

Post-Quantum Cryptography: Securing Blockchains with Dilithium

Mike Mu
Hacken

June 6, 2025

Abstract

Quantum computing threatens the cryptographic foundations of blockchain technologies, which rely on elliptic curve digital signature algorithms (ECDSA) vulnerable to quantum attacks. Post-quantum cryptography (PQC) offers quantum-resistant solutions to secure decentralized systems. This article, the first in a series showcasing our firm's expertise in PQC auditing, focuses on Dilithium, a NIST-standardized lattice-based digital signature scheme (ML-DSA). We explore the quantum threat to blockchains, introduce key PQC signature algorithms, and provide a detailed analysis of Dilithium's design, comparing it to classical Schnorr signatures. We also discuss implementation challenges and auditing strategies, emphasizing our capability to ensure security and efficiency in quantum-resistant blockchain systems.

1 The Quantum Computing Threat

Quantum computers, leveraging Shor's algorithm, can efficiently solve the discrete logarithm problem, compromising the security of the Elliptic Curve Digital Signature Algorithm (ECDSA) used by blockchains like Bitcoin, Ethereum, and Solana [4]. This vulnerability could allow adversaries to forge signatures, steal funds, or disrupt consensus mechanisms, endangering billions in digital assets [5]. For example, an attacker with a sufficiently powerful quantum computer could derive private keys from public keys, undermining the integrity of blockchain transactions.

The National Security Agency (NSA) and the U.S. government have issued mandates urging a rapid transition to post-quantum cryptography (PQC) to mitigate this threat [3]. While some argue that quantum computers capable of breaking ECDSA are still years away [6], the NSA's expertise in cryptographic risk assessment suggests that proactive migration is prudent. As auditors specializing in blockchain security, we advocate following this guidance, given the lack of better information to contradict the NSA's recommendations.

2 Post-Quantum Digital Signature Algorithms

The National Institute of Standards and Technology (NIST) has standardized several PQC digital signature algorithms designed to resist both classical and quantum attacks [3]. These algorithms are critical for securing blockchain systems in a post-quantum era. Key candidates include:

- **Dilithium (ML-DSA):** A lattice-based scheme leveraging the Module Learning With Errors (MLWE)

problem, offering efficient signing and verification without complex Gaussian sampling [1].

- **SPHINCS+**: A stateless hash-based scheme relying on cryptographic hash functions, providing robust security but larger signatures [7].
- **Falcon**: A lattice-based scheme based on the Short Integer Solution (SIS) problem, known for compact signatures and fast verification [8].
- **XMSS**: A stateful hash-based scheme standardized earlier but not recommended for new applications due to state management challenges [9].

Dilithium is particularly promising for blockchain applications due to its balance of security, efficiency, and implementation simplicity, making it a focal point for quantum-secure decentralized systems.

3 Introduction to Dilithium

Dilithium, standardized by NIST as the Module-Lattice-Based Digital Signature Algorithm (ML-DSA), is a cornerstone of PQC [1]. Designed to resist quantum attacks, it leverages the hardness of the MLWE problem, a lattice-based cryptographic primitive. Dilithium adapts the classical Schnorr signature framework to a lattice context using the Fiat-Shamir with Aborts technique, ensuring security against chosen-message attacks [2].

To understand Dilithium's structure, we first examine the Schnorr signature scheme, a classical algorithm widely used in elliptic curve cryptography.

3.1 Schnorr Signature Scheme in Classic ECC

The Schnorr signature scheme is an efficient digital signature algorithm operating over an elliptic curve group [10]. Its simplicity and security make it a benchmark for modern signature schemes.

1. Key Generation:

- Choose a random private key $x \in \mathbb{Z}_q$, where q is the order of the elliptic curve group.
- Compute the public key $y = x \cdot G$, where G is the group generator.

2. Signing:

- For a message m , select a random nonce $k \in \mathbb{Z}_q$ and compute $r = k \cdot G$.
- Compute the challenge $c = H(r||m)$, where H is a cryptographic hash function.
- Compute the signature $s = k + c \cdot x \mod q$.
- Output the signature (r, s) .

3. Verification:

- Compute $c' = H(r||m)$.
- Verify that $r = s \cdot G - c' \cdot y$.

The security of Schnorr signatures relies on the discrete logarithm problem, which is intractable for classical computers but vulnerable to quantum attacks via Shor's algorithm [4].

3.2 Dilithium as a Lattice-Based Schnorr Signature

Dilithium adapts the Schnorr signature framework to a lattice-based setting, replacing the elliptic curve group with a module lattice over a polynomial ring. It uses the Fiat-Shamir with Aborts technique to create a non-interactive signature scheme secure against chosen-message attacks [1].

Following NIST's FIPS 204 notation [1], let:

$R_q = \mathbb{Z}_q[X]/(X^n + 1)$ be the polynomial ring, where q is a prime modulus (e.g., $q = 2^{23} - 2^{13} + 1$) and $n = 256$.

$A \in R_q^{k \times l}$ be a matrix of dimension $k \times l$, where k and l are security parameters (e.g., $k = 8, l = 7$ for ML-DSA-87).

$s_1 \in R_q^l, s_2 \in R_q^k$ be secret key vectors with coefficients bounded by a small norm.

The public key is (A, t) , where $t = A \cdot s_1 + s_2$.

The core idea of Dilithium's signing and verification process can be described as a naive implementation:

1. Key Generation (ML-DSA.KeyGen):

- Generate a random seed $\xi \in \{0, 1\}^{256}$.
- Use ξ to expand into matrix $A \in R_q^{k \times l}$ and vectors $s_1 \in R_q^l, s_2 \in R_q^k$ with small coefficients.
- For ML-DSA-87, the coefficients of s_1 and s_2 are uniformly sampled from $[-2, 2] = \{-2, -1, 0, 1, 2\}$, ensuring small norms critical for the security of the Module Learning With Errors (MLWE) problem.
- Compute $t = A \cdot s_1 + s_2$.
- Output private key (s_1, s_2) and public key (A, t) .

2. Signing (ML-DSA.Sign):

- For a message m , generate a random vector $y \in R_q^l$ with coefficients in $[-\gamma_1, \gamma_1]$.

- Compute the commitment $w = A \cdot y$.
- Compute the challenge $c = H(\mu || \text{highbits}(w))$, where $\mu = H(\text{public key} || m)$, H is SHAKE-256, and highbits extracts high-order coefficients.
- Compute the response $z = y + c \cdot s_1$.
- Apply rejection sampling: if $\|z\|_\infty \geq \gamma_1 - \beta$ or $\text{lowbits}(w, c) \geq \gamma_2 - \beta$, restart.
- Output the signature (z, c) .

3. Verification (ML-DSA.Verify):

- Compute $w' = A \cdot z - c \cdot t$.
- Verify that $c = H(\mu || \text{highbits}(w'))$ and $\|z\|_\infty < \gamma_1 - \beta$.
- *Why use $\text{highbits}(w')$?*
 - **Rationale:** The verifier cannot compute the original commitment $w = A \cdot y$ directly, as it requires the private key component s_2 . Since $z = y + c \cdot s_1$, we have:

$$A \cdot z = A \cdot (y + c \cdot s_1) = A \cdot y + c \cdot (A \cdot s_1) = w + c \cdot (A \cdot s_1).$$

Given $t = A \cdot s_1 + s_2$, so $A \cdot s_1 = t - s_2$, this becomes:

$$A \cdot z = w + c \cdot (t - s_2).$$

Thus:

$$w' = A \cdot z - c \cdot t = [w + c \cdot (t - s_2)] - c \cdot t = w - c \cdot s_2.$$

Since s_2 is private, the verifier cannot compute $w = w' + c \cdot s_2$. However, $s_2 \in S_\eta$ has small coefficients (e.g., in $[-2, 2]$ for ML-DSA-87), and c has at most $\tau = 60$ coefficients of ± 1 , so $c \cdot s_2$ is small (coefficients bounded by $\tau \cdot \eta = 120$). Thus, $w' \approx w$. The $\text{highbits}(w')$ extracts high-order coefficients, which are minimally affected by $c \cdot s_2$, allowing the verifier to check $\text{highbits}(w') \approx \text{highbits}(w)$ via $c = H(\mu || \text{highbits}(w'))$.

- **Ensuring small $c \cdot s_2$:** Key generation ensures s_2 has coefficients in $[-\eta, \eta]$, keeping $c \cdot s_2$ small. For ML-DSA-87, $\eta = 2$, so $c \cdot s_2$ coefficients are at most $60 \cdot 2 = 120$, ensuring $\text{highbits}(w')$ is a good approximation of $\text{highbits}(w)$.
- **Highbits concept:** The highbits function splits a coefficient $r \in \mathbb{Z}_q$ into high and low parts relative to $\alpha = 2 \cdot \gamma_2$ (e.g., $\alpha = 523776$ for ML-DSA-87). It computes $r_1 = \lfloor (r \bmod {}^\pm q) / \alpha \rfloor$, discarding low-order bits. *Example:* Consider a 4-bit number $r = 15$ (binary: 1111), and cut off the last 2 bits (like dividing by $\alpha = 2^2 = 4$):

$$* \text{ Compute } \lfloor 15/4 \rfloor = \lfloor 3.75 \rfloor = 3.$$

* Multiply back: $3 \cdot 4 = 12$ (binary: 1100).

* The highbits is 12, preserving the first 2 bits (11) with error $15 - 12 = 3$.

In Dilithium, $\text{highbits}(w')$ discards small contributions from $c \cdot s_2$, ensuring a reliable approximation for verification.

The security of Dilithium relies on the MLWE problem, which assumes that distinguishing $(A, A \cdot s_1 + s_2)$ from a random pair is computationally hard, even for quantum computers [1]. The Fiat-Shamir transform ensures non-interactivity, while rejection sampling prevents leakage of secret key information.

3.3 Why The Native Implementation Need Fixes

The naive implementation of Dilithium captures its core idea but is insufficient for production use due to several security and efficiency challenges:

- **Parameter Tuning:** Parameters such as n , q , k , l , γ_1 , γ_2 , and β must be carefully selected to achieve NIST security levels (e.g., 128-bit, 192-bit, or 256-bit). Incorrect choices can weaken security or degrade performance [1].
- **Rejection Sampling:** The signing algorithm relies on rejection sampling to ensure z has small coefficients, preventing information leakage about s_1 . Inefficient or incorrect sampling can lead to excessive rejections (reducing performance) or security vulnerabilities [2].
- **Uniform Distribution:** Dilithium uses uniform sampling instead of Gaussian sampling to avoid side-channel attacks. However, precise polynomial arithmetic and random number generation are critical to maintain security [11]. Uniform sampling is superior for side-channel resistance because it employs constant-time integer operations, avoiding the data-dependent branching and floating-point arithmetic of Gaussian sampling, which are vulnerable to timing and power analysis attacks. This ensures that sampling s_1 , s_2 , or y leaks minimal information via execution time or power consumption.
- **Performance Optimizations:** Naive implementations are computationally intensive due to the complexity of polynomial multiplication, matrix-vector operations, rejection sampling, modular arithmetic, and pseudorandom generation. For example, naive polynomial multiplication in $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$ requires $256^2 = 65536$ operations per polynomial, and computing $A \cdot s_1$ for ML-DSA-87 ($k = 8$, $l = 7$) involves 56 such multiplications. Matrix-vector operations involve thousands of coefficient multiplications modulo $q = 2^{23} - 2^{13} + 1$, using costly division-based reductions. Rejection sampling may require multiple iterations, each recomputing $A \cdot y$, while expanding A using SHAKE-128 generates large pseudorandom streams (e.g., 43 KB for ML-DSA-87). Production systems optimize with Number Theoretic Transform (NTT) for $O(n \log n)$ polynomial multiplication, AVX2 vector instructions for parallel coefficient operations, and efficient SHAKE-128 implementations for matrix expansion [2, 1].
- **Side-Channel Resistance:** Naive implementations may be vulnerable to timing, power, or fault injection attacks. Countermeasures, such as constant-time arithmetic and masking, are essential for secure deployment [1].

These challenges highlight the critical role of auditing in Dilithium implementations. Our firm’s expertise in cryptographic design and code review ensures secure parameter selection, efficient rejection sampling, and robust side-channel protections, making us ideal partners for quantum-secure blockchain projects.

4 Auditing Dilithium Implementations

To ensure the security and efficiency of Dilithium implementations, a comprehensive audit must address the challenges identified in Section 3.3. ML-DSA-44, ML-DSA-65, and ML-DSA-87 are predefined parameter sets for the ML-DSA (Dilithium-based) signature scheme, each targeting a different NIST security level: Level 1, Level 3, and Level 5, respectively. These levels correspond to the estimated effort required to break the scheme using known attacks, with Level 1 offering security comparable to brute-forcing AES-128, Level 3 to AES-192, and Level 5 to AES-256. Each profile adjusts key sizes, signature lengths, and other internal parameters to meet its targeted level of cryptographic strength. ML-DSA-44 and ML-DSA-65 differ from ML-DSA-87 primarily in using smaller parameters that offer lower security guarantees. Therefore, for the purpose of this article, we will focus on ML-DSA-87 and its security properties as specified in NIST FIPS 204. [1].

4.1 Parameters of ML-DSA-87

The ML-DSA-87 parameter set defines the cryptographic and performance characteristics of Dilithium, targeting a high security level (256-bit quantum security). Each parameter must be audited for correctness, as deviations can compromise security, performance, or size. Below, we list each parameter for ML-DSA-87, its value, meaning, and importance, as defined in FIPS 204 [1]:

- $\mathbb{Z}_q$, **mod q** , **mods q** , **symmetric q** :
 - *Value*: $\mathbb{Z}_q = \{0, 1, \dots, q - 1\}$, where $q = 2^{23} - 2^{13} + 1$. Operations **mod q** reduce integers to $[0, q - 1]$. **mods q** (symmetric modulo) maps to $[-q/2, q/2] \approx [-4190208, 4190208]$.
 - *Meaning*: $\mathbb{Z}_q$ is the integer ring modulo q , used for all arithmetic in Dilithium. **Mod q** ensures coefficients stay within $[0, q - 1]$, while **mods q** (symmetric representation) centers coefficients around 0 for error terms and norm computations.
 - *Importance*: The choice of q ensures a large field for MLWE security, while **mods q** simplifies error analysis in lattice problems. Incorrect modulo operations can lead to arithmetic errors, weakening security or causing verification failures.
 - *Audit Focus*:
Verify that all arithmetic operations use $q = 2^{23} - 2^{13} + 1$, with correct **mod q** reductions. Ensure **mods q** is used in norm computations (e.g., $\|z\|_\infty$) and high/low bit extractions.
- $R_q = \mathbb{Z}_q[X]/(X^n + 1)$:
 - *Value*: $n = 256, q = 2^{23} - 2^{13} + 1$.

- *Meaning:* R_q is the polynomial ring of degree < 256 , with coefficients in $\mathbb{Z}_q$, quotiented by the cyclotomic polynomial $X^{256} + 1$. Elements are polynomials of the form $a_0 + a_1X + \dots + a_{255}X^{255}$, reduced modulo $X^{256} + 1$.
 - *Importance:* R_q defines the algebraic structure for Dilithium’s lattice operations, enabling efficient polynomial multiplication via Number Theoretic Transform (NTT). The choice of $X^{256} + 1$ ensures a field structure, critical for MLWE security and performance. Incorrect ring operations can break the scheme.
 - *Audit Focus:*
Ensure all polynomial operations respect $X^{256} + 1$ and $q = 2^{23} - 2^{13} + 1$.
Verify NTT implementations for correctness and efficiency.
- S_η :
 - *Value:* $\eta = 2$ for ML-DSA-87.
 - *Meaning:* S_η is the set of polynomials in R_q with coefficients in $[-\eta, \eta] = [-2, 2]$, reduced modulo q .
 - *Importance:* Used for secret key vectors s_1, s_2 , small coefficients ensure the MLWE problem is hard, enhancing security. For ML-DSA-87, $\eta = 2$ balances security and performance, requiring precise sampling. Larger η reduces security.
 - *Audit Focus:*
Verify that s_1, s_2 coefficients are uniformly sampled in $[-2, 2]$.
Check for bounds violations in key generation.
- $\tilde{S}_{\gamma_1}$:
 - *Value:* $\gamma_1 = 2^{19} = 524288$ for ML-DSA-87.
 - *Meaning:* $\tilde{S}_{\gamma_1}$ is the set of polynomials in R_q with coefficients in $(-\gamma_1, \gamma_1] = (-524288, 524288]$, excluding $-\gamma_1$. This ensures an even number of values ($2 \cdot \gamma_1 = 1048576$) for uniform sampling.
 - *Importance:* Used for the random vector y in signing, the large range masks $c \cdot s_1$ in $z = y + c \cdot s_1$, preventing key leakage. The exclusion of $-\gamma_1$ simplifies uniform sampling, avoiding bias. A smaller γ_1 increases rejection rates; a larger γ_1 increases signature size.
 - *Audit Focus:*
Ensure y is sampled uniformly in $(-524288, 524288]$.
Verify rejection sampling checks $\|z\|_\infty < \gamma_1 - \beta$.

- (k, l) **with** $k \geq l$:
 - *Value*: $k = 8, l = 7$ for ML-DSA-87.
 - *Meaning*: k is the number of rows, and l is the number of columns in the public key matrix $A \in R_q^{k \times l}$. The condition $k \geq l$ ensures the matrix is not underdetermined, supporting MLWE security.
 - *Importance*: k affects security (larger k increases MLWE hardness) and public key size (2592 bytes). l impacts signature size (4627 bytes) and signing speed. For ML-DSA-87, $k = 8, l = 7$ provides 256-bit quantum security. The constraint $k \geq l$ prevents weak matrix structures.
 - *Audit Focus*:
Verify A is 8×7 . Ensure matrix-vector operations (e.g., $A \cdot s_1$) respect these dimensions.
- B_τ :
 - *Value*: $\tau = 60$ for ML-DSA-87.
 - *Meaning*: B_τ is the set of polynomials in S_1 (coefficients in $\{-1, 0, 1\}$) with exactly $\tau = 60$ coefficients equal to ± 1 , the rest 0.
 - *Importance*: Defines the challenge polynomial c , where sparsity (60 non-zero coefficients) balances security and efficiency. A larger τ increases signing time; a smaller τ reduces security.
 - *Audit Focus*:
Ensure c has exactly 60 coefficients of ± 1 , generated via SHAKE-256.
Verify the sampling algorithm produces valid B_τ polynomials.
- $\beta = \tau \cdot \eta$:
 - *Value*: $\beta = 60 \cdot 2 = 120$ for ML-DSA-87 ($\tau = 60, \eta = 2$).
 - *Meaning*: The bound for coefficients of $c \cdot s_1$, where $c \in B_\tau$ and $s_1 \in S_\eta$.
 - *Importance*: Determines the size of $c \cdot s_1$ in $z = y + c \cdot s_1$, affecting rejection sampling. A larger β risks key leakage; a smaller β increases rejections, slowing signing. For ML-DSA-87, $\beta = 120$ balances security and performance.
 - *Audit Focus*: Verify $\beta = 120$ in rejection sampling checks ($\|z\|_\infty < \gamma_1 - \beta$).

- γ_2 :
 - *Value*: $\gamma_2 = (q - 1)/32 = 261888$ for ML-DSA-87.
 - *Meaning*: The bound for low-order bits in rejection sampling of $w = A \cdot y - c \cdot s_2$.
 - *Importance*: Ensures low-order bits do not leak private key information. A smaller γ_2 increases rejections, slowing signing; a larger γ_2 risks leakage. For ML-DSA-87, $\gamma_2 = 261888$ balances security and signing efficiency.
 - *Audit Focus*:
Check that $\text{lowbits}(w, c) < \gamma_2 - \beta$ is enforced in rejection sampling.
- $\alpha = 2 \cdot \gamma_2$:
 - *Value*: $\alpha = 2 \cdot 261888 = 523776$ for ML-DSA-87.
 - *Meaning*: The scaling factor for high/low bit decomposition in w and w' .
 - *Importance*: Used in $\text{highbits}(w, \alpha)$ and $\text{lowbits}(w, \alpha)$, α determines the precision of bit splitting. A larger α reduces the number of high bits, shrinking signatures; a smaller α increases rejection rates. For ML-DSA-87, $\alpha = 523776$ optimizes signature size (4627 bytes).
 - *Audit Focus*:
Verify that $\alpha = 523776$ is used in high/low bit algorithms.
- λ :
 - *Value*: $\lambda = 256$ for ML-DSA-87.
 - *Meaning*: Half the bitlength of the signature component $\tilde{c}$, the hash of the challenge polynomial c .
 - *Importance*: Determines the security level of the Fiat-Shamir transform, as $\tilde{c}$ is a 512-bit hash. For ML-DSA-87, $\lambda = 256$ ensures 256-bit security against collision attacks. A smaller λ reduces security.
 - *Audit Focus*:
Ensure $\tilde{c}$ is a 512-bit SHAKE-256 output, with $\lambda = 256$ bits defining its security.
- $d = 13$:
 - *Value*: $d = 13$ for ML-DSA-87.

- *Meaning*: The number of bits dropped from t to form t_0 (low bits), with $t_1 = \text{highbits}(t, 2^d)$.
 - *Importance*: Reduces public key size by storing only t_1 , with t_0 kept private. For ML-DSA-87, $d = 13$ (i.e., $2^{13} = 8192$) balances key size (2592 bytes) and security. A larger d increases key size; a smaller d risks leakage.
 - *Audit Focus*:
Verify $t = t_1 \cdot 2^{13} + t_0$. Ensure t_0 is private and t_1 is correctly computed.
- $\omega = 75$:
 - *Value*: $\omega = 75$ for ML-DSA-87.
 - *Meaning*: The maximum number of 1's in the high-bit array of $w' = A \cdot z - c \cdot t_1 \cdot 2^d$ during verification.
 - *Importance*: Limits signature size by bounding the number of non-zero coefficients in $\text{highbits}(w', \alpha)$. For ML-DSA-87, $\omega = 75$ reduces signature size (4627 bytes) while maintaining verification reliability. A larger ω increases size; a smaller ω rejects valid signatures.
 - *Audit Focus*:
Ensure verification checks that $\text{highbits}(w', \alpha)$ has at most 75 non-zero coefficients.
- $n = 256$:
 - *Value*: $n = 256$ for ML-DSA-87.
 - *Meaning*: The degree of polynomials in R_q , defining the number of coefficients.
 - *Importance*: Balances security and performance. $n = 256$ enables efficient NTT-based polynomial multiplication while supporting MLWE security. A smaller n reduces security; a larger n increases computational cost.
 - *Audit Focus*:
Verify all polynomials have degree < 256 , respecting $X^{256} + 1$.
- $q = 2^{23} - 2^{13} + 1$:
 - *Value*: $q = 2^{23} - 2^{13} + 1$.
 - *Meaning*: The prime modulus of $\mathbb{Z}_q$ and R_q .
 - *Importance*: Ensures a large field for MLWE security and efficient arithmetic. A smaller q weakens security; a larger q increases key/signature sizes.

- *Audit Focus:*
Ensure all modular reductions use $q = 2^{23} - 2^{13} + 1$.
- $k = 8$:
 - *Value:* $k = 8$.
 - *Meaning:* Number of rows in $A \in R_q^{k \times l}$.
 - *Importance:* Increases MLWE security and public key size (2592 bytes). For ML-DSA-87, $k = 8$ achieves 256-bit security.
 - *Audit Focus:*
Verify A has 8 rows.
- $l = 7$:
 - *Value:* $l = 7$.
 - *Meaning:* Number of columns in $A \in R_q^{k \times l}$.
 - *Importance:* Affects signature size (4627 bytes) and signing speed. $l = 7$ balances performance and size.
 - *Audit Focus:*
Verify A has 7 columns.
- $\gamma_1 = 2^{19}$:
 - *Value:* $\gamma_1 = 524288$.
 - *Meaning:* Bound for y coefficients and z in rejection sampling.
 - *Importance:* Controls signature size and signing efficiency. A larger γ_1 increases size; a smaller γ_1 increases rejections.
 - *Audit Focus:*
Ensure $y \in (-\gamma_1, \gamma_1]$ and $\|z\|_\infty < \gamma_1 - \beta$.

Auditing these parameters involves verifying their implementation in code (e.g., Rust or C), ensuring they match FIPS 204 values for ML-DSA-87, and checking that arithmetic operations respect their bounds and properties.

4.2 Rejection Sampling

Rejection sampling is a critical component of Dilithium’s signing algorithm, ensuring that the signature (z, c) does not leak information about the private key s_1 . During signing, the response $z = y + c \cdot s_1$ is computed, where y is a random vector with coefficients in $[-\gamma_1, \gamma_1]$, and $c \cdot s_1$ has coefficients bounded by β . If z ’s coefficients are too large ($\|z\|_\infty \geq \gamma_1 - \beta$) or the low-order bits of $w = A \cdot y$ are too large ($\text{lowbits}(w, c) \geq \gamma_2 - \beta$), the signature is rejected, and the process restarts with a new y .

Importance: Without rejection sampling, the distribution of z could reveal information about s_1 , allowing an attacker to recover the private key through statistical analysis over multiple signatures. Rejection sampling ensures that z is statistically indistinguishable from a uniformly random vector, preserving zero-knowledge properties.

Audit Focus:

- Verify that rejection sampling checks both $\|z\|_\infty < \gamma_1 - \beta$ and $\text{lowbits}(w, c) < \gamma_2 - \beta$.
- Ensure the random number generator for y is cryptographically secure (e.g., using SHAKE-256).
- Check for implementation errors, such as early termination of rejections, which could leak partial information.
- Measure rejection rates to confirm they align with expected probabilities (e.g., low rejection rates for ML-DSA-87).

4.3 Uniform Distribution Parameters

Dilithium uses uniform sampling for the random vector y and secret keys s_1, s_2 , avoiding Gaussian sampling to mitigate side-channel attacks. The parameters enabling uniform distribution are:

- $\gamma_1 = 2^{19}$: Defines the range $(-\gamma_1, \gamma_1]$ for y , ensuring uniform sampling of large coefficients to mask $c \cdot s_1$.
- $\eta = 2$: Bounds s_1, s_2 coefficients in $[-2, 2]$, allowing uniform sampling of small coefficients to maintain security.
- $q = 2^{23} - 2^{13} + 1$: The large modulus ensures that uniform sampling over $\mathbb{Z}_q$ provides sufficient entropy without requiring Gaussian distributions.

Why Uniform Over Gaussian? Gaussian sampling, used in earlier lattice-based schemes, requires precise floating-point operations, which are susceptible to timing and power analysis attacks. Uniform sampling simplifies implementation, uses integer arithmetic, and reduces side-channel risks, as it does not depend on complex distribution shapes. However, it requires careful parameter selection to ensure security, as uniform distributions must cover a sufficiently large space to prevent lattice attacks.

Central Binomial Distribution:

For sampling secret keys s_1, s_2 , Dilithium uses the Central Binomial Distribution (CBD) with parameter $\eta = 2$. The CBD generates coefficients by selecting η pairs of bits (a_i, b_i) , where $1 \leq i \leq \eta$, uniformly at random. The coefficient is computed as $c = \sum_{i=1}^{\eta} (a_i - b_i)$, resulting in $c \in [-\eta, \eta] = \{-2, -1, 0, 1, 2\}$. The probability of $c = j$ for $j \in [-2, 2]$ is:

$$\text{Probability}(c = j) = \frac{\binom{2\eta}{\eta+j}}{2^{2\eta}} = \frac{\binom{4}{2+j}}{16}.$$

For example, $P(c = 0) = \frac{\binom{4}{2}}{16} = \frac{6}{16} = 0.375$, and $P(c = \pm 2) = \frac{\binom{4}{4}}{16} = \frac{1}{16} = 0.0625$. This distribution, centered at zero, approximates a Gaussian but is simpler to implement securely. The figure below shows the probability distribution for c .

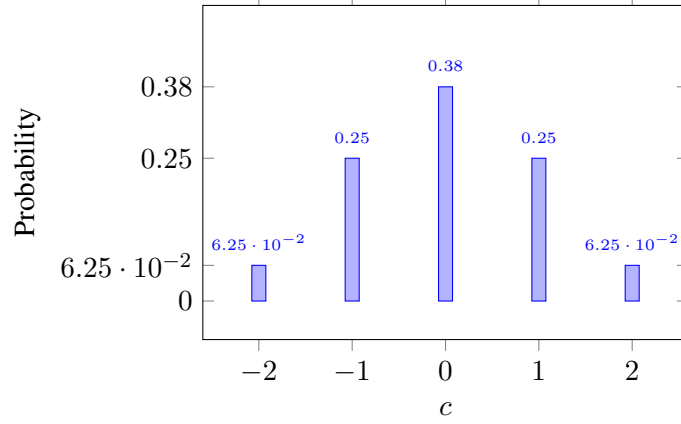


Figure 1: Probability distribution of CBD coefficients for $\eta = 2$.

Audit Focus:

- Verify that y is sampled uniformly in $(-\gamma_1, \gamma_1]$ and s_1, s_2 in $[-\eta, \eta]$.
- Ensure the sampling algorithm (e.g., SHAKE-256) produces uniform outputs without bias.
- Check that modular reductions modulo q preserve uniformity.
- Check that central binomial distribution (CBD) is implemented correctly.

4.4 Error Term and Key Decomposition

The verification step computes $w' = A \cdot z - c \cdot t$, where $z = y + c \cdot s_1$ and $t = A \cdot s_1 + s_2$. Substituting, we get:

$$w' = A \cdot (y + c \cdot s_1) - c \cdot (A \cdot s_1 + s_2) = A \cdot y - c \cdot s_2.$$

Ideally, $w' = A \cdot y$, but the term $-c \cdot s_2$ introduces an error, as s_2 has small coefficients bounded by η . This error is managed by splitting w' into high and low bits and only hashing the high bits to compute the challenge c .

To reduce signature size, the public key t is decomposed into t_1 (high bits) and t_0 (low bits), where:

- $t_1 = \text{highbits}(t, 2^d)$, containing the high-order coefficients.
- $t_0 = \text{lowbits}(t, 2^d)$, containing the low-order coefficients, such that $t = t_1 \cdot 2^d + t_0$.
- For ML-DSA-87, $d = 13$, so $2^d = 8192$.

Why Decompose t ? - The signer keeps t_0 as part of the private key to compute t during signing, but only t_1 is included in the public key, reducing its size (2592 bytes). - During verification, only $\text{highbits}(A \cdot z - c \cdot t_1 \cdot 2^d)$ is used, as the low bits ($c \cdot t_0$) are small and do not affect the high-order coefficients.

Audit Focus:

- Verify that $t = t_1 \cdot 2^{13} + t_0$ is correctly computed.
- Ensure t_0 is kept private and not leaked in the public key.
- Check that verification uses only $\text{highbits}(w')$, ignoring low bits.

4.5 Highbit and Lowbit Algorithms

The highbits and lowbits algorithms round coefficients to powers of 2, specifically $2^d = 2^{13} = 8192$ for ML-DSA-87, to decompose polynomials:

- $\text{highbits}(r, \alpha)$:
- For a coefficient $r \in \mathbb{Z}_q$, compute $r_1 = \lfloor (r \bmod^{\pm} q) / \alpha \rfloor$, where $\bmod^{\pm} q$ maps r to $[-q/2, q/2]$.
- Output r_1 , representing the high-order part.
- $\text{lowbits}(r, \alpha)$:
- Compute $r_0 = (r \bmod^{\pm} q) - \alpha \cdot r_1$.
- Output r_0 , representing the low-order part.
- For $\alpha = 2 \cdot \gamma_2 = 523776$, this splits coefficients relative to 523776.

Audit Focus:

- Ensure highbits and lowbits correctly implement the modulo and division operations.
- Verify that $r = \alpha \cdot \text{highbits}(r, \alpha) + \text{lowbits}(r, \alpha)$.
- Check for constant-time implementation to prevent timing attacks.

4.6 Expansion of A, s_1, s_2 from ψ

The 256-bit random seed $\psi \in \{0, 1\}^{256}$ is expanded using SHAKE-256 to generate the matrix A , secret keys s_1, s_2 , and a secret seed K . The process is:

1. Expand ψ to 1024 bits:
 - Compute $\text{nonce} = \text{SHAKE-256}(\psi)$, producing a 1024-bit string.
2. Split into components:
 - $\rho = \text{nonce}[0 : 256]$: A 256-bit seed for generating A .
 - $\rho' = \text{nonce}[256 : 768]$: A 512-bit seed for generating s_1, s_2 .
 - $K = \text{nonce}[768 : 1024]$: A 256-bit secret seed for signing.
3. Generate A : - Use ρ to seed SHAKE-256, producing coefficients for each entry of $A \in R_q^{8 \times 7}$.
 - Each coefficient is sampled uniformly in $[0, q - 1]$.
4. Generate s_1, s_2 : - Split ρ' into two 256-bit seeds: $\rho'_1 = \rho'[0 : 256]$, $\rho'_2 = \rho'[256 : 512]$.
 - Use ρ'_1 to generate $s_1 \in R_q^7$, sampling coefficients uniformly in $[-2, 2]$.
 - Use ρ'_2 to generate $s_2 \in R_q^8$, sampling coefficients uniformly in $[-2, 2]$.

Audit Focus:

- Verify that SHAKE-256 correctly expands ψ to 1024 bits.
- Ensure ρ, ρ', K are correctly partitioned from the 1024-bit output.
- Check that A coefficients are uniformly sampled in $[0, q - 1]$.
- Confirm s_1, s_2 coefficients are in $[-2, 2]$, using ρ'_1, ρ'_2 .

4.7 Expansion Algorithms for Matrix and Secret Keys

The generation of the public key matrix $A \in R_q^{k \times l}$ and secret key vectors $s_1 \in R_q^l, s_2 \in R_q^k$ is a critical step in Dilithium's key generation process, relying on the cryptographic hash function SHAKE-

256 to expand seeds into uniformly distributed polynomials. The algorithms `ExpandA` and `ExpandS`, defined in NIST FIPS 204 [1], ensure that A , s_1 , and s_2 are generated securely and consistently across implementations. Below, we detail the major steps of these algorithms, focusing on their inputs, outputs, and the role of SHAKE-256.

ExpandA Algorithm:

The `ExpandA` algorithm generates the matrix $A \in R_q^{k \times l}$, where each entry is a polynomial in $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$, with coefficients uniformly sampled in $[0, q - 1]$, where $q = 2^{23} - 2^{13} + 1$. This matrix forms part of the public key and is critical to the security of the Module Learning With Errors (MLWE) problem.

- *Input:* A 256-bit seed $\rho \in \{0, 1\}^{256}$, and parameters k, l (e.g., $k = 8, l = 7$ for ML-DSA-87).
- *Output:* A matrix $A \in R_q^{k \times l}$.
- *Steps:*
 1. Initialize an empty matrix A of size $k \times l$.
 2. For each entry $A[i, j]$ ($0 \leq i < k, 0 \leq j < l$):
 - Form a nonce by concatenating indices: $\text{nonce} = i || j$, where i and j are single-byte values.
 - Compute $\text{stream} = \text{SHAKE-256}(\rho || \text{nonce})$, producing a pseudorandom byte stream.
 - For each of the 256 coefficients of the polynomial $A[i, j]$:
 - * Read 3 bytes from the stream and interpret as a 24-bit integer.
 - * If the integer $\geq 23 \cdot q = 192949391$, reject and read the next 3 bytes (rejection sampling).
 - * Compute the coefficient as $\text{coef} = \text{integer} \bmod q$, ensuring $\text{coef} \in [0, q - 1]$.
 - Assign the 256 coefficients to $A[i, j]$.
 3. Return the matrix A .

ExpandS Algorithm:

The `ExpandS` algorithm generates the secret key vectors $s_1 \in R_q^l$ and $s_2 \in R_q^k$, where each polynomial has coefficients uniformly sampled in $[-\eta, \eta]$, ensuring small norms for MLWE security. For ML-DSA-87, $\eta = 2$, so coefficients are in $\{-2, -1, 0, 1, 2\}$.

- *Input:* A 512-bit seed $\rho' \in \{0, 1\}^{512}$, parameters k, l , and η (e.g., $\eta = 2$ for ML-DSA-87).
- *Output:* Vectors $s_1 \in R_q^l, s_2 \in R_q^k$.
- *Steps:*
 1. Initialize empty vectors s_1 (length l) and s_2 (length k).
 2. For each polynomial in s_1 ($0 \leq j < l$) and s_2 ($l \leq j < l + k$):
 - Set nonce = j , encoded as a single byte.
 - Compute stream = SHAKE-256($\rho' || \text{nonce}$), producing a pseudorandom byte stream.
 - For each of the 256 coefficients of the polynomial:
 - * Read 1 byte from the stream.
 - * For $\eta = 2$, accept bytes in $\{0, 1, 2, 3, 4\}$ and map to $\{-2, -1, 0, 1, 2\}$ (e.g., $0 \rightarrow -2, 1 \rightarrow -1, 2 \rightarrow 0, 3 \rightarrow 1, 4 \rightarrow 2$).
 - * If the byte is outside $\{0, \dots, 2\eta\}$, reject and read the next byte.
 - Assign the 256 coefficients to $s_1[j]$ (if $j < l$) or $s_2[j - l]$ (if $j \geq l$).
 3. Return s_1, s_2 .

Importance:

The ExpandA and ExpandS algorithms ensure that A, s_1 , and s_2 are generated deterministically from seeds ρ and ρ' , enabling reproducible key generation. The use of SHAKE-256 guarantees cryptographic randomness, while rejection sampling ensures uniform distributions within the required bounds. Incorrect implementation could lead to biased coefficients, weakening the MLWE problem's hardness or causing interoperability issues.

Audit Focus:

- Verify that SHAKE-256 is correctly implemented per FIPS 202, producing uniform outputs.
- Ensure nonce concatenation ($\rho || \text{nonce}, \rho' || \text{nonce}$) avoids collisions and respects byte encoding.
- Check rejection sampling in ExpandA enforces integer $< 23 \cdot q$, and in ExpandS maps bytes to $[-\eta, \eta]$.
- Confirm constant-time operations to prevent timing attacks, especially in coefficient sampling.
- Validate that $A \in R_q^{k \times l}, s_1 \in R_q^l, s_2 \in R_q^k$ match the parameter set (e.g., $k = 8, l = 7, \eta = 2$ for ML-DSA-87).

4.8 Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is a specialized form of the Fast Fourier Transform (FFT) adapted for finite fields, used in Dilithium to perform efficient polynomial multiplication in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, where $q = 2^{23} - 2^{13} + 1$ and $n = 256$. This subsection explains the purpose of NTT, its operational mechanism, its reversibility, and key audit considerations.

Purpose:

Polynomial multiplication in R_q is central to Dilithium's operations, such as computing $A \cdot s_1$ during key generation or $A \cdot y$ during signing. Direct multiplication of two degree- n polynomials has a computational complexity of $O(n^2)$, which is inefficient for $n = 256$. The NTT reduces this complexity to $O(n \log n)$ by transforming polynomials into a frequency-like domain, enabling point-wise multiplication, and transforming back to the polynomial domain. This efficiency is critical for Dilithium's performance, particularly in resource-constrained blockchain environments.

How It Works:

The NTT transforms a polynomial into a vector of coefficients suitable for fast multiplication, then reverses the process to recover the product polynomial. The process involves three main steps:

- *Forward NTT (Polynomial to NTT Form):*

- A polynomial $a \in R_q$, represented as $a(X) = a_0 + a_1X + \dots + a_{255}X^{255}$, is transformed into its NTT form, a vector $\hat{a} = (\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{255})$.
- The NTT uses a primitive 512-th root of unity ω in $\mathbb{Z}_q$, satisfying $\omega^{512} \equiv 1 \pmod{q}$ and $\omega^{256} \equiv -1 \pmod{q}$.
- Compute:

$$\hat{a}_i = \sum_{j=0}^{255} a_j \cdot \omega^{ij} \pmod{q}, \quad \text{for } i = 0, 1, \dots, 255.$$

- This is implemented using a Cooley-Tukey FFT-like algorithm with complexity $O(n \log n)$, leveraging precomputed values of ω .

- *Point-Wise Multiplication:*

- For two polynomials $a, b \in R_q$ in NTT form $(\hat{a}, \hat{b})$, compute the NTT of their product:

$$\hat{c}_i = \hat{a}_i \cdot \hat{b}_i \pmod{q}, \quad \text{for } i = 0, 1, \dots, 255.$$

- This produces $\hat{c}$, the NTT form of $c = a \cdot b \pmod{(X^{256} + 1)}$, with complexity $O(n)$.

- *Inverse NTT (NTT Form to Polynomial):*

- Transform $\hat{c}$ back to the polynomial $c \in R_q$:

$$c_j = \frac{1}{256} \sum_{i=0}^{255} \hat{c}_i \cdot \omega^{-ij} \mod q, \quad \text{for } j = 0, 1, \dots, 255.$$

- Here, $\frac{1}{256}$ is the modular inverse of 256 in $\mathbb{Z}_q$, and ω^{-1} is the inverse of $\omega \mod q$.
- The inverse NTT uses an FFT-like algorithm, maintaining $O(n \log n)$ complexity, and ensures reduction modulo $X^{256} + 1$.

Reversibility and Optimization:

The NTT is reversible, satisfying $\text{NTT}^{-1}(\text{NTT}(p)) = p$, due to the properties of ω and modular arithmetic in $\mathbb{Z}_q$. In Dilithium, polynomials such as the matrix A , vectors s_1, s_2 , and intermediate values (e.g., y, z) are often stored in NTT form to optimize both space and computation. Storing A in NTT form avoids repeated transformations during operations like $A \cdot s_1$ or $A \cdot y$, which are performed using point-wise multiplications. This reduces computational overhead, as transformations are only needed when converting inputs or outputs, enhancing performance in signing and verification.

Audit Focus:

- Verify that the NTT uses a correct primitive 512-th root of unity ω , ensuring $\omega^{512} \equiv 1 \mod q$ and $\omega^{256} \equiv -1 \mod q$.
- Ensure all arithmetic operations are performed modulo $q = 2^{23} - 2^{13} + 1$, with proper reductions.
- Check that NTT and inverse NTT implementations are constant-time to prevent timing attacks, particularly in coefficient computations.
- Validate that polynomials transformed to NTT form and back produce results equivalent to direct polynomial multiplication.
- Confirm the implementation respects $n = 256$ and the ring $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$.
- Assess performance optimizations (e.g., butterfly operations, precomputed twiddle factors) for efficiency without compromising security.

4.9 ML-DSA-87 Flow

The complete flow of ML-DSA-87 (KeyGen, Sign, Verify) is detailed below, based on FIPS 204 [1]:

KeyGen (ML-DSA.KeyGen):

1. Generate a 256-bit random seed $\psi \in \{0, 1\}^{256}$.
2. Compute nonce = SHAKE-256(ψ), a 1024-bit string.
3. Set $\rho = \text{nonce}[0 : 256]$, $\rho' = \text{nonce}[256 : 768]$, $K = \text{nonce}[768 : 1024]$.
4. Expand ρ to $A \in R_q^{8 \times 7}$, sampling coefficients in $[0, q - 1]$.
5. Split ρ' into ρ'_1, ρ'_2 , and expand to $s_1 \in R_q^7$, $s_2 \in R_q^8$, with coefficients in $[-2, 2]$.
6. Compute $t = A \cdot s_1 + s_2$.
7. Decompose $t = t_1 \cdot 2^{13} + t_0$, where $t_1 = \text{highbits}(t, 2^{13})$, $t_0 = \text{lowbits}(t, 2^{13})$.
8. Output private key $sk = (\rho, K, t_0, s_1, s_2)$ and public key $pk = (\rho, t_1)$.

Sign (ML-DSA.Sign):

1. Compute $\mu = \text{SHAKE-256}(pk || m)$, where m is the message.

2. Initialize $\kappa = 0$.
3. While no valid signature:
 - Generate $y \in R_q^7$ with coefficients in $(-\gamma_1, \gamma_1]$, using $\text{SHAKE-256}(K||\mu||\kappa)$.
 - Compute $w = A \cdot y$.
 - Compute $w_1 = \text{highbits}(w, 2 \cdot \gamma_2)$.
 - Compute $c = \text{SHAKE-256}(\mu||w_1)$, a polynomial with 60 coefficients of ± 1 .
 - Compute $z = y + c \cdot s_1$.
 - Compute $r_0 = \text{lowbits}(w - c \cdot s_2, 2 \cdot \gamma_2)$.
 - If $\|z\|_\infty \geq \gamma_1 - \beta$ or $\|r_0\|_\infty \geq \gamma_2 - \beta$, increment κ and restart.
4. Output signature $\sigma = (z, c)$.

Verify (ML-DSA.Verify):

1. Compute $\mu = \text{SHAKE-256}(pk||m)$.
2. Compute $w_1 = \text{highbits}(A \cdot z - c \cdot t_1 \cdot 2^{13}, 2 \cdot \gamma_2)$.
3. Compute $c' = \text{SHAKE-256}(\mu||w_1)$.
4. Accept if $c = c'$, $\|z\|_\infty < \gamma_1 - \beta$, and $\text{highbits}(A \cdot z - c \cdot t_1 \cdot 2^{13}, 2 \cdot \gamma_2)$ has at most $\omega = 75$ non-zero coefficients; otherwise, reject.

Audit Focus:

- Verify each step's implementation matches FIPS 204.
- Ensure constant-time operations for KeyGen and Sign to prevent timing attacks.
- Check that verification correctly enforces all conditions, especially coefficient bounds.

4.10 Full Parameter Specification for ML-DSA Variants

To facilitate comprehensive auditing and comparison of ML-DSA implementations, the following table provides the complete parameter sets for ML-DSA-44, ML-DSA-65, and ML-DSA-87, as specified in NIST FIPS 204 [1]. These parameter sets correspond to security levels 2 (256-bit SHA security), 3 (192-bit block cipher security), and 5 (256-bit block cipher security), respectively.

The parameters $\mathbb{Z}_q$, $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, and $\tilde{S}_{\gamma_1}$ are defined consistently across all variants, with $q = 2^{23} - 2^{13} + 1$, $n = 256$, and $\tilde{S}_{\gamma_1}$ covering coefficients in $(-\gamma_1, \gamma_1]$. Auditors must verify that implementations adhere to these values to ensure security and interoperability across ML-DSA variants.

5 Dilithium Implementations

The adoption of Dilithium (ML-DSA) in post-quantum cryptographic systems, particularly for blockchain applications, relies on secure and efficient implementations. This section surveys current open-source Dilithium implementations in various programming languages, focusing on their programming language, repository details, and whether they have undergone formal security audits. Given the high stakes in blockchain systems, where vulnerabilities could lead to significant financial losses, we emphasize the critical need for rigorous auditing.

5.1 Survey of Dilithium Implementations

- **PQ-Crypto (C/C++):**

Table 1: Full Parameter Sets for ML-DSA-44, ML-DSA-65, and ML-DSA-87 [1]

Parameter	ML-DSA-44	ML-DSA-65	ML-DSA-87
Security Level	2 (256-bit SHA)	3 (192-bit block cipher)	5 (256-bit block cipher)
q	$2^{23} - 2^{13} + 1$	$2^{23} - 2^{13} + 1$	$2^{23} - 2^{13} + 1$
n	256	256	256
η	2	4	2
γ_1	$2^{17} = 131072$	$2^{19} = 524288$	$2^{19} = 524288$
k	4	6	8
l	4	5	7
τ	39	49	60
$\beta = \tau \cdot \eta$	78	196	120
γ_2	$(q - 1)/88 = 95232$	$(q - 1)/32 = 261888$	$(q - 1)/32 = 261888$
$\alpha = 2 \cdot \gamma_2$	190464	523776	523776
λ	128	192	256
d	13	13	13
ω	80	55	75
Private Key (bytes)	2560	4032	4896
Public Key (bytes)	1312	1952	2592
Signature Size (bytes)	2420	3309	4627

– *Repository*: pq-crystals/dilithium

– *Description*: The official CRYSTALS-Dilithium reference implementation, developed by the Dilithium team for NIST’s PQC standardization. Written in C, with optimized versions using AVX2 and AES instructions. Supports ML-DSA-44, ML-DSA-65, and ML-DSA-87.

– *Audit Status*: No formal audit report is publicly available. As the reference implementation submitted to NIST, it underwent extensive scrutiny during standardization (2017–2024). The repository mentions testing for correctness and performance but lacks a third-party audit.

– *Notes*: Widely used as a baseline for other implementations. Its adherence to FIPS 204 makes it reliable, but the lack of a formal audit is a concern for blockchain deployments.

• **Open Quantum Safe (liboqs) (C):**

– *Repository*: open-quantum-safe/liboqs

– *Description*: A C library for quantum-resistant algorithms, including Dilithium (ML-DSA-44, ML-DSA-65, ML-DSA-87). Integrates optimized Dilithium implementations and supports multiple platforms. Part of the Open Quantum Safe project for PQC adoption.

– *Audit Status*: No specific audit report for Dilithium. The project notes ongoing security evaluations, with some algorithms (e.g., Kyber) audited, but Dilithium’s audit status is unclear. NIST reviews and community testing provide some assurance.

– *Notes*: Popular for integrating PQC into applications (e.g., OpenSSL forks). Its active main-

tenance makes it a strong candidate, but blockchain use requires a dedicated audit.

- **Rust Crypto (Rust):**

- *Repository:* RustCrypto/signatures
- *Description:* A Rust implementation of Dilithium within the RustCrypto project. Supports ML-DSA variants and leverages Rust's memory safety for security and performance.
- *Audit Status:* No formal audit report. The RustCrypto project encourages community review, and its modular design aids testing, but no third-party audit is documented.
- *Notes:* Rust's safety guarantees make it attractive for blockchain. The lack of an audit report limits its use in high-stakes applications without further verification.

- **Go-PQ-Crypto (Golang):**

- *Repository:* cryptoballot/pq-crypto
- *Description:* A Golang implementation of PQC algorithms, including Dilithium. Designed for simplicity and correctness, supporting ML-DSA variants.
- *Audit Status:* No audit report. The repository is less active (last updated 2023) and has limited community engagement, reducing confidence without formal verification.
- *Notes:* Suitable for prototyping but not recommended for production blockchain systems due to limited maintenance and no audit.

- **Python-PQC (Python):**

- *Repository:* pq-crystals/dilithium-python
- *Description:* A Python reference implementation of Dilithium, maintained by the CRYSTALS team. Prioritizes clarity and correctness for educational and testing purposes.
- *Audit Status:* No formal audit report. As a reference implementation, it benefits from NIST's review but is not optimized for production.
- *Notes:* Useful for understanding Dilithium's algorithms but unsuitable for blockchain due to Python's performance limitations and lack of audit.

5.2 Other Implementations

- **Supersingular (JavaScript)**: The mikelodder7/supersingular repository includes an experimental Dilithium implementation in JavaScript. It lacks an audit report and has limited adoption, making it unsuitable for blockchain applications.
- **PQClean (C)**: PQClean/PQClean provides portable C implementations of PQC algorithms, including Dilithium. No audit report is available, but its focus on portability aids verification. It is a candidate for embedded systems but requires auditing for blockchain use.

5.3 Summary

Table 2: Survey of Dilithium Implementations

GitHub Repository	Programming Language	Audit Report Available
pq-crystals/dilithium	C/C++	No (NIST scrutiny)
open-quantum-safe/liboqs	C	No (partial project evaluations)
RustCrypto/signatures	Rust	No
cryptoballot/pq-crypto	Golang	No
pq-crystals/dilithium-python	Python	No (NIST scrutiny)
mikelodder7/supersingular	JavaScript	No
PQClean/PQClean	C	No

5.4 Auditing Dilithium Implementation for Blockchain

New implementations of Dilithium for blockchain applications must undergo rigorous auditing, as billions of dollars in digital assets are at risk. Blockchain systems rely on cryptographic signatures for transaction integrity, consensus, and asset security. A vulnerability in a Dilithium implementation (e.g., side-channel leakage, incorrect sampling, or parameter deviations) could allow attackers to forge signatures, steal funds, or disrupt networks. None of the surveyed implementations provide a formal audit report, though reference implementations (e.g., pq-crystals/dilithium) benefit from NIST’s review. For blockchain deployments, developers must commission third-party audits focusing on:

- Correctness of FIPS 204 compliance (parameters, algorithms).
- Side-channel resistance (timing, power analysis).
- Cryptographic randomness and sampling uniformity.
- Robustness against fault injection and implementation errors.

Our firm’s expertise in PQC auditing ensures that Dilithium implementations meet these stringent requirements, safeguarding blockchain ecosystems in the post-quantum era.

References

- [1] National Institute of Standards and Technology, “Module-Lattice-Based Digital Signature Standard,” Federal Information Processing Standards Publication (FIPS) NIST FIPS 204, 2024, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf>.

- [2] L. Ducas et al., “CRYSTALS-Dilithium: Algorithm Specification and Supporting Documentation (Version 3.1),” NIST PQC Round 3, 2021, <https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf>.
- [3] NIST, “Post-Quantum Cryptography Standardization,” <https://csrc.nist.gov/projects/post-quantum-cryptography>.
- [4] P. W. Shor, “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer,” SIAM Journal on Computing, 1997.
- [5] AMINA Bank, “The Quantum Threat to Blockchains,” 2024, <https://aminagroup.com/research/the-quantum-threat-to-blockchains/>.
- [6] Cointelegraph, “Why quantum computing isn’t a threat to crypto... yet,” 2022, <https://cointelegraph.com/news/why-quantum-computing-isn-t-a-threat-to-crypto-yet>.
- [7] D. J. Bernstein et al., “SPHINCS+: Submission to the NIST Post-Quantum Cryptography Standardization Project,” 2020.
- [8] P.-A. Fouque et al., “Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU,” NIST PQC Round 3, 2020.
- [9] A. Hülsing et al., “XMSS: eXtended Merkle Signature Scheme,” RFC 8391, 2018, <https://datatracker.ietf.org/doc/html/rfc8391>.
- [10] C. P. Schnorr, “Efficient Signature Generation by Smart Cards,” Journal of Cryptology, 1991.
- [11] DigiCert, “Dilithium: Post-Quantum Cryptography Insights,” <https://www.digicert.com/insights/post-quantum-cryptography/dilithium>.