UDC 004.056.55:004.43
DOI https://doi.org/10.32782/2521-6643-2026-2-72.26

Oleksiichuk Yu. F., Candidate of Physical and Mathematical Sciences,
Associate Professor at the Department of Computer Science and Information Technology
Poltava University of Economics and Trade
ORCID: 0000-0002-0585-3307

POST-QUANTUM CRYPTOGRAPHY IN THE JAVA ECOSYSTEM

The completion of NIST standardization in 2024, with the publication of ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205), has confronted Java developers with a concrete practical question: how to integrate these algorithms into real-world applications using the available platform tools. The Java ecosystem offers several fundamentally different integration paths depending on the JDK version; however, a systematic description of these approaches from a practitioner's perspective remains insufficiently represented in the scientific literature.

This paper analyzes the evolution of post-quantum cryptography support in the Java platform from JDK 17 through JDK 25, with an outlook toward JDK 27. The Java Cryptography Architecture and the role of the provider model as a foundation for implementing the crypto agility concept are examined. Two API levels of the BouncyCastle 1.83 library – the JCA provider and the low-level lightweight API – are compared, and the appropriate use cases for each are identified. It is established that native support in JDK 25 covers only ML-KEM and ML-DSA, while SLH-DSA remains available exclusively through BouncyCastle across all JDK versions, demonstrating the asynchronous nature of NIST standardization and JDK platform integration.

Practical code examples are developed and analyzed for two typical scenarios. The first involves hybrid JWT token signing using both classical ECDSA and post-quantum ML-DSA simultaneously in a Spring Boot application, ensuring backward compatibility with existing infrastructure. The second addresses secure inter-service communication via ML-KEM with public key authentication using ML-DSA to protect against man-in-the-middle attacks. It is shown that direct use of ML-DSA in JWT is incompatible with existing libraries due to the absence of corresponding algorithms in RFC 7515 and RFC 7518, and practical mitigation approaches for the transition period are proposed, including reference tokens and token introspection.

A four-phase migration strategy for enterprise Java systems is proposed: inventory of cryptographic dependencies, introduction of crypto agility through centralized algorithm configuration in Spring Boot, a hybrid mode of parallel classical and post-quantum algorithm usage, and full migration following PQC TLS standardization in JDK 27. The results of this work can be directly applied by Java developers when planning and implementing the migration of enterprise applications to post-quantum cryptographic standards.

Key words: *post-quantum cryptography, ML-KEM, ML-DSA, SLH-DSA, Java, BouncyCastle, Spring Boot, JCA, JWT, migration*

Олексійчук Ю. Ф. Постквантова криптографія в екосистемі Java

Завершення стандартизації NIST у 2024 році з публікацією алгоритмів ML-KEM (FIPS 203), ML-DSA (FIPS 204) та SLH-DSA (FIPS 205) поставило перед Java-розробниками конкретне практичне питання: як інтегрувати ці алгоритми у реальні застосунки, використовуючи наявні інструменти платформи. Екосистема Java пропонує кілька принципово різних шляхів такої інтеграції залежно від версії JDK, однак систематизований опис цих підходів з практичної точки зору залишається недостатньо представленим у науковій літературі.

У статті проаналізовано еволюцію підтримки постквантової криптографії у платформі Java від JDK 17 до JDK 25 та окреслено перспективи JDK 27. Розглянуто архітектуру Java Cryptography Architecture та роль моделі провайдерів як основи для реалізації концепції crypto agility. Порівняно два рівні API бібліотеки BouncyCastle 1.83 – JCA провайдер та низькорівневий lightweight API – і визначено сценарії застосування кожного. Встановлено що нативна підтримка у JDK 25 охоплює лише ML-KEM і ML-DSA, тоді як SLH-DSA залишається доступним виключно через BouncyCastle, що свідчить про асинхронність між стандартизацією NIST і інтеграцією алгоритмів у платформу Java.

Розроблено та проаналізовано практичні приклади коду для двох типових сценаріїв. Перший – гібридний підпис JWT токенів із одночасним використанням класичного ECDSA та постквантового ML-DSA у Spring Boot застосунку, що забезпечує зворотну сумісність з існуючою інфраструктурою. Другий – захищена міжсервісна взаємодія через ML-KEM із автентифікацією публічного ключа засобами ML-DSA для захисту від атак типу «людина посередині». Виявлено що пряме використання ML-DSA у JWT є несумісним з існуючими бібліотеками через відсутність відповідних алгоритмів у стандартах RFC 7515 і RFC 7518, і запропоновано практичні підходи для перехідного періоду.

Запропоновано чотирифазову стратегію міграції корпоративних Java-систем до постквантових стандартів: інвентаризація криптографічних залежностей, впровадження crypto agility через централізовану конфігурацію алгоритмів у Spring Boot, гібридний режим паралельного використання класичних і постквантових алгоритмів, та

повна міграція після стандартизації PQC TLS у JDK 27. Результати роботи можуть бути безпосередньо використані Java-розробниками при плануванні та реалізації міграції корпоративних застосунків до постквантових стандартів.
Ключові слова: *постквантова криптографія, ML-KEM, ML-DSA, SLH-DSA, Java, BouncyCastle, Spring Boot, JCA, JWT, міграція*

Formulation of the problem. The threat posed by quantum computing [1–2] to modern cryptographic systems has moved from theory to practice: experts estimate the emergence of cryptographically relevant quantum computers within the next 10–15 years, while the «harvest now, decrypt later» attack strategy renders migration an immediate concern [3]. The completion of NIST standardization [4–6] in 2024 has removed a major barrier to the practical adoption of post-quantum cryptography; however, it also introduces new challenges for Java developers. The Java ecosystem offers several fundamentally different integration approaches – through the Java Cryptography Architecture / Java Cryptography Extension (JCA/JCE) provider model with the BouncyCastle library, through the KEM API introduced in JDK 21 (JEP 452) [7], and through native support in JDK 25 – each with its own characteristics, limitations, and compatibility implications for existing infrastructure. At the same time, there is no systematic treatment of these approaches from the perspective of practicing developers: which APIs to use for specific scenarios, how to ensure compatibility with the Spring framework, how to structure hybrid schemes for the transitional period, and what pitfalls arise when integrating into real-world applications. Developers must navigate library documentation, JEP specifications, and release notes across multiple JDK versions independently – a situation that significantly complicates and slows the migration of enterprise Java systems to post-quantum standards.

Analysis of recent research and publications. The challenge of transitioning to post-quantum cryptography has taken on practical significance following the completion of NIST standardization in 2024. Research in this area is advancing along several concurrent tracks: analysis of threats and migration timelines, the concept of cryptographic agility, and PQC support across specific platforms and protocols.

Regarding timelines and threats, the Global Risk Institute's annual report finds that the majority of experts place the emergence of cryptographically relevant quantum computers within a 10–20 year horizon [8]. A study on enterprise migration timelines [9] concluded that realistic estimates substantially exceed optimistic projections, ranging from 5–7 years for small organizations to 12–15 years for large corporations, creating a gap that, when set against projected quantum computing timelines, creates a critical security window.

On cryptographic agility as an architectural principle, NIST published document CSWP 39 [10], which systematizes approaches to achieving crypto agility, defined as the ability of a system to replace cryptographic algorithms without significant architectural changes. A survey in [11] examined crypto-agility concepts in the context of quantum resistance, defining it as the capacity to rapidly and securely replace cryptographic algorithms and associated data when those algorithms are compromised. The authors of [12] proposed the concept of software-defined cryptography as a mechanism for realizing cryptographic agility through centralized management of cryptographic policies and their automated enforcement.

The Java platform already provides the building blocks for hybrid key exchange schemes, having introduced the KEM API in Java 21 (JEP 452) [13] and the ML-KEM algorithm in Java 24 (JEP 496) [14]. The next step is native support for hybrid TLS key exchange via JEP 527 (QUBIP), targeted for JDK 27 [15]. PQC integration into the Java platform is thus incremental and remains incomplete even in JDK 25.

At the protocol level, the authors of [16] proposed a framework for evaluating the performance impact of PQC on TLS, documenting increased handshake latency when using ML-KEM and ML-DSA algorithms. The NCSC guidance emphasizes that, except in the simplest systems, classical and post-quantum cryptography must coexist throughout the transitional period; systems must therefore support cryptographic agility, i.e., the ability to accommodate alternative algorithm suites [17].

In summary, despite active research in adjacent areas, a systematic treatment of practical approaches to integrating ML-KEM, ML-DSA, and SLH-DSA directly into Java applications, accounting for the specifics of the JCA/JCE platform, multiple JDK versions, and the Spring framework, remains insufficiently represented in the scientific literature.

The purpose of this article is to systematize practical approaches to integrating the post-quantum cryptographic algorithms ML-KEM, ML-DSA, and SLH-DSA into Java applications – by analyzing the evolution of PQC support across the Java platform for different JDK versions, comparing available implementations and APIs at the level of architectural approaches, developing practical recommendations and code examples for common usage scenarios including integration with Spring Boot applications, and outlining a phased migration strategy for enterprise Java systems toward post-quantum standards, accounting for backward compatibility requirements and the transitional period.

Presenting main material. Post-quantum cryptography (PQC) encompasses cryptographic algorithms whose security is not undermined by known quantum algorithms. Unlike classical asymmetric algorithms – RSA, ECDSA, and ECDH – whose security can be broken by a sufficiently powerful quantum computer, post-quantum algorithms rely on mathematical problems for which no efficient quantum algorithms are currently known. The following problem classes arise most frequently in practice.

1. **Secure key exchange.** Two parties – for example, two microservices or a client and a server – wish to establish a shared secret key for subsequent data encryption, communicating over an open channel where an adversary can read all messages. The classical solution – the Diffie-Hellman protocol or ECDH – is vulnerable to quantum attacks. ML-KEM implements a key encapsulation mechanism: one party generates a shared secret and «wraps» it into a ciphertext using the other party's public key; the recipient «unwraps» the ciphertext with their private key and recovers the same shared secret. This secret is then used as a key for AES-256 symmetric encryption.

2. **Digital signatures.** One party wishes to sign data – a document, a token, a message – such that any other party can verify that the signature was produced by the holder of the private key and that the data has not been altered since signing. The classical solutions – ECDSA and RSA-PSS – are vulnerable to quantum attacks. The post-quantum alternatives are ML-DSA (FIPS 204) and SLH-DSA (FIPS 205). Both algorithms address the same problem but exhibit different characteristics. ML-DSA is based on lattice problems and generally offers a better balance between speed and signature size for most practical applications. SLH-DSA relies exclusively on hash functions, making it the most conservative choice from a security standpoint; however, it is significantly slower at signature generation and produces larger signatures.

3. **Data encryption.** Protecting the confidentiality of data at rest or in transit. A fundamental constraint must be understood here: none of the three NIST PQC standards are used for directly encrypting large volumes of data. They are asymmetric primitives, whereas bulk data encryption is performed by symmetric algorithms. AES-256 and ChaCha20 are already quantum-resistant at appropriate key lengths: Grover's algorithm merely halves the effective key length, reducing the security margin of AES-128, while AES-256 remains a more conservative choice [18], [19]. Consequently, post-quantum migration for data encryption reduces, in the base case, to a single step: replacing the symmetric key transport mechanism from ECDH to ML-KEM. The data encryption procedure itself remains unchanged.

The Java platform provides a unified mechanism for working with cryptographic algorithms through the Java Cryptography Architecture (JCA) and its extension, the Java Cryptography Extension (JCE). The central element of JCA is the provider model. A provider is a named software module that implements one or more cryptographic services – signature algorithms, key pair generators, encryption mechanisms, and so on. An application interacts not with a specific algorithm implementation, but with an algorithm-agnostic API – classes such as KeyPairGenerator, Signature, Cipher, KeyAgreement, and KEM. The concrete implementation is selected at runtime based on the requested algorithm name and the registered providers.

This architecture naturally supports the concept of crypto agility – the ability of a system to replace cryptographic algorithms without modifying business logic. If an algorithm name is not hardcoded directly in the application but externalized to configuration, swapping algorithms reduces to changing a single configuration value.

When an algorithm is requested without an explicit provider, the JVM iterates over registered providers in priority order and returns the first matching implementation found. If a provider is specified explicitly, only that provider is used.

A standard JDK installation includes several built-in providers. The most relevant for developers are:

- SUN – the base provider for signature and hashing algorithms;
- SunJCE – the provider for symmetric encryption and cryptographic primitives;
- SunEC – the provider for elliptic-curve algorithms: ECDSA, ECDH, and X25519;
- SunPKCS11 – the provider for interacting with hardware cryptographic modules via the PKCS#11 interface.

Prior to JDK 21, the JCA architecture lacked a standard service type for key encapsulation mechanisms – KEM. Key exchange was implemented through KeyAgreement, which is semantically misaligned with the KEM model, where one party is active (generating the ciphertext) and the other is passive (performing decapsulation). JEP 452 in JDK 21 introduced the new class javax.crypto.KEM, which accurately reflects the semantics of key encapsulation mechanisms.

In JDK 17, javax.crypto.KEM is absent, and working with ML-KEM requires the low-level BouncyCastle API via the MLKEMGenerator and MLKEMExtractor classes.

One practically important point is that even in JDK 25, native support for SLH-DSA is absent. This means that developers requiring the full stack of NIST PQC algorithms must use BouncyCastle regardless of the JDK version.

The BouncyCastle library integrates with JCA through the standard provider mechanism and provides two providers with distinct roles. The BC provider is the primary provider, containing implementations of most algorithms, including all three NIST-standardized PQC algorithms. The BCPQC provider was historically used for post-quantum algorithms in Bouncy Castle. As of version 1.76, the majority of PQC algorithms have been consolidated into the main BC provider, making it unnecessary to register BCPQC explicitly. As a result, BCPQC is rarely used in new projects.

In addition to the JCA provider, BouncyCastle offers a low-level lightweight API consisting of classes in the org.bouncycastle.pqc.crypto.* packages, that operates independently of the JCA infrastructure. This API provides full control over algorithm parameters and remains the only viable option for JDK 17, where javax.crypto.KEM is unavailable. Table 2 summarizes the differences between the two API levels.

Table 1

Evolution of PQC Support in the Java Platform

JDK Version	Support Type	What's New
JDK 17 (LTS)	Third-party library	PQC available only via BouncyCastle; no native support
JDK 21 (LTS)	API + third-party library	JEP 452: new javax.crypto.KEM class; ML-KEM via BC provider
JDK 24	Preview	JEP 496: ML-KEM preview; JEP 497: ML-DSA preview
JDK 25 (LTS)	Partial native	ML-KEM and ML-DSA finalized; SLH-DSA absent from native JDK
JDK 26 (TBD)	Partial native	ML-DSA support for JAR file signing
JDK 27 (TBD)	Partial native	JEP 527: hybrid key exchange ML-KEM + X25519 for TLS 1.3

Table 2

Comparison of BouncyCastle API Levels for PQC

Characteristic	JCA via BC Provider	Lightweight API
Interface	Standard JCA (KeyPairGenerator, Signature, KEM)	BC classes (MLKEMGenerator, MLDSASigner)
Portability	High – provider can be replaced easily	Low – tightly coupled to BC
JDK 17 compatibility	Partial – KEM API unavailable	Full
Recommended scenario	Production code, Spring integration	JDK 17, low-level operations

As of 2026, Spring Boot does not provide built-in support for post-quantum algorithms at the application protocol level (e.g., JWT and JWS). Although the Java platform, as of JDK 25, includes standardized implementations of ML-KEM (FIPS 203) and ML-DSA (FIPS 204), these algorithms have not yet been incorporated into the current JSON Web Signature standards (RFC 7515 [20], RFC 7518 [21]). Their standardization within the JOSE framework is ongoing and addressed in several active IETF Internet-Draft documents.

As a result, using ML-DSA for JWT signing requires custom solutions. Directly substituting ML-DSA for standard JWS algorithms leads to incompatibility with existing libraries that validate the alg header field. A more practical approach is a hybrid scheme consistent with current IETF standardization efforts: a standard JWT is signed using a classical algorithm such as ECDSA, while an additional post-quantum ML-DSA signature is computed over the same message. This approach is currently under consideration in relevant IETF Internet-Draft documents on composite signatures.

This scheme preserves backward compatibility with existing infrastructure and enables the incremental adoption of post-quantum cryptography ahead of the completion of JOSE standardization.

The following example demonstrates the use of ML-DSA-65 for token signing in a Spring Boot project (version 3.5.12, JDK 25). Configuration:

```
@Configuration
public class PqcKeyConfig {
    @Bean
    public KeyPair ecdsaKeyPair() throws Exception {
        KeyPairGenerator kpg = KeyPairGenerator.getInstance(«EC»);
        kpg.initialize(256);
        return kpg.generateKeyPair();
    }
    @Bean
    public KeyPair mlDsaKeyPair() throws Exception {
        KeyPairGenerator kpg =
            KeyPairGenerator.getInstance(«ML-DSA-65»);
        return kpg.generateKeyPair();
    }
}
Service for token generation and validation:
@Service
public class HybridJwtService {
    private final KeyPair ecdsaKeyPair;
    private final KeyPair mlDsaKeyPair;
    public HybridJwtService(KeyPair ecdsaKeyPair,
                            KeyPair mlDsaKeyPair) {
        this.ecdsaKeyPair = ecdsaKeyPair;
        this.mlDsaKeyPair = mlDsaKeyPair;
    }
```

```
    private final Base64.Encoder encoder =
        Base64.getUrlEncoder().withoutPadding();
    private final Base64.Decoder decoder =
        Base64.getUrlDecoder();
    public String generate(String subject) throws Exception {
        String header = encoder.encodeToString(
“{\”alg\“:\”ES256\“,\”typ\“:\”JWT\“}”.getBytes()
        );
        String payload = encoder.encodeToString(
            (“{\”sub\“:\”“ + subject + “\”}”)
                    .getBytes()
        );
        String signingInput = header + «.» + payload;
        Signature ecdsa = Signature.getInstance(«SHA256withECDSA»);
        ecdsa.initSign(ecdsaKeyPair.getPrivate());
        ecdsa.update(signingInput.getBytes());
        String ecdsaSig = encoder.encodeToString(ecdsa.sign());
        Signature pqc = Signature.getInstance(«ML-DSA-65»);
        pqc.initSign(mlDsaKeyPair.getPrivate());
        pqc.update(signingInput.getBytes());
        String pqcSig = encoder.encodeToString(pqc.sign());
        return signingInput + «.» + ecdsaSig + «.» + pqcSig;
    }
    public boolean verify(String token) throws Exception {
        String[] parts = token.split(«\\.»);
        if (parts.length != 4) return false;
        String signingInput = parts[0] + «.» + parts[1];
        byte[] ecdsaSig = decoder.decode(parts[2]);
        byte[] pqcSig = decoder.decode(parts[3]);
        Signature ecdsa = Signature.getInstance(«SHA256withECDSA»);
        ecdsa.initVerify(ecdsaKeyPair.getPublic());
        ecdsa.update(signingInput.getBytes());
        if (!ecdsa.verify(ecdsaSig)) return false;
        Signature pqc = Signature.getInstance(«ML-DSA-65»);
        pqc.initVerify(mlDsaKeyPair.getPublic());
        pqc.update(signingInput.getBytes());
        return pqc.verify(pqcSig);
    }
}
```

A simple controller for testing purposes:

```
@RestController
@RequestMapping(“/auth”)
public class AuthController {
    private final HybridJwtService jwtService;
    public AuthController(HybridJwtService jwtService) {
        this.jwtService = jwtService;
    }
    @GetMapping(«/token»)
    public Map<String, String> token(@RequestParam String user)
            throws Exception {
        String token = jwtService.generate(user);
        return Map.of(«token», token);
    }
    @GetMapping(«/verify»)
    public Map<String, Boolean> verify(@RequestParam String token)
            throws Exception {
        return Map.of(«valid», jwtService.verify(token));
    }
}
```

The primary practical limitation of using post-quantum signature algorithms in JWT is the substantial size of the resulting signatures, which can increase token size by an order of magnitude and may exceed the HTTP header size limits imposed by web servers and reverse proxies.

Several approaches are used in practice to mitigate this issue. First, the use of reference tokens: instead of a self-contained JWT, the client receives an opaque session identifier while the server maintains authentication state internally. This resolves the size problem but requires abandoning stateless architecture. Second, selective application of PQC: the post-quantum signature is used only during initial authentication to establish a secure session, while subsequent requests are authorized using short-lived classical tokens of smaller size. Third, token introspection (RFC 7662 [22]): the resource server validates the token by querying the authorization server rather than verifying the signature locally. This allows large cryptographic signatures to be handled centrally, at the cost of additional network latency.

The choice among these approaches depends on the system's scalability requirements, acceptable latency, and the required level of cryptographic strength.

In a microservices architecture, services often require a secure channel for transmitting sensitive data. The ML-KEM key encapsulation mechanism (FIPS 203) can be used in combination with the ML-DSA digital signature algorithm (FIPS 204) for this purpose.

An important property of KEM is that it does not provide party authentication on its own. Consequently, using ML-KEM without additional protective mechanisms leaves the system vulnerable to man-in-the-middle attacks. To address this, an authenticated KEM approach can be used: the recipient's public key is signed by a trusted signing key, and the sender verifies this signature before using the key.

Once a shared secret has been established via ML-KEM, subsequent data encryption is performed using a symmetric algorithm such as AES-GCM.

Migrating an enterprise Java system to post-quantum cryptography is not a one-time algorithm replacement but a phased process that may span several years. The complexity is determined by the scale of the system, the number of cryptographic dependencies, and backward compatibility requirements. Research in [9] indicates that realistic migration timelines for large corporations range from 12 to 15 years. This figure, when set against projected quantum computing timelines, makes early initiation of the process essential.

The proposed strategy consists of four sequential phases.

Phase 1. Cryptographic dependency inventory. Before migration can begin, it is necessary to identify precisely what needs to be migrated. Most enterprise Java systems contain cryptographic dependencies at multiple levels: explicit JCA/JCE calls in application code, cryptography embedded within frameworks such as Spring Security and Jakarta EE, transitive dependencies introduced through third-party libraries such as OkHttp, gRPC, and the Kafka client, and configuration-level dependencies in the form of TLS cipher suites and KeyStore parameters.

Bytecode-level dependency discovery can be performed through reflective analysis or static analyzers. The output of this phase is a cryptographic dependency registry: a catalog of algorithms in use, their locations within the codebase, and the libraries that depend on them. This registry serves as the input for all subsequent phases.

Phase 2. Implementing crypto agility. Before algorithm replacement begins, the application architecture must be prepared to accommodate future changes with minimal disruption. This entails eliminating hardcoded algorithm names and centralizing cryptographic configuration.

A centralized cryptographic algorithm provider implemented as a Spring Bean represents the foundational pattern:

```
@Configuration
@ConfigurationProperties(prefix = “crypto”)
public class CryptoConfig {
    private String signingAlgorithm = “ML-DSA-65”;
    private String kemAlgorithm = “ML-KEM-768”;
    private String symmetricAlgorithm = “AES/GCM/NoPadding”;
    private String provider = “BC”;
    @Bean
    public CryptoProvider cryptoProvider() {
        return new CryptoProvider(
            signingAlgorithm,
            kemAlgorithm,
            symmetricAlgorithm,
            provider
        );
    }
}
```

```
@Component
public class CryptoProvider {
    private final String signingAlg;
    private final String kemAlg;
    private final String symmetricAlg;
    private final String provider;
    public CryptoProvider(String signingAlg, String kemAlg,
                    String symmetricAlg, String provider) {
        this.signingAlg = signingAlg;
        this.kemAlg = kemAlg;
        this.symmetricAlg = symmetricAlg;
        this.provider = provider;
        if (Security.getProvider(provider) == null) {
            Security.addProvider(new BouncyCastleProvider());
        }
    }
    public KeyPair generateSigningKeyPair() throws Exception {
        return KeyPairGenerator
            .getInstance(signingAlg, provider)
            .generateKeyPair();
    }
    public KeyPair generateKEMKeyPair() throws Exception {
        return KeyPairGenerator
            .getInstance(kemAlg, provider)
            .generateKeyPair();
    }
    public Signature getSigner(PrivateKey key) throws Exception {
        Signature sig = Signature.getInstance(signingAlg, provider);
        sig.initSign(key,
SecureRandom.getInstance(«NativePRNGNonBlocking»));
        return sig;
    }
    public Signature getVerifier(PublicKey key) throws Exception {
        Signature sig = Signature.getInstance(signingAlg, provider);
        sig.initVerify(key);
        return sig;
    }
}
application.yml for different environments:
# Current environment — classical algorithms (Phase 2)
crypto:
signing-algorithm: SHA256withECDSA
kem-algorithm: ECDH
provider: SunEC
# Transitional environment — hybrid mode (Phase 3)
spring:
config:
activate:
on-profile: hybrid
crypto:
    signing-algorithm: ML-DSA-65
    kem-algorithm: ML-KEM-768
    provider: BC
# Post-quantum environment (Phase 4)
spring:
    config:
        activate:
            on-profile: pqc
crypto:
    signing-algorithm: ML-DSA-87
    kem-algorithm: ML-KEM-1024
    provider: BC
```

Phase 3. Hybrid mode. Hybrid mode involves the concurrent use of classical and post-quantum algorithms. This is the longest phase: it begins now and will conclude once PQC-capable clients have been widely deployed. The hybrid approach is recommended by the NCSC [17] and aligns with current IETF standardization efforts.

For signing, a dual-signature scheme is implemented: ECDSA for compatibility with existing clients and ML-DSA for post-quantum resilience. Verification follows a policy requiring both signatures to be valid, ensuring that the system cannot be compromised through either algorithm individually.

For key exchange, a hybrid scheme combining X25519 and ML-KEM-768 is the recommended approach and will be standardized in JDK 27 via JEP 527. Before the release of JDK 27, this scheme can be implemented manually.

Phase 4. Full migration. Full migration, i.e., the complete retirement of classical algorithms, becomes feasible once three conditions are met: all clients have been updated and support PQC; PQC TLS has been standardized and is supported in the JDK (expected in JDK 27); and the operational infrastructure, including HSMs, PKI, and certificate management systems, has been updated to handle PQC keys.

At this stage, the CryptoProvider configuration is switched to the PQC profile, hybrid signatures are replaced with standalone ML-DSA signatures, and TLS is configured to use only post-quantum cipher suites.

Conclusions. This paper systematizes practical approaches to the integration of post-quantum cryptographic algorithms ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205) into Java applications, while accounting for the specifics of the JCA/JCE platform and multiple JDK versions.

The results show that the Java platform provides several levels of PQC support depending on the JDK version. In JDK 17, the BouncyCastle library remains the only practical option, using its low-level API for ML-KEM and its JCA provider for signature algorithms. JDK 21 introduced the standard class javax.crypto.KEM (JEP 452), which unifies the handling of key encapsulation mechanisms. JDK 25 added native support for ML-KEM and ML-DSA through the SunJCE and SUN providers, respectively; however, SLH-DSA remains available only via BouncyCastle across all JDK versions. This reflects a mismatch between the completion of NIST standardization and the integration of algorithms into the platform.

The results further show that the JCA provider architecture naturally supports the concept of crypto agility: externalizing algorithm names into Spring Boot application configuration allows cryptographic primitives to be changed without recompiling the codebase.

Directions for future research include assessing the performance impact of PQC algorithms on Spring Boot applications under realistic workloads; analyzing PQC support in TLS 1.3 following the release of JDK 27 with JEP 527; and developing methods for automated discovery of cryptographic dependencies in existing Java applications to support migration planning.

Bibliography:

1. Prokopovych-Tkachenko D. I., Khrushkov B. S., Derkach Y. O. Post-quantum threats to information security: challenges at the global and national levels. *Systems and Technologies*, 2025, 69(1). P. 118–123. https://doi.org/10.32782/2521-6643-2025-1-69.14

2. Prokopovych-Tkachenko D. I. Emergent-adaptive method of assessing the impact of the post-quantum environment on the information security of the state. *Systems and Technologies*, 2024, 68(2). P. 86–94. https://doi.org/10.32782/2521-6643-2024-2-68.10

3. Federal Office for Information Security. Status of Quantum Computer Development, V2.2. 2025. URL: https://www.bsi.bund.de/dok/study_status_quantum_computer

4. NIST. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standard. National Institute of Standards and Technology. Gaithersburg, MD. 2024. https://doi.org/10.6028/NIST.FIPS.203.

5. NIST. FIPS 204: Module-Lattice-Based Digital Signature Standard. Federal Information Processing Standard. National Institute of Standards and Technology. Gaithersburg, MD. 2024. https://doi.org/10.6028/NIST.FIPS.204

6. NIST. FIPS 205: Stateless Hash-Based Digital Signature Standard. Federal Information Processing Standard. National Institute of Standards and Technology. Gaithersburg, MD. 2024. https://doi.org/10.6028/NIST.FIPS.205

7. Nita S. L., Mihailescu M. I. JDK 21: New Features. In Cryptography and Cryptanalysis in Java: Creating and Programming Advanced Algorithms with Java SE 21 LTS and Jakarta EE 11 (pp. 19–37). Berkeley, CA: Apress. 2024. https://doi.org/10.1007/979-8-8688-0441-0_2

8. Mosca M., Piani M., Neill B. Quantum Threat Timeline Research Report 2024. Global Risk Institute. December 2024. URL: https://www.evolutionq.com/publications/quantum-threat-timeline-research-report-2024

9. Campbell R. Enterprise Migration to Post-Quantum Cryptography: Timeline Analysis and Strategic Frameworks. *Computers*. 2026; 15(1):9. https://doi.org/10.3390/computers15010009

10. NIST. (2025). Considerations for Achieving Cryptographic Agility: Strategies and Practices. NIST CSWP 39. https://doi.org/10.6028/NIST.CSWP.39

11. Marchesi L., Marchesi M., Tonelli R. Reviewing Crypto-Agility and Quantum Resistance in the Light of Agile Practices. Agile Processes in Software Engineering and Extreme Programming – Workshops. XP XP 2022 2023. Lecture Notes in Business Information Processing, vol 489. Springer, Cham, 2024. https://doi.org/10.1007/978-3-031-48550-3_21

12. Cho J., Lee C., Kim E., Lee J., Cho B. Software-Defined Cryptography: A Design Feature of Cryptographic Agility. Cryptology ePrint Archive, Paper 2024/518, 2024. URL: https://eprint.iacr.org/2024/518

13. OpenJDK. JEP 452: Key Encapsulation Mechanism API. 2023. URL: https://openjdk.org/jeps/452

14. OpenJDK. JEP 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism. 2024. URL: https://openjdk.org/jeps/496

15. OpenJDK. JEP 527: Post-Quantum Hybrid Key Exchange for TLS 1.3. 2026. URL: https://openjdk.org/jeps/527

16. José A. Montenegro Ruben Rios, & Javier Lopez-Cerezo. A performance evaluation framework for post-quantum TLS. Future Generation Computer Systems, Volume 175, 2026. https://doi.org/10.1016/j.future.2025.108062

17. National Cyber Security Centre. Timelines for Migration to Post-Quantum Cryptography. NCSC Guidance. 2025. URL: https://www.ncsc.gov.uk/guidance/pqc-migration-timelines

18. Grover L. K. A fast quantum mechanical algorithm for database search. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, 1996, pp. 212–219. https://doi.org/10.1145/237814.237866

19. Grassl, M., Langenberg, B., Roetteler, M., Steinwandt, R. Applying Grover's algorithm to AES: quantum resource estimates. In International Workshop on Post-Quantum Cryptography, 2016, pp. 29–43. Cham: Springer International Publishing. https://doi.org/10.48550/arXiv.1512.04965

20. Jones M., Bradley J., Sakimura N. JSON Web Signature (JWS), RFC 7515, May 2015. https://doi.org/10.17487/RFC7515

21. Jones M. JSON Web Algorithms (JWA), RFC 7518, May 2015. https://doi.org/10.17487/RFC7518

22. Richer J. OAuth 2.0 Token Introspection, RFC 7662, October 2015. https://doi.org/10.17487/RFC7662

References:

1. 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

2. Prokopovych-Tkachenko, D. I. (2024). Emergent-adaptive method of assessing the impact of the post-quantum environment on the information security of the state. Systems and Technologies, 68(2), 86–94. https://doi.org/10.32782/2521-6643-2024-2-68.10

3. Federal Office for Information Security (2025). Status of Quantum Computer Development, V2.2. Retrieved from: https://www.bsi.bund.de/dok/study_status_quantum_computer

4. NIST (2024). FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standard. National Institute of Standards and Technology. Gaithersburg, MD. https://doi.org/10.6028/NIST.FIPS.203.

5. NIST (2024). FIPS 204: Module-Lattice-Based Digital Signature Standard. Federal Information Processing Standard. National Institute of Standards and Technology. Gaithersburg, MD. https://doi.org/10.6028/NIST.FIPS.204.

6. NIST (2024). FIPS 205: Stateless Hash-Based Digital Signature Standard. Federal Information Processing Standard. National Institute of Standards and Technology. Gaithersburg, MD. https://doi.org/10.6028/NIST.FIPS.205

7. Nita, S. L., & Mihailescu, M. I. (2024). JDK 21: New Features. In Cryptography and Cryptanalysis in Java: Creating and Programming Advanced Algorithms with Java SE 21 LTS and Jakarta EE 11 (pp. 19-37). Berkeley, CA: Apress. https://doi.org/10.1007/979-8-8688-0441-0_2

8. Mosca, M., Piani, M., Neill B. (2024, December). Quantum Threat Timeline Research Report 2024. Global Risk Institute. Retrieved from: https://www.evolutionq.com/publications/quantum-threat-timeline-research-report-2024

9. Campbell, R. (2026). Enterprise Migration to Post-Quantum Cryptography: Timeline Analysis and Strategic Frameworks. Computers. 15(1):9. https://doi.org/10.3390/computers15010009

10. NIST. Considerations for Achieving Cryptographic Agility: Strategies and Practices. NIST CSWP 39. 2025. https://doi.org/10.6028/NIST.CSWP.39

11. Marchesi, L., Marchesi, M., & Tonelli, R. (2024). Reviewing Crypto-Agility and Quantum Resistance in the Light of Agile Practices. Agile Processes in Software Engineering and Extreme Programming – Workshops. XP XP 2022 2023. Lecture Notes in Business Information Processing, vol 489. Springer, Cham. https://doi.org/10.1007/978-3-031-48550-3_21

12. Cho, J., Lee, C., Kim, E., Lee, J., & Cho, B. (2024). Software-Defined Cryptography: A Design Feature of Cryptographic Agility. Cryptology ePrint Archive, Paper 2024/518. Retrieved from: https://eprint.iacr.org/2024/518

13. OpenJDK. (2023). JEP 452: Key Encapsulation Mechanism API. Retrieved from: https://openjdk.org/jeps/452
14. OpenJDK. (2024). JEP 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism. Retrieved from: https://openjdk.org/jeps/496
15. OpenJDK. (2026). JEP 527: Post-Quantum Hybrid Key Exchange for TLS 1.3. Retrieved from: https://openjdk.org/jeps/527
16. José A. Montenegro, Ruben Rios, & Javier Lopez-Cerezo (2026). A performance evaluation framework for post-quantum TLS. Future Generation Computer Systems, Volume 175. https://doi.org/10.1016/j.future.2025.108062
17. National Cyber Security Centre. (2025). Timelines for Migration to Post-Quantum Cryptography. NCSC Guidance. Retrieved from: https://www.ncsc.gov.uk/guidance/pqc-migration-timelines
18. Grover, L. K. (1996, July). A fast quantum mechanical algorithm for database search. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (pp. 212–219). https://doi.org/10.1145/237814.237866
19. Grassl, M., Langenberg, B., Roetteler, M., & Steinwandt, R. (2016, February). Applying Grover's algorithm to AES: quantum resource estimates. In International Workshop on Post-Quantum Cryptography (pp. 29–43). Cham: Springer International Publishing. https://doi.org/10.48550/arXiv.1512.04965
20. Jones, M., Bradley, J., & Sakimura, N. (May 2015). JSON Web Signature (JWS), RFC 7515. https://doi.org/10.17487/RFC7515
21. Jones, M. (May 2015). JSON Web Algorithms (JWA), RFC 7518. https://doi.org/10.17487/RFC7518
22. Richer, J. (October 2015). OAuth 2.0 Token Introspection, RFC 7662. https://doi.org/10.17487/RFC7662

Дата першого надходження статті до видання: 24.03.2026
Дата прийняття статті до друку після рецензування: 17.04.2026
Дата публікації (оприлюднення) статті: 30.05.2026