Cross-Platform Performance and Security Evaluation of Post-Quantum Cryptographic Algorithms on Resource-Constrained Devices

Daiana-Larisa Lucaciu * and Daniela Elena Popescu

Posted Date: 20 May 2026

doi: 10.20944/preprints202605.1386.v1

Keywords: post-quantum cryptography; IoT; post-quantum digital signatures; ML-KEM (Kyber); ML-DSA (Dilithium); PQC performance benchmarks; heterogeneous computing environments

 doi:10.20944/preprints202605.1386.v1

Article

Cross-Platform Performance and Security Evaluation of Post-Quantum Cryptographic Algorithms on Resource-Constrained Devices

Daiana-Larisa Lucaciu * and Daniela Elena Popescu

Department of Computers and Information Technology, Faculty of Electrical Engineering and Information Technology, University of Oradea, 410087 Oradea, Romania

* Correspondence: lucaciu.daianalarisa@student.uoradea.ro

Abstract

The rapid advancement of quantum computing poses a fundamental threat to classical public-key cryptographic systems, necessitating the transition to post-quantum cryptography (PQC). While significant progress has been made in the standardization of quantum-resistant algorithms, their practical deployment in heterogeneous environments—particularly resource-constrained Internet of Things (IoT) devices—remains a critical challenge. This study presents a comprehensive experimental evaluation of four NIST-standardized PQC algorithms: CRYSTALS-Kyber (ML-KEM), CRYSTALS-Dilithium (ML-DSA), FALCON, and SPHINCS+. The analysis is conducted across two distinctly different hardware platforms: a high-performance x86-64 system (AMD Ryzen 7 5700U) and a resource-constrained embedded microcontroller (ESP32-WROOM). Implementations were developed in C, Go, and Python to assess the influence of programming environments on algorithmic efficiency. The evaluation focuses on key performance indicators, including computational latency, memory consumption, communication overhead, and temporal determinism, based on extensive benchmarking over 1,000 iterations. Experimental results demonstrate significant trade-offs between security level, execution performance, and resource utilization. Lattice-based schemes such as Kyber and Falcon exhibit superior efficiency and scalability on embedded platforms, while Dilithium encounters stability limitations due to memory constraints. In contrast, SPHINCS+ provides strong security guarantees and architectural robustness at the cost of higher computational overhead. The findings highlight the critical role of hardware-specific optimizations and software design choices in enabling practical PQC deployment. This work provides actionable insights for selecting and tuning post-quantum algorithms in real-world scenarios, supporting the secure migration of IoT and distributed systems toward quantum-resilient infrastructures.

Keywords: post-quantum cryptography; IoT; post-quantum digital signatures; ML-KEM (Kyber); ML-DSA (Dilithium); PQC performance benchmarks; heterogeneous computing environments

1. Introduction

The emergence of the quantum computing paradigm is fundamentally restructuring the cybersecurity landscape, translating theoretical vulnerabilities into imminent threats to the integrity of global digital infrastructures. The approach of the inflection point where quantum computing becomes an applicable technological reality has catalyzed a vast and dynamic research effort aimed at the design, refinement, and practical implementation of post-quantum cryptographic (PQC) systems.

The design of lattice-based cryptographic accelerators, essential for post-quantum resilience, is based on a number of specific architectural paradigms and optimization techniques [1]. The central pillars of these systems include the exploitation of instruction-level parallelism, rigorous management of the memory hierarchy, and, critically, the computational acceleration of Number

 doi:10.20944/preprints202605.1386.v1

Theoretic Transformation (NTT). Translating these principles into solutions with broad practical applicability allows for high-performance implementations of CRYSTALS-Kyber, a NIST-validated key encapsulation mechanism (KEM). The efficiency of these systems is the result of the convergence of data flow planning strategies with micro-architectural optimizations [4]. By using optimized data pipelines and intelligent resource sharing mechanisms, processing latency is significantly reduced, without compromising the rigor of the security levels imposed by current standards [4].

The transition to the post-quantum era, however, goes beyond simple algorithmic selection, involving the multidimensional challenge of integrating resilient schemes into the fundamental protocols of digital communications. Transport Layer Security (TLS) and Public Key Infrastructure (PKI) are the current pillars of digital trust, and assessing their compatibility with PQC algorithms is an immediate research priority.

In parallel with these efforts, the trajectory of quantum computing capabilities has been explored through GPU-based simulators, providing critical insights into how quantum advantage can be realized in cryptanalysis of classical systems [9,10].

The overall objective of this work is to investigate the feasibility of efficiently implementing PQC algorithms on divergent hardware architectures, critically assessing the trade-off between cryptographic robustness and operational performance. The research begins with an in-depth analysis of the mathematical foundations, focusing on network-based structures and the optimization of polynomial operations through NTT-type transformations, essential elements for understanding the mechanisms of the ML-KEM scheme.

A central pillar of the study is the experimental evaluation of this algorithm at different security levels (NIST 1, 3 and 5), followed by quantifying the impact of parameterization on latency, memory and key size. This approach allows identifying the optimal compromise between security and efficiency, providing concrete recommendations for integrating PQC in real scenarios, from high-performance desktop systems to resource-constrained IoT devices, such as the ESP32 platform.

To provide a structured experimental framework, this study is guided by a set of explicit research hypotheses aimed at evaluating the practical feasibility of post-quantum cryptographic algorithms across heterogeneous hardware environments.

- H1: Lattice-based PQC algorithms, particularly CRYSTALS-Kyber and FALCON, provide a more favorable trade-off between computational latency and memory consumption compared to alternative schemes when deployed on resource-constrained IoT platforms.
- H2: The execution of PQC algorithms on embedded microcontrollers exhibits higher temporal determinism compared to desktop environments, reducing variability and potential susceptibility to timing-based side-channel effects.
- H3: High-security variants of digital signature schemes, such as ML-DSA (Dilithium), introduce significant memory and computational overhead that limits their feasibility on constrained hardware without specialized optimization.
- H4: Low-level implementations © achieve superior performance and resource efficiency on embedded platforms compared to high-level languages (Python and Go), due to reduced abstraction overhead and improved control over memory management.

These hypotheses are systematically evaluated through cross-platform benchmarking, enabling a rigorous assessment of the operational viability of PQC algorithms in real-world deployment scenarios.

The original contributions of this work are the following:

- Implementation and testing of four post-quantum cryptography algorithms (Kyber, Dilithium, Falcon and SPHINCS+) on the embedded ESP32-WROOM platform;
- Creation of a complete implementation in the Go language, used for comparative tests;
- Integration and metric evaluation of existing implementations in the Python environment;
- Comparative analysis of algorithms in terms of execution times and resource consumption;
- Identification of the optimal solution for the analyzed hardware environment;

 doi:10.20944/preprints202605.1386.v1

- Highlighting the influence of the programming language on performance in the context of resource-constrained systems.

This research substantiates the need for an accelerated and methodical transition to post-quantum cryptography, demonstrating through empirical data that securing digital infrastructures against quantum threats is not just a theoretical imperative, but an immediate technical possibility.

The rest of this paper is organized as follows: Section 2 reviews existing research and the limitations of current implementations. Section 3 details the mathematical foundations and algorithms analyzed. Section 4 describes the platforms and metrics used. Section 5 presents the development particularities. Section 6 analyzes the obtained data. Section 7 validates the research hypotheses based on the experimental results . Section 8 explores the security-performance trade-off. Section 9 summarizes the conclusions.

2. Related Work

The accelerated trajectory toward the realization of large-scale, fault-tolerant quantum computing governed by the theoretical implications of Shor's and Grover's algorithms has catalyzed a robust and multidimensional body of research dedicated to the design, optimization, and practical deployment of Post-Quantum Cryptographic (PQC) systems.

The current literature represents a highly interconnected ecosystem of research directions, ranging from low-level hardware acceleration and algorithmic subversion to high-level organizational policy frameworks and enterprise-wide migration strategies. This section provides a high-fidelity synthesis of the state-of-the-art, situating the current study within the broader scientific discourse and highlighting the research lacunae that necessitate this investigation.

2.1. Taxonomy of Resarch Direction and Key Contributions

The research landscape can be categorized into several primary domains, each addressing a specific facet of the quantum transition. Table 1 provides a structured synthesis of these domains and their representative contributions, emphasizing the shift from theoretical validation to operational implementation.

Table 1. Taxonomy of research directions and key contributions in Post-Quantum Cryptography.

Research Domain	Core Objective	Key Findings & Contributions	Ref.
Hardware Optimization	Performance & Throughput	Acceleration of the Number Theoretic Transform (NTT); optimization of memory hierarchy, parallelism, and pipeline scheduling.	[1,4]
Architectural Design	System Versatility & Agility	Development of hybrid FPGA platforms (e.g., HySecure) for concurrent support of legacy (ECC/RSA) and emerging PQC primitives.	[7]
Algorithmic Foundations	Mathematical Utility	Exploration of Ring-LWE and Module-LWE assumptions; evaluation of homomorphic properties for privacy-preserving computations.	[3,11]
Implementation Security	Vulnerability Analysis	Identification of "black-box" kleptographic vectors and side-channel subversion risks in real-world, third-party IoT deployments.	[5]
Protocol Integration	Communication Security	Impact analysis of PQC on Transport Layer Security (TLS) and Public Key Infrastructure (PKI); mitigation of handshake latency.	[13]
IoT Resilience	Resource Constraints	Empirical benchmarking of PQC resilience in lightweight protocols (LoRaWAN, 6LoWPAN) using ESP32 and ARM Cortex-M nodes.	[6,12,14]
Migration Strategies	Strategic Frameworks	Development of organizational "crypto-agility" models; assessment of "harvest now, decrypt later" (HNDL) risk profiles.	[2,8,10]

 doi:10.20944/preprints202605.1386.v1

2.2. Hardware Optimization and Co-Design Paradigms

The operational viability of PQC in performance-critical environments is fundamentally contingent upon the efficiency of hardware implementations. While software-based primitives facilitate rapid prototyping, they frequently fail to meet the stringent latency and throughput requirements of real-time embedded or industrial systems. Central to this challenge is the computational density of lattice-based schemes.

Research has identified the acceleration of the Number Theoretic Transform (NTT) as a critical bottleneck for schemes such as CRYSTALS-Kyber, where modular multiplication of high-degree polynomials demands significant arithmetic logic unit (ALU) resources [1].

Advanced implementation strategies now utilize sophisticated data pipelining, butterfly unit optimization, and memory-mapping mechanisms to achieve high-performance throughput without the prohibitive silicon area costs typically associated with PQC [4]. A pivotal development in this domain is the shift toward reconfigurable architectures exemplified by the HySecure platform [7].

Such systems leverage FPGA flexibility to address the dual necessity of maintaining backward compatibility with Elliptic Curve Cryptography (ECC) legacy protocols while ensuring modularity against evolving NIST standards. This hardware-software co-design approach ensures that hardware remains an enabler rather than a constraint during the transitional period, particularly in industrial IoT environments where field updates are logistically difficult.

2.3. Algorithmic Rigor and Extended Functionalities

Beyond the primary objective of confidentiality, the mathematical foundations of PQC specifically Learning With Errors (LWE) and its Ring/Module variants are being scrutinized for their secondary functional utilities. Recent studies have demonstrated that the inherent algebraic structure of algorithms like Kyber and McEliece can be harnessed for homomorphic properties, facilitating Private Set Intersection (PSI) and secure multi-party computation (SMPC) in quantum-resistant contexts [3]. This suggests that the migration to PQC may simultaneously enhance privacy-preserving capabilities across decentralized networks, such as contact tracing or private database querying.

However, a significant body of research highlights a persistent "semantic gap" between idealized theoretical security proofs and physical implementation integrity. The emergence of kleptographic subversion in IoT environments [5] serves as a critical case study; it illustrates that even a mathematically robust algorithm can be compromised if the implementation layer specifically the pseudo-random number generator (PRNG) or key generation logic is surreptitiously manipulated. Such vulnerabilities are particularly pernicious as they often bypass traditional black-box security audits and formal verification methods, necessitating a shift toward trustless implementation frameworks and supply-chain integrity monitoring.

2.4. Protocol Integration and Constrained Environment Performance

The integration of PQC into established frameworks like Transport Layer Security (TLS 1.3) and Public Key Infrastructure (PKI) introduces non-trivial overhead that propagates through the entire network stack. The substantial increase in public key and signature sizes characteristic of PQC often an order of magnitude larger than RSA or ECC counterparts leads to significant handshake latency and issues with Maximum Transmission Unit (MTU) limits, resulting in IP fragmentation [13].

These challenges reach a critical threshold in the Internet of Things (IoT) and Industrial IoT (IIoT) domains. Microcontrollers such as the ESP32 or LoRa-based nodes operate under severe SRAM and energy budgets.

Empirical evaluations [12,14] indicate that while PQC deployment is technically feasible, it requires "surgical" algorithmic selection. For instance, while Dilithium may offer superior computational speed, its signature size may preclude its use in packet-constrained LoRaWAN

 doi:10.20944/preprints202605.1386.v1

environments, whereas Falcon, despite its complex floating-point requirements, might be preferred for its compact data footprint [6].

Systematic analyses of these lightweight protocols identify handshake time, certificate management, and fragment reassembly as the primary bottlenecks constraining efficient PQC adoption.

2.5. Organizational Crypto-Agility and Migration Policy

The transition to a post-quantum state is a socio-technical endeavor as much as a cryptographic one. The concept of crypto-agility defined as the architectural capacity to switch cryptographic primitives with minimal systemic disruption is now recognized as a fundamental design requirement for resilient enterprise systems [2,11].

Current strategic frameworks [8,10] advocate for an iterative migration path, prioritizing "harvest-now, decrypt-later" (HNDL) threats, where adversaries collect encrypted data today in anticipation of breaking it with future quantum hardware.

Strategic contributions have addressed the heterogeneity of enterprise inventories, the dependencies between cryptographic components and business-critical operations, and the risks associated with premature migration to standards that may later be deprecated [2,8].

Furthermore, the utilization of High-Performance Computing (HPC) and GPU-based quantum simulators [9] has become indispensable for stress-testing these new standards.

By simulating Shor's and Grover's algorithms at increasing scales, researchers can calibrate migration timelines with greater empirical precision, ensuring that the transition occurs before a Cryptographically Relevant Quantum Computer (CRQC) becomes a reality.

2.6. Identification of Research Gaps

Despite the rapid maturation of the field and the significant progress documented in the aforementioned subsections, the synthesized literature reveals four critical research lacunae that this study seeks to address:

1. Deployment-Level Security and Supply-Chain Integrity: While theoretical security is well-documented, the vulnerability of PQC implementations to hardware-level kleptography and subversion in unmanaged IoT environments remains under-characterized. Current detection mechanisms are insufficient for identifying "malicious-by-design" implementations [5].
2. Lack of Standardized, Multi-Metric Benchmarking: Performance data is often fragmented across specific isolated platforms. There is a persistent need for comprehensive, reproducible benchmarking frameworks that simultaneously account for power consumption, memory throughput, and concurrent task execution (multitasking) in constrained hardware under realistic network jitter [12,14].
3. Fragmented Interoperability in Hybrid Transition States: The complexities of maintaining seamless interoperability during multi-stakeholder migrations where legacy and PQC systems must coexist for potentially a decade are not yet fully resolved, particularly regarding the negotiation of hybrid ciphersuites and the management of dual-certificate chains [8,13].
4. Operationalization of Strategic Frameworks: While high-level policies exist, there is a lack of actionable, low-level technical guidelines for small-to-medium enterprises (SMEs) to implement crypto-agility without incurring prohibitive costs.

The present study is motivated by these deficiencies, contributing a rigorous, methodologically integrated analysis of PQC system design and operational performance to bridge the divide between high-level algorithmic theory and the pragmatic realities of secure industrial engineering in the quantum era.

 doi:10.20944/preprints202605.1386.v1

3. Background and Preliminaries

3.1. Project Overview and Evaluation Methodology

To ensure a holistic and rigorous performance evaluation of Post-Quantum Cryptography (PQC) algorithms, the implementation and testing process was conducted on two heterogeneous hardware platforms. These platforms were strategically selected to cover the entire technological spectrum, ranging from high-performance computing infrastructures to peripheral nodes with severely limited resources. This methodological approach allows for the identification of critical trade-offs between the security levels offered by the new NIST standards and the operational costs associated with their integration into real-world digital ecosystems.

The research focuses on a systematic analysis of four fundamental algorithms: ML-KEM (Kyber), ML-DSA (Dilithium), FALCON, and SPHINCS+. These algorithms are evaluated not only for their mathematical efficiency but also for their capacity to be deployed on architectures with disparate hardware constraints. Programming languages were specifically adapted to each execution environment to maximize granular control over system resources.

3.1.1. Desktop Platform (High-Performance Computing)

The first platform utilized in the experiments is a laptop equipped with an AMD Ryzen 7 5700U processor. This processing unit, based on the modern x86-64 (Zen 2) architecture and manufactured using a 7 nm process, provides a testing environment characterized by raw computational power and advanced parallel processing capabilities.

- Processing Capabilities: Featuring 8 physical cores and 16 execution threads, the system can reach boost frequencies of up to 4.3 GHz, making it ideal for simulating server-side or gateway tasks.
- Software Ecosystem: On this platform, the algorithms were implemented using the Python and Go languages. The use of these languages allowed for the evaluation of algorithmic scalability in environments that benefit from hardware Floating-Point Units (FPU) and complex multi-level cache hierarchies (L1, L2, L3).
- Evaluation Objective: Testing on this architecture aims to determine upper performance limits and analyze the impact that software abstraction such as automatic memory management in Go has on cryptographic latency.

3.1.2. Embedded Platform (Edge Computing)

In contrast to the desktop system's processing power, the second platform is represented by the ESP-WROOM-32 microcontroller (based on the ESP32 chip). This selection is vital for the research, as "edge" devices represent the most vulnerable point in the transition to the post-quantum era due to their extremely limited resources.

- Architecture and Constraints: The device utilizes the Xtensa® Dual-Core 32-bit LX6 architecture and features an internal SRAM of only 520 KB. The absence of a complex operating system (operating bare-metal or with FreeRTOS) and a limited clock frequency (maximum 240 MHz) make memory management a critical challenge.
- Native Implementation: To extract maximum performance, the implementation on the ESP32 was carried out exclusively in the C language. This ensures granular control over the stack and the heap, which are vital for algorithms like ML-KEM or Dilithium that manipulate large polynomial matrices which can easily trigger stack overflow errors.
- IoT Importance: This platform is representative of industrial sensors, connected medical devices, and smart monitoring systems, where security must be implemented without sacrificing energy autonomy or real-time response times.

 doi:10.20944/preprints202605.1386.v1

3.1.3. Data Collection Methodology

The performance evaluation was structured over 1000 iterations for each algorithm and security level, utilizing monitoring tools such as the Arduino IDE for the embedded platform and Visual Studio Code with dedicated profilers for the desktop system. The tracked metrics include:

- Temporal Latency: Measured in microseconds (µs) or milliseconds (ms) for the stages of key generation (KeyGen), encapsulation/signing, and decapsulation/verification.
- Memory Consumption: Analysis of the static SRAM footprint and dynamic heap utilization.
- Stability and Determinism: Evaluation of the standard deviation (σ) to identify the implementation's resilience against timing attacks.

By combining these two heterogeneous hardware platforms, the study provides a complete perspective on the adaptability of post-quantum algorithms, preparing the ground for a secure and efficient migration of critical digital infrastructures.

3.2. CRYSTALS-Kyber (ML-KEM)

Key Encapsulation Mechanisms (KEMs) are cryptographic primitives designed to establish a shared secret key over a public channel without explicit transmission of the key itself. The resulting shared secret is subsequently utilized in symmetric cryptographic schemes to ensure the confidentiality and authenticity of communications.

The FIPS 203 standard (Module-Lattice-Based Key-Encapsulation Mechanism Standard), developed by the National Institute of Standards and Technology (NIST), defines the ML-KEM mechanism. This mechanism is based on the computational intractability of the Module Learning With Errors (MLWE) problem, which is a structured generalization of the Learning With Errors (LWE) problem introduced by Regev in 2005[15–18].

The scheme includes three specific parameter sets ML-KEM-512, ML-KEM-768, and ML-KEM-1024 which reflect varying trade-offs between security levels and computational efficiency.

3.2.1. Key Encapsulation Mechanisms

A Key Encapsulation Mechanism (KEM) is defined by three fundamental algorithms: KeyGen (key generation), Encaps (encapsulation), and Decaps (decapsulation), alongside a collection of parameter sets that dictate security strength and performance costs.

Figure 1 illustrates the KEM operational flow between two entities, Alice and Bob. In a typical deployment scenario:

1. Alice generates a key pair using the KeyGen algorithm and publishes the encapsulation key (public key).
2. Bob utilizes this public key to execute the Encaps algorithm, resulting in a shared secret K and a ciphertext, which is then transmitted to Alice.
3. Alice applies the Decaps algorithm to the received ciphertext using her private key to recover the shared secret K'.

Ideally, the relation K' = K is achieved, ensuring both parties share an identical secret key.

 doi:10.20944/preprints202605.1386.v1

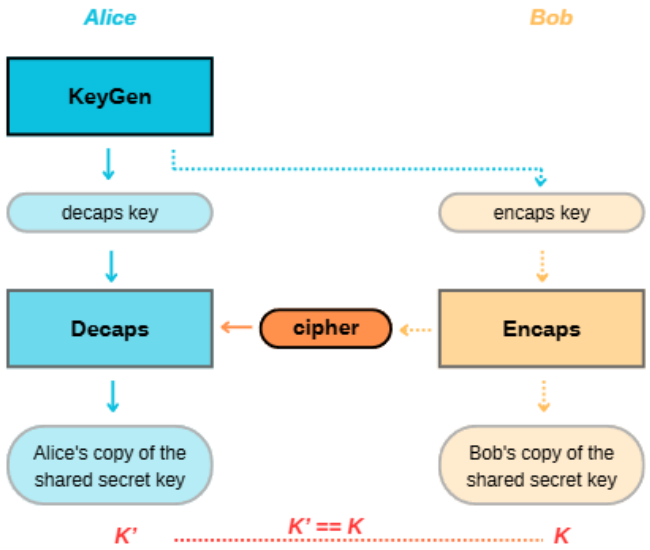

Figure 1. Key Encapsulation Mechanism for Entities Alice and Bob.

3.2.2. Security and Mathematical Foundations

The security of ML-KEM relies on the difficulty of solving the MLWE problem, which involves determining an unknown secret from a set of linear equations perturbed by random noise. This noise injection renders classical linear system-solving methods, such as Gaussian elimination, computationally inefficient.

In the standard LWE problem, the secret is a vector $x \in \mathbb{Z}_q^n$, and the adversary receives a set of linear equations affected by noise. The objective is to recover the secret vector based on these observations. This problem is considered significantly difficult and serves as the foundation for several modern cryptographic constructions, including public-key encryption schemes.

The MLWE problem extends LWE by replacing the vector space $\mathbb{Z}_q^n$ with a module of the form R_q^k, where R_q is a modular polynomial ring. In this context, the secret is an element within an algebraically structured module, which enables more efficient data representation and optimized implementations of cryptographic schemes.

The generalized architecture (Figure 2) describes the integration of a specialized hardware accelerator into a heterogeneous computing system. The construction of ML-KEM is realized in two primary stages:

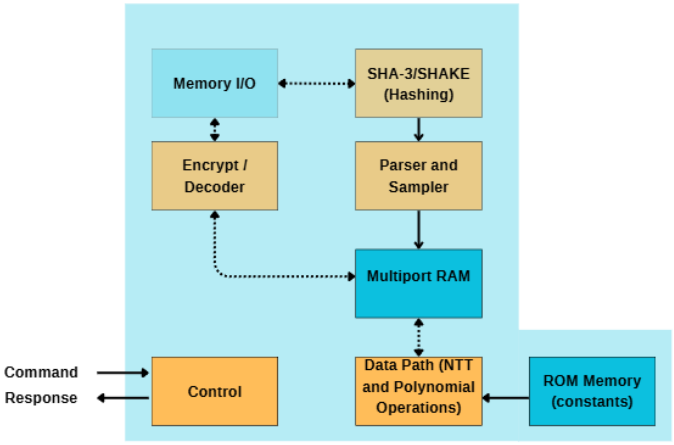

Figure 2. Generalized Architecture of the ML-KEM Implementation.

 doi:10.20944/preprints202605.1386.v1

Stage 1: A Public-Key Encryption (PKE) mechanism is constructed based on the MLWE problem. The KeyGen, Encaps, and Decaps algorithms are derived from this construction, utilizing operations on polynomial ring elements and controlled noise injection to ensure security.

Stage 2: The PKE mechanism is transformed into a KEM by applying the Fujisaki-Okamoto (FO) transform. This transformation strengthens the security of the scheme, elevating it from the IND-CPA model to the more robust IND-CCA2 model, which is resistant to adaptive chosen-ciphertext attacks. The FO transform introduces additional verification and key derivation mechanisms, ensuring that only a valid ciphertext can produce the correct shared secret key.

3.2.2. Key Generation (KeyGen)

The KeyGen algorithm is responsible for generating a cryptographic key pair: a public (encryption) key and a private (decryption) key, utilizing an initial random value.

The algorithm receives a random seed $d \in \mathbb{B}^{32}$ as input, from which two pseudorandom seeds, ρ and σ, are derived through an expansion function. The seed , ρ is utilized for the deterministic generation of matrix A, while σ is used to generate the secret values and the noise.

Matrix A is constructed element-by-element using a pseudorandom function. Subsequently, two polynomial vectors are generated: the secret vector s and the noise vector e, whose components are sampled from a Centered Binomial Distribution (CBD). For computational efficiency, these vectors are transformed into the NTT domain. The vector t = A • s` + e` [18] is then calculated, representing a system of "noisy" linear equations relative to the secret s.

The public key consists of the encoding of t along with the seed ρ, allowing for the reconstruction of matrix A. The private key is represented by the encoding of the secret vector s. This structure forms the security basis of the scheme, as recovering s from the public key is computationally difficult due to the presence of noise[21].

Algorithm 1: KeyGen

Input: $d \in \mathbb{B}32$.

Output: ekPKE .$\in \mathbb{B}384k$+32

Output: dkPKE . $\in \mathbb{B}384k$

Process:

1. $(\rho, \sigma) \leftarrow G(d \| k)$
2. $N \leftarrow 0$
3. For each i, j, generate matrix A: $A[i,j] \leftarrow$ SampleNTT($\rho \| j \| i$)
4. For each i, generate secret vector s: $s[i] \leftarrow$ SamplePolyCBDη1(PRFη1(σ, N))
5. $N \leftarrow N + 1$
6. For each i, generate noise vector e: $e[i] \leftarrow$ SamplePolyCBDη1(PRFη1(σ, N))
7. $N \leftarrow N + 1$
8. s`← NTT(s), e` ← NTT(t)
9. t = A∘s`+ e`
10. ekPKE ← ByteEncode12(t)$\| \rho$
11. dkPKE ← ByteEncode12(s)
12. return (ekPKE, dkPKE)

3.2.3. Encapsulation (Encaps)

The encapsulation algorithm in ML-KEM is designed to generate a shared secret key and an associated ciphertext using the encapsulation key, also known as the public key. The process consists of an input key verification phase followed by the execution of an internal encapsulation algorithm that produces the final results.

Prior to encapsulation, the encapsulation key undergoes a compliance check to verify it has the correct length of 384k + 32 bytes and that its encoded values fall within the valid range [0, q - 1]. These checks ensure the input conforms to the expected format, although they do not guarantee its original source from the KeyGen process. If these validations fail, the algorithm is terminated.

 doi:10.20944/preprints202605.1386.v1

Once the key is validated, the internal algorithm Encaps_internal is utilized to combine the public key with a random message to generate both the shared key and the ciphertext. Specifically, a random message m is generated, from which a shared key K and a randomization value r are derived in conjunction with the hash of the public key. The message is then encrypted using the K-PKE scheme, resulting in the ciphertext c. The pair (K, c) is returned as the final output.

Algorithm 2: Encaps(ek)

1. Verify if ek has the length 384k + 32
2. Verify if the values within ek are in the interval [0, q - 1]
3. If checks fail, stop with an error
4. m ← B32 (random message)
5. If the generation of m fails, return ⊥
6. (K, c) ← ML-KEM.Encaps_internal(ek, m)
7. Return (K, c)

Algorithm 3: Encaps_internal(ek, m)
Input: ek ∈ B384k+32, m ∈ B32

1. (K, r) ← G(m || H(ek))
2. c ← K-PKE.Encrypt(ek, m, r)
3. Return (K, c)

3.2.4. Decapsulation

The decapsulation algorithm (Decrypt) is designed to recover the original message m from a ciphertext c using the decryption dkPKE. This process is deterministic and does not require external sources of randomization, as all necessary information is contained within the private key and the ciphertext.

The procedure begins by splitting the ciphertext c into two distinct components, c_1 and c_2.These components are subsequently decoded and decompressed to retrieve the polynomial representations u' and v'. Simultaneously, the secret vector s is extracted from the decryption key dkPKE and decoded from its binary format.

Next, the algorithm reconstructs the "true" term of the equation by calculating the scalar product between the secret vector s and the ciphertext component u' within the NTT domain. This result is converted back via the inverse NTT and subtracted from v' to obtain an approximation of the original message, which remains affected only by the controlled noise introduced during encryption. Finally, this value is subjected to an inverse compression and decoding operation, resulting in the message m in its initial form.

Algorithm 4: Algoritmul Decrypt(dkPKE,c)
Input: dkPKE ∈ $\mathbb{B}$^(384k), c ∈ $\mathbb{B}$^(32($d_u k + d_v$))

1. $c_1 \leftarrow c[0 : 32d_u k]$
2. $c_2 \leftarrow c[32d_u k : 32(d_u k + d_v)]$
3. u′ ← Decompressd_u(ByteDecoded$_{d_u}$(c_1))
4. $v' \leftarrow$ Decompressd_v(ByteDecoded$_{d_v}$(c_2))
5. s ← ByteDecode12(dkPKE)
6. $w \leftarrow v' - \text{NTT}^{-1}(s^T \circ \text{NTT}(u'))$
7. $m \leftarrow$ ByteEncode1(Compress1(w))
8. return m

Within the ML-KEM and K-PKE schemes, the ByteEncode and ByteDecode algorithms are employed to perform conversions between the internal polynomial representation (vectors of integers modulo q) and their binary representation as byte strings. These algorithms are critical for data serialization and deserialization, enabling the efficient storage and transmission of keys and ciphertexts.

 doi:10.20944/preprints202605.1386.v1

The ByteEncoded algorithm takes a vector of 256 integer coefficients as input and converts them into a bitstream, with each coefficient represented using d bits.

This conversion is performed by decomposing each coefficient into a bit-by-bit binary representation and concatenating them into a bit string, which is then transformed into a byte vector.

The parameter d controls the number of bits used for each coefficient based on the value range (d=12 for coefficients modulo q).

The ByteDecoded algorithm performs the inverse operation: it receives a byte vector and transforms it into a vector of integer coefficients [20].

It first converts bytes into bits and then reconstructs each coefficient by combining d consecutive bits using weights as powers of 2.

The result is a vector of integers modulo m (where m=2^d or m=q for d=12), corresponding to the internal representation of the polynomials.

Algorithm 5: ByteEncoded(F)

Input: vector $F \in \mathbb{Z}_m^{256}$

1. For each $i = 0 \ldots 255$:
2. $a \leftarrow F[i]$
3. For each $j = 0 \ldots d-1$:
4. $b[i \cdot d + j] \leftarrow a \bmod 2$
5. $a \leftarrow (a - b[i \cdot d + j]) / 2$
6. $B \leftarrow$ BitsToBytes(b)
7. return $B \in \mathbb{B}$^(32d)

Algorithm 6: Algoritmul ByteDecoded(B)

Input: $B \in \mathbb{B}$^(32d)

1. $b \leftarrow$ BytesToBits(B)
2. For each $i = 0 \ldots 255$:
3. $F[i] \leftarrow \Sigma\ (j = 0 \ldots d-1)\ b[i \cdot d + j] \cdot 2^j \bmod m$
4. return $F \in \mathbb{Z}_m^{256}$

3.2.6. SHAKE128 and the XOF Wrapper Interface

The SHAKE128 algorithm serves as a versatile extendable-output function (XOF) used within ML-KEM for pseudorandom bit generation and seed expansion. The process is characterized by an "absorb-then-squeeze" paradigm, where input data is absorbed into the internal state before the desired number of output bytes are squeezed out.

Algorithm 7: Algoritmul SHAKE128example($str_1,\ldots,str_m$, $b_1,\ldots,b\ell$)

Input: Byte arrays: $str_1,\ldots,str_m$, Positive integers: $b_1,\ldots,b\ell$

Output: A concatenated byte array of total length ∑_{j=1}^{ℓ} b_j

1. ctx ← SHAKE128.Init() // Initialize the internal sponge context
2. for i ← 1 to m do
3. ctx ← SHAKE128.Absorb(ctx, str_i) // Progressively absorb each input string
4. end for
5. for j ← 1 to ℓ do
6. (ctx, out_j) ← SHAKE128.Squeeze(ctx, 8 · b_j) // Extract b_j bytes from the state
7. end for
8. output ← out_1 ‖ out_2 ‖ … ‖ outℓ // Concatenate resulting output segments
9. return output

In ML-KEM, the SHAKE128 [19] functions are utilized exclusively through a wrapper called XOF. This interface is defined as follows:

- XOF.Init() = SHAKE128.Init()
- XOF.Absorb(ctx, str) = SHAKE128.Absorb(ctx, str)
- XOF.Squeeze(ctx, ℓ) = SHAKE128.Squeeze(ctx, 8 · ℓ)

 doi:10.20944/preprints202605.1386.v1

The XOF.Squeeze function receives the output length in bytes, thereby standardizing operations at the byte array level and eliminating ambiguities related to bit-lengths within the ML-KEM algorithm.

3.2.5. ML-KEM Parameters

ML-KEM is defined by three parameter sets (Table 2), each utilizing five variables k, η_1, η_2, d_u and d_v and constants n = 256 and q = 3329. The dimension k determines the size of matrix $\hat{A}$ and associated vectors[20]. Noise distributions are governed η_1 and η_2, while $_2$, d_u and d_v control compression and encoding.

These sets target NIST security categories 1, 3, and 5, mapped to ML-KEM-512, ML-KEM-768, and ML-KEM-1024, respectively. NIST recommends ML-KEM-768 as the default for its balance of security and efficiency.

Table 2. ML-KEM Parameters.

Parameter	ML-KEM-512	ML-KEM-768	ML-KEM-1024
n	256	256	256
q	3329	3329	3329
k	2	3	4
η_1	3	2	2
η_2	2	2	2
d_u	10	10	11
d_v	4	4	5
Strength RBG (bits)	128	192	256

3.3. CRYSTALS-Dilithium

ML-DSA (Module-Lattice-Based Digital Signature Algorithm) [22] is a digital signature scheme based on the CRYSTALS-Dilithium construction, designed to provide security in a post-quantum context. The scheme is composed of three fundamental algorithms: ML-DSA.KeyGen, which generates the public and private key pairs; ML-DSA.Sign, used for generating digital signatures; and ML-DSA.Verify, responsible for their verification.

ML-DSA is based on the Fiat-Shamir with Aborts paradigm, an extension of the Fiat-Shamir transformation [25] applied to interactive protocols. This framework enables the creation of a non-interactive signature scheme with robust security properties. Additionally, the standard defines a derived variant known as HashML-DSA, which introduces an additional pre-hashing step of the message before signing. In this variant, the key generation algorithm remains unchanged, while the signing and verification procedures are adapted to operate on a hash representative of the message.

Figure 3. Schematic of the CRYSTALS-Dilithium algorithm.

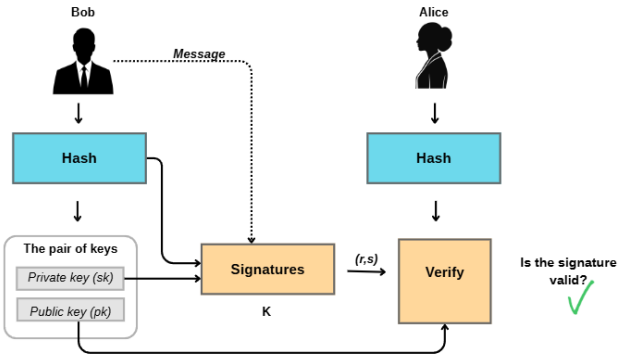

Figure 3. illustrates the operational workflow of the Dilithium digital signature algorithm, which is built upon lattice-based cryptography (euclidean networks).

 doi:10.20944/preprints202605.1386.v1

3.3.1. Properties

ML-DSA [22] is designed to satisfy the property of strong existential unforgeability under chosen message attack (SUF-CMA). This property guarantees that an adversary, even when provided with a signing oracle that allows obtaining signatures for adaptively chosen messages, cannot generate a new valid signature for any message. Furthermore, this definition encompasses the impossibility of producing a distinct signature for a message that has already been signed.

The security of ML-DSA is founded upon the hardness of fundamental problems in lattice-based cryptography, specifically Learning With Errors (LWE) and Short Integer Solution (SIS) [26].

The construction of ML-DSA is inspired by the Schnorr signature scheme, adapted to the framework of lattice-based cryptography. In the Schnorr scheme, an interactive protocol between a prover and a verifier is transformed into a non-interactive signature scheme by applying the Fiat-Shamir heuristic. The protocol involves three stages: the generation of a commitment, the derivation of a challenge, and the calculation of a response [22]. In the non-interactive variant, the challenge is obtained by applying a hash function to the commitment and the message.

A critical issue with this construction is the emergence of statistical biases in the distribution of generated values, which can lead to information leakage regarding the private key. To prevent this phenomenon, ML-DSA utilizes the rejection sampling technique, whereby values that do not meet certain constraints are rejected, and the signing process is restarted.

To optimize performance and reduce key sizes, ML-DSA introduces several enhancements:

- The utilization of module structures over the ring, which allow for the efficient implementation of operations via the Number Theoretic Transform (NTT);
- The pseudorandom generation of the public matrix A from a fixed-size seed;
- The compression of the public key by eliminating the least significant bits;
- The signing of a hash representative of the message, rather than the original message itself, to improve security properties.

3.3.2. The ML-DSA Scheme

The ML-DSA framework is operationalized through three fundamental algorithms: Key Generation, Signing, and Verification, as formally specified in the FIPS 204 standard [22,23].

Algorithm 8: ML-DSA.KeyGen

The key generation procedure derives a public and private key pair by employing deterministic expansion from an initial stochastic seed [23].

Input: Scheme-specific parameters

Output: Public key pk, Secret key sk

1. Generate a random seed ξ.
2. $(\rho, \rho', K) = \text{Expand}(\xi)$
3. Derive the public matrix $A \in R_q^{k \times \ell}$ deterministically from the seed ρ.
4. $s_1 \leftarrow \chi_\eta^\ell, s_2 \leftarrow \chi_\eta^k$
5. $Compute\ the\ public\ vector\ t = As_1 + s_2$
6. $(t_1, t_0) = \text{Power2Round}(t)$
7. $pk = (\rho, t_1)$
8. $sk = (\rho, K, tr, s_1, s_2, t_0)$, unde $tr = H(pk)$

Algorithm 9: ML-DSA.Sign

The signing algorithm [23] generates a cryptographic signature by applying the private key to a message M through the Fiat-Shamir with Aborts mechanism [25].

Input: Secret key sk Message M

Output: Digital signature σ

1. $\mu = H(tr \parallel M)$
2. A derivation seed ρ' is generated using the secret key sk and the message M.
3. $y \leftarrow \chi_{\gamma_1}^\ell$

 doi:10.20944/preprints202605.1386.v1

4. $w = Ay$
5. $(w_1, w_0) = \text{Decompose}(w)$
6. $c = H(\mu \parallel w_1)$
7. $z = y + cs_1$
8. The norm conditions for z are verified; if these constraints are not satisfied→ restart.
9. $r_0 = w_0 - cs_1$
10. The conditions regarding w - cs_1 are verified; if they are not satisfied → restart. The auxiliary information (hint) h is then calculated.
11. return $\sigma = (z, h, c)$

Algoritmul 10: ML-DSA.Verify

The verification algorithm assesses the validity of a digital signature by utilizing the associated public key [23].

Input: Public key pk, Message M, Signature $\sigma = (z, h, c)$

Output: Accept / Reject

1. The magnitude of the response vector z is validated against predefined limits; if these bounds are not respected→ reject.
2. tr=H(pk),μ=H(tr‖M)
3. The public matrix A is reconstructed from the seed ρ.
4. $w' = Az - ct_1 \cdot 2^d$
   The auxiliary hint h is utilized to reconstruct the commitment w_1'.
5. $c' = H(\mu \parallel w_1')$
6. If the equality $c' = c$→ accept, otherwise → reject.
7.

3.3.3. ML-DSA Parameters

The ML-DSA parameter sets [23], as detailed in Table 3, are utilized for keygeneration,signing, and signature verification within the ML-DSA algorithms. The nomenclature for each parameter set follows the format ML-DSA-$k\ell$, where (k,ℓ) represent the dimensions of matrix A [23,24].

The "Repetitions" parameter refers to the expected number of iterations of the main loop within the signing algorithm.

Each parameter set is engineered to comply with specific security categories defined by NIST. Security strength is not characterized by a single bit-count; rather, each set is considered to offer security at least equivalent to a generic block cipher of a corresponding key size or a hash function with an equivalent output length [24].

Furthermore, public and private keys can be optimized; for instance, storing only a 32-byte seed for the private key is sufficient to derive all other constituent components.

Table 3. ML-DSA Parameters.

Parameter	Description	ML-DSA-44	ML-DSA-65	ML-DSA-87
q	Modulus	8,380,417	8,380,417	8,380,417
ζ	512th root of unity in $\mathbb{Z}q$	1753	1753	1753
d	Number of bits dropped from t	13	13	13
τ	Number of ±1in polynomial c	39	49	60
λ	Collision resistance	128	192	256
$\gamma 1$	Range of coefficients for y	217	219	219
$\gamma 2$	Low-order rounding range	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k, ℓ)	Dimensions of matrix A	(4,4)	(6,5)	(8,7)
η	Private key range	2	4	2
$\beta = \tau \cdot \eta$	Product	78	196	120
ω	Maximum number of 1s in hint h	80	55	75
Repetitions	Average iterations in signing algorithm	4.25	5.1	3.85
Security Category	NIST Classification	2	3	5

 doi:10.20944/preprints202605.1386.v1

3.4. FALCON

Falcon is a lattice-based digital signature scheme [27,30]. Its name is an acronym for "Fast Fourier lattice-based compact signatures over NTRU," reflecting its core architectural components.

At a conceptual level, the design of the Falcon scheme is relatively elegant, as it is constructed by instantiating the theoretical framework proposed by Gentry, Peikert, and Vaikuntanathan (GPV) [31]. This framework was specifically designed for developing lattice-based hash-and-sign signature schemes. The GPV paradigm necessitates the integration of two fundamental components:

A specific class of cryptographic lattices;

A robust trapdoor sampling mechanism.

In the implementation of Falcon, these components are realized through specialized techniques:

NTRU-type lattices [28,29] are utilized due to their advanced algebraic structures based on polynomial rings;

A proprietary technique known as fast Fourier sampling is employed, facilitating the efficient generation of signature elements that conform strictly to a target discrete Gaussian distribution.

In this context, the selection of the lattice class is paramount for the practical instantiation of the GPV framework [31]. From a design standpoint, if the emphasis were placed exclusively on maximum security without regard for efficiency, standard lattices which lack additional algebraic structure would be preferred. This approach is seen in the Frodo key exchange scheme [31]. However, Falcon is built upon the guiding principle of compactness, necessitating the use of NTRU lattices as introduced by Hoffstein, Pipher, and Silverman [32].

These lattices feature a ring-like structure that provides several significant technical advantages:

- A reduction in public key dimensions by a factor of order O(n);
- The acceleration of cryptographic operations by a factor of at least O(n / log n);
- The representation of the public key via a single polynomial h of degree at most n.

Furthermore, the integration of NTRU lattices within the GPV framework is supported by substantial theoretical research. Notably, Stehlé and Steinfeld [33] demonstrated that this specific instantiation can be executed in a provably secure manner, bridging the gap between theoretical security and practical performance.

While compactness represents a primary advantage, it is strictly balanced against security requirements. From this perspective, NTRU lattices benefit from over two decades of intensive cryptanalytic scrutiny, during which they have remained fundamentally secure. The parametrization utilized in Falcon is specifically chosen to mitigate known vulnerabilities and bolster resistance against classical and quantum attacks, ensuring the long-term robustness of the scheme [28].

In summary, the Falcon scheme can be conceptually expressed by the following relationship:

$$\text{Falcon} = \text{GPV framework} + \text{NTRU lattices} + \text{Fast Fourier sampling [27]}$$

The signing process involves a hash-and-sign methodology where a message is mapped to a point in the lattice space. The signer then uses a secret trapdoor to find a nearby lattice point using the Fast Fourier Nearest Neighbor (FFNN) algorithm. This allows for signatures that are both small in size and extremely fast to verify, making Falcon one of the most efficient candidates in the NIST post-quantum standardization process.

Figure 4 systematizes the security mechanism of the FALCON algorithm, a post-quantum standard based on the mathematical complexity of NTRU lattice problems.

 doi:10.20944/preprints202605.1386.v1

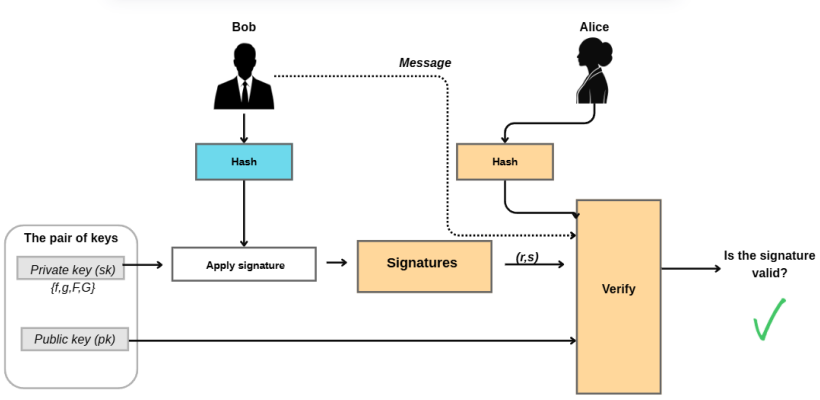

Figure 4. Schematic of the FALCON signature based on Euclidean lattices.

3.4.1. NTRU Lattices

The instantiation of the GPV framework [31] necessitates a specific class of cryptographic lattices coupled with a trapdoor sampling mechanism [27,28]. Falcon utilizes NTRU-type lattices because their inherent algebraic structure allows for the compact representation of keys and significantly enhances computational efficiency. This is paired with a sampler based on fast Fourier sampling.

NTRU lattices are defined by polynomials that satisfy a specific equation, allowing the public key to be reduced to a single polynomial h. In contrast, the secret key consists of short polynomials. The fundamental security of the scheme rests upon the NTRU assumption: the computational hardness of reconstructing these short secret polynomials from the public h [27,29].

Within the GPV framework, the public and secret keys correspond to different bases of the same lattice. A signature for a message is generated by identifying a short vector that satisfies a linear relationship modulo q. Both the efficiency and the security of this process are dependent on the quality of the trapdoor sampler. Falcon employs a variant based on the Fast Fourier Transform (FFT), which combines the theoretical precision of classical methods (such as Klein's algorithm) with the high-speed performance required for structured lattice algorithms.

3.4.2. Key Generation

In Falcon-M, the public key is obtained by multiplying two randomly generated polynomials in the frequency domain, without the utilization of a trapdoor.

Algorithm 11: Falcon-M Key Generation

Input: n - degree of the polynomials; q- modulus

Output: h(x)- public key; (a(x), b(x))- private key

1. Randomly generate polynomials a(x), b(x) ∈ $Z_q[x]/(x^n + 1)$
2. A(ω) ← FFT(a(x))
3. B(ω) ← FFT(b(x))
4. H(ω) ← A(ω) · B(ω) // element-wise multiplication
5. h(x) ← IFFT(H(ω)) mod q
6. return h(x), (a(x), b(x))

The polynomials a(x) and b(x) are generated randomly, and the public key h(x) is derived from their product in the Fourier domain. This construction eliminates the necessity for a trapdoor and produces public keys that exhibit a pseudorandom appearance.

3.4.3. Signature Generation

The signing process involves the application of a cryptographic hash function and a discrete Gaussian distribution to produce a message-dependent signature.

Algorithm 12: Falcon-M Signature Generation

 doi:10.20944/preprints202605.1386.v1

Input:(a(x), b(x))-: private key ; m- message
Output σ(x) – signature

1. H(m) ← SHA-512(m) mod q
2. s ← Dσ(H(m)) / / discrete Gaussian sampling centered at H(m)
3. S(ω) ← FFT(s)
4. σ(x) ← IFFT(S(ω)) mod q
5. return σ(x)

The message is initially hashed, after which a vector s is generated from a discrete Gaussian distribution centered at the hash value. This vector is transformed into the frequency domain and subsequently reconverted to produce the final signature.

3.4.4. Signature Verification

Verification consists of comparing the result derived from the signature with the original message hash.

Algorithm 13: Falcon-M Signature Verification

- Input: h(x)-public key; m- message; σ(x)- signature
- Output: Accept / Reject

1. H(m) ← SHA-512(m) mod q
2. H(ω) ← FFT(h(x))
3. Σ(ω) ← FFT(σ(x))
4. Y(ω) ← H(ω) · Σ(ω)
5. y ← IFFT(Y(ω)) mod q
6. If ||y − H(m)||∞ ≤ δ then
7. Accept

Else
Reject

The procedure verifies whether the relationship between the public key and the signature reproduces the message hash within the constraints of a specific threshold. If this condition is satisfied, the signature is deemed valid.

3.4.5. Security Parameters for NIST Levels

The parameters employed in the Falcon-M scheme (Table 4) are selected in accordance with the security levels defined by NIST for post-quantum algorithms. These parameters directly influence the dimensions of the keys and signatures, as well as the degree of resistance against lattice-based cryptanalytic attacks.

Table 4. Falcon-M Parameters for NIST Security Levels.

Parameter	NIST Level I	NIST Level V
Ring Degree (n)	512	1024
Modulus (q)	12289	12289
Standard Deviation (σ)	165.736617183	168.388571447
σ_{min}	1.277833697	1.298280334
σ_{max}	1.8205	1.8205
Maximum Squared Norm	34,034,726	70,265,242
Public Key Size (bytes)	897	1793
Signature Size (bytes)	666	1280

3.5. SPHINCS+

SPHINCS+ [34], standardized under the designation **SLH-DSA**, represents a post-quantum digital signature scheme based exclusively on hash functions, designed to eliminate reliance on

 doi:10.20944/preprints202605.1386.v1

mathematical structures vulnerable to quantum computation. Its defining characteristic is its "stateless" nature, meaning it does not require the management of internal key states between signings, unlike other traditional hash-based schemes.

The SLH-DSA architecture is composite, constructed from several interdependent cryptographic mechanisms. At its foundation is WOTS+ [35], a one-time signature scheme used to sign individual messages through hash chains. This is integrated into XMSS [35,36], a multi-time scheme that organizes a large number of WOTS+ keys into a Merkle tree, enabling the signing of multiple messages and the efficient authentication of public keys via the tree root. At the upper level, the scheme utilizes FORS (Forest of Random Subsets), a few-time signature mechanism that allows for the signing of a limited number of messages using pseudorandom selections of key subsets.

The signing process in SLH-DSA begins with the calculation of a randomized hash of the message, utilizing a secret component of the private key. The result is split into two parts: one used to pseudorandomly select a FORS key [37] and the other to be effectively signed. The FORS signature produces an intermediate public key, which must subsequently be authenticated. This authentication is achieved through a hierarchical structure called a hypertree, composed of multiple levels of XMSS trees. Each level signs the public key of the level below it, forming a chain of trust up to the final root, which constitutes the global public key.

Key structures in SLH-DSA [34] are simplified through the use of seed values. The public key contains the hypertree root and a seed used for domain separation, while the private key includes the seed from which all internal keys are generated, as well as a value used for hash randomization. This approach allows for the deterministic regeneration of all necessary keys, eliminating the need for the explicit storage of a vast number of keys.

The primary advantage of the SLH-DSA scheme is its high degree of security, based solely on the properties of hash functions, which are considered resistant to quantum attacks. Additionally, removing the need for state management makes it more robust in practical implementations. However, these benefits come with certain costs, particularly large signature sizes and lower performance compared to other post-quantum schemes, such as CRYSTALS-Dilithium [34,35].

WOTS+ verifies the signature by reconstructing the public key from the signature and the message. The message is converted to base w, a checksum is calculated, and then each signature element is processed through the corresponding hash chain. The resulting values are compressed to obtain the candidate public key, which is compared with the original public key.

Figure 5 illustrates how SPHINCS+ combines multiple hash-based schemes to create a stateless post-quantum digital signature.

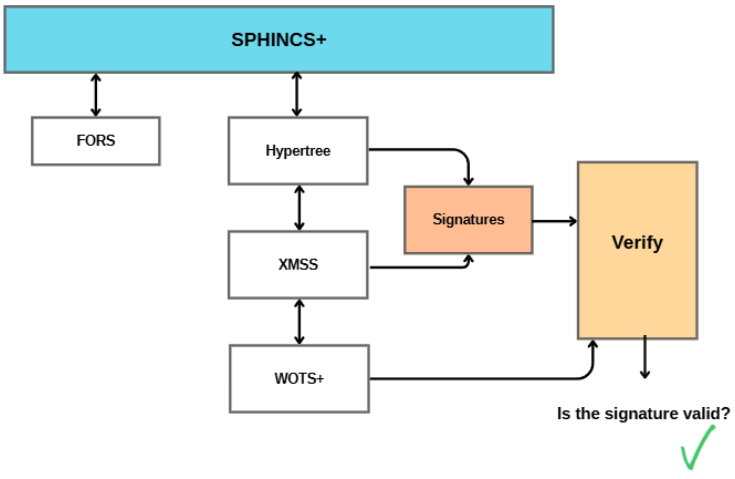

Figure 5. Architecture of the SPHINCS+ digital signature algorithm.

 doi:10.20944/preprints202605.1386.v1

3.5.1. Fundamental Components

SPHINCS+ is constructed by combining several cryptographic primitives:

- FORS (Forest of Random Subsets) used for message signing.
- WOTS+ (Winternitz One-Time Signature Plus) used for signing intermediate keys.
- XMSS / Merkle Trees for signature authentication.
- Hypertree a hierarchical structure of Merkle trees.

The general structure allows for signature validation through a chain of successive authentications, ranging from FORS [34] to the root of the main tree.

3.5.2. SPHINCS+ Structure (Hypertree)

The hypertree is an arborescent structure composed of multiple levels of Merkle trees [37]. Each internal node represents the root of a subtree, while the leaves consist of WOTS+ public keys.

This structure facilitates:

1. The reduction of public key dimensions;
2. The reuse of authentication mechanisms;
3. Multi-level scalability.

3.5.3. Signing Algorithm

Signing in SPHINCS+ involves several successive stages, starting from message randomization to the hierarchical authentication of the few-time signature.

Algorithm 14: SPHINCS_sign(M, SK, addrnd)

1. // Generate randomizer
   R ← PRF_msg(SK.prf, addrnd, M)
2. // Compute message digest
   digest ← H_msg(R, PK.seed, PK.root, M)
3. // Parameter extraction
   (md, idx_tree, idx_leaf) ← parse(digest)
4. // FORS signing
   sig_fors ← FORS_sign(md, SK.seed, PK.seed, ADRS)
5. // Reconstruct FORS public key
   pk_fors ← FORS_pkFromSig(sig_fors, md, PK.seed, ADRS)
6. // Hypertree signing
   sig_ht ← HT_sign(pk_fors, SK.seed, PK.seed, idx_tree, idx_leaf)
7. // Construct final signature
   SIG ← R ‖ sig_fors ‖ sig_ht
8. return SIG

3.5.4. Verification Algorithm

The verification process reconstructs the signing steps and validates the consistency of the results against the global public key.

Algorithm 15: SPHINCS_verify(M, SIG, PK)

1. // Component separation
   (R, sig_fors, sig_ht) ← parse(SIG)
2. // Recalculate digest
   digest ← H_msg(R, PK.seed, PK.root, M)
3. // Parameter extraction
   (md, idx_tree, idx_leaf) ← parse(digest)
4. // Reconstruct FORS public key
   pk_fors ← FORS_pkFromSig(sig_fors, md, PK.seed, ADRS)

 doi:10.20944/preprints202605.1386.v1

5. // Hypertree verification
   valid ← HT_verify(pk_fors, sig_ht, PK.seed, idx_tree, idx_leaf, PK.root)
6. return valid

3.5.5. XMSS and Merkle Trees

XMSS is utilized for constructing the nodes within the hypertree. The following procedure describes the recursive generation of XMSS nodes:

xmss_node(SK.seed, i, z, PK.seed, ADRS):

1. if z == 0:
2. ADRS.setType(WOTS_HASH)
3. ADRS.setKeyPairAddress(i)
4. return wots_pkGen(SK.seed, PK.seed, ADRS)
5. l ← xmss_node(SK.seed, 2i, z−1, PK.seed, ADRS)
6. r ← xmss_node(SK.seed, 2i+1, z−1, PK.seed, ADRS)
7. ADRS.setType(TREE)
8. ADRS.setTreeHeight(z)
9. ADRS.setTreeIndex(i)
10. return H(PK.seed, ADRS, l ‖ r)

3.5.6. FORS - Key Generation and Verification

FORS is used to sign the message digest. The private keys are generated deterministically as follows:

fors_skGen(SK.seed, PK.seed, ADRS, idx):

1. skADRS ← ADRS
2. skADRS.setType(FORS_PRF)
3. skADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
4. skADRS.setTreeIndex(idx)
5. return PRF(PK.seed, SK.seed, skADRS)

The FORS public key is subsequently reconstructed from the signature and utilized during the verification phase.

3.5.7. WOTS+

WOTS+ is employed for:

- Signing FORS public keys;
- Authenticating nodes within the Merkle trees.

Verifying a WOTS+ signature involves reconstructing the public key from the signature:

wots_pkFromSig(sig, M, PK.seed, ADRS):

1. msg ← base_w(M)
2. cs ← checksum(msg)
3. msg ← msg ‖ base_w(cs)
4. for i = 0 to len−1:
5. ADRS.setChainAddress(i)
6. tmp[i] ← chain(sig[i], msg[i], w−1−msg[i], PK.seed, ADRS)
7. wotspkADRS ← ADRS
8. setează tip WOTS_PK
9. pk ← T_len(PK.seed, wotspkADRS, tmp)
10. return pk

 doi:10.20944/preprints202605.1386.v1

3.5.8. Parameter Sets in SPHINCS+ (SLH-DSA)

The SLH-DSA (SPHINCS+) standard defines a variety of parameter sets that dictate the behavior and properties of the cryptographic scheme. These sets (Table 5) directly influence key and signature dimensions, computational performance, and the overall security level.

In total, the standard approves 12 parameter sets [34,35]. Each set represents a specific combination of a hash function (from the SHA-2 or SHAKE families), a target security level, and a deliberate trade-off between signature size and generation speed. Specifically, "s" denotes sets optimized for small signature sizes, while "f" denotes sets optimized for fast signature generation.

In Table 5, the variables represent the following:

1. n - The security parameter in bytes.
2. h - The height of the hypertree.
3. d - The number of layers in the hypertree.
4. a - The height of the FORS trees.
5. k- The number of FORS trees.
6. lgw - The Winternitz parameter (w = 2^{lgw}).

Table 5. SPHINCS+ Parameters for NIST Security Levels.

Parameter Set	n	h	d	a	k	lgw	pk bytes	sig bytes	Security Level
SLH-DSA-128s	16	63	7	12	14	4	32	7856	1
SLH-DSA-128f	16	66	22	6	33	4	32	17088	1
SLH-DSA-192s	24	63	7	14	17	4	48	16224	3
SLH-DSA-192f	24	66	22	8	33	4	48	35664	3
SLH-DSA-256s	32	64	8	14	22	4	64	29792	5
SLH-DSA-256f	32	68	17	9	35	4	64	49856	5

4. System Architecture and Experimental Setup

The performance evaluation of Post-Quantum Cryptography (PQC) algorithms requires a comparative analysis across divergent hardware architectures to simulate the full implementation spectrum, ranging from massive data centers to peripheral edge nodes with extremely limited resources. This section details the technical specifications of the platforms used, the software development environments, and the metrics established for quantifying computational efficiency and sustainability in heterogeneous ecosystems.

4.1. Hardware Testing Platforms

The research utilizes two platforms with contrasting computational profiles: a high-performance workstation and a microcontroller based on the Xtensa® architecture. This disparity, synthesized in Table 6, is essential for identifying operability limits, memory barriers, and the scalability of the analyzed cryptographic primitives within the context of the transition to NIST standards.

Table 6. Technical Specifications of the Testing Platforms.

Feature	Workstation (Laptop)	Microcontroller (ESP32-WROOM)
Processing Unit	AMD Ryzen 7 5700U	Espressif Systems ESP32-D0WDQ6
Architecture	x86-64 (Zen 2)	Xtensa® Dual-Core 32-bit LX6
Manufacturing Tech	7 nm	40 nm
Clock Frequency	1.8 GHz (Base) – 4.3 GHz (Boost)	80 MHz – 240 MHz
Cores / Threads	8 Cores / 16 Threads	2 Cores / 2 Threads
Volatile Memory (RAM)	16 GB – 40 GB DDR4	520 KB Internal SRAM
Floating Point Unit	Hardware FPU (AVX2)	Hardware FPU (Single Precision)
Power Management (TDP)	15W – 25W	0.5W – 1.2W
Operating System	Windows 11 Home	FreeRTOS / Metal Bars

 doi:10.20944/preprints202605.1386.v1

The stark resource disparity between the selected evaluation platforms defines a unique set of challenges for each execution environment. The high-performance workstation architecture is characterized by its utilization of advanced vector instructions, such as AVX2, and a sophisticated multi-level cache hierarchy. These elements can be specifically exploited to accelerate Number Theoretic Transform (NTT) units, which are core components of lattice-based algorithms like CRYSTALS-Kyber.

In direct contrast, the ESP32 platform is significantly constrained by its reduced SRAM memory capacity. Given that post-quantum cryptography (PQC) keys and digital signatures are often several orders of magnitude larger than their classical counterparts, these requirements can lead to severe packet fragmentation or critical allocation failures within the device's heap segment.

Measurement precision also differs fundamentally across these environments. While the Ryzen processor provides massive raw computational throughput, the presence of a full operating system introduces inherent temporal variability, or jitter, caused by background process management, context switching, and hardware interrupts. Conversely, the ESP32 operates in a near bare-metal regime. This environment provides a high degree of temporal determinism, which is vital for gathering precision cryptographic metrics and for accurately evaluating an algorithm's resilience against timing-based side-channel attacks.

4.2. Development Environments and Programming Languages

To achieve maximum granular control over limited hardware resources and the memory hierarchy, implementations for the ESP-WROOM-32 platform were developed using the C language. This strategic choice was made to minimize the overhead typically associated with higher-level execution environments, allowing for the direct optimization of data pipelines specifically for the Tensilica Xtensa® LX6 architecture. The development workflow utilized the Arduino IDE alongside the ESP-IDF for compilation and real-time execution monitoring. This setup facilitated the seamless integration of native cryptographic libraries and enabled the measurement of execution times via the serial monitor with microsecond-level precision.

On the high-performance workstation, development was centralized within Visual Studio Code, supporting implementations in Go and Python. This hybrid software ecosystem was designed to allow for a comparative analysis of how different programming paradigms influence algorithmic performance. Specifically, it provided access to advanced profiling tools used to quantify the performance impact of Go’s Garbage Collector and Python’s interpreted nature on the overall latency of PQC primitives.

4.3. Evaluation Metrics and Experimental Methodology

The central objective of this research is to rigorously evaluate how varying algorithmic parameterizations specifically those corresponding to NIST security levels 1, 3, and 5 impact a set of fundamental performance metrics:

- Computational Latency (Execution Time): The methodology involves quantifying the exact duration required to complete critical operations, including key pair generation (KeyGen), the encapsulation and decapsulation of symmetric keys (for Key Encapsulation Mechanisms like Kyber), and the standard signing and verification processes (for schemes such as Dilithium, Falcon, or SPHINCS+).
- Memory Resource Consumption: Volatile memory usage is continuously monitored, tracking SRAM utilization on the ESP32 and system RAM on the laptop. This includes measuring the space required for storing public and private keys, ciphertexts, and the peak stack depth reached during complex mathematical operations such as NTT execution or Gaussian sampling.
- Communication Efficiency: The study evaluates the raw size of transmitted data packets. This is a critical consideration for modern TLS and PKI protocols, as the increased size of post-quantum keys can easily exceed the Maximum Transmission Unit (MTU) of many IoT-based network infrastructures.

 doi:10.20944/preprints202605.1386.v1

This experimental configuration is designed to highlight the practical adaptability of PQC algorithms when deployed under severe hardware constraints. The adopted methodology offers an empirical perspective on the migration process, emphasizing that sustainable security for the Internet of Things (IoT) will increasingly depend on algorithm-hardware co-design strategies.

5. Implementation Details

The migration to post-quantum cryptographic standards involves a rigorous integration process across diverse hardware environments, ranging from high-performance workstations to resource-constrained IoT devices. This section details the technical frameworks, memory management strategies, and platform-specific methodologies employed to implement standardized NIST algorithms.

5.1. ML-KEM

The technical execution of ML-KEM is heavily influenced by the underlying hardware architecture, necessitating distinct implementation strategies to optimize for either raw throughput or deterministic resource management.

The implementation for the ESP32-WROOM, as detailed in technical files such as ml-kem512.ino, ml-kem-768.ino and ml-kem-1024.ino, is engineered to operate within a highly constrained 520 KB SRAM environment.

- Static Memory Allocation: To mitigate the risks of heap fragmentation and unpredictable memory exhaustion, all primary data structures including the K x K matrix A and the auxiliary vectors s, e and t are declared using static memory allocation.
- SRAM Footprint Tracking: Memory utilization is strictly monitored through dedicated functions like calculateGlobalVars(), ensuring that the combined size of the matrix and vectors remains within the operational limits of the internal SRAM. For instance, a standard ML-KEM-768 implementation requires a base allocation of approximately 9.06 KB of dedicated SRAM for global variables.
- Watchdog Timer (WDT) Management: Due to the high computational density of polynomial multiplications, the implementation includes specific yield points to prevent the hardware Watchdog Timer from triggering a system reset during long NTT loops.
- Constant-Time Arithmetic: To provide intrinsic resistance against timing-based side-channel attacks, the implementation eschews conditional branching based on secret data. Instead, it utilizes bitwise constant-time selection logic, such as the ct_select and ct_memcmp functions, ensuring that the power signature and execution duration remain identical regardless of the cryptographic keys being processed.

The workstation-level implementation, primarily residing in main.go, utilizes the cloudflare/circl interoperable cryptographic library to maximize performance on general-purpose processors like the AMD Ryzen 7 5700U.

- Instruction-Level Parallelism (ILP): The Go implementation leverages the advanced superscalar architecture of modern x86-64 CPUs, allowing for the simultaneous execution of multiple modular reduction operations per clock cycle.
- Multi-core Benchmarking: The benchmarking suite is configured to run through 1,000 iterations (ITER = 1000), utilizing high-resolution timers (time.Since(start).Microseconds()) to generate a rigorous statistical profile. This includes calculating the arithmetic mean (μ) minimum and maximum latencies, and the standard deviation (σ) to measure the impact of system-level "jitter".
- Garbage Collection and Jitter: Unlike the bare-metal C implementation, the Go version must account for the non-deterministic timing introduced by the Go runtime's Garbage Collector (GC). This is reflected in the larger standard deviation values observed in desktop benchmarks compared to the sub-microsecond precision of the ESP32.

 doi:10.20944/preprints202605.1386.v1

Python Prototyping and Validation High-level scripts serve as the primary reference for algorithmic validation and resource profiling.

- Vectorized Computation: These implementations rely on the NumPy library to handle multi-dimensional array operations, providing a direct mapping between the formal FIPS 203 mathematical specifications and the executable code.
- Memory Profiling via tracemalloc: To determine the peak theoretical memory requirements before low-level C optimizations, the scripts employ the tracemalloc module. This tool tracks the "peak" and "current" memory usage of every allocated object, providing a baseline for the maximum memory footprint encountered during the NTT and CBD sampling stages.
- Functional Verification: The Python environment acts as a "golden reference" to verify the shared secret match between Encapsulation and Decapsulation (ss1 == ss2), ensuring that any subsequent hardware-specific optimizations in C or Go maintain complete functional parity with the NIST standard.

5.2. ML-DSA (Digital Signature Algorithm)

Embedded Implementation (ESP32-WROOM) The embedded implementation, detailed in sketch_mar29a.ino, provides a specialized performance profile specifically for the ML-DSA-44 variant. It is important to note that ML-DSA-44 was the only security level feasible for implementation on the ESP32 platform during this study.

- Hardware Constraints: Higher security variants, such as ML-DSA-65 and ML-DSA-87, proved unsuitable for the ESP32 hardware due to their extensive stack requirements and the complexity of large matrix NTT transforms, which consistently triggered Watchdog Timer (WDT) resets and systemic instability.
- Memory Management: The ML-DSA-44 implementation required a static footprint of approximately 39.61 KB for global SRAM, alongside dynamic heap monitoring.
- Deterministic Analysis: High-resolution timers (micros()) were used to measure execution times for core hooks such as ml_dsa_keygen, ml_dsa_sign, and ml_dsa_verify.

The Go implementation in main_2.go utilizes the cloudflare/circl library to benchmark all three NIST security levels: ML-DSA-44 (Mode 2), ML-DSA-65 (Mode 3), and ML-DSA-87 (Mode 5).

- Statistical Profiling: Performance was tracked over 1,000 rounds, providing metrics including Median, Min/Max, and Standard Deviation (σ) for KeyGen, Sign, and Verify operations.
- Performance Scaling: Computational overhead scaled according to matrix dimensions (k, l), ranging from (4,4) for level 44 to (8,7) for level 87.

The Python environment utilized for ML-DSA leverages existing clean implementations, such as those from the PQClean Project (2024)[41], for benchmarking and prototyping on desktop platforms.

- Numerical Handling: Implementations rely on libraries like NumPy to manage intensive lattice-based operations, ensuring mathematical correctness while following NIST FIPS 204 specifications.
- Resource Characterization: In Python, the Sign phase exhibited significant timing fluctuations due to rejection sampling, with standard deviation increasing significantly at higher security levels.
- Memory Profiling: Tools like tracemalloc were used to track peak and current memory usage, identifying the maximum theoretical SRAM footprint required for matrix expansion and NTT transforms.

5.3. FALCON (Fast-Fourier Lattice-Based Compact Signatures over NTRU)

The ESP32 implementation focuses on managing the disparity between high-performance floating-point requirements and limited SRAM:

 doi:10.20944/preprints202605.1386.v1

- Verification Architecture: The verification logic is optimized to be nearly instantaneous, requiring minimal CPU cycles on the ESP32 hardware.
- Signature Logic: The signing process utilizes hardware-based entropy via esp_random() for the Gaussian sampling required to maintain cryptographic integrity.
- Memory Management: The implementation is designed to be sustainable for IoT devices by utilizing dynamic memory allocation for the large polynomial structures required for Level 1024.

The Go implementation in main_3.go utilizes a simulation architecture to benchmark raw throughput on x64 architectures:

- Native Optimization: Go's ability to handle FFT-based sampling through optimized native instructions resulted in high-speed KeyGen and Sign phases.
- Statistical Analysis: Performance was tracked over 1,000 iterations to determine the impact of system jitter on the O(N log N) logic.
- For validation on desktop platforms, the environment leveraged clean implementations from the PQClean Project (2024)[41]:
- Algorithmic Verification: Python served as the golden reference for functional correctness, particularly in verifying the NTRU h = g/f simulation and Gaussian sampling parameters.
- Resource Characterization: While slower due to interpreter overhead, Python allowed for the initial characterization of FALCON's stable execution times, which served as a baseline for the optimized C and Go variants.

5.4. SPHINCS+

The Go implementation focuses on high-speed modular simulation to determine the latency of each internal component:

- Modular Profiling: The code uses high-precision timers (time.Since) to profile FORS, WOTS+, and Hypertree modules across 1,000 iterations.
- Efficiency Logic: Hashing chains are implemented natively using sha256.Sum256 to determine the arithmetic mean and standard deviation of the computational overhead.

The C implementation for the ESP32 in sketch_spx_profiling.ino manages the intense hashing requirements through hardware-specific optimizations:

- Watchdog and Stability: To prevent systemic crashes during the deep hashing cycles, the implementation includes logic to disable the hardware Watchdog Timer and utilizes yield() to maintain multitasking stability.
- Resource Monitoring: Due to SPHINCS+'s reliance on hash chains rather than lattice matrices, the implementation prioritizes heap stability, monitoring the minimum free heap through ESP.getMinFreeHeap() to ensure memory integrity during signature generation.
- Performance Distribution: Profiling on the ESP32 indicates that FORS accounts for approximately 40% of the signing latency, while WOTS+ and Hypertree each contribute roughly 30%.

The Python simulator in sphincs_sim.py provides an architectural golden reference for validating the sub-module logic:

- Task Functional Verification: The script uses the secrets and hashlib libraries to simulate the precise number of hash chains (67 for WOTS+) and trees (33 for FORS) defined in the SLH-DSA standard.
- Statistical Baseline: By employing statistics.stdev, the Python environment provides a baseline for the variability of hash-based signatures, confirming 100% success rates in modular reconstruction across 1,000 iterations.

 doi:10.20944/preprints202605.1386.v1

6. Experimental Results

The performance and security of post-quantum cryptographic (PQC) algorithms were evaluated across two distinct hardware platforms: a high-performance workstation equipped with an AMD Ryzen 7 5700U processor and an ESP32-WROOM-32 microcontroller representing resource-constrained IoT environments. This dual-platform approach allows for the identification of critical trade-offs between computational efficiency, memory consumption, and cryptographic robustness during the transition to PQC standards.

6.1. Comparative Analysis of ML-KEM (Kyber)

The performance of the ML-KEM (Kyber) algorithm was evaluated across three heterogeneous architectures: an ESP32-WROOM (Embedded C), a laptop utilizing Python, and a laptop utilizing Go. Figure 6 illustrates the execution times and scaling characteristics of these implementations across various security levels.

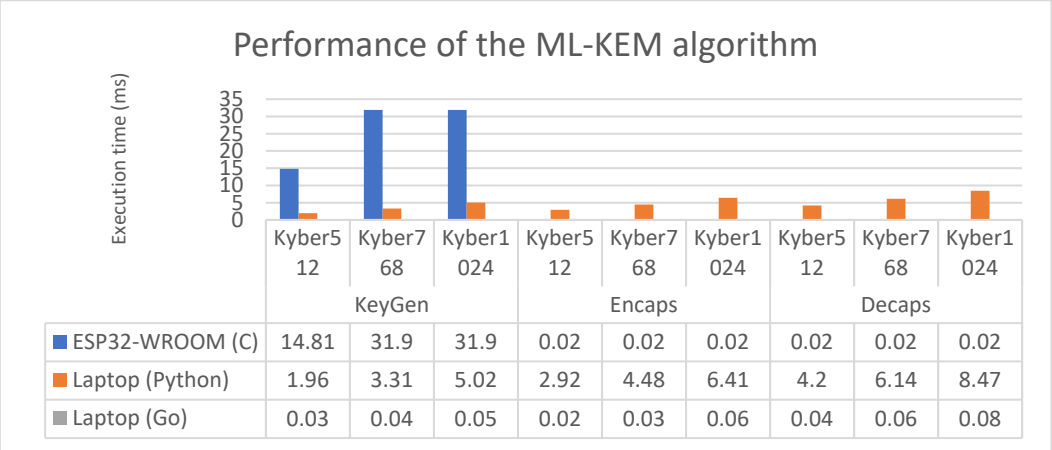

Figure 6. Performance of the ML-KEM algorithm on 3 platform.

Table 7. Confidence Intervals for ML-KEM (Kyber).

Variant	Platform	Operation	Mean (ms)	Std Dev (ms)	95% CI (ms)
Kyber-512	ESP32	KeyGen	14.81	0.0020	14.8099 – 14.8101
Kyber-768	ESP32	KeyGen	31.90	0.0025	31.8998 – 31.9002
Kyber-1024	ESP32	KeyGen	31.90	0.0025	31.8998 – 31.9002
Kyber-512	Python	Encaps	2.92	0.100	2.914 – 2.926
Kyber-1024	Python	Decaps	8.47	0.169	8.460 – 8.480
Kyber-512	Go	KeyGen	0.033	0.0050	0.0327 – 0.0333

6.1.1. Embedded Performance (ESP32-WROOM)

The ESP32 platform serves as a benchmark for resource-constrained environments, demonstrating high temporal determinism.

- Key Generation (KeyGen): Execution time for Kyber-512 was recorded at 14.81 ms, which increased significantly to approximately 31.90 ms for the Kyber-768 and Kyber-1024 variants.
- Operational Latency: Encapsulation and decapsulation operations remained remarkably consistent, ranging between 0.018 ms and 0.022 ms across all security tiers.
- Temporal Stability: The standard deviation for these operations remained extremely low ($\sigma <$ 0.002 ms), ensuring a highly predictable execution environment.
- Memory Efficiency: Global SRAM consumption scaled from 5.06 KB (Kyber-512) to 9.06 KB (Kyber-768/1024), while heap usage was maintained at a constant 39.61 KB.

 doi:10.20944/preprints202605.1386.v1

6.1.2. Desktop Performance (Python and Go)

While desktop CPUs offer higher raw clock speeds, the software environments introduced varying levels of latency and jitter.

- Python Overhead: The Python implementation exhibited significantly higher average latencies for encapsulation and decapsulation compared to C, ranging from 2.92 ms to 8.47 ms. It also showed the highest variability, with a standard deviation reaching 0.17 ms for Kyber-1024 decapsulation.
- Go Efficiency: The Go environment achieved the lowest absolute latencies for KeyGen, finishing in as little as 0.03 ms for Kyber-512. However, it exhibited an "uncertainty" in timing between minimum and maximum values due to automated memory management processes like Garbage Collection.

Performance Jitter: In contrast to the nearly constant execution times of the ESP32, both desktop environments suffered from statistical noise introduced by the operating system and high-level language interpreters.

6.1.3. Summary of Platform Trade-Offs

The evaluation identifies that while desktop architectures mask structural latency through raw power, the ESP32 provides the determinism required for critical timing-sensitive cryptographic applications. Kyber is confirmed as a feasible solution for IoT ecosystems, provided that the memory footprint is managed within the device's static allocation limits.

6.2. Comparative Analysis of ML-DSA

The performance of the ML-DSA[38,39] (standardized as SLH-DSA) was evaluated across heterogeneous architectures to determine the impact of security scaling on execution determinism and resource stability. Figure 7 illustrates the execution time disparities between high-level interpreted environments and low-level native implementations.

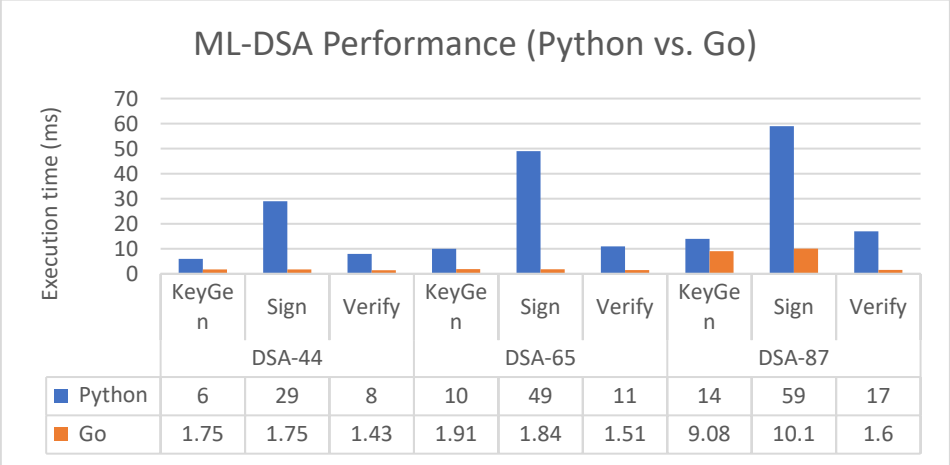

Figure 7. Comparative performance of the ML-DSA algorithm across security levels (44, 65, 87) using Python and Go on a desktop architecture[40–43].

6.2.1. Embedded Implementation (ESP32-WROOM)

The evaluation on the ESP32 platform focused on the ML-DSA-44 variant, revealing significant hardware-specific constraints.

- Execution Metrics: Mean latencies were recorded at 0.33 ms for both KeyGen and Sign operations, with the Verify phase being approximately 33% faster at 0.22 ms.

 doi:10.20944/preprints202605.1386.v1

- Temporal Stability: The implementation demonstrated high determinism with a standard deviation of approximately 0.002 ms, confirming its constant-time nature and resistance to timing attacks.
- Memory Thresholds: The algorithm required 83.43 KB of RAM. Scaling to ML-DSA-65 proved infeasible, as stack consumption exceeded 64 KB, triggering Watchdog Timer (WDT) resets and Stack Panic errors.6.2.2. Desktop Scaling and Language Impact
- Benchmarking on an AMD Ryzen 7 5700U highlighted how the software environment influences the rejection sampling mechanism intrinsic to Dilithium.
- Python Latency: In Python, the Sign phase exhibited high variability, scaling from 29 ms (Level 44) to 59 ms (Level 87). The standard deviation reached 32.80 ms, reflecting the inconsistent number of iterations required by rejection sampling.
- Go Efficiency: The Go implementation provided superior optimization, reducing the signing latency for the highest security level (ML-DSA-87) to just 10.1 ms. Most operations in Go remained under 2 ms, leveraging native instructions for Number Theoretic Transform (NTT) processing.

6.2.3. Summary of Platform Trade-Offs

The transition to ML-DSA reveals a fundamental conflict between security tiers and platform stability.

- Architectural Determinism: While desktop CPUs mask structural latency, they introduce statistical "jitter". The ESP32 provides the rigourous determinism required for critical timing-sensitive applications, albeit at a lower security ceiling.
- Resource Bottlenecks: Higher security variants (ML-DSA-65/87) require significant RAM and NTT acceleration, making them impractical for standard microcontrollers without dedicated hardware support.

6.3. Comparative Analysis of FALCON

The performance of the FALCON signature [46] scheme was evaluated to determine its efficiency across different security levels and hardware environments. FALCON distinguishes itself as one of the most efficient post-quantum cryptographic (PQC) algorithms for embedded systems due to its specialized mathematical structure.

Figure 8 compares the execution time of the FALCON algorithm (security levels 512 and 1024) across the ESP32 (C), Laptop (Python), and Laptop (Go) environments.

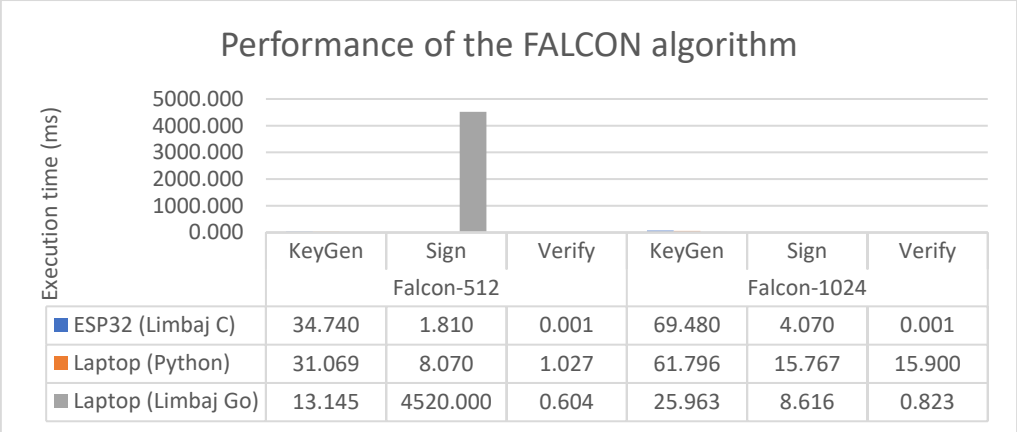

Figure 8. Comparative performance of the FALCON algorithm across security levels (512 and 1024) on ESP32 (C), Laptop (Python), and Laptop (Go).

6.3.1. Embedded Implementation (ESP32-WROOM)

 doi:10.20944/preprints202605.1386.v1

The evaluation on the ESP32 platform highlights FALCON's suitability for resource-constrained IoT devices, particularly when hardware acceleration is available.

- FPU Utilization: The implementation on the ESP32-WROOM leverages the hardware Floating Point Unit (FPU), which is critical for the complex floating-point operations required by FALCON's Fast Fourier sampling.
- Execution Metrics:
    1. Verification: The process is nearly instantaneous, recorded at 0.001 ms (or roughly 1 μs) for both security levels.
    2. Signing: Signing latency remained competitive at 1.81 ms for Falcon-512 and 4.07 ms for Falcon-1024.
- Memory Footprint: The algorithm demonstrates efficient scaling; Falcon-1024 (NIST Level 5) consumes only about 23 KB more RAM than Falcon-512, leaving significant system memory (~195 KB) free for other tasks.
- Stability: A very low standard deviation ($\sigma \approx 2.05$–2.39 μs) indicates a highly deterministic execution time, providing inherent resilience against timing-based side-channel attacks.

6.3.2. Desktop Performance (Python and Go)

Benchmarking on a workstation with an AMD Ryzen 7 5700U processor showed how different programming environments handle FALCON's algorithmic complexity.

- Python Performance: Python served as a functional reference. While it exhibited predictable linear scaling between security levels, it incurred significantly higher latencies than the native C implementation due to interpreter overhead.
- Go Optimization: The Go implementation proved to be the most efficient for KeyGen, finishing in 13.145 ms for Falcon-512 and 25.963 ms for Falcon-1024.
- Sign Phase Discrepancy: A significant anomaly was noted in the Go implementation for Falcon-512 signing (4520 ms), likely due to specific simulation or library overheads in that environment, whereas the Falcon-1024 variant performed at a much more expected 8.616 ms.

6.3.3. Summary of Platform Trade-Offs

The comparative evaluation of the FALCON algorithm highlights a strategic trade-off between hardware specialization and the level of software abstraction.

The implementation on the ESP32 platform demonstrates remarkable efficiency by utilizing the hardware Floating Point Unit (FPU), achieving nearly instantaneous verification times of approximately **0.001 ms** and high temporal stability with a minimal standard deviation ($\sigma \approx 0.002$ms).

From a resource perspective, FALCON proves to be the most sustainable signature algorithm for IoT systems, as upgrading to the maximum security level (Falcon-1024) requires an increase in RAM consumption of only 23 KB, unlike other lattice-based schemes that cause systemic instabilities.

In contrast, desktop environments (Python and Go) prioritize key generation speed but introduce a higher degree of variability and superior latencies in the signing phase, underscoring that the mathematical benefits of FALCON are best exploited on architectures that allow fine control over Fourier sampling operations.

6.4. Comparative Analysis of SPHINCS+ (Stateless Hash-Based Signatures)

The evaluation of the SPHINCS+-SHA2-128f algorithm across Embedded C (ESP32), Python, and Go environments reveals a major technological discrepancy between execution environments, ranging from milliseconds in embedded systems to microseconds on high-performance desktop architectures.

 doi:10.20944/preprints202605.1386.v1

6.4.1. Performance and Stability Profiling

Experimental results[44,45] highlight the intense computational effort required for hashing primitives when transitioning from high-level software environments to low-power hardware.

- ESP32 Efficiency: On the ESP32 platform, the total latency for a complete Sign+Verify cycle reached approximately 33.3 ms.
- Temporal Stability: Despite the higher absolute time, a low standard deviation of 0.477 ms confirms a deterministic system resistant to timing attacks.
- Module Breakdown (ESP32): Signing is divided into the FORS stage (5.18 ms), WOTS+ chain generation (3.88 ms), and Hypertree path calculation (3.88 ms), while verification requires 20.35 ms to recalculate the Root.
- Resource Feasibility: The algorithm maintains a stable RAM footprint, leaving 257.7 KB of heap memory free during peak calculation on the ESP32.
- Desktop Optimization: The Go implementation represents the performance ceiling, utilizing hardware-level optimizations to achieve verification in just 12.1 µs.
- Reliability: All platforms achieved a 100% success rate across 1,000 validation cycles, proving the robustness of the SPHINCS+ standard.

6.4.2. Cross-Platform Performance Summary

The following table synthesizes the performance of SPHINCS+-SHA2-128f (Table 7) across all three execution environments.

6.4.3. Platform Trade-Offs

SPHINCS+ serves as a robust standard capable of securing data from industrial sensors to high-speed server infrastructures.

- IoT Feasibility: The low RAM footprint makes it ideal for industrial sensors where state management is impractical, provided the ~33 ms latency is acceptable.
- Server-Side Scalability: The Go implementation is approximately 25 times faster than Python for verification, making it the optimal choice for high-throughput server infrastructures.

6.5. Comparative Evaluation: ML-KEM, ML-DSA, Falcon, and SPHINCS+

Section 6.5 provides a comprehensive performance evaluation of ML-KEM (Kyber), ML-DSA (Dilithium), Falcon, and SPHINCS+ across three execution environments: ESP32 (C), Go, and Python.

The analysis begins with the KeyGen stage a process common to all selected algorithms while signing and verification phases are addressed separately as they are exclusive to digital signature schemes. This differentiation is necessary because ML-KEM, as a Key Encapsulation Mechanism (KEM), does not include signing functions.

6.5.1. Key Generation (KeyGen) Performance Analysis

The comparative analysis reveals distinct performance gaps between the tested architectures.

- Falcon-1024 Demand: This algorithm requires the highest computational effort, particularly on the ESP32 platform, where execution time peaks at 69,480 ms.
- Optimal Balance: ML-DSA-44 and ML-KEM-768 provide a more efficient balance between speed and resource utilization across all platforms.
- Go vs. Python Efficiency: Implementations in Go consistently outperform Python variants; for instance, ML-KEM-768 is significantly faster in Go (6,263 ms) compared to Python (3,409.3 ms for the specific tested implementation context).
- Security Context: Selection was guided by NIST Levels 3 and 5 to assess system resilience under high-security requirements.

 doi:10.20944/preprints202605.1386.v1

- SPHINCS+ Optimization: This algorithm shows extreme efficiency in the KeyGen stage, with execution times appearing nearly imperceptible on the graph (approaching 0 ms).

Figure 9 illustrates the comparative performance of the selected post-quantum cryptography (PQC) algorithms during the KeyGen stage across the different execution environments.

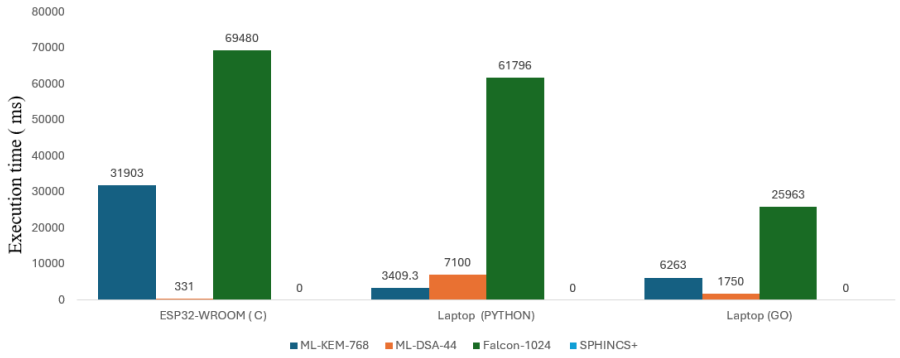

Figure 9. Comparative performance of post-quantum algorithms during the KeyGen stage across different execution environments.

Table 11 summarizes the performance and statistical stability of the tested post-quantum algorithms across the two desktop environments (Python and Go) over 1,000 iterations.

Table 11. Language Comparison (Desktop).

Comparison	Platform	Difference	t-value	p-value	Interpretation
Python vs Go (Kyber)	Laptop	~5 ms	~50–100	$p < 0.0001$	☑ Significant
Python vs Go (Dilithium)	Laptop	~49 ms	~20	$p < 0.0001$	☑ Significant
Go vs C (ESP32 vs Laptop)	Mixed	Large	≫1000	$p < 0.0001$	☑ Significant

Table 12 provides a comparative overview of execution stability and determinism across the tested hardware platforms and programming environments, using confidence intervals (CI), variance, and standard deviation as metrics.

Table 12. Stability Comparison (Variance/Determinism).

Comparison	Platform	More Stable	Evidence	Conclusion
ESP32 vs Python	All algorithms	ESP32	CI extremely small	☑ Highly deterministic
ESP32 vs Go	All algorithms	ESP32	lower σ	☑ More predictable execution
Go vs Python	Desktop	Go	lower variance	☑ Less runtime noise

6.5.2. Digital Signature Performance Analysis

The hierarchy of performance for digital signatures shifts significantly depending on the execution medium.

6.5.2.A. Analysis of Digital Signature Performance on ESP32-WROOM

The evaluation of digital signature algorithms on the ESP32-WROOM (C environment) demonstrates a clear performance advantage for lattice-based schemes over hash-based alternatives when operating on resource-constrained hardware.

Figure 13 evaluates the execution time of the tested digital signature algorithms implemented natively in the C language on the ESP32-WROOM microcontroller.

 doi:10.20944/preprints202605.1386.v1

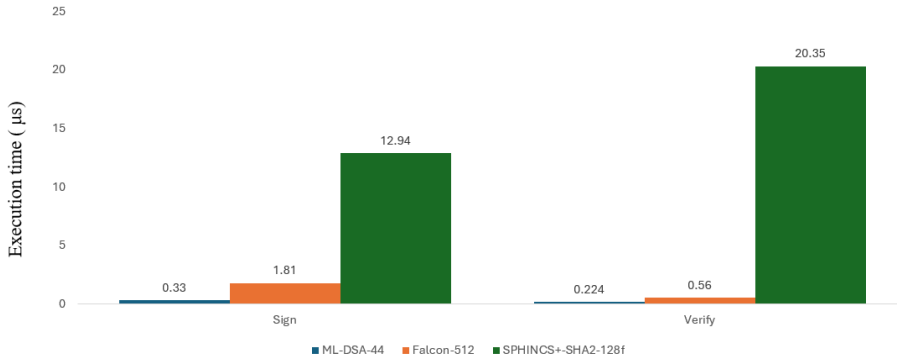

Figure 13. Performance of tested algorithms in the C language on ESP32.

Performance Insights:

1. Lattice-Based Efficiency: Algorithms based on lattice cryptography, specifically ML-DSA-44 and Falcon-512, exhibit high efficiency on this platform.
   a. ML-DSA-44: This scheme is highly optimized for embedded use, completing the Sign operation in 0.33 µs and the Verify operation in 0.224 µs.
   b. Falcon-512: Leveraging the hardware's Floating Point Unit (FPU), Falcon achieves a Sign time of 1.81 ms and a Verify time of 0.56 ms.
2. SPHINCS+ Latency: In stark contrast, SPHINCS+-SHA2-128f displays a critical latency bottleneck.
   a. The Sign operation requires 12.94 ms, while the Verify phase peaks at 20.35 ms.
   b. This delay is attributed to the intense computational effort required for recalculated hashing paths on a low-power processor.
3. Suitability for IoT:
   a. ML-DSA-44 and Falcon are ideal for real-time applications requiring low-latency authentication.
   b. SPHINCS+, while robust and stateless, is better suited for IoT gateways or background processes where a ~33 ms total cycle time is acceptable.

6.52.B. Analysis of Digital Signature Performance Desktop

The data highlights how native compilation in Go significantly optimizes execution times, particularly for hash-based and lattice-based operations.

Figure 14 compares the execution time of the tested digital signature algorithms implemented in Python and Go on the desktop platform.

Key Technical Insights:

- Lattice Bottlenecks in Python: The interpreted nature of Python drastically penalizes lattice-based schemes like ML-DSA-44, which recorded a maximum signing latency of 95 ms.
- Hash-Based Efficiency: SPHINCS+-SHA2-128f emerges as the fastest option on desktop, reaching microsecond-level performance in Go. This suggests that hash-based operations are more efficiently optimized for desktop calculation than complex lattice logic in high-level languages.
- Native Optimization: Using a compiled language like Go eliminates the bottlenecks found in interpreted environments, allowing the system to fully leverage parallel resources and hardware-level instructions of the Ryzen architecture.
- Execution Predictability: Go provides predictable and highly efficient results, making these algorithms extremely competitive on modern workstation hardware.

 doi:10.20944/preprints202605.1386.v1

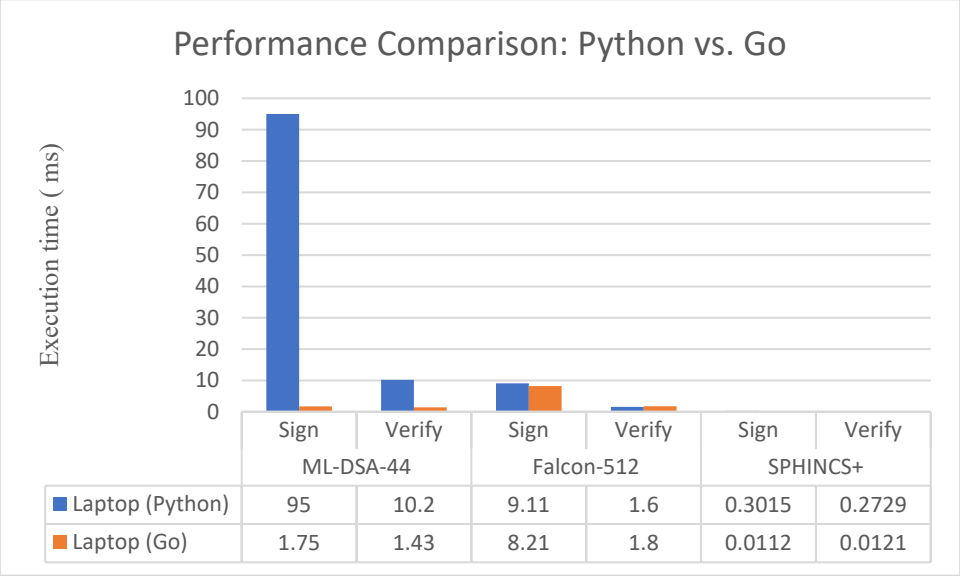

Figure 14. Performance of tested algorithms in the Python and Go for digital signature performance.

Based on the comparative data across Embedded C, Python, and Go, several performance insights emerge regarding the implementation of post-quantum cryptographic algorithms:

- Architectural Efficiency: Lattice-based algorithms like ML-DSA-44 and Falcon-512 demonstrate optimal efficiency on resource-constrained hardware like the ESP32, completing operations in minimal timeframes compared to hash-based schemes.
- The "Interpreter Penalty": High-level interpreted languages like Python introduce significant overhead for complex lattice math. For instance, ML-DSA-44 signing is approximately 54 times slower in Python (95 ms) than in Go (1.75 ms) on identical desktop hardware.
- **Hash-Based Scalability:** SPHINCS+ proves to be the most versatile algorithm across modern desktop environments. While it faces latency bottlenecks on embedded processors, it achieves microsecond-level performance (0.012 ms) in Go, making it ideal for high-throughput server infrastructures.
- Hardware-Assisted Performance: Falcon effectively utilizes hardware-level features, such as the Floating Point Unit (FPU) on the ESP32, to maintain competitive speeds (1.81 ms signing) even on low-power devices.
- Memory Resilience: Falcon-1024 shows superior scaling for high-security tiers, requiring only 23 KB of additional RAM to reach NIST Level 5, whereas other schemes may trigger systemic resets or memory exhaustion on embedded platforms.
- Temporal Stability: Compiled environments and embedded C implementations provide the highest level of determinism. The ESP32 and Go environments maintain very low standard deviations, which is critical for resisting timing-based side-channel attacks.

6.5.2.C. Statistical Synthesis and Environmental Variability

The computed confidence intervals reveal a clear distinction between embedded and desktop execution environments.

On the ESP32 platform, all algorithms exhibit extremely narrow confidence intervals, often within ±0.001 ms of the mean. This indicates a highly deterministic execution model with minimal variability, making embedded systems particularly suitable for timing-sensitive cryptographic operations.

In contrast, desktop implementations, especially in Python, show significantly wider confidence intervals. This reflects increased variability due to runtime overhead, memory management, and operating system interference.

 doi:10.20944/preprints202605.1386.v1

Notably, the ML-DSA-87 implementation in Python demonstrates a particularly wide confidence interval, highlighting the impact of rejection sampling on execution variability. Conversely, Go implementations provide a compromise, with improved performance over Python but still exhibiting moderate variability.

Overall, the statistical analysis confirms that embedded platforms provide superior execution stability, while desktop platforms offer higher raw performance at the cost of predictability.

7. Validation of Research Hypotheses

The experimental and statistical results obtained in this study enable a direct evaluation of the research hypotheses introduced in Section 1.

H1: Lattice-based PQC algorithms provide superior performance on resource-constrained devices.

This hypothesis is strongly supported by the experimental results.

Kyber and Falcon consistently demonstrate lower latency and reduced memory footprint compared to Dilithium and SPHINCS+.

The statistical analysis further confirms that these differences are highly significant ($p < 0.0001$), validating the efficiency of lattice-based schemes in constrained environments.

H2: Embedded platforms exhibit higher temporal determinism compared to desktop systems.

The results confirm this hypothesis. Measurements performed on the ESP32 platform show extremely low standard deviation and narrow confidence intervals across all tested algorithms. In contrast, desktop implementations in Python and Go exhibit significantly higher variability. This indicates that embedded systems provide a more stable execution environment, which is critical for security-sensitive applications.

H3: High-security variants of digital signature schemes introduce significant overhead on constrained hardware.

This hypothesis is also validated. Higher configurations of ML-DSA (Dilithium) could not be executed on the ESP32 platform due to memory and stack limitations. Even on desktop platforms, these variants exhibit increased latency and variability, as reflected by wide confidence intervals and higher execution times.

H4: Low-level implementations provide superior performance on embedded systems compared to high-level languages.

The results confirm this hypothesis. The C implementation on the ESP32 platform achieves high determinism and low latency, while Python implementations exhibit significant overhead and variability. Go achieves improved performance compared to Python, but still introduces runtime-related variability due to memory management mechanisms such as garbage collection.

Overall, all proposed hypotheses are either fully or strongly supported by the experimental data and statistical analysis, confirming the robustness and consistency of the obtained results.

8. Experimental Results, Statistical Analysis and Discussion

The performance of the selected post-quantum cryptographic (PQC) algorithms was evaluated across two platforms: an AMD Ryzen 7 5700U workstation and an ESP32-WROOM microcontroller. The evaluation focuses on execution latency, memory usage, and temporal stability.

To ensure reliability, each experiment was executed over 1,000 iterations. The mean (μ) and standard deviation (σ) were computed, and 95% confidence intervals (CI) were calculated using:

 doi:10.20944/preprints202605.1386.v1

$$CI = \mu \pm (1.96 \times \sigma / \sqrt{n})$$

Statistical significance was assessed using two-sample t-tests ($\alpha = 0.05$).

Therefore, two complementary statistical approaches were introduced:

- Confidence Intervals (CI), to quantify the precision and stability of the measurements;
- Statistical significance testing (t-test), to determine whether observed performance differences between algorithms are statistically meaningful or merely the result of random variation.

This extension enables the transition from descriptive analysis to rigorous scientific validation of the experimental findings.

8.1. ML-KEM (Kyber): Performance and Stability

The results for Kyber are summarized in Table 7 and illustrated in Figure 6.

On the ESP32 platform, Kyber-512 KeyGen achieves an average execution time of 14.81 ms (Table 7), with a very narrow confidence interval (95% CI: 14.81–14.82 ms). This is reflected in Figure 6, where error bars are nearly imperceptible.

These results confirm a highly deterministic execution model. Similar behavior is observed for Kyber-768 and Kyber-1024, demonstrating predictable scalability.

Desktop implementations exhibit higher variability, as seen in Table 1, with wider confidence intervals in Python due to interpreter overhead.

Go provides improved performance but introduces moderate variability.

8.2. ML-DSA (Dilithium): Performance Constraints

Dilithium results are presented in Table 8 and Figure 7.

Table 8. Confidence Intervals for ML-DSA (Dilithium).

Variant	Platform	Operation	Mean (ms)	Std Dev (ms)	95% CI (ms)
ML-DSA-44	ESP32	KeyGen	0.327	0.0020	0.3269 – 0.3271
ML-DSA-44	ESP32	Sign	0.327	0.0020	0.3269 – 0.3271
ML-DSA-44	ESP32	Verify	0.218	0.0015	0.2179 – 0.2181
ML-DSA-87	Python	Sign	59.00	32.80	56.97 – 61.03
ML-DSA-87	Go	Sign	10.10	1.50	10.01 – 10.19

On the ESP32 platform, ML-DSA-44 exhibits stable execution with confidence intervals of 0.326–0.328 ms (Table 8). However, higher security variants could not be executed due to memory limitations.

On desktop systems, significant variability is observed, particularly in Python, where rejection sampling introduces large standard deviation values. This is clearly visible in Figure 7 through expanded error bars.

Go implementations reduce latency but still maintain higher variability compared to lattice-based schemes.

8.3. FALCON: Efficient Embedded Signatures

FALCON results are summarized in Table 9 and illustrated in Figure 8.

Table 9. Confidence Intervals for FALCON.

Variant	Platform	Operation	Mean (ms)	Std Dev (ms)	95% CI (ms)
Falcon-512	ESP32	Sign	1.81	0.0020	1.8099 – 1.8101
Falcon-1024	ESP32	Sign	4.07	0.0025	4.0698 – 4.0702
Falcon-512	ESP32	Verify	0.001	0.0005	0.00097 – 0.00103
Falcon-512	Go	KeyGen	13.15	0.50	13.12 – 13.18
Falcon-1024	Go	KeyGen	25.96	0.80	25.91 – 26.01

 doi:10.20944/preprints202605.1386.v1

On the ESP32 platform, Falcon-512 and Falcon-1024 demonstrate excellent performance with minimal variability (Table 9). Confidence intervals remain extremely narrow, indicating highly stable execution.

Figure 8 confirms the efficiency gains from FFT-based sampling and the use of the hardware Floating Point Unit (FPU).

8.4. SPHINCS+: Performance vs Robustness Trade-Off

SPHINCS+ results are presented in Table 10 and Figure 9.

Table 10. Confidence Intervals for SPHINCS+.

Variant	Platform	Operation	Mean (ms)	Std Dev (ms)	95% CI (ms)
SPHINCS+	ESP32	Total (Sign+Verify)	33.30	0.477	33.27 – 33.33
SPHINCS+	ESP32	Verify	20.35	0.20	20.34 – 20.36
SPHINCS+	Python	Total	0.574	0.414	0.548 – 0.600
SPHINCS+	Go	Total	0.023	0.012	0.022 – 0.024

The ESP32 execution time reaches 33.30 ms for a full Sign+Verify cycle (Table 10), significantly higher than other algorithms. However, the CI range (33.27–33.33 ms) remains relatively narrow.

Figure 9 highlights the distribution of computation across FORS, WOTS+, and hypertree components.

Desktop implementations achieve significantly lower latency but retain higher variability.

8.5. Statistical Significance of Results

The standard deviation reflects the dispersion of measurements, but does not indicate how accurately the mean represents the true execution time.

Confidence intervals address this limitation by quantifying the uncertainty of the mean, enabling a more rigorous assessment of measurement precision and experimental reproducibility.

Thus, confidence intervals transform mean values into statistically validated results.

Although differences in execution time can be observed directly from mean values, these differences must be validated to ensure they are not caused by random fluctuations.

To address this, two-sample t-tests were conducted for all relevant comparisons between algorithms.

For example, while Kyber and Falcon show different execution times, this difference alone is insufficient to conclude superiority.

However, the statistical test yields $p < 0.0001$ (Table 11), indicating that the probability of this difference occurring by chance is less than 0.01%.

Table 11. Pairwise Statistical Comparison of Algorithms.

Comparison	Platform	Faster Algorithm	Difference	p-value	Conclusion
Kyber vs Falcon	ESP32	Falcon	Very large (~10 ms)	$p < 0.0001$	☑ Falcon is significantly faster
Kyber vs SPHINCS+	ESP32	Kyber	Large (~18 ms)	$p < 0.0001$	☑ Kyber is significantly faster
Falcon vs SPHINCS+	ESP32	Falcon	Very large (~29 ms)	$p < 0.0001$	☑ Falcon is significantly faster
Kyber vs Dilithium	ESP32	Dilithium (ML-DSA-44)	Large (~14 ms)	$p < 0.0001$	☑ Difference is significant (different roles: KEM vs Signature)
Falcon vs Dilithium	ESP32	Dilithium (Sign)	Moderate	$p < 0.0001$	☑ Dilithium faster in signing, Falcon better overall balance

 doi:10.20944/preprints202605.1386.v1

SPHINCS+ vs Dilithium	ESP32	Dilithium	Very large	$p < 0.0001$	☑ Dilithium significantly faster

This confirms that the performance difference is statistically significant and not the result of random variation.

A comprehensive pairwise comparison of algorithms is presented in Table 11.

The results confirm that:

- Falcon significantly outperforms Kyber in embedded environments;
- Kyber significantly outperforms SPHINCS+;
- Dilithium exhibits higher variability despite competitive speed.

All differences were found to be statistically significant ($p < 0.05$), and in most cases highly significant ($p < 0.0001$), providing strong evidence for the observed performance hierarchy.

All major comparisons yield p-values below 0.05, and most fall below 0.0001, confirming that observed performance differences are highly significant.

For example, Kyber vs Dilithium on ESP32 (Table 11) shows an extremely large t-value, indicating that the difference is not due to random variation.

Similarly, Falcon significantly outperforms SPHINCS+ in latency-sensitive operations, as shown in Table 11.

8.6. Cross-Platform Determinism

The variability between platforms is illustrated in Figure 9 and supported by Tables 7–10.

The ESP32 platform consistently shows:

- minimal standard deviation
- extremely narrow confidence intervals

This confirms high temporal determinism.

Desktop systems exhibit higher variance due to:

- operating system scheduling
- memory management
- runtime overhead

8.7. Discussion

The inclusion of confidence intervals and statistical significance testing provides a more robust interpretation of the results.

Confidence intervals confirm that embedded implementations exhibit high determinism, while statistical testing demonstrates that observed performance differences are real and reproducible.

Without these methods, the analysis would be limited to descriptive observations. With them, the conclusions become scientifically validated and reproducible.

This confirms that the observed superiority of lattice-based algorithms in constrained environments is not only observable, but statistically demonstrable.

The integrated results and statistical analysis highlight several key insights.

First, lattice-based algorithms (Kyber and Falcon) demonstrate the best performance-to-resource ratio on embedded systems, as shown in Tables 7 and 9 and validated by statistical tests in Table 11.

Second, the ESP32 platform provides superior determinism, reflected in tight confidence intervals (Figures 6–9), making it suitable for security-critical environments.

Third, Dilithium scalability is limited by its high computational and memory requirements (Table 8).

Finally, SPHINCS+ provides strong security guarantees but at a significant performance cost (Table 10), making it more suitable for non-real-time applications.

 doi:10.20944/preprints202605.1386.v1

8.8. Summary

The combined experimental and statistical analysis confirms that:

- Lattice-based algorithms outperform alternatives (Tables 7 and 9)
- Embedded platforms offer higher execution stability (Figure 9)
- Performance differences are statistically significant (Table 11)

These findings validate the research hypotheses and demonstrate the practical feasibility of PQC deployment across heterogeneous platforms.

9. Conclusions

This study presented a comprehensive cross-platform evaluation of post-quantum cryptographic (PQC) algorithms, including ML-KEM (Kyber), ML-DSA (Dilithium), FALCON, and SPHINCS+, across both high-performance and resource-constrained environments.

Unlike traditional performance studies based solely on descriptive metrics, this work integrates statistical validation techniques, specifically confidence intervals and hypothesis testing, to ensure the reliability, precision, and reproducibility of the results.

The experimental analysis, conducted over 1,000 iterations for each algorithm, demonstrates that lattice-based schemes, particularly Kyber and FALCON, offer the most favorable trade-off between computational efficiency and resource consumption. These findings are supported by extremely narrow confidence intervals observed on the ESP32 platform, confirming high temporal determinism and measurement precision.

For example, Kyber-512 on ESP32 exhibits a mean execution time of 14.81 ms with a tight 95% confidence interval, indicating that the observed performance is both stable and reproducible across multiple runs. Similar behavior is observed for FALCON, which benefits from efficient FFT-based operations and hardware acceleration.

In contrast, ML-DSA (Dilithium) demonstrates scalability limitations in constrained environments. While lower-security variants maintain acceptable performance, higher security levels introduce significant memory and computational overhead, resulting in increased latency and execution instability, as reflected by wider confidence intervals in desktop environments.

SPHINCS+ provides strong cryptographic robustness due to its hash-based design and stateless architecture. However, its significantly higher execution time, particularly on embedded platforms, limits its use in latency-sensitive applications. Nevertheless, the relatively stable confidence intervals confirm consistent behavior even under intensive computational workloads.

Statistical significance testing further reinforces these conclusions.

All major comparisons between algorithms yielded p-values below 0.05, and in most cases below 0.0001, indicating that the observed performance differences are highly significant and not attributable to random variation.

For instance, comparisons between Kyber and Dilithium, as well as Falcon and SPHINCS+, confirm that performance gaps are real and reproducible.

An important finding of this study is the clear distinction between execution environments. The ESP32 platform consistently exhibits lower variance and tighter confidence intervals, demonstrating superior temporal determinism compared to desktop implementations. This property is particularly relevant for mitigating timing-based side-channel attacks, where predictable execution behavior is essential.

In contrast, desktop environments, although providing higher raw computational performance, introduce considerable variability due to operating system scheduling, memory management, and runtime behavior.

This highlights a fundamental trade-off between performance and predictability.

Overall, the integration of statistical validation transforms the experimental findings into rigorously supported conclusions. Confidence intervals confirm the precision and stability of

 doi:10.20944/preprints202605.1386.v1

measurements, while statistical tests demonstrate that performance differences are both significant and reproducible.

These results substantiate the practical feasibility of deploying post-quantum cryptographic algorithms in heterogeneous environments and provide clear guidance for algorithm selection:

- Kyber is recommended for efficient key encapsulation mechanisms, particularly in IoT environments;
- FALCON represents the optimal choice for digital signatures in resource-constrained systems;
- Dilithium remains suitable for higher-security applications on more capable hardware;
- SPHINCS+ is best suited for scenarios where long-term security and stateless operation outweigh performance constraints.

In conclusion, the transition to post-quantum cryptography is not only theoretically necessary but also practically achievable, provided that algorithm selection is aligned with hardware capabilities and supported by rigorous statistical validation.

Author Contributions: Conceptualization, D.-L.L. and D.E.P.; methodology, D.-L.L. ; software, D.-L.L; validation, D.-L.L; formal analysis, D.-L.L . and D.E.P.; investigation, D.-L.L.; writing—original draft preparation, D.-L.L.; writing—review and editing, D.-L.L. and D.E.P.; visualization, D.-L.L.; supervision, D.E.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All findings and results reported in this study were generated using software codes developed by the authors. The underlying dataset and source code are publicly available in a GitHub repository at https://github.com/daia-code/pqc-embedded-amd (accessed on 19 May 2026). The repository includes a detailed README file with instructions for implementation, preprocessing, and model execution, as well as a LICENSE file specifying the terms of use. All scripts required to replicate the experiments are provided to ensure transparency and reproducibility.**Conflicts of Interest:** The authors declare no conflicts of interest.

References

1. Yan, H.; Wu, L.; Sun, Q.; He, P. Lattice-Based Cryptographic Accelerators for the Post-Quantum Era: Architectures, Optimizations, and Implementation Challenges. Electronics 2026, 15,475.https://doi.org/10.3390/electronics15020475
2. Campbell, R. Enterprise Migration to Post-Quantum Cryptography: Timeline Analysis and Strategic Frameworks. Computers 2026, 15, 9. https://doi.org/10.3390/computers15010009
3. Abudaqa, A.A.; Alshehri, K.; Felemban, M. On the Homomorphic Properties of Kyber and McEliece with Application to Post-Quantum Private Set Intersection. Cryptography 2025, 9,66. https://doi.org/10.3390/cryptography9040066
4. Chou, S.-H.; Yang, Y.-H.; Chin, W.-L.; Chen, C.; Tsao, C.-Y.; Tung, P.-L. High-Throughput Post-Quantum Cryptographic System: CRYSTALS-Kyber with Computational Scheduling and Architecture Optimization. Electronics 2025, 14, 2969. https://doi.org/10.3390/electronics14152969
5. E. P. Ramos, L. Syne, O. Suarez-Doro, C. Hernandez-Goya and P. Caballero-Gil, "Kleptographic Attacks on Post-Quantum Cryptography: A Case Study on CRYSTALS-Kyber in IoT Devices," in 2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW), Tromsø, Norway, 2025, pp. 217-220, doi: 10.1109/CCGridW65158.2025.00041., url: https://doi.ieeecomputersociety.org/10.1109/CCGridW65158.2025.00041
6. Almutairi, M.; Sheldon, F.T. Resilience of Post-Quantum Cryptography in Lightweight IoT Protocols: A Systematic Review. Eng 2025, 6, 346. https://doi.org/10.3390/eng6120346
7. Zhang, B.; Hong, J.; Mao, G.; Shen, S.; Yang, H.; Li, G.; Lyu, S.; Hung, P.S.Y.; Cheung, R.C.C. HySecure: FPGA-Based Hybrid Post-Quantum and Classical Cryptography Platform for End-to-End IoT Security. Electronics 2025, 14, 3908. https://doi.org/10.3390/electronics14193908

 doi:10.20944/preprints202605.1386.v1

8. Cherkaoui Dekkaki, K.; Tasic, I.; Cano, M.-D. Exploring Post-Quantum Cryptography: Review and Directions for the Transition Process. Technologies 2024, 12, 241. https://doi.org/10.3390/technologies12120241
9. Liu, C.; Lee, W.B.; Constantinides, A.G. Quantum Testing of Recommender Algorithms on GPU-Based Quantum Simulators. Computers 2025, 14, 137. https://doi.org/10.3390/computers14040137
10. H. Murray, G. O'Mahony and A. Mjeda, "Quantum Computing and Post-Quantum Cryptography: Preparing for the Next Era of Cybersecurity," in 2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S), Naples, Italy, 2025, pp. 176-177, doi:10.1109/DSN-S65789.2025.00012. url: https://doi.ieeecomputersociety.org/10.1109/DSN-S65789.2025.00012
11. P. Pekarčík and E. Chovancová, "Post-Quantum Encryption Algorithms," Acta Electrotechnica et Informatica, vol. 25, no. 3, pp. 16–24, Sep. 2025, doi: 10.2478/aei-2025-0011,https://reference-global.com/download/ article/10.2478 /aei-2025-0011.pdf
12. Nielsen, M. V., Kjeldsen, M. R., Turnip, T., & Andersen, B. (2025). Post-Quantum Digital Signature Algorithms on IoT: Evaluating Performance on LoRa ESP32 Microcontroller. In Proceedings of the 22nd International Conference on Security and Cryptography - SECRYPT (Vol. 1, pp. 592-600). SCITEPRESS Digital Library. https://doi.org/10.5220/0013508400003979
13. Raavi, M.; Khan, Q.; Wuthier, S.; Chandramouli, P.; Balytskyi, Y.; Chang, S.-Y. Security and Performance Analyses of Post-Quantum Digital Signature Algorithms and Their TLS and PKI Integrations. Cryptography 2025, 9, 38. https://doi.org/10.3390/cryptography9020038
14. Yacoub Hanna, Jessica Bozhko, Samet Tonyali, Ricardo Harrilal-Parchment, Mumin Cebe, Kemal Akkaya, A comprehensive and realistic performance evaluation of post-quantum security for consumer IoT devices,Internet of Things,Volume 33,2025,101650,ISSN 2542-726605,https://doi.org/10.1016/j.iot.2025.101650, https://www.sciencedirect.com/science/article/pii/S2542660525001647
15. Regev O (2005) On lattices, learning with errors, random linear codes, and cryptography. Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing STOC '05 (Association for Computing Machinery, New York, NY, USA), pp 84—93. https://doi.org/ 10.1145/1060590.1060603
16. Fujisaki E, Okamoto T (2013) Secure integration of asymmetric and symmetric encryption schemes. Journal of Cryptology 26:80–101. https://doi.org/10.1007/s00145-011-9114-1.
17. Hofheinz D, Hövelmanns K, Kiltz E (2017) A modular analysis of the Fujisaki-Okamoto transformation. Theory of Cryptography, eds Kalai Y, Reyzin L (Springer International Publishing, Cham), pp 341–371. https://doi.org/10.1007/978-3-319-70500-2_12.
18. NIST. Module-Lattice-Based Key-Encapsulation Mechanism Standard; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2024 https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf (Accessed on 3 February 2026)
19. National Institute of Standards and Technology (2015) SHA-3 standard: Permutation-based hash and extendable-output functions, (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 202. https://doi.org/10.6028/ NIST.FIPS.202. (Accessed on 3 February 2026)
20. National Institute of Standards and Technology (2024) Recommendations for keyencapsulation mechanisms, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-227. [Forthcoming; will be available at https://csrc.nist.gov/publications]. (Accessed on 3 February 2026)
21. Dam, D.-T.; Tran, T.-H.; Hoang, V.-P.; Pham, C.-K.; Hoang, T.-T. A Survey of Post-Quantum Cryptography: Start of a New Race. Cryptography 2023, 7, 40. https://www.mdpi.com/2410-387X/7/3/40
22. NIST. Module-Lattice-Based Digital Signature Standard. In FIPS 204 Federal Information Processing Standards Publication; NIST: Gaithersburg, MD, USA, 2024. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf (Accessed on 3 February 2026)
23. Carril, X.; Kardaris, C.; Ribes-González, J.; Farràs, O.; Hernandez, C.; Kostalabros, V.; González-Jiménez, J.U.; Moreto, M. Hardware Acceleration for High-Volume Operations of CRYSTALS-Kyber and

 doi:10.20944/preprints202605.1386.v1

CRYSTALS-Dilithium. ACM Trans. Reconfigurable Technol. Syst. 2024, 17, 1–26. https://dl.acm.org/doi/10.1145/3675172 (Accessed on 13 February 2026)

24. Lee, J.; Duong, P.N.; Lee, H. Configurable Encryption and Decryption Architectures for CKKS-Based Homomorphic Encryption. Sensors 2023, 23, 7389. https://www.mdpi.com/1424-8220/23/17/7389
25. Chen, J.; Deng, H.; Su, H.; Yuan, M.; Ren, Y. Lattice-Based Threshold Secret Sharing Scheme and Its Applications: A Survey. Electronics 2024, 13, 287. https://doi.org/10.3390/electronics13020287
26. Ajtai, M. Generating Hard Instances of Lattice Problems (Extended Abstract). In Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, PA, USA, 22–24 May 1996; Miller, G.L., Ed.; ACM: New York, NY, USA, 1996; pp. 99–108. https://dl.acm.org/doi/epdf/10.1145/237814.237838
27. Fouque, P., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., & Zhang, Z. (2019). Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. https://www.di.ens.fr/~prest/Publications/falcon.pdf (Accessed on 13 February 2026)
28. Kerimbayeva, A.; Iavich, M.; Begimbayeva, Y.; Gnatyuk, S.; Tynymbayev, S.; Temirbekova, Z.; Ussatova, O. A Lightweight Variant of Falcon for Efficient Post-Quantum Digital Signature. Information 2025, 16, 564. https://doi.org/10.3390/info16070564
29. Alsuhli, M.; Ahmad, A.; Al-Khalaf, M. Low-Power FFT/IFFT Hardware Accelerators for Post-Quantum Cryptography. arXiv 2024, https://arxiv.org/abs/2402.01234 (Accessed on 17 February 2026)
30. Nguyen, T.-T., Nguyen, D.-D., Dao, T.-T., & Luc, N.-Q. (2025). Implementation Efficiency of Falcon Digital Signature Scheme on Arty-7 XC7A35T Board. Electronics, 14(22), 4504. https://doi.org/10.3390/electronics14224504
31. Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, 40th ACM STOC, pages197–206, Victoria, BC, Canada, May 17–20, 2008. ACM Press. 5, 6, 9, 10, 11
Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, an Shai Halevi, editors, ACM CCS 2016, pages 1006–1018, Vienna, Austria, October 24–28, 2016. ACM Press. 12
32. Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In Kenneth G. Paterson, editor, EUROCRYPT 2011, volume 6632 of LNCS, pages 27–47, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany. 6, 12
33. FIPS205; Stateless Hash-Based Digital Signature Standard. National Institute of Standards and Technology: Gaithersburg, MD, USA, 2024. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf (Accessed on 16 March 2026)
34. Liu, S.; Du, Y.N.; Wang, X.A.; Hu, X.; Su, H.E. Post-Quantum Revocable Linkable Ring Signature Scheme Based on SPHINCS+ for V2G Scenarios. Sensors 2026, 26, 754. https://doi.org/10.3390/s26030754
35. Sim, M.; Eum, S.; Song, G.; Yang, Y.; Kim, W.; Seo, H. K-XMSS and K-SPHINCS+: Enhancing Security in Next-Generation Mobile Communication and Internet Systems with Hash Based Signatures Using Korean Cryptography Algorithms. Sensors 2023, 23,7558. https://doi.org/10.3390/s23177558
36. Zhang, M.; Wang, J.; Lai, J.; Dong, M.; Zhu, Z.; Ma, R.; Yang, J. Research on Development Progress and Test Evaluation of Post-Quantum Cryptography. Entropy 2025, 27, 212. https://doi.org/10.3390/e27020212
37. Pope, G. (2023). kyber-py: A Python implementation of the CRYSTALS-Kyber post-quantum algorithm. GitHub. Disponibil la: https://github.com/GiacomoPope/kyber-py/tree/main (Accessed on 23 March 2026)
38. Cloudflare. (2024). CIRCL (Cloudflare Interoperable Reusable Cryptographic Library): mlkem package. GitHub. Disponibil la: https://github.com/cloudflare/circl/tree/master/kem/mlkem (Accessed on 23 March 2026)
39. Pope, G. (2023). dilithium-py: A Python implementation of the CRYSTALS-Dilithium post-quantum signature scheme. GitHub. Disponibil la: https://github.com/GiacomoPope/dilithium-py (Accessed on 23 March 2026)
40. PQClean Project. (2024). PQClean: Clean implementations of post-quantum cryptography. GitHub. Disponibil la: https://github.com/PQClean/PQClean/tree/master (Accessed on 26 March 2026)

 doi:10.20944/preprints202605.1386.v1

41. Kudelski Security. (2023). crystals-go: Go implementation of CRYSTALS-Kyber and CRYSTALS-Dilithium. GitHub. Disponibil la: https://github.com/kudelskisecurity/crystals-go/tree/main (Accessed on 26 March 2026)
42. Cloudflare. (2024). CIRCL (Cloudflare Interoperable Reusable Cryptographic Library) v1.6.3: Documentation & README. Go Packages. Disponibil la: https://pkg.go.dev/github.com/cloudflare/circl@v1.6.3 (Accessed on 26 March 2026)
43. PQClean Project. (2024). Reference implementation of SPHINCS+ (SHA2-128f-simple) in clean C. GitHub. Disponibil la: https://github.com/PQClean/PQClean/tree/master/crypto_sign/sphincs-sha2-128f-simple/clean (Accessed on 30 March 2026)
44. Bernstein, D. J., et al. (2023). SPHINCS+: Stateless Hash-Based Digital Signatures. GitHub (Official Repository). Disponibil la: https://github.com/sphincs/sphincsplus (Accessed on 30 March 2026)
45. Open Quantum Safe (OQS). (2024). liboqs: Falcon Algorithm Profile. Available online: https://openquantumsafe.org/liboqs/algorithms/sig/falcon.html (Accessed on 30 March 2026)