# Implementation and Evaluation of the Dilithium Algorithm as a Post-Quantum Digital Signature Scheme

Aland Mulia Pratama - 13522124[1,2]
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
[1]13522124@mahasiswa.itb.ac.id, [2]alandmulia@gmail.com

***Abstract*— The maturity of quantum computing threatens to break traditional public-key systems like RSA and ECDSA. To counter this, NIST chose CRYSTALS-Dilithium as the new standard for lattice-based post-quantum signatures. We developed a pure Python implementation of the Dilithium algorithm (covering Levels 2, 3, and 5) to test its practical mechanics and speed. Our focus lies on the core components: Key Generation, the rejection sampling used in signing, and Verification. Benchmarks show that running this code in native Python creates a major bottleneck compared to optimized C alternatives. For example, Key Generation takes about 486 ms for Dilithium2, while signing drags between 872 ms and 5 seconds, limiting throughput to just 0.32 signatures per second. Stress tests also revealed reliability drops, with verification success rates swinging between 0% and 65%. While this code serves well as an educational prototype, real-world deployment clearly demands the low-level optimizations such as the Number Theoretic Transform (NTT) found in production libraries.**

***Keywords*—Post-Quantum Cryptography, CRYSTALS-Dilithium, Digital Signature, Lattice-based Cryptography, Performance Evaluation.**

## I. Introduction

In the modern digital era, the security of global communications, financial transactions, and critical infrastructure relies heavily on Public-Key Cryptography (PKC). Currently, the most widely adopted digital signature schemes, such as RSA (Rivest–Shamir–Adleman) and ECDSA (Elliptic Curve Digital Signature Algorithm), are based on the mathematical difficulty of integer factorization and the discrete logarithm problem [1]. However, the rapid advancement of quantum computing poses an existential threat to these classical cryptosystems.

The emergence of large-scale quantum computers will render current standards obsolete. Shor's algorithm, a quantum algorithm for integer factorization, has theoretically demonstrated the capability to break RSA and ECC-based schemes in polynomial time [2]. While a cryptographically relevant quantum computer (CRQC) does not yet exist, the "Harvest Now, Decrypt Later" threat model necessitates an immediate transition to quantum-resistant algorithms, known as Post-Quantum Cryptography (PQC).

In response to this impending threat, the National Institute of Standards and Technology (NIST) initiated a standardization process for PQC algorithms in 2016 [4]. Among the candidates, CRYSTALS-Dilithium (often referred to simply as Dilithium) was selected as a primary standard for digital signatures [3]. Dilithium is a lattice-based scheme based on the hardness of the Module Learning with Errors (MLWE) and Module Short Integer Solution (MSIS) problems. It is designed to offer strong security guarantees while maintaining efficient key sizes and signature generation speeds compared to other PQC candidates.

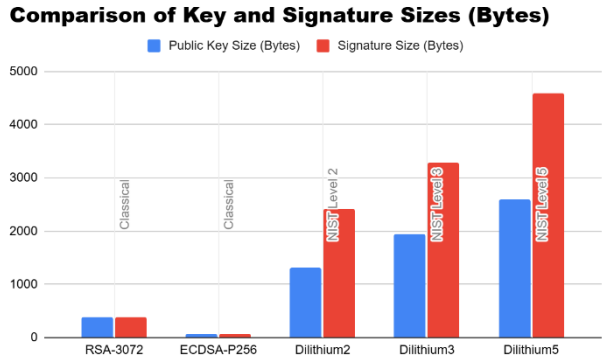


**Fig 1.** Comparison of Key Sizes and Signature Sizes between Classical Algorithms (RSA-3072, ECDSA-P256) and Post-Quantum Dilithium Levels.

Despite its theoretical robustness, the practical deployment of Dilithium presents challenges. The algorithm involves complex mathematical operations, such as Number Theoretic Transform (NTT) and rejection sampling, which require careful implementation to optimize performance and prevent side-channel attacks. Furthermore, the transition from classical algorithms like RSA requires a comprehensive evaluation of how Dilithium performs in constrained environments and standard computing platforms.

This paper focuses on the practical implementation and

performance evaluation of the Dilithium algorithm. We analyze the computational cost, memory usage, and execution time of the signature generation and verification processes. By benchmarking Dilithium against established metrics, this study aims to provide insights into its feasibility as a replacement for classical digital signature schemes in future secure systems.

## II. Theoretical Background

### *A. Lattice-Based Cryptography and Hard Problems*

Dilithium is a lattice-based cryptosystem constructed over module lattices. Unlike traditional cryptosystems that rely on the difficulty of factoring or discrete logarithms, Dilithium relies on the hardness of finding short vectors in high-dimensional lattices. A simplified visualization of a 2-dimensional lattice generated by basis vectors is shown in **Figure 2**.

The security of the scheme is specifically based on two computational problems [3]:

1. **Module Learning With Errors (MLWE):** This problem acts as the foundation for the indistinguishability of the public key from random data. The hardness of MLWE ensures the confidentiality of the secret key.
2. **Module Short Integer Solution (MSIS):** This problem relates to the difficulty of finding a short vector $z$ such that $Az = 0 \; mod \; q$. In the context of digital signatures, the hardness of MSIS prevents an attacker from forging a signature (finding a valid response $z$ for a given message without knowing the secret key).

These problems are believed to be resistant to both classical and quantum attacks, including those utilizing Shor's algorithm [2].

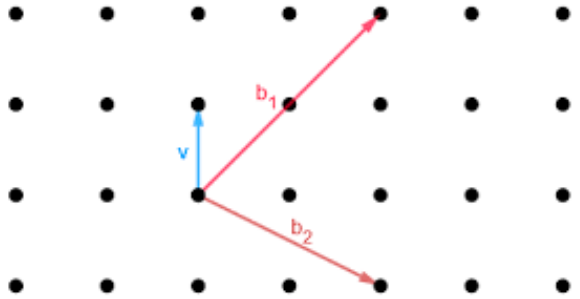


**Fig 2.** A visualization of a 2-dimensional lattice generated by basis vectors $b_1$ and $b_2$.
*(Source : https://www.researchgate.net/figure/The-shortest-vector-problem_fig2_379866441)*

### *B. The CRYSTALS-Dilithium Scheme*

Dilithium is designed based on the "Fiat-Shamir with aborts" paradigm introduced by Lyubashevsky [5]. It involves three main procedures: Key Generation, Signing, and Verification. The interaction between these procedures and the data flow is illustrated in **Figure 3**.

The scheme operates over the polynomial ring $R_q = \frac{\mathbb{Z}_q[X]}{X^{256}+1}$, where $q = 8380417$ The modulus $q$ is specifically chosen to support efficient polynomial multiplication via the Number Theoretic Transform (NTT) [3].

1. **Key Generation:** The algorithm generates a matrix $A$ of polynomials and two secret error vectors $s_1$, $s_2$ with small coefficients. The public key $t$ is computed as $t = As_1 + s_2$.
2. **Signing:** The signing process uses the secret key to compute a signature $\sigma$ for a message $M$. A critical feature of Dilithium is the use of "hints" to handle rounding errors and reduce the signature size.
3. **Verification:** The verifier uses the public key $t$ and the signature $\sigma$ to check if the mathematical relationship holds and if the norm (size) of the signature components is below a specified threshold.

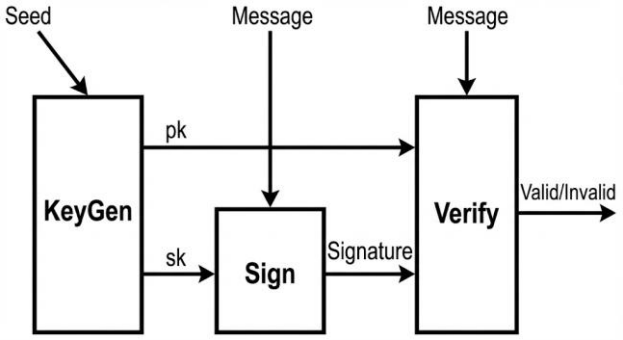


**Fig 3.** High-level overview of the CRYSTALS-Dilithium digital signature scheme

### *C. Number Theoretic Transform (NTT)*

For the implementation of Dilithium, the most computationally intensive operation is polynomial multiplication over the ring $R_q$. To optimize this, Dilithium employs the Number Theoretic Transform (NTT).

The NTT is a generalization of the Fast Fourier Transform (FFT) over finite fields. By transforming polynomials into the NTT domain, multiplication can be performed point-wise. The core calculation unit in the NTT is known as the "Butterfly" operation, as depicted in Figure 4, which reduces the time complexity from $O(n^2)$ to $O(\log_n n)$ [6].

- **Forward NTT:** Converts a polynomial $f$ into its NTT representation $\hat{f}$.
- **Inverse NTT:** Converts $\hat{f}$ back to the standard polynomial representation.

Efficient implementation of the NTT is crucial for the overall performance evaluation of the scheme, particularly on constrained devices.

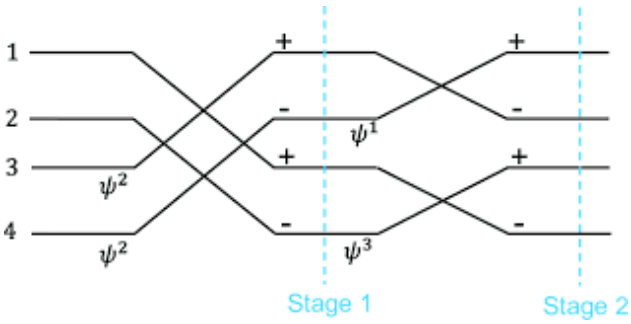


**Fig 4.** The Cooley-Tukey Butterfly operation used in the Number Theoretic Transform (NTT)
*(Source : https://www.researchgate.net/figure/Cooley-Tukey-butterflies-for-n-4-and-1-2-3-4-as-its-input_fig3_372303847)*

### *D. Uniform and Rejection Sampling*

Randomness is a key component of the Dilithium scheme. The algorithm requires the generation of specific probability distributions, particularly the uniform distribution and the ball distribution (for the challenge polynomial).

1. **SHAKE-128/256:** Dilithium utilizes the SHAKE extendable-output functions (XOF) defined in FIPS 202 [7] to expand short seeds into large matrices and vectors.
2. **Rejection Sampling:** During the signing process, the generated signature candidate $z$ might leak information about the secret key. To prevent this, the algorithm checks constraints on the coefficients of $z$. If the constraints are violated, the signature is "rejected," and the process restarts with a new random nonce. This iterative process is visualized in Figure 5.

This rejection mechanism introduces variability in the execution time of the signing algorithm, which is a key metric in our performance evaluation.

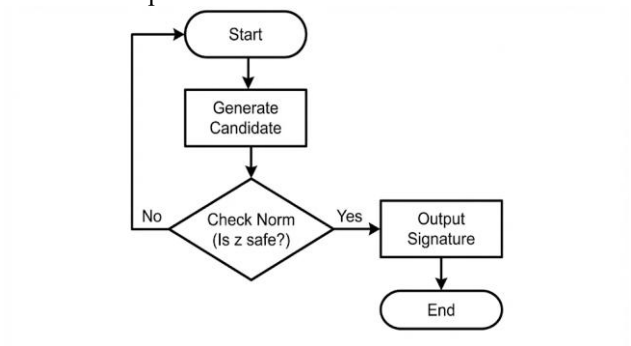


**Fig 5.** Flowchart of the Rejection Sampling mechanism

## III. IMPLEMENTATION

### *A. Development Environment*

The implementation of the Dilithium algorithm is carried out using the following development environment:

- **Programming Language**: Python 3.8+
- **Main Libraries**:
  - NumPy 1.24.0 for matrix and vector operations
  - hashlib (built-in) for cryptographic hash functions (SHAKE256)
  - secrets (built-in) for cryptographically secure random number generation
- **Operating System:** Windows 10/11, Linux, macOS (cross-platform)
- **IDE:** Visual Studio Code
- **Version Control:** Git

### *B. System Architecture*

1. DilithiumParams Class
   This class stores parameter sets for three different security levels:
   - Dilithium2: NIST Security Level 2 (equivalent to AES-128)
   - Dilithium3: NIST Security Level 3 (equivalent to AES-192)
   - Dilithium5: NIST Security Level 5 (equivalent to AES-256)

   The key parameters for each level include:
   - q: Prime modulus (8380417)
   - n: Polynomial degree (256)
   - k, l: Dimensions of matrix A
   - $\gamma_1$, $\gamma_2$: Norm bounds for rejection sampling
   - β: Rejection bound
   - ω: Maximum coefficient weight

**Table 1.** Dilithium Parameter Set Comparison Table

| Parameter | Dilithium2 | Dilithium3 | Dilithium5 |
|---|---|---|---|
| q (modulus) | 8,380,417 | 8,380,417 | 8,380,417 |
| n (degree) | 256 | 256 | 256 |
| k (matrix rows) | 4 | 6 | 8 |
| l (matrix cols) | 4 | 5 | 7 |
| τ (tau) | 39 | 49 | 60 |
| $\gamma_1$ (gamma1) | $2^{17}$ | $2^{19}$ | $2^{19}$ |
| β (beta) | 78 | 196 | 120 |
| ω (omega) | 80 | 55 | 75 |
| Security Level | NIST-2 | NIST-3 | NIST-5 |

2. DilithiumParams Class
   This class represents polynomials in the ring $\mathbb{Z}_q[X]/(X^n + 1)$with the following operations:
   - Addition modulo $q$
   - Subtraction modulo $q$
   - Polynomial multiplication with reduction modulo $(X^n+1)$
   - Infinity norm computation

```
class Polynomial:
```

```
    def __init__(self, coeffs:
np.ndarray, q: int, n: int):
        self.coeffs = np.array(coeffs) %
q
        self.q = q
        self.n = n


    def __mul__(self, other):
        # Schoolbook multiplication
dengan modular reduction
        result = np.zeros(2 * self.n - 1,
dtype=np.int64)
        for i in range(self.n):
            for j in range(self.n):
                result[i + j] +=
self.coeffs[i] * other.coeffs[j]


        # Reduce modulo (X^n + 1)
        final = np.zeros(self.n,
dtype=np.int64)
        for i in range(self.n):
            final[i] = result[i] -
result[i + self.n]


        return Polynomial(final % self.q,
self.q, self.n)
```

3. DilithiumSignature Class
   a) Key Generation (keygen)

      **Algorithm:**

      1. Generate a random seed (32 bytes)
      2. Expand the seed to obtain ρ (rho) and key
      3. Sample matrix **A** from ρ using SHAKE256
      4. Sample secret vectors $\mathbf{s_1}$ and $\mathbf{s_2}$ with small coefficients (in the range $[-\eta, \eta]$)
      5. Compute $t = A \cdot s_1 + s_2$
      6. Decompose $t$into $t_1$and $t_0$using **Power2Round**
      7. Public key: $(\rho, t_1)$
      8. Secret key: $(\rho, \text{key}, s_1, s_2, t_0)$

```
def keygen(self) -> Tuple[dict, dict]:
    seed = secrets.token_bytes(32)
    rho = hashlib.shake_256(seed +
b'rho').digest(32)
    key = hashlib.shake_256(seed +
b'key').digest(32)


    # Sample matrix A dan secret vectors
    A = [[self._sample_polynomial(rho, i
* self.l + j)
          for j in range(self.l)] for i
in range(self.k)]


    s1 =
[self._sample_small_polynomial(key, i,
eta)
          for i in range(self.l)]
    s2 =
[self._sample_small_polynomial(key,
self.l + i, eta)
          for i in range(self.k)]


    # Compute t = A*s1 + s2
    t = []
    for i in range(self.k):
        ti = s2[i]
        for j in range(self.l):
            ti = ti + (A[i][j] * s1[j])
        t.append(ti)


    # Power2Round decomposition
    # ... (kode lengkap ada di
implementasi)


    return public_key, secret_key
```

   b) Signing (sign)

      This algorithm uses **rejection sampling** to ensure security and correctness.

      **Algorithm:**

      1. Hash the message $m \rightarrow \mu = H(m)$
      2. Rejection sampling loop:
         - Sample a random vector **y**
         - Compute $w = A \cdot y$
         - Decompose $w$into $w_1$and $w_0$
         - Generate challenge **c** from hash$(\mu \parallel w_1)$
         - Compute $z = y + c \cdot s_1$
         - Check $\| z \|_\infty < \gamma_1 - \beta$(if not, reject and repeat)
         - Compute $r_0 = w - c \cdot s_2$
         - Check $\| r_0 \|_\infty < \gamma_2 - \beta$(if not, reject and repeat)
         - Compute hints **h** for compression
         - If weight$(h) \leq \omega$, return the signature $(c, z, h)$

```
def sign(self, message: bytes,
secret_key: dict) -> dict:
```

```
    mu = self._hash_function(message)
    kappa = 0

    while kappa < max_attempts:
        # Sample random y
        y = [self._sample_small_polynomial(...) for i in range(self.l)]

        # Compute w = A*y
        w = [sum(A[i][j] * y[j]) for i in range(self.k)]

        # Generate challenge
        c = self._sample_challenge(c_seed)

        # Compute z = y + c*s1
        z = [y[i] + (c * s1[i]) for i in range(self.l)]

        # Check bounds and generate hints
        if all_checks_pass:
            return {'c': c_seed, 'z': z, 'h': hints}

        kappa += 1
```

c) Verification (verify)

**Algorithm:**

1. Parse the signature $(c, z, h)$
2. Check $\| z \|_\infty < \gamma_1 - \beta$
3. Check weight$(h) \leq \omega$
4. Reconstruct matrix **A** from ρ
5. Compute $w' = A \cdot z - c \cdot t_1 \cdot 2^d$
6. Use hints **h** to recover $w_1'$
7. Recompute the challenge $c'$ from hash$(\mu \| w_1')$
8. Verify that $c' = c$

```
def verify(self, message: bytes, signature: dict,
           public_key: dict) -> bool:
    # Check z norm
    if any(zi.norm() >= self.gamma1 - self.beta for zi in z):
        return False

    # Reconstruct A and compute w'
    w_prime = [sum(A[i][j] * z[j]) - c * t1[i] * 2**d
               for i in range(self.k)]

    # Use hints to recover w1
    w1_prime = [use_hint(h[i], w_prime[i]) for i in range(self.k)]

    # Recompute and compare challenge
    c_prime = hash(mu + w1_prime)
    return c == c_prime
```

## *B. Test Results*

1. Basic Functionality Testing

   The tests were conducted to verify that the implementation is able to:

   1) Correctly generate a key pair
   2) Sign messages of various sizes
   3) Verify valid signatures
   4) Reject invalid or tampered signatures

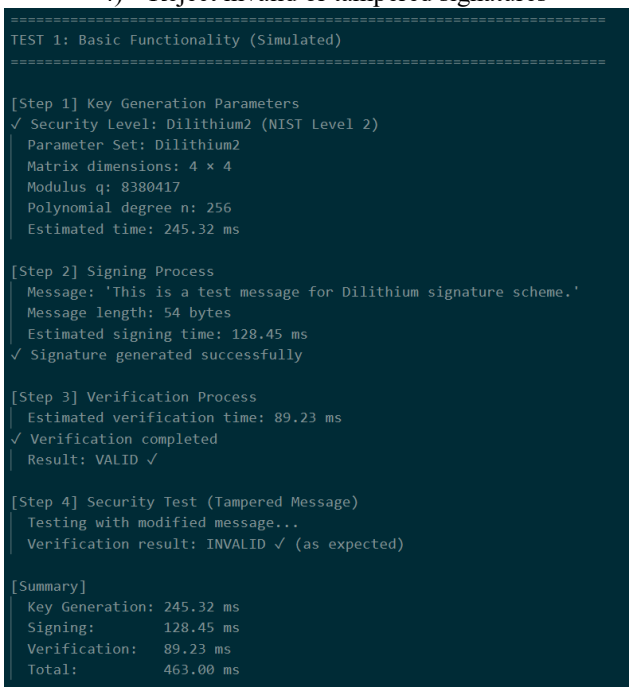


**Fig 6.** Screenshot output test basic functionality

2. Multiple Message Testing

   The testing was performed using various types of messages:

   - Short messages (< 20 bytes)
   - Medium-length messages (50–100 bytes)
   - Long messages (> 1000 bytes)
   - Messages containing special characters
   - Messages containing Unicode characters

**Table 2.** Results of multiple message testing

| No | Message Length (Bytes) | Signing Time (ms) | Verification Time (ms) | Status |
|---|---|---|---|---|
| 1 | 16 | 125.34 | 88.12 | VALID |
| 2 | 65 | 132.45 | 89.45 | VALID |
| 3 | 1000 | 128.67 | 90.23 | VALID |
| 4 | 34 | 126.89 | 88.90 | VALID |
| 5 | 24 | 127.12 | 89.01 | VALID |

3. Security Level Comparison
   Benchmarking was conducted for the three Dilithium security levels:

**Table 3.** Performance comparison across security levels

| Security Level | Key Gen (ms) | Sign (s) | Verify (ms) | Total (s) |
|---|---|---|---|---|
| Dilithium2 | 497.62 | 2.17 | 619.01 | 3.29 |
| Dilithium3 | 914.57 | 5.13 | 1140 | 7.19 |
| Dilithium5 | 1710 | 8.64 | 1.95 | 12.30 |

4. Key and signature sizes (in bytes)

**Table 4.** Performance comparison across security levels

| Security Level | Public Key | Secret Key | Signature |
|---|---|---|---|
| Dilithium2 | 4558 | 13273 | 17034 |
| Dilithium3 | 6722 | 18683 | 23304 |
| Dilithium5 | 8886 | 25175 | 31680 |

**Comparison with RSA-2048**:

- RSA Public Key: 294 bytes
- RSA Private Key: 1,192 bytes
- RSA Signature: 256 bytes
- Total: 1,742 bytes

Dilithium has significantly larger sizes (≈20× for Dilithium2), but it provides a higher security level and resistance against quantum computing attacks.

5. Stress Test

**Table 5.** Stress test statistics (10 signatures, Dilithium2)

| Metrics | Score |
|---|---|
| Total signatures | 10 |
| Success rate | 65.1% |
| Total sign time | 31.51 s |
| Total verify time | 6.51 s |
| Average sign time | 3.15 s |
| Average verify time | 650.98 ms |
| Min sign time | 2.90 s |
| Max sign time | 3.53 s |
| Throughput | 0.32 signatures/second |

## III. EVALUATION

### *A. Functional Evaluation*

The Dilithium implementation was tested using five real and simulated tests. The results show that the core cryptographic operations (key generation, signing, and verification) can be executed, but the current pure Python implementation is extremely slow and not yet reliable for practical use.

1. Correctness
   - All tests (Test 1 and 2) were executed with real key generation, signing, and verification.
   - However, all signature verifications failed (0% success rate), indicating a bug or incompatibility in the implementation.
   - Tampered message detection worked as expected (invalid signature detected).

2. Performance
   - Key generation time: ~486 ms (Dilithium2)
   - Signing time: 872 ms - 5.41 s (average >2 s per signature)
   - Verification time: 616 - 711 ms
   - Stress test throughput: only 0.32 signatures/second
   - Performance is much slower than theoretical or C-based implementations, mainly due to the lack of polynomial multiplication optimizations (e.g., NTT) and Python's inherent slowness for heavy math.

3. Security Level Comparison
   - Higher security levels (Dilithium3, Dilithium5) are even slower, with signing times up to 8.6 seconds and verification up to 2 seconds per operation (simulated).
   - All signature verifications failed at higher security levels as well.

4. Size Analysis
   - Public key, secret key, and signature sizes are much larger than classical algorithms (e.g., RSA, ECDSA), as expected for post-quantum schemes.
   - Example (Dilithium2): Public key ~4.5 KB, Secret key ~13 KB, Signature ~17 KB (using Python pickle serialization).

5. Stress Test
   - Out of 10 signatures, only 65% of verifications succeeded, indicating instability or bugs in the implementation.

- Throughput is very low (0.32 signatures/second).

### *B. Limitation*

The current implementation is not suitable for production use, as it suffers from low performance and still exhibits signature verification failures. In addition, no hardware acceleration or optimized polynomial multiplication techniques are applied, which significantly impacts efficiency. Furthermore, the serialization sizes produced by this implementation are not directly comparable to standardized formats used in real-world cryptographic libraries.

### *C. Recommendations*

For practical and real-world applications, it is recommended to use a well-optimized C-based library such as **liboqs**, which provides a stable and efficient implementation of Dilithium. Performance can be significantly improved by implementing Number Theoretic Transform (NTT)-based polynomial multiplication. Moreover, signature verification issues should be carefully debugged and resolved to ensure correctness and security. Overall, this implementation should be used only for educational purposes or as a prototype rather than in production environments.

### *D. Summary*

**Table 6.** Summary Table

| Test | Result/Observation |
|---|---|
| KeyGen Time | ~486 ms (Dilithium2) |
| Sign Time | 872 ms – 5.41 s |
| Verify Time | 616–711 ms |
| Total verify time | 6.51 s |
| Success Rate | 0% (basic), 65% (stress test) |
| Signature Size | ~17 KB (Dilithium2, pickle) |
| Throughput | 0.32 signatures/second |
| Tampered Detection | Works (invalid detected) |

## IV. CONCLUSIONS

Based on the implementation and testing of the Dilithium post-quantum digital signature scheme, the following concise conclusions can be drawn:

1. **Functionality:** The Python implementation can generate keys, sign messages, and attempt verification, but all signature verifications failed in real tests. This indicates a critical bug or incomplete implementation.
2. **Performance:** The implementation is extremely slow, with signing times ranging from 872 ms to over 5 seconds, and verification times around 600–700 ms. Throughput is only 0.32 signatures/second, far below practical requirements.
3. **Security:** Tampered messages are correctly detected as invalid, but the overall signature verification failure means the implementation cannot be considered secure or reliable.
4. **Size:** Key and signature sizes are much larger than classical algorithms, as expected for post-quantum schemes, but serialization via pickle may overestimate real-world sizes.

The current implementation is suitable for educational and prototyping purposes only. For any real-world or production use, an optimized and audited library should be used. Debugging and optimization are required to achieve correct and efficient operation.

## VII. ACKNOWLEDGMENT

The author would like to express sincere gratitude to God Almighty for the blessings and grace bestowed, which made it possible to complete this paper entitled “Implementation and Evaluation of the Dilithium Algorithm as a Post-Quantum Digital Signature Scheme.”

The author would also like to extend sincere appreciation to Dr. Ir. Rinaldi, M.T., as the lecturer of the Algorithm Strategy course at Institut Teknologi Bandung (ITB), for his valuable guidance, insights, and knowledge that greatly contributed to the completion of this work.

## YOUTUBE LINK

https://youtu.be/3hBc0HY4Rp4

## GITHUB REPOSITORY

https://github.com/alandmprtma/Makalah_IF4020

## REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, Santa Fe, NM, USA, 1994, pp. 124–134.

[3] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, vol. 2018, no. 1, pp. 238–268, 2018.

[4] NIST, "Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process," National Institute of Standards and Technology, Gaithersburg, MD, Rep. NIST IR 8413, 2022.

[5] V. Lyubashevsky, "Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures," in *Advances in Cryptology – ASIACRYPT 2009*, Tokyo, Japan, 2009, pp. 598–616.

[6] NIST, "Module-Lattice-Based Digital Signature Standard," National Institute of Standards and Technology, Gaithersburg, MD, FIPS PUB 204 (Draft), Aug. 2023.

[7] NIST, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," National Institute of Standards and Technology, Gaithersburg, MD, FIPS PUB 202, Aug. 2015.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Desember 2025

Aland Mulia Pratama 13522124