Article

A Comparative Study of NIST Post-Quantum Cryptography Algorithms: CRYSTALS-Kyber and CRYSTALS-Dilithium

Zainab Al-Ramadhan [1], Shatha Aldosssary [2], Munirah Aljrayan [3] and Zahra Al-Muslem [4]

College of Computer Science and Information Technology, Imam Abdulrahman Bin Faisal University, Dammam, Saudi Arabia;

[1] 2240005053@iau.edu.sa
[2] 2240005624@iau.edu.sa
[3] 2240006702@iau.edu.sa
[4] 2240005262@iau.edu.sa

Abstract

Quantum computing is transforming cybersecurity and rendering traditional cryptographic systems increasingly obsolete. In response, this study evaluates two of NIST's newly standardized algorithms: CRYSTALS-Kyber for key exchange and CRYSTALS-Dilithium for digital signatures. Using both literature review and hands-on implementation, the project examines their mathematical foundations, performance, efficiency, implementation challenges, and real-world applicability. By comparing these lattice-based schemes from theoretical and experimental perspectives, the study clarifies how they can enable secure communication in the post-quantum era. This work supports efforts to prepare modern systems for the quantum transition and underscores the need to assess post-quantum algorithms in practice as well as in theory.

Keywords: CRYSTALS-Kyber; CRYSTALS-Dilithium; Post-quantum cryptography; Quantum-resistant algorithms; Key encapsulation mechanism (KEM);

1. Introduction

The digital world we rely on today was built on cryptographic systems that have served us well for decades. Protocols such as RSA and elliptic-curve cryptography protect everything from online banking to private messaging. However, this security foundation faces a serious challenge: the arrival of quantum computing. A sufficiently powerful quantum computer could break these classical systems in a matter of hours [1], exposing even well-protected data. This possibility has transformed quantum computing from a scientific curiosity into a real cybersecurity concern [2].

To address this threat, the field of post-quantum cryptography has come to the forefront. Its goal is simple yet ambitious: design cryptographic algorithms that remain secure even when attackers have access to quantum computers. Among the many mathematical approaches explored, lattice-based cryptography has emerged as one of the strongest candidates. It is built on structured, high-dimensional mathematical problems that are exceptionally difficult to solve, even with advanced quantum algorithms [5].

Recognizing the importance of transitioning to quantum-safe solutions, the National Institute of Standards and Technology (NIST) launched a global competition to identify practical and secure replacements for classical cryptographic standards [3]. After years of evaluation, NIST selected two algorithms as part of its first post-quantum standardization effort: CRYSTALS-Kyber, used to establish secure shared keys, and CRYSTALS-Dilithium, used to create and verify digital signatures [4]. These two algorithms form the foundation of what many expect to be the next generation of secure communication protocols [5].

Despite being designed for different tasks, Kyber and Dilithium share a common mathematical foundation and are often deployed together. Understanding how they compare in real-world scenarios is essential for designing secure systems that are ready for the quantum era [10]. This project investigates both algorithms by reviewing previous research and performing hands-on coding experiments. The comparison focuses on security strength, computational performance, efficiency, implementation complexity, and practical applicability.

Through a side-by-side, theoretical and experimental comparison of Kyber and Dilithium, this project provides actionable insights into how post-quantum cryptography will shape modern security, highlighting the immediate challenges and opportunities this transformation presents.

2. Literature Review and Background

2.1 Foundations of PQC

The rise of quantum computers has presented a new challenge to the cryptography community over the past decade. These machines can break classical systems such as RSA and elliptic-curve cryptography by solving their underlying mathematical problems way more faster than any classical computer [1]. This has caused researchers to explore alternative paths, with post-quantum cryptography emerging as one of the most promising [2].

Lattice-based cryptography is now the most widely used post-quantum family. It is based on multi-dimensional mathematical structures called lattices, where it is very difficult for even quantum computers to find certain vectors or reverse complicated equations. Due to their perceived difficulty, problems such as Learning With Errors, Module-LWE, and Module-SIS are yet to experience an effective attack. As a result, they are the best options for future digital communication security [8].

This strong mathematical foundation explains why NIST selected two lattice-based algorithms, CRYSTALS-Kyber and CRYSTALS-Dilithium, as official standards for the post-quantum era [4]. Dilithium offers digital signatures, and Kyber manages safe key exchange. They are working together to create the security of the future.

2.2 CRYSTALS-Kyber in Research

Establishing secure session keys between two parties is the primary purpose of CRYSTALS-Kyber. It has an unexpected advantage because it employs the Module-LWE problem and uses efficient polynomial arithmetic to carry out its operations. Kyber's algorithm is incredibly quick, despite the fact that its keys are larger compared to traditional RSA keys [5].

Kyber's outstanding performance is highlighted in a number of studies. For instance, extensive tests conducted by organizations like Google and Cloudflare demonstrated that Kyber functions flawlessly within actual TLS handshakes [6]. Even though the algorithms are performing much more complicated mathematics in the background, users hardly notice a difference in speed.

Researchers also looked at Kyber in constrained situations like microcontrollers and Internet of Things (IoT) devices. Thanks to its structure and efficient number-theoretic transforms, Kyber was able to run with limited memory and power [7]. This has made Kyber one of the most practical post-quantum KEMs available today and one of the main reasons NIST selected it [4].

2.3 CRYSTALS-Dilithium in Research

Kyber serves as a key exchange protocol, whereas CRYSTALS-Dilithium functions as a digital signature scheme. Dilithium is engineered to provide authenticity, integrity, and non-repudiation. It employs Module-SIS and Module-LWE [8], and avoids floating-point operations, which facilitates secure implementation and mitigates the risk of side-channel attacks [9].

Studies indicate that Dilithium is notable for its rapid verification speed, a critical feature for systems that must validate large volumes of signatures daily. Such systems include certificate authorities, firmware update mechanisms, secure boot processes, and identity management platforms [8,9].

Although Dilithium produces larger signatures compared to traditional schemes, it delivers robust security and consistent performance. Its straightforward design facilitates reliable implementation and enhances resistance to quantum-based attacks. Consequently, Dilithium has become the most extensively evaluated post-quantum signature scheme [8].

2.4 Comparative Research Findings

Comparative analysis of Kyber and Dilithium reveals a distinct pattern: Kyber performs better in key exchange, whereas Dilithium is more effective for digital signatures [10]. These algorithms are designed to function in complementary ways, analogous to the use of ECDH for key agreement and ECDSA for signatures in classical cryptographic systems.

Studies show that Kyber is usually faster and uses fewer resources, especially during key generation and encapsulation; for instance, Kyber512 performs key generation in approximately 0.7 milliseconds on a standard CPU, whereas comparable schemes take longer [7]. Dilithium stands out for its very fast verification, making it ideal for authentication; for example, Dilithium2 achieves verification in around 0.1 milliseconds. Both have larger keys and outputs than classical cryptography, but this is a normal trade-off for quantum resistance [8].

The research also examines challenges such as coping with memory limitations, defending against side-channel attacks, and maximizing hardware efficiency. Despite these difficulties, Kyber and Dilithium surpass other post-quantum cryptographic candidates in both practicality and security [10].

2.5 Summary and Justification

A review of the literature shows why Kyber and Dilithium deserve closer study. They are the first post-quantum algorithms standardized by NIST, backed by solid mathematics, real-world testing, and years of public cryptanalysis [4]. They also cover the two main parts of secure communication: key exchange and signatures.

The literature points out differences in speed, output sizes, memory use, and how easy each algorithm is to implement. This project will look at these factors through hands-on coding experiments. The goal is to implement both algorithms, test their performance, and make practical comparisons. This way, we can see how each post-quantum tool works in real-world situations.

This literature review prepares the way for the experimental portion of the project, where we will implement, test, and compare Kyber and Dilithium.

3. algorithm implementation

This section will show our demonstration of Kyber (Post quantum key encapsulation mechanism) and Dilithium (Post-quantum digital signature). Our goal was not to fully implement the official schemes , but to create simplified simulation that illustrate the core ideas behind key generation, encryption/encapsulation, and signature verification.

3.1. Kyber (Post quantum key encapsulation mechanism)-TOY-KEM-

3.1.1. libraries used

The libraries shown in **Figure 1**

```python
import os
import hashlib
import random
import time
from Cryptodome.Cipher import AES
```

Figure 1. libraries used.

OS: Secure randomness for keys, kyber use random number generator like randombytes()

Time: to calculate the time difference between our algorithms

Hashlib: kyber use SHAKE128, SHAKE256, however on our demo we use sha256()

Cryptodome : it provides AES-GCM which lets us encrypt/decrypt massages using shared secret keys produced in this demo just like real kyber uses AES after it establishes a key

3.1.2. key generator

The key generator shown in **Figure 2**

```python
def keygen(matrix_size=64):  1 usage

    # Secret vector s
    s = [random.randint(a: 0, b: 255) for _ in range(matrix_size)]

    # Matrix A
    A = []
    for _ in range(matrix_size):
        row = [random.randint(a: 0, b: 255) for _ in range(matrix_size)]
        A.append(row)

    # A * s
    result = []
    for i in range(matrix_size):
        total = 0
        for j in range(matrix_size):
            total += A[i][j] * s[j]
        result.append(total % 256)

    # Public key
    pk = hashlib.sha256(bytes(result)).digest()
    # Secret key is vector s
    sk = bytes(s)

    return A, sk, pk
```

Figure 2. Key generator in kyber.

Simulated KeyGen:
- Build matrix A
- Build secret vector s
- Compute A * s
- Hash result -> public key

Returns A, s, pk so decaps can rebuild the same pk.

3.1.3. encapsulation

The encapsulation shown in **Figure 3**

```python
def encaps(pk):  1 usage
    """
    Bob generates:
      - Random shared secret ss
      - Ciphertext = ss XOR pk
    """
    ss = os.urandom(32)
    ciphertext = bytes([ss[i] ^ pk[i] for i in range(32)])
    return ciphertext, ss
```

Figure 3. encapsulation in kyber.

Our KEMGenerates a random shared secret and Hides it using XOR with the public key

Ciphertext = shared_secret XOR pk

However, Real Kyber Generates vector r that

Computes:

$u = A \cdot r + e1$

$v = t \cdot r + e2 + H(m)$

Ciphertext = (u, v)

3.1.4. decapsulation

The decapsulation shown in **Figure 4**

```python
def decaps(sk, ciphertext):  1 usage
    """
    Alice recovers:
      ss = ciphertext XOR H(sk)
    """
    pk = hashlib.sha256(sk).digest()
    ss = bytes([ciphertext[i] ^ pk[i] for i in range(32)])
    return ss
```

Figure 4. decapsulation in kyber.

Our KEM Recomputes pk from sk, Reverses XOR to get the same shared secret however, Real Kyber Computes:($v' = v - u \cdot s$) then Applies hashing to reconstruct the secret

3.1.5. encrypt

The encryption shown in **Figure 5**

```python
def aes_encrypt(key, plaintext):
    cipher = AES.new(key, AES.MODE_GCM)
    ciphertext, tag = cipher.encrypt_and_digest(plaintext)
    return ciphertext, tag, cipher.nonce
```

Figure 5. encryption in kyber.

This function encrypts the actual message using AES-GCM. in Real Kyber it only creates the shared secret key that AES uses.

3.1.6. decrypt

The decryption shown in **Figure 6**

```python
def aes_decrypt(key, ciphertext, tag, nonce):
    cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
    return cipher.decrypt_and_verify(ciphertext, tag)
```

Figure 6. decryption in kyber.

It uses the shared secret key from the KEM: The nonce replays the exact encryption environment, The tag ensures correctness, It returns the original message.

3.1.7. the scenario

The scenario shown in **Figure 7**

```python
def demo():

    print("\n--- Alice generates key pair ---")
    pk, sk = keygen()
    print("Alice's public key:", pk.hex())
    print("Alice's secret key:", sk.hex())

    print("\n--- Bob receives Alice's public key and encapsulates ---")
    ct, ss_bob = encaps(pk)
    print("Ciphertext sent to Alice:", ct.hex())
    print("Bob's shared secret:", ss_bob.hex())

    print("\n--- Alice decapsulates ciphertext to recover secret ---")
    ss_alice = decaps(sk, ct)
    print("Alice's recovered shared secret:", ss_alice.hex())

    print("\nShared secret match:", ss_bob == ss_alice)
```

Figure 7. scenario created to understand the process of kyber.

At the beginning of the program, Alice creates a key pair via keygen().
In the Toy-KEM, this is intentionally simple: we create a random secret key and derive the public key by hashing it with SHA-256. This emulates the spirit of Kyber's KeyGen-a

private key that only Alice knows and a corresponding public key that others can use safely-without the heavy lattice math. Real Kyber uses vectors of polynomials, noise values, and a matrix multiplication step in key generation, but the high-level concept is the same .

Then, Bob gets Alice's public key and does the encapsulation. In the Toy-KEM, Bob generates a random shared secret and masks it by XORing it with Alice's public key. This yields a ciphertext that only Alice can invert. This is similar to Kyber's: Bob takes Alice's public key and generates a ciphertext and a shared secret. Real Kyber does this via polynomial equations like:

$-u = A * r + e1$

$-v = t \times r + e2 + H(m)$

Different math, same purpose: Bob makes a secret in such a way that only Alice can recover it.

When Alice receives the ciphertext she does decapsulation. In the toy version she simply recomputes the public key from her secret key and reverses the XOR. Real Kyber instead computes:

$v' = v - (u * s)$

And then reconstructs the shared secret from that. The math is different, but the logic is identical:

only the party having the correct secret key can recover the shared secret.

The program then checks whether Bob and Alice derived the same secret. In Toy-KEM, they always match because XOR is perfectly reversible. In real Kyber, they also match unless the ciphertext is intentionally corrupted, in which case Kyber uses extra protections to avoid leaking information. We then use AES-GCM to encrypt a message using the shared secret. Kyber itself does not encrypt messages; it only produces the shared key. But in real systems like TLS 1.3, Kyber is immediately followed by AES-GCM or ChaCha20 to encrypt application data. We include AES-GCM in the toy version to show how Kyber is meant to be used in practice.

The scenario shown in **Figure 8**

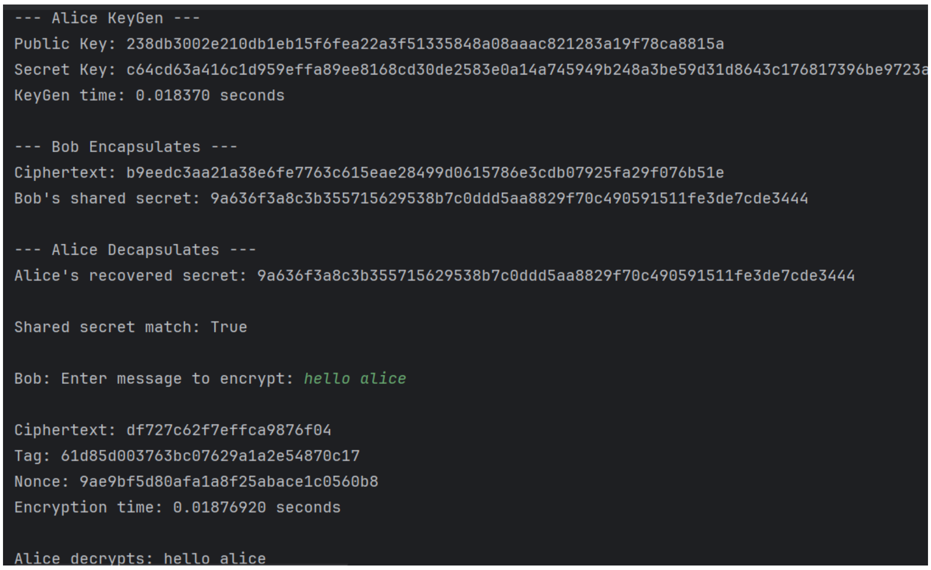
```
    # ---------- ENCRYPT MESSAGE ----------
    user_message = input("\nBob: Enter message to encrypt for Alice: ").encode()

    ciphertext, tag, nonce = aes_encrypt(ss_bob, user_message)
    print("\nBob sends encrypted message, tag, nonce to Alice:")
    print("Ciphertext:", ciphertext.hex())
    print("Tag:", tag.hex())
    print("Nonce:", nonce.hex())

    # ---------- ALICE DECRYPTS ----------
    decrypted = aes_decrypt(ss_alice, ciphertext, tag, nonce)
    print("\nAlice decrypts the message:")
    print("Decrypted:", decrypted.decode())

if __name__ == "__main__":
    demo()
```

Figure 8. scenario created to understand the process of kyber.

This part is not Kyber itself, but it shows exactly how Kyber is used in real systems: Kyber creates the shared key, and AES uses that key to encrypt the message.

3.1.8. output

The output shown in **Figure 9**



```
--- Alice KeyGen ---
Public Key: 238db3002e210db1eb15f6fea22a3f51335848a08aaac821283a19f78ca8815a
Secret Key: c64cd63a416c1d959effa89ee8168cd30de2583e0a14a745949b248a3be59d31d8643c176817396be9723a
KeyGen time: 0.018370 seconds

--- Bob Encapsulates ---
Ciphertext: b9eedc3aa21a38e6fe7763c615eae28499d0615786e3cdb07925fa29f076b51e
Bob's shared secret: 9a636f3a8c3b355715629538b7c0ddd5aa8829f70c490591511fe3de7cde3444

--- Alice Decapsulates ---
Alice's recovered secret: 9a636f3a8c3b355715629538b7c0ddd5aa8829f70c490591511fe3de7cde3444

Shared secret match: True

Bob: Enter message to encrypt: hello alice

Ciphertext: df727c62f7effca9876f04
Tag: 61d85d003763bc07629a1a2e54870c17
Nonce: 9ae9bf5d80afa1a8f25abace1c0560b8
Encryption time: 0.01876920 seconds

Alice decrypts: hello alice
```

Figure 9. output of kyber.

3.2. *CRYSTALS Dilithium(post-quantum digital signature scheme)-demo-*

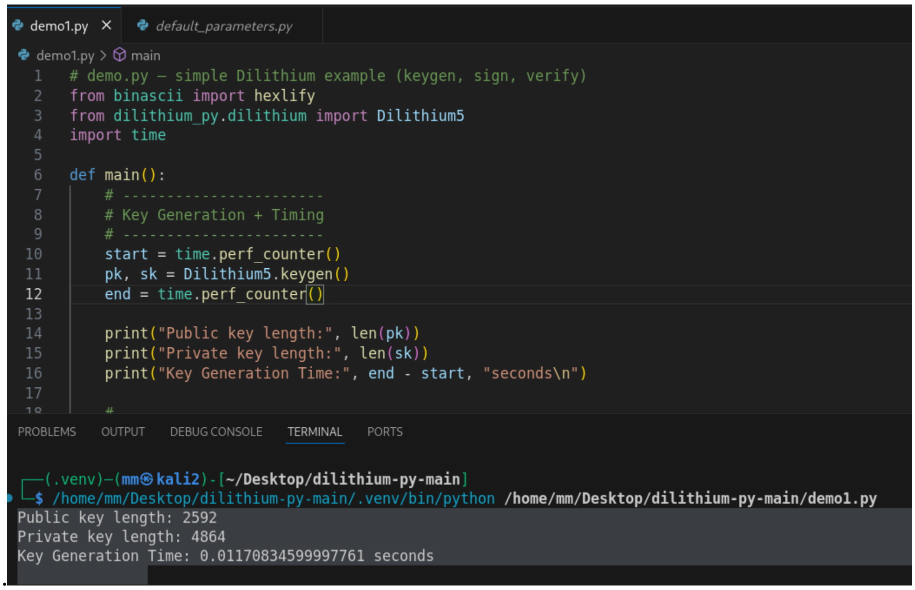
We implemented CRYSTALS-Dilithium using a publicly available GitHub repository.

The implementation includes key generation, message signing, signature verification, and execution time measurement .

All experiments were conducted in Python on a machine equipped with 6 CPU cores and 8 GB of RAM [13].

3.2.1. Key generation:

The output of the key-generation code is shown in **Figure 10**



```
demo1.py  default_parameters.py
demo1.py > main
# demo.py – simple Dilithium example (keygen, sign, verify)
from binascii import hexlify
from dilithium_py.dilithium import Dilithium5
import time

def main():
    # ----------------------
    # Key Generation + Timing
    # ----------------------
    start = time.perf_counter()
    pk, sk = Dilithium5.keygen()
    end = time.perf_counter()

    print("Public key length:", len(pk))
    print("Private key length:", len(sk))
    print("Key Generation Time:", end - start, "seconds\n")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

┌──(.venv)─(mm㉿kali2)-[~/Desktop/dilithium-py-main]
└─$ /home/mm/Desktop/dilithium-py-main/.venv/bin/python /home/mm/Desktop/dilithium-py-main/demo1.py
Public key length: 2592
Private key length: 4864
Key Generation Time: 0.01170834599997761 seconds
```

Figure 10. Dilithium5 key-generation output and execution time.

For the key generation part, we used the Dilithium5 function to create both the public and private keys.

The process was very fast and took about 0.0117 seconds.

The generated key sizes were:

Public key: 2592 bytes

Private key: 4864 bytes

3.2.2. Sign messages:

The signing output is shown in Figure 11.

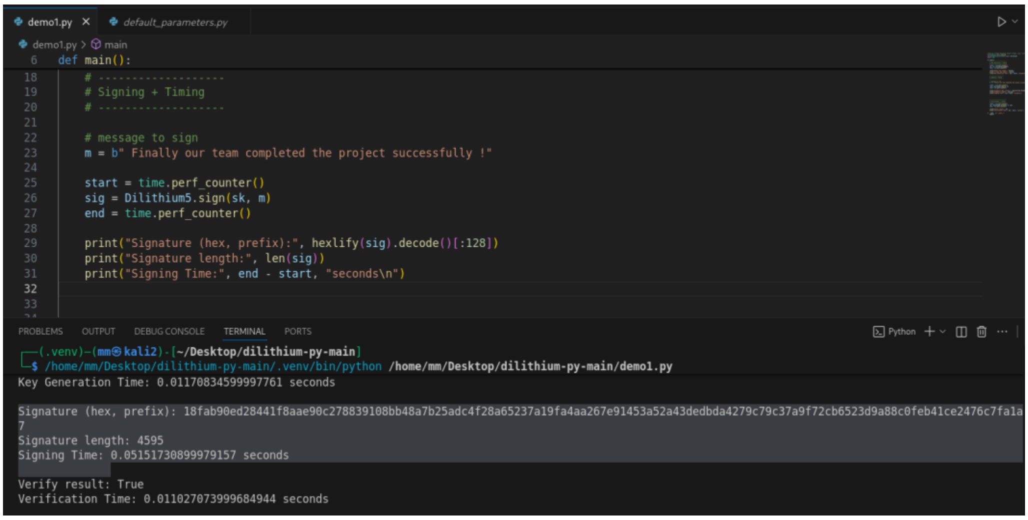


Figure 11. Dilithium5 signing output and signature length.

To test the signing process, we used a fixed message and recorded how long it took to generate a signature. The Dilithium5 signature size was about 4595 bytes, which matches what is expected from the standard.

We noticed that the signing time was not constant and ranged between 0.05 and 0.15 seconds.

3.2.3. Verification:

The verification output is shown in Figure 12.

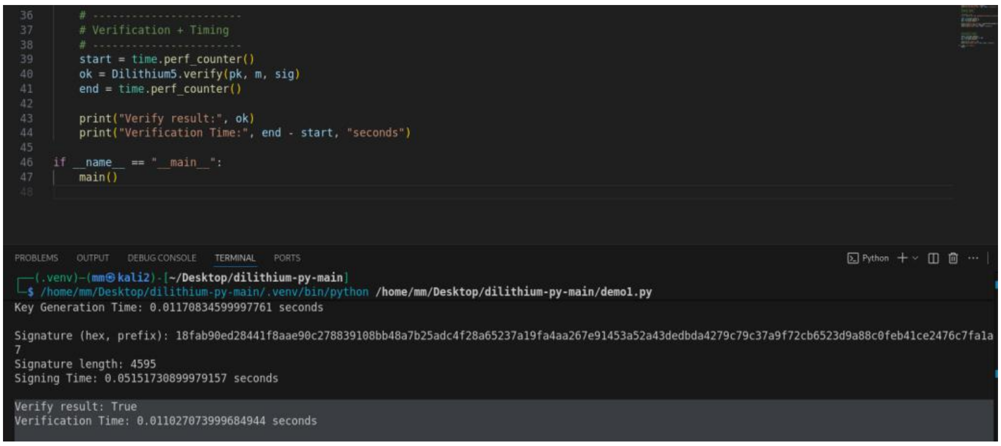


Figure 12. Verification output showing a valid signature (True result).

For verification, we checked the validity of the generated signature using the corresponding public key, and the result is True.

3.2.4. Scenario:

Alice and Bob use Dilithium5 to exchange a secure update about their project. Alice first generates her key, producing a 2592-byte public key and a 4864-byte private key, with the key generation taking around 0.01 seconds.

She then writes a message and signs it using her private key. The signature is about 4595 bytes, and the signing process takes 0.05–0.15 seconds. Alice sends Bob both the message and the signature.

When Bob receives them, he verifies the signature using Alice's public key. Verification completes almost 0.01 seconds and confirms that the message is coming from Alice [14].

3.3. *Comparison of Key Generation Time Between Kyber and Dilithium*

The key-generation outputs for Kyber and Dilithium are shown in Figure 13 and Figure 14, respectively.

These outputs include the generated keys and the execution time recorded during the KeyGen phase.

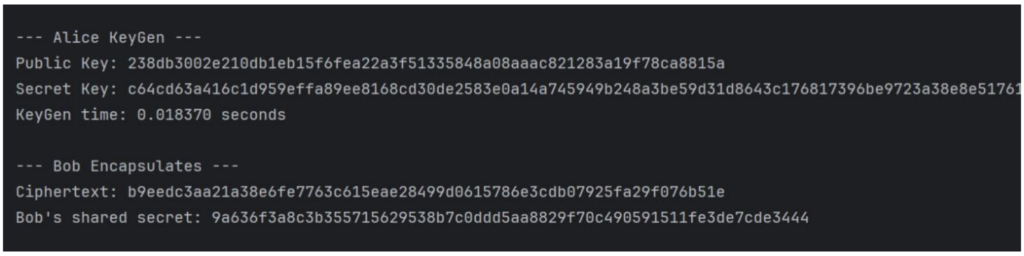


Figure 13. Kyber Toy-KEM key-generation output and execution time.

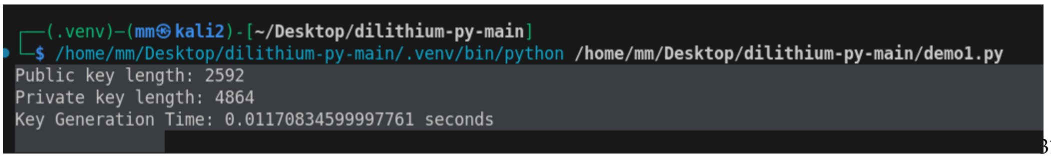


Figure 14. Dilithium5 key-generation output and execution time.

Based on the results, it is noticeable that the time for generating the public and private keys is different. Kyber took about 0.01837 seconds to generate the keys, while Dilithium was a little faster with around 0.0117 seconds.

So we conclude from our test that Dilithium is faster than Kyber by about 0.006 seconds in generating the keys.

4. Discussion

CRYSTALS-Kyber and CRYSTALS-Dilithium represent two basic but different post-quantum cryptographic primitives. Kyber is a KEM that has been designed to provide secure key establishment based on the hardness of the Module-LWE problem. It enables shared symmetric key generation and provides IND-CCA2 security, therefor making it suitable for encrypted communication protocols like TLS. According to the specification of Kyber, efficiency and small ciphertext size along with very fast encapsulation and decapsulation operations are achieved through structured lattices and the use of SHAKE-based hashing functions [11]. Dilithium, on the other hand, is a digital signature scheme that is obtained by combining Module-SIS and Module-LWE problems in order to attain SUF-CMA unforgeability. This corresponds to authentication provided by Dilithium through deterministic signing, rejection sampling, and the Fiat–Shamir transform using SHAKE functions, rather than key exchange. In the specification of Dilithium, strong security guarantees, fast verification times, and a structured design have been emphasized-optimally suited for real-world deployment in signature-heavy applications [2]. Both schemes rely on hardness assumptions of lattices, and both employ similar hash functions. However, their goals are different: Kyber ensures confidentiality by the derivation of a shared secret, while Dilithium provides integrity and authenticity in the form of digital signatures. Their parameter sets also differ: Kyber offers three levels, 512, 768, and 1024, where each subsequent level represents increased security level; Dilithium provides three levels, 2, 3, and 5, each representing a certain trade-off between signature size, performance, and security [12].

Appendix

AI Tools Usage

AI tools such as ChatGPT were only used to clarify concepts in post-quantum cryptography, improve writing clarity, assist in structuring the MDPI style, and answer questions related to formatting or creating scenarios

References

1. P. W. Shor, “Algorithms for quantum computation: discrete logarithms and factoring,” Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 124–134. https://doi.org/10.1109/SFCS.1994.365700
2. D. J. Bernstein and T. Lange, “Post-Quantum Cryptography,” Nature, vol. 549, pp. 188–194, 2017.
3. https://doi.org/10.1038/nature23461
4. J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, et al., “CRYSTALS-Kyber: A CCA-secure Module-LWE-based KEM,” 2018 IEEE European Symposium on Security and Privacy, pp. 353–367. https://doi.org/10.1109/EuroSP.2018.00032
5. NIST, “Post-Quantum Cryptography Standardization: Final Portfolio,” 2024. Available online: https://csrc.nist.gov/projects/post-quantum-cryptography (accessed on 8 December 2025).
6. Cloudflare, “Experimenting with Post-Quantum Cryptography,” 2019. Available online: https://blog.cloudflare.com/post-quantum-cryptography/ (accessed on 10 December 2025).
7. PQShield, “Benchmarking Lattice-Based Post-Quantum Cryptography Algorithms,” Technical Report, 2022. Available online: https://pqshield.com/resources (accessed on 9 December 2025).
8. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, “CRYSTALS-Dilithium: Digital Signatures from Module Lattices,” Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 236–253. https://doi.org/10.1145/3133956.3133982
9. NIST, “FIPS 204: CRYSTALS-Dilithium Digital Signature Standard,” 2024. Available online: https://doi.org/10.6028/NIST.FIPS.204
10. D. Kales, J. M. Baldimtsi, J. van der Merwe, and D. Stebila, “On the Implementation of Kyber and Dilithium in TLS,” NDSS Workshop on Post-Quantum Cryptography, 2021. Available online: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_imsw_paper_2.pdf (accessed on 10 December 2025).
11. J. Bos et al., “CRYSTALS-Kyber: Algorithm Specifications and Supporting Documentation (Round 3 Submission),” NIST PQC Project, 2021. [Online]. Available: https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf
12. L. Ducas et al., “CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Round 3 Submission),” NIST PQC Project, 2021. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf
13. G. Pope, “dilithium-py,” GitHub Repository. Available online: https://github.com/GiacomoPope/dilithium-py (accessed on 08 December 2025).

14. ChatGPT Conversation, “Discussion on PQC Research and Methods,” Available online: https://chatgpt.com/share/69360f26-3684-8000-850a-357e8187eb51 (accessed on 09 December 2025).