A TECHNICAL NOTE ON UNDERSTANDING FIPS 204 (ML-DSA)

Bo Lin

Version history

Version	Date	Author	Status
A	02 Feb 2026	B LIN	Release

Change history

Version	Changes

Table of Contents

1. Overview and notations

An initial challenge for those who directly dive into the FIPS 204 ML-DSA specification (*FIPS 204, Module-Lattice-Based Digital Signature Standard, August 13, 2024*, https://csrc.nist.gov/pubs/fips/204/final) is that data compression and security checks are interleaved in various functions in the lattice-based signature scheme, making it difficult to map a conceptual lattice-based signature scheme into the FIPS 204 ML-DSA specification. The motivation of this technical note is to bridge this gap to overcome the challenge.

This technical note begins with a brief description of the hard problem in lattice-based cryptography by examining the difference between the complexity of solving $\boldsymbol{Ax} = \boldsymbol{t} \bmod p$ and the complexity of solving $\boldsymbol{Ax} + \boldsymbol{e} = \boldsymbol{t} \bmod p$ (a.k.a. the Learning with Errors (LWE) problem), where p is a pre-defined prime, $\boldsymbol{A}$ is a known $k \times l$ matrix, $\boldsymbol{t}$ a known k-dimensional vector, $\boldsymbol{x}$ an unknown l-dimensional vector, and $\boldsymbol{e}$ an unknown l-dimensional vector.

Then, it proceeds by tabulating two simplified versions of the FIPS 204 ML-DSA – one is a simple math version (SMV) and the other a simplified specification version (SSV) – to explain the FIPS 204 ML-DSA specification. The SMV intends to explain the math fundamentals on how a lattice-based digital signature works while the SSV serves a stepping-stone from the SMV to the FIPS 204 ML-DSA specification.

The technical note ends with a simplified flow-chart for Algorithm 7 ML-DSA.Sign_internal() of the FIPS 204 ML-DSA specification to visualise some key steps.

In this technical note, a bold letter, such as $\boldsymbol{A}$ or $\boldsymbol{x}$, stands for a matrix or a vector. Its entry, as that in the FIPS 204 ML-DSA specification, is a polynomial in $Z_p[X]/(X^{256} + 1)$. On the other hand, a regular letter, such as c, stands for a single element. It can be a polynomial in $Z_p[X]/(X^{256} + 1)$, a data string, or an integer.

If a symbol is the same as that in the FIPS 204 ML-DSA specification, the symbol follows its definition in the specification to make the cross-reference easy.

2. Learning with Errors (LWE)

For $\boldsymbol{Ax} = \boldsymbol{t} \bmod p$, the system of equations can be solved, or the $\boldsymbol{x}$ can be *learned*, efficiently by solving $(\boldsymbol{A} \ \ -\boldsymbol{t}) \cdot \binom{\boldsymbol{x}}{1} = \boldsymbol{0}$. However, if the $\boldsymbol{Ax} = \boldsymbol{t} \bmod p$ is changed to $\boldsymbol{Ax} + \boldsymbol{e} = \boldsymbol{t} \bmod p$ by adding a k-dimensional vector $\boldsymbol{e}$ whose components are small random numbers, i.e., by adding small random errors in the system, this system of equations can written as $(\boldsymbol{A}\ \boldsymbol{I}_k \ -\boldsymbol{t}) \cdot \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{e} \\ 1 \end{pmatrix} = \boldsymbol{0} \bmod p$, where the $\boldsymbol{I}_k$ is an identity matrix of order k. The "Given $\boldsymbol{A}$ and $\boldsymbol{t}$ to find $\boldsymbol{x}$ and $\boldsymbol{e}$" becomes an instance of Learning with Errors (LWE). It can be shown that the vector $(\boldsymbol{x}\ \boldsymbol{e}\ 1)$ is a shortest vector in the lattice pertaining to the $\boldsymbol{A}$, meaning that solving the LWE problem yields to solving the Shortest Vector Problem (SVP) which is extremely difficult for general cases.

There are several variants of LWE. NIST selected CRYSTALS-Dilithium (https://pq-crystals.org/dilithium/), a digital signature scheme based on module LWE, to standardise the Post-Quantum Cryptography digital signature algorithm, ML-DSA (Module-Lattice-Based Digital Signature Standard).

In the FIPS 204 ML-DSA specification, the $\boldsymbol{x}$ is denoted as $\boldsymbol{s}_1$, $\boldsymbol{e}$ as $\boldsymbol{s}_2$. As a result, the $\boldsymbol{Ax}+\boldsymbol{e}=\boldsymbol{t} \bmod p$ is denoted as $\boldsymbol{As_1}+\boldsymbol{s_2}=\boldsymbol{t} \bmod q$ with $q=8380417$.

3. Two simplified FIPS 204 versions

As mentioned in section *1. Overview and notations*, the SMV and SSV are tabulated for contrast. The SMV provides a mathematics proof for the correctness of the signature scheme and the SSV serves as a stepping-stone to the FIPS 204 specification.

3.1. Key generation – KeyGen()

<table>
<tr><th>#</th><th>SMV (Simple Math Version)</th><th>SSV (Simplified Spec Version)</th></tr>
<tr><td>1</td><td colspan="2">Generate $\boldsymbol{A} \in R_q^{k\times l}$. That is, the $\boldsymbol{A}$ is a $k \times l$ matrix with each entry is a polynomial in $Z_q[X]/(X^{256}+1)$. Simply put, an entry of $\boldsymbol{A}$ is a $GF(q)$ polynomial modulo $X^{256}+1$ with its coefficients being random numbers in $[0, q-1]$.
Note, in the FIPS 204 ML-DSA specification, $\widehat{\boldsymbol{A}}$, the $\boldsymbol{A}$'s “spectrum” representation, is generated to facilitate the NTT (Number Theory Transform) operation for polynomial multiplications because a random $\boldsymbol{A}$'s NTT result, $\widehat{\boldsymbol{A}}=\mathrm{NTT}(\boldsymbol{A})$, is a random matrix anyway, so the random $\widehat{\boldsymbol{A}}$ can be regarded as NTT($\boldsymbol{A}$) for some random $\boldsymbol{A}$.</td></tr>
<tr><td>2</td><td colspan="2">Sample $\boldsymbol{s_1} \in R_q^l$ with coefficients in $[-\eta, \eta]$ where $\eta=2$ for ML-DSA-44 while $\eta=4$ for ML-DSA-65 and ML-DSA-87. That is, the $\boldsymbol{s_1}$ is a vector of l components and each component is a 255-degree polynomial with coefficients in $[-\eta, \eta]$.</td></tr>
<tr><td>3</td><td colspan="2">Sample $\boldsymbol{s}_2 \in R_q^k$ with coefficients in $[-\eta, \eta]$ where $\eta=2$ for ML-DSA-44 while $\eta=4$ for ML-DSA-65 and ML-DSA-87. The $\boldsymbol{s}_2$ is a vector of k components and each component is a 255-degree polynomial with coefficients in $[-\eta, \eta]$.</td></tr>
<tr><td>4</td><td colspan="2">Calculate $\boldsymbol{t}=\boldsymbol{As_1}+\boldsymbol{s_2}$.
Note, the $\boldsymbol{As_1}$ operation is implemented as $\mathrm{NTT}^{-1}(\widehat{\boldsymbol{A}} \circ \mathrm{NTT}(\boldsymbol{s_1}))$ where the ∘ is component-wise multiplication.</td></tr>
<tr><td>5</td><td>–</td><td>$\boldsymbol{t}=\boldsymbol{t_1}\cdot 2^d+\boldsymbol{t_0} \bmod q$
$\boldsymbol{t_0}=(t \bmod q) \bmod^{\pm} 2^d$
$d=13$</td></tr>
<tr><td>6</td><td>$pk=(\boldsymbol{A},\boldsymbol{t})$, $sk=\boldsymbol{s_1}$</td><td>$pk=(\boldsymbol{A},\boldsymbol{t_1})$, $sk=(\boldsymbol{s_1},\boldsymbol{s_2},\boldsymbol{t_0})$</td></tr>
</table>

It is noted that the $\boldsymbol{s}_2$ is not part of sk in the SMV, but it is part of sk in the SSV. In addition, the $\boldsymbol{t}$ is split into $\boldsymbol{t}_1$ and $\boldsymbol{t}_0$ in the SSV, where the $\boldsymbol{t}_1$ (higher part of $\boldsymbol{t}$) is used as part of pk while the $\boldsymbol{t}_0$ (lower part of $\boldsymbol{t}$) as part of sk.

In the FIPS 204 ML-DSA specification, the $\boldsymbol{s}_2$ is used for security check and it is also used with $\boldsymbol{t}_0$ to make a “hint” vector $\boldsymbol{h}$ in Sign_internal(). The vector $\boldsymbol{h}$ later used in Verify_internal() to adjust the result from HighBits().

3.2. Signing – Sign_internal()

<table>
<tr><th>#</th><th>SMV (Simple Math Version)</th><th>SSV (Simplified Spec Version)</th></tr>
<tr><td>1</td><td colspan="2">Sample $\boldsymbol{y} \in R_q^l$ with small coefficients in $[-\gamma_1 + 1, \gamma_1]$ where $\gamma_1 = 2^{17}$ for ML-DSA-44 while $\gamma_1 = 2^{19}$ for ML-DSA-65 and ML-DSA-87. That is, the $\boldsymbol{y}$ is a vector of l components and each component is a 255-degree polynomial with coefficients in $[-\gamma_1 + 1, \gamma_1]$.</td></tr>
<tr><td>2</td><td colspan="2">Calculate $\boldsymbol{w} = \boldsymbol{A}\boldsymbol{y}$.
Note, the $\boldsymbol{A}\boldsymbol{y}$ operation is implemented as $\mathrm{NTT}^{-1}(\hat{A} \circ \mathrm{NTT}(\boldsymbol{y}))$ where the $\circ$ is component-wise multiplication.</td></tr>
<tr><td>3</td><td colspan="2">Calculate $\boldsymbol{w_1} = \mathrm{HighBits}(\boldsymbol{w})$ // the $\mathrm{HighBits}()$ is a rounding function</td></tr>
<tr><td>4</td><td colspan="2">Calculate $\tilde{c} = \mathrm{SHAKE256}(\mu || \boldsymbol{w_1})$ // the $\tilde{c}$ is a bit-string</td></tr>
<tr><td>5</td><td colspan="2">Transform the bit-string $\tilde{c}$ to a polynomial c.</td></tr>
<tr><td>6</td><td colspan="2">Calculate $\boldsymbol{z} = \boldsymbol{y} + c\boldsymbol{s_1}$
Note, the $c\boldsymbol{s_1}$ operation is implemented as $\mathrm{NTT}^{-1}(\mathrm{NTT}(c) \circ \mathrm{NTT}(\boldsymbol{s}_1))$ where the $\circ$ is component-wise multiplication.</td></tr>
<tr><td>7</td><td>–</td><td>Use $\boldsymbol{s_2}$ and γ_1 to check security. If fails, go to #1.
Note, the γ_1 is defined in the FIPS 204 ML-DSA specification</td></tr>
<tr><td>8</td><td>–</td><td>Use $\boldsymbol{t_0}$ and $\boldsymbol{s_2}$ to $\mathrm{MakeHint}()$ $\boldsymbol{h}$. If the $\boldsymbol{h}$ is not legitimate, go to #1.</td></tr>
<tr><td>9</td><td>Output signature $\sigma = (\boldsymbol{z}, c)$</td><td>Output signature $\sigma = \mathrm{sigEncode}(\boldsymbol{z}, \tilde{c}, \boldsymbol{h})$</td></tr>
</table>

As can be seen in Sign_internal().SMV, the $\boldsymbol{s}_2$ is not used to work out σ explicitly, neither in the Sign_internal().SSV, but it is used for security check and the $\boldsymbol{h}$ calculation in Sign_internal().SSV.

Line #4 of Sign_internal() needs attention that the bit-string $\tilde{c}$ is not suitable for multiplying a polynomial vector $\boldsymbol{s}_1$. It needs to be transformed to be a polynomial c as that in line #5. This is implemented in Algorithm 29 SampleBall() of the FIPS 204 ML-DSA specification.

The output of Sign_internal().SMV $\sigma = (\boldsymbol{z}, c)$ is conceptual while the output $\sigma = \mathrm{sigEncode}(\boldsymbol{z}, \tilde{c}, \boldsymbol{h})$ of Sign_internal().SSV is a bit-string by compressing and coding $\boldsymbol{z}$, $\tilde{c}$ and $\boldsymbol{h}$.

3.3. Verifying – Verify_internal()

#	SMV (Simple Math Version)	SSV (Simplified Spec Version)
1	Recover c from $\tilde{c}$. // $\tilde{c}$ is a bit string and c is a polynomial	
2	–	Calculate $\boldsymbol{w}'_{\boldsymbol{Approx}} = \boldsymbol{Az} - c\boldsymbol{t_1} \cdot 2^d$ Note, the $\boldsymbol{Az} - c\boldsymbol{t_1} \cdot 2^d$ operation is implemented as $\mathrm{NTT}^{-1}(\hat{A} \circ \mathrm{NTT}(\boldsymbol{z}) - \mathrm{NTT}(c) \circ \mathrm{NTT}(\boldsymbol{t_1} \cdot 2^d))$ where the $\circ$ is component-wise multiplication.
3	Calculate $\boldsymbol{w}'_{\boldsymbol{1}} = \mathrm{HighBits}(\boldsymbol{Az} - c\boldsymbol{t})$	$\boldsymbol{w}'_{\boldsymbol{1}} = \mathrm{UseHint}(\boldsymbol{h}, \boldsymbol{w}'_{\boldsymbol{Approx}})$
4	Calculate $\tilde{c}' = \mathrm{SHAKE256}(\mu \| \boldsymbol{w}'_{\boldsymbol{1}})$ // the $\tilde{c}'$ is a bit-string	
5	Output: valid if $\tilde{c} == \tilde{c}'$, else invalid // the $\tilde{c}$ is received and the $\tilde{c}'$ calculated.	

In line #3 of Verify_internal().SMV,

$$\begin{aligned}\boldsymbol{w}'_{\boldsymbol{1}} = \mathrm{HighBits}(\boldsymbol{Az} - c\boldsymbol{t}) &= \mathrm{HighBits}\big(\boldsymbol{A}(\boldsymbol{y} + c\boldsymbol{s_1}) - c(\boldsymbol{As_1} + \boldsymbol{s_2})\big)\\ &= \mathrm{HighBits}(\boldsymbol{Ay} + \boldsymbol{A}c\boldsymbol{s_1} - \boldsymbol{A}c\boldsymbol{s_1} - c\boldsymbol{s_2})\\ &= \mathrm{HighBits}(\boldsymbol{Ay} - c\boldsymbol{s_2})\end{aligned}$$

On the other hand, in line #3 of Sign_internal().SMV, $\boldsymbol{w_1} = \mathrm{HighBits}(\boldsymbol{Ay})$. So, when $\boldsymbol{s_2}$ is small, the rounded result leads to $\boldsymbol{w}'_{\boldsymbol{1}} = \boldsymbol{w_1}$, resulting in $\tilde{c} = \tilde{c}'$ if the message μ is not tampered.

Line #2 and line #3 of Verify_internal().SSV implement the rounding operation. The higher part of $\boldsymbol{t}$ (i.e., $\boldsymbol{t_1}$) is used in line #2 to work out an approximate rounded value of $(\boldsymbol{Az} - c\boldsymbol{t})$ as $\boldsymbol{w}'_{\boldsymbol{Approx}}$. The $\boldsymbol{w}'_{\boldsymbol{Approx}}$ is further rendered with $\boldsymbol{h}$ to get $\boldsymbol{w}'_{\boldsymbol{1}}$ in line #3 to achieve the final rounded value of $(\boldsymbol{Az} - c\boldsymbol{t})$. The $\boldsymbol{w}'_{\boldsymbol{1}}$ in line #3 of Verify_internal().SSV matches the $\boldsymbol{w_1}$ in line #3 of Sign_internal().

The key point is that although the two results, $\boldsymbol{Ay}$ and $\boldsymbol{Az} - c\boldsymbol{t}$, are different, after the rounding operation, their rounded results, $\mathrm{HighBits}(\boldsymbol{Ay})$ and $\mathrm{HighBits}(\boldsymbol{Az} - c\boldsymbol{t})$, are the same.

4. The refinement in FIPS 204

By contrasting the SMV and the SSV as illustrated in section

3. Two simplified FIPS 204 versions, the SSV performs the rounding operation by using the higher part of $\boldsymbol{t}$ (i.e., the $\boldsymbol{t_1}$) in line #2 of Verify_internal() and then improving the result with the hint vector $\boldsymbol{h}$ in line #3 of Verify_internal(). The hint vector $\boldsymbol{h}$ is calculated with the lower part of $\boldsymbol{t}$ (i.e., the $\boldsymbol{t_0}$) in line #8 of Sign_internal() which “hints” whether a carry has happened during the signing process. So, the $\boldsymbol{h}$ is included in the signature to “inform” Verify_internal() to improve the rounded value as shown in line #3 of Verify_internal().

The use of $\boldsymbol{t}_1$ makes the public key being compressed from $pk = (\boldsymbol{A}, \boldsymbol{t})$ to $pk = (\boldsymbol{A}, \boldsymbol{t}_1)$ but the $\boldsymbol{h}$ makes the signature being expanded from $= (\boldsymbol{z}, \tilde{c})$ to $\sigma = \mathrm{sigEncode}(\boldsymbol{z}, \tilde{c}, \boldsymbol{h})$. Because an entry of $\boldsymbol{h}$ only has two possible values: 0 or 1, the $\boldsymbol{h}$ can be compressed and coded in a small bit-string according to Algorithm 39 MakeHint() and Algorithm 26 sigEncode() in the FIPS 204 ML-DSA specification.

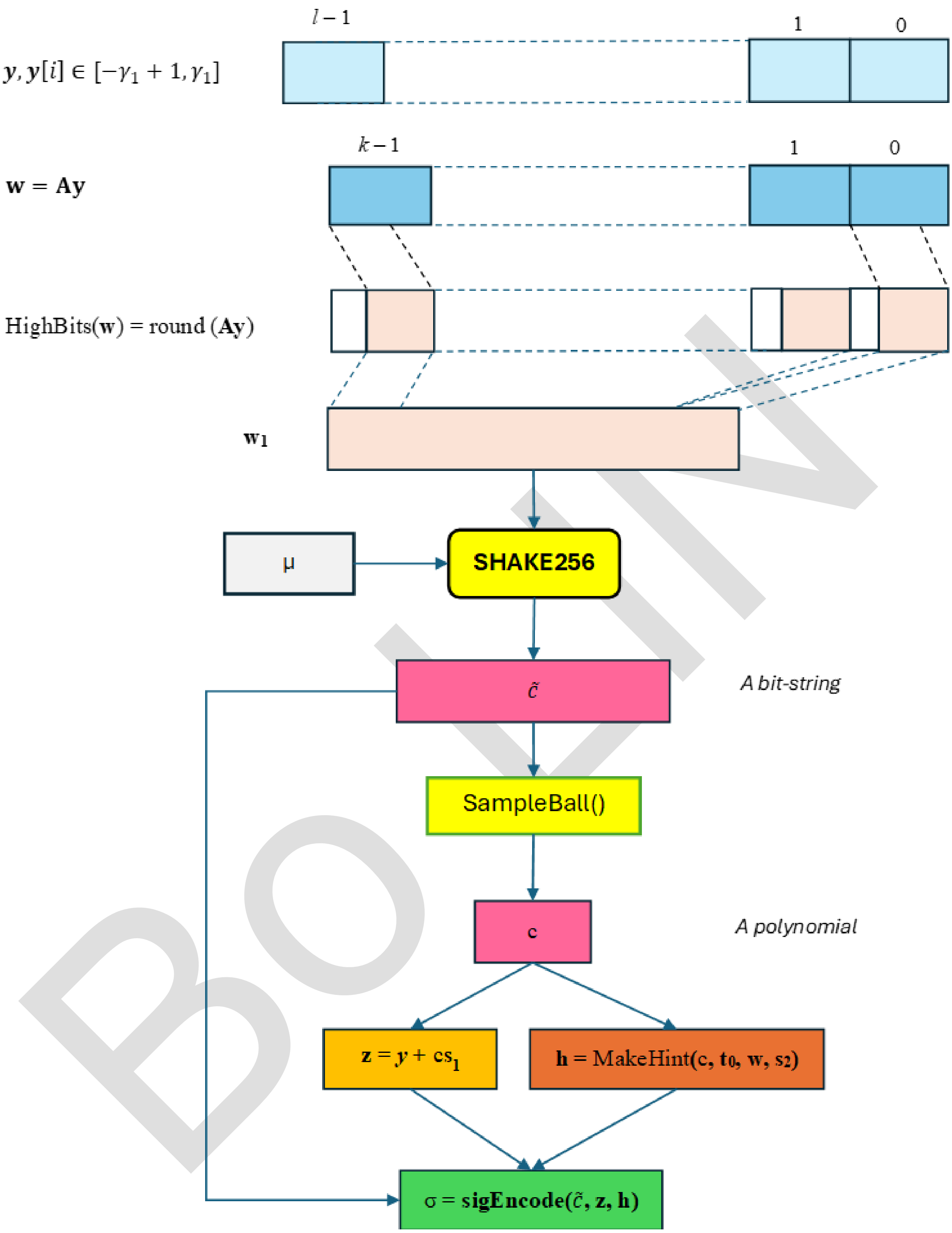
In the FIPS 204 standard, the $\boldsymbol{A}$ is compressed by having its NTT representation (i.e., $\widehat{\boldsymbol{A}}$) generated from a public random 32-byte seed ρ. The 32-byte ρ represents an $k \times l$ matrix $\widehat{\boldsymbol{A}}$ through Algorithm 32 ExpandA() in the FIPS 204 ML-DSA specification. This leads to $pk = (\rho, \boldsymbol{t_1})$ that is a significant reduction of the public key size. The reduction on $\boldsymbol{A}$ and $\boldsymbol{t}$ much overweighs the small expansion of the signature size by appending the coded $\boldsymbol{h}$.

In addition to the data compression above, security check is also considered. The result on line #6 of Sign_internal() is checked with $\boldsymbol{s_2}$ and γ_1 on line #7. If the signature result does not meet a set of security criteria, the signature is discarded and a new signature is generated until a legitimate signature is generated (see line #23 of Algorithm 7 ML-DSA.Sign_internal() of the FIPS 204 ML-DSA specification).

In the FIPS 204 ML-DSA specification, some loops are not bounded, these unbounded loops seem to cause an infinitive loop in the signature generation, leading to a potential “signature failure”, that is, no signature will be generated, but, as discussed in *Appendix C – Loop Bounds* of the FIPS 204 ML-DSA specification, the failure probability is negligible (<2^{-256}).

5. Summary

This technical note concludes with a simplified flow-chat that illustrates the key steps of the Sign_internal() in the FIPS 204 ML-DSA specification.



$l-1$
1
0
$y, y[i] \in [-\gamma_1 + 1, \gamma_1]$
$k-1$
1
0
$\mathbf{w} = \mathbf{Ay}$
HighBits(w) = round (Ay)
$\mathbf{w}_1$
μ
SHAKE256
$\tilde{c}$
A bit-string
SampleBall()
c
A polynomial
$\mathbf{z} = y + cs_1$
$\mathbf{h}$ = MakeHint(c, $\mathbf{t}_0$, w, $\mathbf{s}_2$)
σ = sigEncode($\tilde{c}$, z, h)