的随机性。ρ''的计算为$\rho'' = \mathrm{H}(K||rnd||\mu, 64)$。在默认的对称变体中，rnd是一个RBG的输出，而在确定的变体中，rnd是一个32字节的字符串，它完全由零组成。这是ML-DSA的确定性变体和对冲变体的唯一区别。
- 10-32：签名算法的主要部分，由一个拒绝采样循环组成，在该循环的每次迭代中，要么产生一个有效的签名，要么产生一个无效的签名，这些签名的释放会泄露私钥的信息。循环重复进行，直到产生有效的签名，然后可以将其编码为字节数组并输出。
- 11：使用`ExpandMask`函数、随机种子ρ''以及计数器,从系数为$[-\gamma_1+1, \gamma_1]$的多项式向量集合中采样一个长度为l的多项式向量$\boldsymbol{y}$
- 12-13：计算$\boldsymbol{w} = \boldsymbol{A}\boldsymbol{y}$，通过`Highbits`计算承诺$\boldsymbol{w}_1$，即提取多项式向量$\boldsymbol{w}$中每个系数的高位bit（相当于压缩函数），$\boldsymbol{w}_1$的包含k个多项式
- 15：将承诺$\boldsymbol{w}_1$的编码和消息代表μ级联，并通过H函数哈希到长度$\lambda/4$的承诺哈希$\tilde{c}$（字节数组）
- 16：将承诺哈希$\tilde{c}$（字节数组）用于伪随机采样一个系数为{-1,0,1}以及汉明权重为τ的多项式c（验证者挑战）,是一个长度为256的整数数组
- 17-20：通过计算$\boldsymbol{z} = \boldsymbol{y} + c\boldsymbol{s}_1$获取响应多项式向量$\boldsymbol{z}$
- 21:提取$\boldsymbol{w} - c\boldsymbol{s}_2$的低比特
- 23-32：执行有效性检查，通过则生成提示$\boldsymbol{h}$并退出循环，不通过则继续
- 33：将承诺哈希$\tilde{c}$、响应$z \bmod^{\pm} q$和提示$\boldsymbol{h}$的字节编码获得签名

(3)ML_DSA Verifing验证算法

验证消息和签名是否对应。

Algorithm 8 ML-DSA.Verify_internal(pk, M', σ)

Internal function to verify a signature σ for a formatted message M'.

又来生成A_hat了。

Input: Public key $pk \in \mathbb{B}^{32+32k(\mathrm{bitlen}\,(q-1)-d)}$ and message $M' \in \{0,1\}^*$.
Input: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\mathrm{bitlen}\,(\gamma_1-1))+\omega+k}$.
Output: Boolean

从签名中恢复承诺哈希c_hat，响应z和提示h。

$(\rho, \mathbf{t}_1) \leftarrow \mathsf{pkDecode}(pk)$
$(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow \mathsf{sigDecode}(\sigma)$ ▷ signer's commitment hash $\tilde{c}$, response $\mathbf{z}$, and hint $\mathbf{h}$
if $\mathbf{h} = \bot$ **then return** false ▷ hint was not properly encoded
end if

相同的tr。相同的生成方式

$\hat{\mathbf{A}} \leftarrow \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
$tr \leftarrow \mathsf{H}(pk, 64)$
$\mu \leftarrow (\mathsf{H}(\mathsf{BytesToBits}(tr)||M', 64))$ ▷ message representative that may optionally be computed in a different cryptographic module
$c \in R_q \leftarrow \mathsf{SampleInBall}(\tilde{c})$ ▷ compute verifier's challenge from $\tilde{c}$
$\mathbf{w}'_{\mathrm{Approx}} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathsf{NTT}(\mathbf{z}) - \mathsf{NTT}(c) \circ \mathsf{NTT}(\mathbf{t}_1 \cdot 2^d))$ ▷ $\mathbf{w}'_{\mathrm{Approx}} = \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d$
$\mathbf{w}'_1 \leftarrow \mathsf{UseHint}(\mathbf{h}, \mathbf{w}'_{\mathrm{Approx}})$ ▷ reconstruction of signer's commitment
▷ UseHint is applied componentwise (see explanatory text in Section 7.4)
$\tilde{c}' \leftarrow \mathsf{H}(\mu||\mathsf{w1Encode}(\mathbf{w}'_1), \lambda/4)$ ▷ hash it; this should match $\tilde{c}$
return $[[\,||\mathbf{z}||_\infty < \gamma_1 - \beta]]$ **and** $[[\tilde{c} = \tilde{c}']]$

用于和之前的承诺哈希比较　　重建签名者的承诺

- 功能：将编码为字节数组的公钥pk、编码为比特数组的消息M'和编码为字节数组的签名σ作为输入。它产生一个布尔值作为输出，如果签名对消息和公钥有效,则该值为真;如果签名无效,则该值为假。
- 输入：
  - 公钥pk,长度为$32+32k(\text{bitlen}(q-1)-d)$的字节数组
  - 格式化消息M':比特数组
  - 签名σ,长度为$\lambda/4+32l(1+bitlen(\gamma_1-1))+w+k$,其中$\lambda/4$为碰撞强度，$\gamma_1$为多项式向量$\boldsymbol{y}$的系数范围，$k,l$为多项式矩阵的维度，$w$为提示$\boldsymbol{h}$的中1bit的个数
- 输出：布尔值
- 1-2：验证者首先从公钥pk中提取出公共的随机种子ρ和压缩多项式向量$\boldsymbol{t}_1$，从签名中解码出承诺哈希$\tilde{c}$、响应$\boldsymbol{z}$和提示$\boldsymbol{h}$
- 3-4：检验提示信息有没有被正确的字节编码，用符号'⊥'表示，在这种情况下验证算法会立即返回错误以表明签名无效。
- 5：通过公共随机种子ρ恢复出矩阵$\hat{\boldsymbol{A}}$
- 6：将公钥pk哈希到64字节的tr上
- 7：将消息M'与公钥哈希tr进行级联，并使用H将其哈希到一个64字节的消息代表μ上
- 8：将承诺哈希$\tilde{c}$（字节数组）用于伪随机采样一个系数为{-1,0,1}以及汉明权重为τ的多项式c（有称为验证者挑战）,是一个长度为256的整数数组
- 9-10：验证者从公钥pk和签名σ重建签名者的承诺，即多项式向量$\boldsymbol{w}_1$。核心过程如下：

$$\begin{aligned}
\boldsymbol{w} &= \boldsymbol{A}\boldsymbol{y} \\
&= \boldsymbol{A}(\boldsymbol{z}-c\boldsymbol{s}_1) \\
&= \boldsymbol{A}\boldsymbol{z}-c\boldsymbol{A}\boldsymbol{s_1} \\
\because \boldsymbol{t} = \boldsymbol{A}\boldsymbol{s_1}+\boldsymbol{s_2} & \\
&= \boldsymbol{A}\boldsymbol{z}-c(\boldsymbol{t}-\boldsymbol{s_2}) \\
&= \boldsymbol{A}\boldsymbol{z}-c\boldsymbol{t}+c\boldsymbol{s}_2
\end{aligned}$$

由于$c,\boldsymbol{s}_2$的系数很小，因此$\boldsymbol{t}_1\cdot 2^d\approx\boldsymbol{t}$。验证者计算$\boldsymbol{w}'_{Approx}$：

$$\boldsymbol{w}'_{Approx}=\boldsymbol{A}\boldsymbol{z}-c\boldsymbol{t}_1\cdot 2^d$$

然后利用签名者的提示$\boldsymbol{h}$从$\boldsymbol{w}'_{Approx}$中恢复$\boldsymbol{w}'_1$。

- 12：将计算的消息代表μ和重建的承诺$\boldsymbol{w}'_1$级联，然后哈希得到承诺哈希$\tilde{c}'$
- 13：最后验证响应$\boldsymbol{z}$的所有系数是否都在$(-(\gamma_1-\beta),\gamma_1-\beta)$范围内，并且重建的承诺哈希$\tilde{c}'$与签名者的承诺哈希$\tilde{c}$是否一致。若都一致，则返回true，表示签名有效,否则返回false，签名无效。

3.ML_DSA外部组件方案（external）

(1)ML-DSA_KeyGen

随机种子是RBG生成的

Algorithm 1 ML-DSA.KeyGen()

Generates a public-private key pair.

Output: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}\,(q-1)-d)}$

and private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen}\,(2\eta)+dk)}$.

$\xi \leftarrow \mathbb{B}^{32}$ ▷ choose random seed
if $\xi =$ NULL **then**
  return $\perp$ ▷ return an error indication if random bit generation failed
end if
return ML-DSA.KeyGen_internal (ξ)

- 功能：使用RBG来产生一个32字节的随机种子ξ，并将其作为ML_DSA_KeyGen_internal的输入，产生公钥和私钥

(2)ML-DSA_Sign

上下文字符串（小于255字节）

Algorithm 2 ML-DSA.Sign(sk, M, ctx)

Generates an ML-DSA signature.

Input: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen}\,(2\eta)+dk)}$, message $M \in \{0,1\}^*$,
context string ctx (a byte string of 255 or fewer bytes).
Output: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}\,(\gamma_1-1))+\omega+k}$.

1: **if** $|ctx| > 255$ **then**
2:   **return** $\perp$ ▷ return an error indication if the context string is too long
3: **end if**
4:
5: $rnd \leftarrow \mathbb{B}^{32}$ ▷ for the optional deterministic variant, substitute $rnd \leftarrow \{0\}^{32}$
6: **if** $rnd =$ NULL **then**
7:   **return** $\perp$ ▷ return an error indication if random bit generation failed
8: **end if**
9:
10: $M' \leftarrow$ BytesToBits(IntegerToBytes$(0,1)$ $\|$ IntegerToBytes$(|ctx|,1)$ $\|$ $ctx)$ $\|$ M
11: $\sigma \leftarrow$ ML-DSA.Sign_internal(sk, M', rnd)
12: **return** σ

构造格式化的M

- 功能：将私钥和消息以及小于255字节的文本作为输入，输出编码为字节数组的签名
- 1-3：验证文本是否小于255字节，是则执行下一步
- 5：产生一个32字节的随机数组rnd
- 6-9：验证rnd是否成功生成
- 10：将消息M和ctx级联后转换为bit数组
- 11：调用内部签名函数生成签名

(3)ML-DSA_Verify

Algorithm 3 ML-DSA.Verify(pk, M, σ, ctx)

Verifies a signature σ for a message M.

Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}\,(q-1)-d)}$, message $M \in \{0,1\}^*$,
signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}\,(\gamma_1-1))+\omega+k}$,
context string ctx (a byte string of 255 or fewer bytes).

先构造格式化的M

Output: Boolean.

if $|ctx| > 255$ **then**
  return $\perp$ ▷ return an error indication if the context string is too long
end if

$M' \leftarrow$ BytesToBits(IntegerToBytes$(0,1)$ $\|$ IntegerToBytes$(|ctx|, 1)$ $\|$ ctx) $\|$ M
return ML-DSA.Verify_internal(pk, M', σ)

- 功能：以公钥pk、消息M、签名σ和文本字符串作为ctx输入。公钥、签名和文本字符串均编码为字节数组，而消息为比特字符串。ML-DSA_Verify输出一个布尔值，如果该签名相对于消息和公钥有效，则为真；如果该签名无效，则为假。

然后调用内部的函数。很快！