КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (33), 2026
ISSN 2663 – 4023

DOI 10.28925/2663-4023.2026.33.1210
UDC 004.056.55:004.43

Yurii Oleksiichuk
Candidate of Physical and Mathematical Sciences, Associate Professor at the Department of Computer Science and Information Technology
Poltava University of Economics and Trade, Poltava, Ukraine
ORCID: 0000-0002-0585-3307
olexijchuk@gmail.com

EMPIRICAL PERFORMANCE EVALUATION OF NIST POST-QUANTUM CRYPTOGRAPHIC ALGORITHMS ML-KEM, ML-DSA, AND SLH-DSA ACROSS JDK VERSIONS

Abstract. The standardization of the ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205) algorithms by NIST in 2024 marked the beginning of a practical transition to post-quantum cryptography in enterprise software systems. The Java platform remains one of the most widely used in corporate environments; however, developers face a practical choice: whether to use the BouncyCastle library, which supports the new algorithms across all current JDK versions, or the native capabilities introduced in JDK 25. The absence of systematic empirical data on the performance of these algorithms in the Java environment – across platform versions, security levels, and operation types – makes it difficult to justify this choice when planning the migration of enterprise Java systems to post-quantum standards. This paper presents a systematic comparative performance study of ML-KEM, ML-DSA, and SLH-DSA in the Java environment based on a reproducible experiment using the Java Microbenchmark Harness in a containerized setting across three platform versions – JDK 17, JDK 21, and JDK 25 – comparing BouncyCastle 1.83 and the native JDK 25 implementation. For each algorithm, three core operations were measured: key pair generation, encapsulation or signing, and decapsulation or verification. For signature algorithms, the message size was additionally varied across 256 bytes, 1024 bytes, and 65536 bytes. Statistical significance of observed differences was assessed using the Mann-Whitney and Kruskal-Wallis tests, with post-hoc analysis performed using Dunn's method with Bonferroni correction. It is established that the native JDK 25 implementation consistently underperforms BouncyCastle across all algorithms and operations. A dedicated benchmark of provider initialization overhead confirmed that this gap is attributable to the algorithmic implementation rather than infrastructure costs. Upgrading from JDK 17 to JDK 25 yields a statistically significant performance improvement for BouncyCastle implementations. Based on the experimental results, practical recommendations are formulated for Java developers regarding the choice of implementation and platform version when migrating to post-quantum standards. The source code and Docker Compose configuration are published in an open GitHub repository to enable independent reproduction of the results.

Keywords: post-quantum cryptography; ML-KEM; ML-DSA; SLH-DSA; Java; BouncyCastle; JDK; JMH.

INTRODUCTION

The standardization by NIST in August 2024 of the ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205) algorithms [1-3] marked the beginning of a practical transition to post-quantum cryptography in enterprise software systems. The Java platform remains one of the most widely used in corporate environments; however, developers face a practical choice: whether to use the BouncyCastle (BC) library, which supports the new algorithms across all current JDK versions, or the native capabilities introduced in JDK 25. At the same time, systematic empirical data on the performance of these algorithms in a pure Java environment – across platform versions, security levels, and cryptographic operation types – are lacking. Existing benchmarks focus predominantly on C and Python implementations [4-8], while Java-specific characteristics – the impact of JIT compilation, differences between providers, and the evolution of support across JDK versions – remain insufficiently studied.

Problem statement. The absence of systematic empirical data on the performance of NIST post-quantum algorithms in the Java environment across platform versions and implementation types makes it difficult to

© Ю. Ф. Олексійчук, 2026

justify the choice between BouncyCastle and the native JDK 25 implementation when planning the migration of enterprise Java systems to post-quantum standards.

Analysis of recent research and publications. The potential emergence of quantum computers of sufficient computational power may bring not only a range of benefits but also significant security challenges [9-11]. Research into the performance of post-quantum cryptographic algorithms has been developing actively following the completion of NIST standardization in 2024.

In [5], the efficiency of NIST-standardized digital signature algorithms was studied across C++, Python, and Golang environments in a containerized setting with varying file sizes; it was established that the implementation language significantly affects performance even when the same underlying library is used.

In [6], a comprehensive cross-platform performance study of CRYSTALS-Kyber, NTRU, BIKE, and CRYSTALS-Dilithium was conducted across three hardware profiles – ranging from high-performance servers to embedded devices – measuring latency, memory usage, and energy consumption. The results demonstrate a substantial dependence of algorithm characteristics on the hardware platform and security level.

In [7], a framework for evaluating the impact of post-quantum primitives ML-KEM, ML-DSA, and SLH-DSA on TLS protocol performance was proposed using OpenSSL and liboqs; an increase in TLS handshake latency compared to classical algorithms was recorded.

In [8], the performance of CRYSTALS-Dilithium, Falcon, and SPHINCS+ was analyzed in comparison with RSA using the liboqs library in Python, with a focus on signing and verification operations for files of varying sizes.

Regarding the Java platform, native support for post-quantum algorithms first appeared in JDK 24 through JEP 496 (ML-KEM) and JEP 497 (ML-DSA) as preview APIs and was finalized in JDK 25 [12]. Prior to this, the BouncyCastle library – which has supported ML-KEM, ML-DSA, and SLH-DSA since version 1.80 [13] – remained the only practical option for Java developers. No systematic comparative studies of post-quantum algorithm performance in the JVM environment accounting for platform version and implementation type have been found to date, which defines the scientific novelty of this research.

Article aims. The aim of this paper is to conduct a systematic comparative performance study of ML-KEM, ML-DSA, and SLH-DSA in the Java environment based on a reproducible experiment using the Java Microbenchmark Harness (JMH) [14] in a containerized setting across three platform versions – JDK 17, JDK 21, and JDK 25 – comparing BouncyCastle 1.83 and the native JDK 25 implementation, as well as to formulate practical recommendations regarding the choice of implementation and platform version for Java applications migrating to post-quantum standards.

THEORETICAL FOUNDATIONS OF RESEARCH

Post-quantum cryptography (PQC) is a branch of cryptography concerned with developing algorithms that are resistant to attacks by both classical and quantum computers [15]. Unlike classical algorithms such as RSA [16] and ECDSA [17], whose security is based on the computational hardness of integer factorization and discrete logarithm problems, post-quantum algorithms rely on mathematical problems for which no efficient quantum algorithms are known.

In August 2024, NIST published three final post-quantum cryptography standards – FIPS 203, FIPS 204, and FIPS 205 – defining the ML-KEM, ML-DSA, and SLH-DSA algorithms respectively. Each is described in detail below.

ML-KEM (Module Lattice-based Key Encapsulation Mechanism, FIPS 203) [1] is a key encapsulation mechanism based on module lattices, standardized from the CRYSTALS-Kyber algorithm. It is designed for the secure establishment of a shared secret key between two parties and is functionally analogous to the Diffie-Hellman and ECDH protocols in classical cryptography. The security of ML-KEM is based on the computational hardness of the Module Learning With Errors (MLWE) problem. The standard defines three security levels: ML-KEM-512 (level 1, equivalent to AES-128), ML-KEM-768 (level 3, equivalent to AES-192), and ML-KEM-1024 (level 5, equivalent to AES-256). The algorithm operates through three procedures: key pair generation, encapsulation (the sender generates a ciphertext and a shared secret), and decapsulation (the recipient recovers the shared secret from the ciphertext using the private key).

ML-DSA (Module Lattice-based Digital Signature Algorithm, FIPS 204) [2] is a digital signature algorithm based on module lattices, standardized from the CRYSTALS-Dilithium algorithm. It is designed to ensure data authenticity and integrity and is functionally analogous to RSA-PSS and ECDSA. The security of ML-DSA is based on the hardness of the MLWE and Module Short Integer Solution (MSIS) problems. The standard defines three security levels: ML-DSA-44 (level 2), ML-DSA-65 (level 3), and ML-DSA-87 (level 5). The algorithm supports three operations: key pair generation, signature generation (using the private key and a pseudorandom number generator), and signature verification (using the public key). Unlike RSA, where

signature sizes are fixed and relatively small, ML-DSA key and signature sizes are substantially larger: the public key for ML-DSA-65 is 1952 bytes and the signature is 3309 bytes.

SLH-DSA (Stateless Hash-based Digital Signature Algorithm, FIPS 205) [3] is a hash-based digital signature algorithm, standardized from the SPHINCS+ algorithm. Unlike ML-DSA, the security of SLH-DSA relies exclusively on the properties of cryptographic hash functions (SHA-256 or SHAKE), making it independent of the hardness of lattice problems. The algorithm is stateless – each signature is generated independently without maintaining state between calls. The standard defines twelve parameter sets differing in hash function (SHA2 or SHAKE), security level (128, 192, or 256 bits), and operating mode: the s (small) variant produces smaller signatures at the cost of slower generation, while the f (fast) variant accelerates signature generation at the cost of increased signature size [18]. he fundamental difference between ML-DSA and SLH-DSA lies in the trade-off between artifact size and performance: ML-DSA provides significantly smaller signatures and faster generation, whereas SLH-DSA offers a minimal public key size (50 bytes at the 128-bit security level) and more conservative security assumptions.

For measuring the performance of cryptographic primitives in the Java environment, the Java Microbenchmark Harness (JMH) [14] is used – a framework for writing, running, and analyzing microbenchmarks developed by the OpenJDK team. JMH accounts for JVM-specific behaviors including JIT compiler warm-up, dead code elimination optimizations, and other effects that can distort the results of naive measurements. The framework ensures statistically sound results through the separation of warm-up and measurement phases, execution across multiple independent JVM processes (forks), and the computation of confidence intervals.

METHODOLOGY AND EXPERIMENTAL SETUP

To ensure reproducibility and isolation of results, all benchmarks were executed in Docker containers based on eclipse-temurin images for JDK 17, JDK 21, and JDK 25. Each container was allocated 2 CPUs and 2 GB of RAM. To guarantee sequential execution without competition for hardware resources, containers were launched one at a time using the run-all.sh script with the docker compose run command. The project source code and Docker Compose configuration are published in an open GitHub repository [19] to enable independent reproduction of the results. The experiments were conducted on the hardware and software described in Table 1.

Table 1

Hardware and Software Configuration

Component	Specification
CPU	AMD A10-5700 APU with Radeon(tm) HD Graphics, 4 cores @ 3.4 GHz
RAM	16 GB DDR3
Operating System	Ubuntu 24.04.2 LTS (Linux kernel 6.17.0-19-generic)
Docker	Docker Engine 29.0.2, Docker Compose 2.40.3
JDK17	eclipse-temurin:17-jre-jammy, 17.0.18
JDK21	eclipse-temurin:21-jre-jammy, 21.0.10
JDK25	eclipse-temurin:25-jre-jammy, 25.0.2
BouncyCastle	1.83
JMH	1.37

Measurements were performed using JMH 1.37 in average time mode (AverageTime) with microseconds as the unit of measurement (µs/op). The benchmark configuration was as follows: 5 warm-up iterations of 1 second each, 10 measurement iterations of 1 second each, and 2 independent JVM processes (forks), yielding 20 measurements per parameter combination. To eliminate the impact of pseudorandom number generator blocking, SecureRandom was instantiated with the NativePRNGNonBlocking algorithm.

For each algorithm, three core operations were measured: key pair generation (keyGeneration), encapsulation or signing (encapsulation / signing), and decapsulation or verification (decapsulation / verification). For the ML-DSA and SLH-DSA signature algorithms, the message size was additionally varied across 256 bytes, 1024 bytes, and 65536 bytes in order to investigate the dependence of performance on input length. All security levels defined by the respective standards were evaluated for each algorithm: ML-KEM-512/768/1024, ML-DSA-44/65/87, and SLH-DSA-SHA2-128S/128F/192S/192F/256S/256F.

To assess the statistical significance of differences between implementations and JDK versions, non-parametric tests were applied: the Mann–Whitney test [20] for pairwise comparison of two independent samples (BC versus Native in JDK 25), and the Kruskal-Wallis test for comparison across three JDK versions. When a

statistically significant result was obtained, post-hoc pairwise comparisons were performed using Dunn's method with Bonferroni correction. The choice of non-parametric methods is justified by the small sample sizes (10 measurements per group) and the absence of guarantees of normality in the distribution of execution times.

RESEARCH RESULTS

ML-KEM Performance. The experimental results for different JDK versions are presented in Table 2. For JDK 25, the results for BouncyCastle (BC) and the native implementation are shown separately.

Table 2

ML-KEM Performance (average operation time, µs)

Operation	Parameter	JDK-17 (BC)	JDK-21(BC)	JDK-25 (BC)	JDK-25 (Native)
Key generation	ML-KEM-512	82.1	81.7	78.1	78.6
	ML-KEM-768	132.0	125.5	125.5	128.2
	ML-KEM-1024	199.3	187.9	188.8	201.3
Encapsulation	ML-KEM-512	90.4	88.0	88.0	93.4
	ML-KEM-768	148.0	140.0	141.5	146.9
	ML-KEM-1024	215.0	212.1	205.5	221.2
Decapsulation	ML-KEM-512	118.5	119.9	118.4	127.6
	ML-KEM-768	182.4	186.0	180.9	197.6
	ML-KEM-1024	259.0	267.2	258.5	281.0

Analysis of the results revealed the following patterns. The execution time of all operations increases monotonically with the security level: the transition from ML-KEM-512 to ML-KEM-1024 increases key generation time by a factor of 2.43, encapsulation by 2.37, and decapsulation by 2.2. The native JDK 25 implementation underperforms BouncyCastle across all security levels, which is confirmed statistically ($p < 0.05$ by the Mann-Whitney test for all pairwise comparisons). The difference can reach up to 10%. A comparison across JDK versions shows a marginal advantage of JDK 25 over older versions. The differences are small but statistically significant in most cases according to the Kruskal-Wallis test, with the exception of ML-KEM-512 decapsulation. Pairwise post-hoc comparisons using Dunn's method with Bonferroni correction also confirm statistical significance in most cases. However, for example, ML-KEM-512 key generation for JDK 17 and JDK 21 yields statistically equivalent results. The corresponding bar charts are shown in Fig. 1.

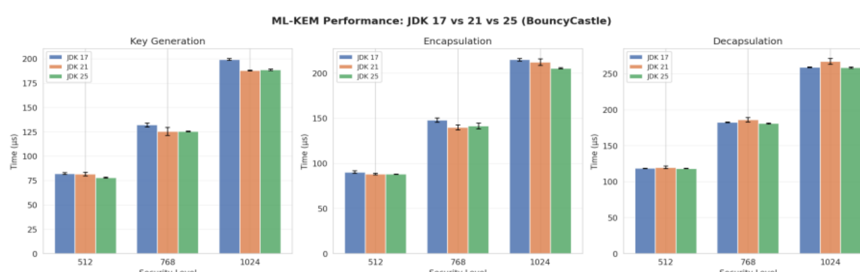

Fig. 1. ML-KEM performance comparison across JDK versions

ML-DSA Performance. The results of the ML-DSA performance measurements are presented in Tables 3 - 5. Table 3 shows the key generation time for different parameter sets across JDK versions.

Table 3

ML-DSA Performance: Key Generation (average time, µs)

Parameter	JDK-17 (BC)	JDK-21 (BC)	JDK-25 (BC)	JDK-25 (Native)
ML-DSA-44	262.2	220.8	220.1	283.6
ML-DSA-65	437.9	378.3	374.3	479.9
ML-DSA-87	706.0	610.8	595.4	741.0

ML-DSA key generation demonstrates a consistent speedup when transitioning to newer JDK versions. For all parameter sets (44, 65, 87), the Kruskal-Wallis test confirms statistically significant differences across JDK versions. For all parameter sets, the native JDK 25 implementation exhibits worse performance compared to the BC implementation, and this difference is statistically significant. Regardless of JDK version or

implementation, ML-DSA-65 is slower than ML-DSA-44, and ML-DSA-87 is substantially slower than ML-DSA-65. Table 4 presents the signing time for messages of varying lengths across JDK versions.

Table 4

ML-DSA Performance: Signing (average operation time, μs)

Message size	Parameter	JDK-17 (BC)	JDK-21 (BC)	JDK-25 (BC)	JDK-25 (Native)
256 B	ML-DSA-44	1073.3	1022.4	924.5	1175.7
	ML-DSA-65	1743.0	1693.3	1518.0	1972.0
	ML-DSA-87	2206.3	1940.8	1819.5	2394.2
1024 B	ML-DSA-44	1074.3	989.6	944.6	1214.8
	ML-DSA-65	1750.9	1545.7	1495.7	1944.2
	ML-DSA-87	2137.5	1883.2	1851.0	2368.3
65536 B	ML-DSA-44	1552.5	1438.1	1419.6	1662.1
	ML-DSA-65	2236.6	2083.0	1973.5	2409.2
	ML-DSA-87	2604.8	2436.8	2317.6	2840.0

For the signing operation, the improvement of JDK 25 over JDK 17 is statistically significant in all scenarios. However, differences between JDK 17 and JDK 21 are not always statistically significant – for example, for ML-DSA-44/65 at 256 B or ML-DSA-87 at 1024 B. The comparison between JDK 21 and JDK 25 is significant in most cases, but isolated exceptions exist – for example, ML-DSA-65 signing at 1024 B or ML-DSA-44 signing at 65536 B. Regardless of the parameter set or message size, signing with the native JDK 25 implementation is substantially slower than with BC. Verification times for messages of varying lengths across JDK versions are presented in Table 5.

Table 5

ML-DSA Performance: Verification (average operation time, μs)

Message size	Parameter	JDK-17 (BC)	JDK-21 (BC)	JDK-25 (BC)	JDK-25 (Native)
256 B	ML-DSA-44	270.6	253.4	239.8	304.4
	ML-DSA-65	462.5	406.4	379.2	488.2
	ML-DSA-87	723.7	661.1	622.0	770.5
1024 B	ML-DSA-44	279.8	257.7	245.9	310.3
	ML-DSA-65	440.3	404.7	386.9	491.7
	ML-DSA-87	715.7	653.9	623.5	783.0
65536 B	ML-DSA-44	745.6	728.1	720.6	770.8
	ML-DSA-65	892.6	877.6	857.0	936.6
	ML-DSA-87	1164.9	1129.2	1095.2	1214.7

For the verification operation, the statistical tests reveal the most consistent pattern: for all ML-DSA parameter sets and all message sizes, the differences between JDK 17, JDK 21, and JDK 25 are statistically significant. The advantage of JDK 21 over JDK 17 and of JDK 25 over JDK 21 is observed in nearly all scenarios, although isolated cases – for example, ML-DSA-44 verification at 65536 B – do not show a significant difference between JDK 21 and JDK 25. All BC vs. Native comparisons indicate a statistically significant degradation in the performance of the native JDK 25 implementation.

Key generation for ML-DSA-87 takes 2.7 times longer than for ML-DSA-44. For signing and verification, this ratio decreases with increasing message size: from 2.0 to 1.7 and from 2.6 to 1.6, respectively.

For both signing and verification operations, execution time increases with message size, with verification being substantially more sensitive to message length than signing. Figure 2 shows the benchmark results for a message of 65536 B.

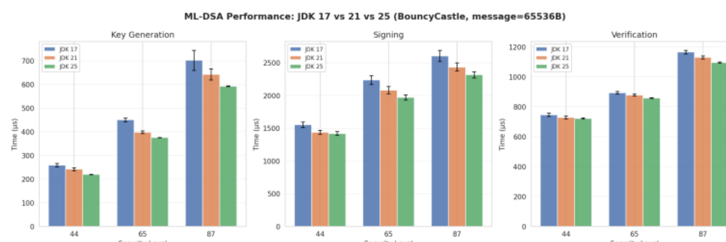

Fig. 2. ML-DSA performance comparison across JDK versions

Figure 3 shows the distributions of signing time measurements for ML-DSA-65 across BC implementations on different JDK versions and the native implementation. The native implementation is slower than BC across all investigated JDK versions. Similar results are observed for other parameter sets of the algorithm.

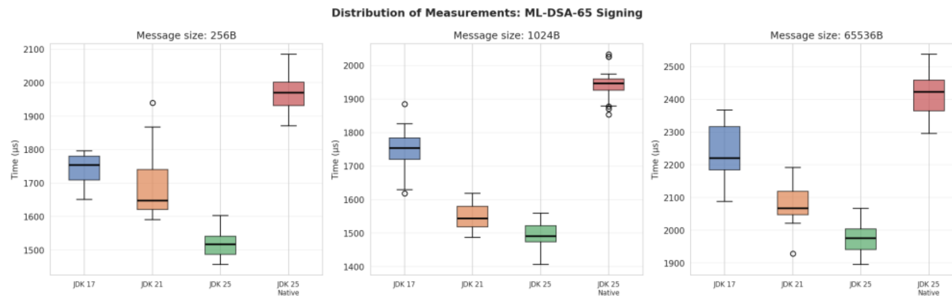

Fig. 3. Distribution of measurements for ML-DSA signing

SLH-DSA Performance. The results of the SLH-DSA performance measurements are presented in Tables 6-8. Since native support for SLH-DSA is absent in JDK 25, no comparison between implementations was conducted for this algorithm.

Table 6

SLH-DSA Performance: Key Generation (average time, µs)

Parameter	JDK-17 (BC)	JDK-21 (BC)	JDK-25 (BC)
SLH-DSA-SHA2-128S	247056.8	230597.4	223381.6
SLH-DSA-SHA2-128F	4058.5	3843.2	3713.3
SLH-DSA-SHA2-192S	373125.5	349864.1	341958.2
SLH-DSA-SHA2-192F	5965.2	5631.5	5321.1
SLH-DSA-SHA2-256S	249488.0	236068.8	224845.0
SLH-DSA-SHA2-256F	15437.8	14781.0	13824.6

For all SLH-DSA variants (128S/F, 192S/F, 256S/F), key generation demonstrates statistically significant differences across JDK 17, JDK 21, and JDK 25. All post-hoc tests confirm a significant speedup at both the 17→21 and 21→25 transitions without exception.

Table 7

SLH-DSA Performance: Signing (average operation time, µs)

Message size	Parameter	JDK-17 (BC)	JDK-21 (BC)	JDK-25 (BC)
256 B	SLH-DSA-SHA2-128S	2093410.5	1989249.3	1933437.2
	SLH-DSA-SHA2-128F	97227.4	93151.4	89928.3
	SLH-DSA-SHA2-192S	3843636.5	3621224.0	3542584.4
	SLH-DSA-SHA2-192F	161755.0	154135.9	146022.6
	SLH-DSA-SHA2-256S	3336503.1	3131803.2	3029779.3
	SLH-DSA-SHA2-256F	334843.5	313671.6	300944.7
1024 B	SLH-DSA-SHA2-128S	2115876.9	2001849.3	1915007.2
	SLH-DSA-SHA2-128F	99617.3	93239.6	89853.0
	SLH-DSA-SHA2-192S	3874433.1	3626796.4	3468723.4
	SLH-DSA-SHA2-192F	164628.8	152319.9	148032.8
	SLH-DSA-SHA2-256S	3275318.6	3155733.6	3059654.1
	SLH-DSA-SHA2-256F	325674.6	313447.8	302208.3
65536 B	SLH-DSA-SHA2-128S	2121660.7	1987320.2	1920345.2
	SLH-DSA-SHA2-128F	99293.4	93671.0	91287.3
	SLH-DSA-SHA2-192S	3849109.6	3607313.2	3468239.5
	SLH-DSA-SHA2-192F	154572.5	153547.3	149385.2
	SLH-DSA-SHA2-256S	3219025.8	3128659.1	3055066.6
	SLH-DSA-SHA2-256F	324813.9	316968.5	304664.4

Signing results for all SLH-DSA parameter sets exhibit the expected large execution times, but also a clear advantage for newer JDK versions. The general trend is that JDK 25 provides a statistically significant speedup across all message sizes, while the difference between JDK 17 and JDK 21 is not significant in one case (SLH-DSA-SHA2-192F at a message size of 65536 B).

Table 8

SLH-DSA Performance: Verification (average operation time, μs)

Message size	Parameter	JDK-17 (BC)	JDK-21 (BC)	JDK-25 (BC)
256 B	SLH-DSA-SHA2-128S	1841.4	1749.6	1734.4
	SLH-DSA-SHA2-128F	5538.1	5400.3	5301.3
	SLH-DSA-SHA2-192S	2809.3	2811.5	2630.9
	SLH-DSA-SHA2-192F	8088.9	7810.4	7566.3
	SLH-DSA-SHA2-256S	4096.1	4168.5	3934.0
	SLH-DSA-SHA2-256F	8152.8	8172.5	7557.0
1024 B	SLH-DSA-SHA2-128S	1855.8	1779.3	1752.7
	SLH-DSA-SHA2-128F	5369.3	5431.8	5243.7
	SLH-DSA-SHA2-192S	2802.7	2745.6	2727.4
	SLH-DSA-SHA2-192F	7747.0	7817.6	7771.8
	SLH-DSA-SHA2-256S	4117.6	4006.5	3960.9
	SLH-DSA-SHA2-256F	8129.2	8040.1	7696.3
65536 B	SLH-DSA-SHA2-128S	2625.5	2554.2	2483.2
	SLH-DSA-SHA2-128F	6337.3	6229.7	6022.2
	SLH-DSA-SHA2-192S	3248.9	3170.8	3178.0
	SLH-DSA-SHA2-192F	8551.9	8538.0	8198.3
	SLH-DSA-SHA2-256S	4559.2	4535.3	4483.9
	SLH-DSA-SHA2-256F	8844.7	8462.4	8058.8

For SLH-DSA signature verification, the overall picture is similar but the effect is less pronounced than for signing. Nevertheless, formal statistical tests confirm a significant difference between versions in most cases, particularly between JDK 21 and JDK 25.

Although the SLH-DSA-SHA2-192S parameter set provides a lower security level, it exhibited worse key generation and signing performance than SLH-DSA-SHA2-256S. This is explained by the internal structure of the algorithm [3].

A key characteristic of SLH-DSA is the trade-off between signing speed and signature size between the s and f variants. The f variant produces signatures faster than the s variant at the same security level, but at the cost of a larger signature size (Fig. 4). The key generation speed of SLH-DSA-SHA2-128F and SLH-DSA-SHA2-192F is 60-64 times greater than that of SLH-DSA-SHA2-128S and SLH-DSA-SHA2-192S respectively, while SLH-DSA-SHA2-256F is 16 times faster than SLH-DSA-SHA2-256S for key generation. The same trend holds for signing: SLH-DSA-SHA2-128F and SLH-DSA-SHA2-192F are 21-24 times faster than their counterparts, while SLH-DSA-SHA2-256F is only 10 times faster.

For verification the relationship is reversed: SLH-DSA-SHA2-128S is 2.4-3.0 times faster than its f counterpart, SLH-DSA-SHA2-192S is 2.4-2.9 times faster, and SLH-DSA-SHA2-256S is 1.8-2.0 times faster.

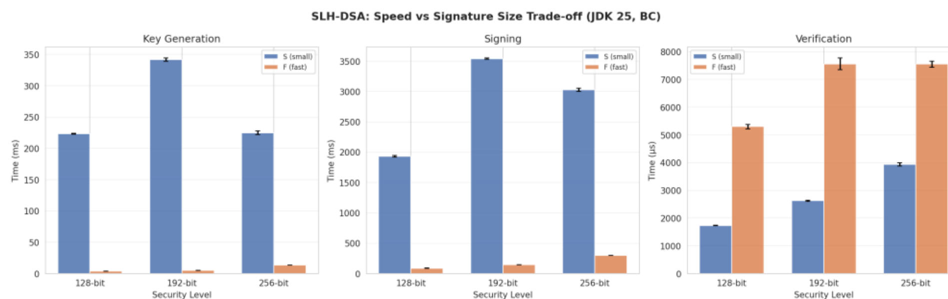

Fig. 4. SLH-DSA s and f variant performance comparison

In addition to performance, the sizes of keys and signatures are of practical interest as they directly affect network throughput and storage requirements. The corresponding data are summarized in Table 9.

Table 9

Cryptographic Artifact Sizes (bytes)

Algorithm	PublicKey	PrivateKey	Signature
ML-DSA-44	1334	2626	2420
ML-DSA-65	1974	4098	3309
ML-DSA-87	2614	4962	4627
SLH-DSA-SHA2-128S	50	84	7856
SLH-DSA-SHA2-128F	50	84	17088
SLH-DSA-SHA2-192S	66	116	16224
SLH-DSA-SHA2-192F	66	116	35664
SLH-DSA-SHA2-256S	82	150	29792
SLH-DSA-SHA2-256F	82	150	49856

A distinctive characteristic of SLH-DSA is the disproportionately small key size relative to the signature size: the public key at the 128-bit security level is only 50 bytes, while the signature ranges from 7856 to 17088 bytes depending on the variant. ML-DSA demonstrates a more balanced ratio between key and signature sizes.

To exclude the influence of getInstance() call overhead on the main benchmark results, a dedicated measurement was conducted. It was established that the getInstance() overhead does not exceed 0.5 µs and accounts for less than 1% of the total execution time of any cryptographic operation. Consequently, the performance gap between the BC and native implementations cannot be attributed to provider initialization costs. Additionally, to eliminate any potential influence of the Docker environment, the benchmarks were also executed in a standard non-containerized setting. In this case, BC performance was likewise superior.

CONCLUSIONS AND PROSPECTS FOR FURTHER RESEARCH

This paper presents a systematic comparative performance study of the post-quantum cryptographic algorithms ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205) in the Java environment, based on a reproducible experiment using the Java Microbenchmark Harness in a containerized setting across three platform versions – JDK 17, JDK 21, and JDK 25.

The following conclusions are drawn from the results of the study.

1. The execution time of all investigated operations increases monotonically with the security level of the algorithm. For ML-KEM, the transition from security level 512 to 1024 increases key generation time by a factor of 2.43, encapsulation by 2.37, and decapsulation by 2.2. For ML-DSA, an analogous relationship is observed across levels 44, 65, and 87 for all operations and JDK versions.

2. The native JDK 25 implementation consistently underperforms BouncyCastle 1.83 across all investigated algorithms and operations. For ML-KEM the gap is 3-19%, and for ML-DSA 14-35%, depending on the operation and security level. The statistical significance of these differences is confirmed by the Mann-Whitney test ($p < 0.05$) for all pairwise comparisons. A dedicated measurement of getInstance() call overhead showed that it does not exceed 0.5 µs and accounts for less than 1% of the total operation time – thus the underperformance of the native implementation is attributable to the algorithmic implementation itself rather than to infrastructure costs. The results are confirmed both in the containerized Docker environment and in direct execution on the test hardware.

3. The JDK version has a substantial effect on the performance of BC implementations. Upgrading from JDK 17 to JDK 25 yields a speedup of 14-17% for ML-DSA signing depending on the security level, and 8-11% for SLH-DSA key generation. Statistical tests confirm the significance of both the 17→21 and 21→25 transitions for most operations. This indicates that upgrading the JVM itself constitutes a free performance improvement for PQC operations – even without any change to the algorithm implementations.

4. The SLH-DSA algorithm is absent from the native JDK 25 provider and is available exclusively through BouncyCastle across all three investigated platform versions. This reflects the asynchronous nature of NIST standardization and the integration of algorithms into the Java platform. Java developers requiring the full stack of NIST PQC algorithms are required to use BouncyCastle regardless of the JDK version.

5. SLH-DSA exhibits a fundamental trade-off between the s and f variants: the f variant produces signatures 10-25 times faster than the s variant at the same security level, but with a signature size that is 1.7-2.2 times larger. Verification for the f variant is 3-4 times slower than for the s variant. The choice between variants should be determined by the requirements of the specific use case: the f variant is preferable for online document signing, while the s variant is more appropriate for long-term storage where signature size is critical. Compared to ML-DSA, both SLH-DSA variants are substantially slower for signing and key generation, but offer a

significantly smaller public key size (50 bytes versus 1334-2614 bytes) and rely on more conservative security assumptions.

6. For most scenarios, the optimal choice is BouncyCastle 1.83 via the BC provider, regardless of the JDK version. This ensures the full stack of NIST algorithms including SLH-DSA and superior performance compared to the native JDK 25 implementation. Adopting the native JDK 25 implementation without BC may be premature: the native implementation underperforms BC and does not support SLH-DSA. Among JDK versions, JDK 25 (LTS) with the latest BouncyCastle 1.83 is recommended.

Directions for further research include: investigating PQC algorithm performance in the context of TLS 1.3 following the release of JDK 27 with JEP 527 (hybrid key exchange ML-KEM + X25519); evaluating the impact of PQC on the performance of Spring Boot microservices under real-world load; studying the memory consumption characteristics of PQC operations through GC profiling; analyzing PQC algorithm performance on ARM architectures, which is relevant for cloud deployments; and developing methods for automated detection of cryptographic dependencies in existing Java applications to support migration planning.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. National Institute of Standards and Technology. (2024a). *FIPS 203: Module-lattice-based key-encapsulation mechanism standard*. https://doi.org/10.6028/NIST.FIPS.203
2. National Institute of Standards and Technology. (2024b). *FIPS 204: Module-lattice-based digital signature standard*. https://doi.org/10.6028/NIST.FIPS.204
3. National Institute of Standards and Technology. (2024c). *FIPS 205: Stateless hash-based digital signature standard*. https://doi.org/10.6028/NIST.FIPS.205
4. Paquin, C., Stebila, D., & Tamvada, G. (2020). Benchmarking post-quantum cryptography in TLS. In J. Ding & J.-P. Tillich (Eds.), *Post-quantum cryptography (PQCrypto 2020)* (Lecture Notes in Computer Science, Vol. 12100). Springer. https://doi.org/10.1007/978-3-030-44223-1_5
5. Dziechciarz, D., & Niemiec, M. (2025). Efficiency analysis of NIST-standardized post-quantum cryptographic algorithms for digital signatures in various environments. *Electronics, 14*(1), 70. https://doi.org/10.3390/electronics14010070
6. Abbasi, M., Cardoso, F., Váz, P., Silva, J., & Martins, P. (2025). A practical performance benchmark of post-quantum cryptography across heterogeneous computing environments. *Cryptography, 9*(2), 32. https://doi.org/10.3390/cryptography9020032
7. Montenegro, J. A., Rios, R., & Lopez-Cerezo, J. (2026). A performance evaluation framework for post-quantum TLS. *Future Generation Computer Systems, 175*, 108062. https://doi.org/10.1016/j.future.2025.108062
8. Opiłka, F., Niemiec, M., Gagliardi, M., & Kourtis, M. A. (2024). Performance analysis of post-quantum cryptography algorithms for digital signature. *Applied Sciences, 14*(12), 4994. https://doi.org/10.3390/app14124994
9. Averichev, I., Rozhenko, A., & Kykhtenko, Y. (2025). Innovative approaches to improving the level of cybersecurity of corporate networks using cloud technologies. *Cybersecurity: Education, Science, Technique, 1*(29), 732-747. https://doi.org/10.28925/2663-4023.2025.29.934
10. Zarudnyi, I., & Liubchak, V. (2025). Methods and information technologies for secure integration of the Ethereum blockchain with the Internet of Things (IoT). *Cybersecurity: Education, Science, Technique, 4*(28), 104-114. https://doi.org/10.28925/2663-4023.2025.28.758
11. Prokopovych-Tkachenko, D. I., Khrushkov, B. S., & Derkach, Y. O. (2025). Post-quantum threats to information security: Challenges at the global and national levels. *Systems and Technologies, 69*(1), 118-123. https://doi.org/10.32782/2521-6643-2025-1-69.14
12. Oracle JEP 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism
13. Bouncy Castle PQC and Lightweight Cryptography Updates
14. Miqdad, A. (2024). *Analyzing Java Microbenchmark Harness (JMH) performance in open-source systems*. https://hdl.handle.net/2077/84478
15. Kumar, M., & Pattnaik, P. (2020). Post-quantum cryptography (PQC): An overview. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC 2020)* (pp. 1-9). IEEE. https://doi.org/10.1109/HPEC43674.2020.9286147
16. Shand, M., & Vuillemin, J. (1993, June). Fast implementations of RSA cryptography. In *Proceedings of the IEEE Symposium on Computer Arithmetic* (pp. 252-259). https://doi.org/10.1109/ARITH.1993.378085

17. Johnson, D., Menezes, A., & Vanstone, S. (2001). The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security, 1*, 36-63. https://doi.org/10.1007/s102070100002
18. Deshpande, S., Lee, Y., Karakuzu, C., Szefer, J., & Paek, Y. (2025). SPHINCSLET: An area-efficient accelerator for the full SPHINCS+ digital signature algorithm. *ACM Transactions on Embedded Computing Systems, 24*(5), Article 69, 1-19. https://doi.org/10.1145/3728469
19. Oleksiichuk, Y. (2026). *PQC JDK benchmark: Performance benchmarks for NIST PQC algorithms ML-KEM, ML-DSA, SLH-DSA across JDK 17, 21, 25* [Computer software]. GitHub repository
20. Birnbaum, Z. W. (1956). On a use of the Mann-Whitney statistic. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability* (Vol. 1, pp. 13-18). University of California Press.

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (33), 2026
ISSN 2663 – 4023

Олексійчук Юрій Федорович
кандидат фізико-математичних наук,
доцент кафедри комп'ютерних наук та інформаційних технологій
Полтавський університет економіки і торгівлі, Полтава, Україна
ORCID: 0000-0002-0585-3307
olexijchuk@gmail.com

ЕМПІРИЧНЕ ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ПОСТКВАНТОВИХ КРИПТОГРАФІЧНИХ АЛГОРИТМІВ ML-KEM, ML-DSA ТА SLH-DSA У РІЗНИХ ВЕРСІЯХ JDK

Анотація. Стандартизація NIST у 2024 році алгоритмів ML-KEM (FIPS 203), ML-DSA (FIPS 204) та SLH-DSA (FIPS 205) ознаменувала початок практичного переходу до постквантової криптографії в промислових програмних системах. Платформа Java залишається однією з найпоширеніших у корпоративному середовищі, однак розробники стикаються з практичною проблемою вибору: використовувати бібліотеку BouncyCastle, яка підтримує нові алгоритми на всіх актуальних версіях JDK, чи нативні можливості що з'явилися у JDK 25. Відсутність систематичних емпіричних даних про продуктивність цих алгоритмів у середовищі Java в залежності від версії платформи, рівня безпеки та типу операції ускладнює обґрунтований вибір при плануванні міграції корпоративних Java-систем до постквантових стандартів. У статті проведено систематичне порівняльне дослідження продуктивності алгоритмів ML-KEM, ML-DSA та SLH-DSA у середовищі Java на основі відтворюваного експерименту з використанням Java Microbenchmark Harness у контейнеризованому середовищі для трьох версій платформи – JDK 17, JDK 21 та JDK 25 – з порівнянням реалізацій BouncyCastle 1.83 та нативної підтримки JDK 25. Для кожного алгоритму вимірювались три основні операції: генерація пари ключів, інкапсуляція або формування підпису, декапсуляція або верифікація. Для алгоритмів підпису додатково варіювався розмір повідомлення – 256 байтів, 1024 байти та 65536 байтів. Статистична значущість відмінностей оцінювалась за критеріями Манна-Уітні та Краскела-Уолліса з post-hoc аналізом за методом Данна з поправкою Бонферроні. Встановлено що нативна реалізація JDK 25 стабільно поступається BouncyCastle для всіх алгоритмів і операцій. Окремий бенчмарк overhead ініціалізації провайдера підтвердив що ця різниця зумовлена алгоритмічною реалізацією а не накладними витратами інфраструктури. Перехід з JDK 17 на JDK 25 забезпечує статистично значуще прискорення BC реалізацій. За результатами дослідження сформульовано практичні рекомендації для Java-розробників щодо вибору реалізації та версії платформи при міграції до постквантових стандартів. Вихідний код і конфігурація Docker Compose опубліковані у відкритому репозиторії GitHub для незалежного відтворення результатів.

Ключові слова: постквантова криптографія; ML-KEM; ML-DSA; SLH-DSA; Java; BouncyCastle; JDK; JMH.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. National Institute of Standards and Technology. (2024a). *FIPS 203: Module-lattice-based key-encapsulation mechanism standard*. https://doi.org/10.6028/NIST.FIPS.203
2. National Institute of Standards and Technology. (2024b). *FIPS 204: Module-lattice-based digital signature standard*. https://doi.org/10.6028/NIST.FIPS.204
3. National Institute of Standards and Technology. (2024c). *FIPS 205: Stateless hash-based digital signature standard*. https://doi.org/10.6028/NIST.FIPS.205
4. Paquin, C., Stebila, D., & Tamvada, G. (2020). Benchmarking post-quantum cryptography in TLS. In J. Ding & J.-P. Tillich (Eds.), *Post-quantum cryptography (PQCrypto 2020)* (Lecture Notes in Computer Science, Vol. 12100). Springer. https://doi.org/10.1007/978-3-030-44223-1_5
5. Dziechciarz, D., & Niemiec, M. (2025). Efficiency analysis of NIST-standardized post-quantum cryptographic algorithms for digital signatures in various environments. *Electronics, 14*(1), 70. https://doi.org/10.3390/electronics14010070

6. Abbasi, M., Cardoso, F., Váz, P., Silva, J., & Martins, P. (2025). A practical performance benchmark of post-quantum cryptography across heterogeneous computing environments. *Cryptography, 9*(2), 32. https://doi.org/10.3390/cryptography9020032
7. Montenegro, J. A., Rios, R., & Lopez-Cerezo, J. (2026). A performance evaluation framework for post-quantum TLS. *Future Generation Computer Systems, 175*, 108062. https://doi.org/10.1016/j.future.2025.108062
8. Opiłka, F., Niemiec, M., Gagliardi, M., & Kourtis, M. A. (2024). Performance analysis of post-quantum cryptography algorithms for digital signature. *Applied Sciences, 14*(12), 4994. https://doi.org/10.3390/app14124994
9. Averichev, I., Rozhenko, A., & Kykhtenko, Y. (2025). Innovative approaches to improving the level of cybersecurity of corporate networks using cloud technologies. *Cybersecurity: Education, Science, Technique, 1*(29), 732-747. https://doi.org/10.28925/2663-4023.2025.29.934
10. Zarudnyi, I., & Liubchak, V. (2025). Methods and information technologies for secure integration of the Ethereum blockchain with the Internet of Things (IoT). *Cybersecurity: Education, Science, Technique, 4*(28), 104-114. https://doi.org/10.28925/2663-4023.2025.28.758
11. Prokopovych-Tkachenko, D. I., Khrushkov, B. S., & Derkach, Y. O. (2025). Post-quantum threats to information security: Challenges at the global and national levels. *Systems and Technologies, 69*(1), 118-123. https://doi.org/10.32782/2521-6643-2025-1-69.14
12. Oracle JEP 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism
13. Bouncy Castle PQC and Lightweight Cryptography Updates
14. Miqdad, A. (2024). *Analyzing Java Microbenchmark Harness (JMH) performance in open-source systems*. https://hdl.handle.net/2077/84478
15. Kumar, M., & Pattnaik, P. (2020). Post-quantum cryptography (PQC): An overview. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC 2020)* (pp. 1-9). IEEE. https://doi.org/10.1109/HPEC43674.2020.9286147
16. Shand, M., & Vuillemin, J. (1993, June). Fast implementations of RSA cryptography. In *Proceedings of the IEEE Symposium on Computer Arithmetic* (pp. 252-259). https://doi.org/10.1109/ARITH.1993.378085
17. Johnson, D., Menezes, A., & Vanstone, S. (2001). The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security, 1*, 36-63. https://doi.org/10.1007/s102070100002
18. Deshpande, S., Lee, Y., Karakuzu, C., Szefer, J., & Paek, Y. (2025). SPHINCSLET: An area-efficient accelerator for the full SPHINCS+ digital signature algorithm. *ACM Transactions on Embedded Computing Systems, 24*(5), Article 69, 1-19. https://doi.org/10.1145/3728469
19. Oleksiichuk, Y. (2026). *PQC JDK benchmark: Performance benchmarks for NIST PQC algorithms ML-KEM, ML-DSA, SLH-DSA across JDK 17, 21, 25* [Computer software]. GitHub repository
20. Birnbaum, Z. W. (1956). On a use of the Mann-Whitney statistic. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability* (Vol. 1, pp. 13-18). University of California Press.

Отримано редакцією журналу / Received: 06.02.26
Прорецензовано / Revised: 20.02.26
Схвалено до друку / Accepted: 25.06.26

This work is licensed under Creative Commons Attribution-noncommercial-sharealike 4.0 International License.