QuantaCipher: Zero-Trust Post-Quantum Infrastructure for the Enterprise

A Technical Whitepaper

QuantaLabs Research Team

August 2026

Abstract

The rapid advancement of quantum computing threatens the foundations of modern cryptography. Nation-state adversaries are currently engaged in "Harvest Now, Decrypt Later" (HNDL) attacks, intercepting and storing encrypted data to be decrypted once Cryptanalytically Relevant Quantum Computers (CRQCs) become available. This whitepaper introduces QuantaCipher, a zero-trust, developer-native post-quantum cryptographic platform. By combining NIST's FIPS 203 ML-KEM-1024 standard with a hybrid KEM/DEM architecture running entirely within a local WebAssembly (WASM) and native Rust execution environment, QuantaCipher guarantees that plaintext never leaves the host runtime. We detail the mathematical underpinnings, system architecture, operational modes, and security proofs that make QuantaCipher the definitive solution for immediate enterprise compliance and data sovereignty in the quantum era.

Contents

1 Introduction

1.1 The Quantum Threat Landscape

For decades, secure digital communication has relied on public-key cryptographic systems such as RSA, Diffie-Hellman (DH), and Elliptic Curve Cryptography (ECC). The security of these algorithms rests on the computational difficulty of specific mathematical problems—namely, integer factorization and the discrete logarithm problem.

In 1994, Peter Shor demonstrated that a sufficiently powerful quantum computer could solve these problems in polynomial time. While theoretical for decades, sustained engineering breakthroughs have moved Cryptanalytically Relevant Quantum Computers (CRQCs) from a distant possibility to an imminent reality.

1.2 Harvest Now, Decrypt Later (HNDL)

The primary threat vector facing enterprises today is not immediate decryption, but the exfiltration and prolonged storage of encrypted data. Nation-states and advanced persistent threats (APTs) are actively scraping high-value encrypted traffic—such as financial records, health data, and intellectual property—under the premise that they will decrypt it retroactively. This strategy is known as "Harvest Now, Decrypt Later" (HNDL). Consequently, data encrypted today using classical asymmetric cryptography must be considered compromised if its required secrecy lifespan extends beyond the horizon of CRQC realization.

1.3 The NIST Standardization Process

In response to this threat, the National Institute of Standards and Technology (NIST) initiated a multi-year process to standardize Post-Quantum Cryptographic (PQC) algorithms. In August 2024, NIST published the final FIPS 203 standard, formalizing the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM), derived from the CRYSTALS-Kyber submission.

QuantaCipher fundamentally relies on ML-KEM at parameter set 1024 (Kyber-1024), providing Security Category 5—the highest possible margin of security, roughly equivalent to breaking AES-256.

2 Mathematical Foundations

2.1 Module Learning With Errors (MLWE)

The security of ML-KEM is based on the Module Learning With Errors (MLWE) problem. Let R_q be the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ where $n = 256$ and $q = 3329$.

The MLWE problem asks an adversary to distinguish between uniformly random samples from $R_q^k \times R_q$ and samples of the form $(A, As + e)$, where:

- $A \in R_q^{k \times k}$ is a randomly chosen matrix.
- $s \in R_q^k$ is a secret vector drawn from a centered binomial distribution.
- $e \in R_q^k$ is a small error vector also drawn from a centered binomial distribution.

For Kyber-1024, the module dimension k is set to 4.

2.2 The KEM / DEM Paradigm

A Key Encapsulation Mechanism (KEM) is used to establish a shared symmetric key, while a Data Encapsulation Mechanism (DEM) uses that symmetric key to encrypt the actual payload. This is the hybrid architecture employed by QuantaCipher.

2.2.1 Encapsulation

The encapsulation function takes a public key pk and produces a shared secret K and a ciphertext c_{KEM}.

$$(c_{KEM}, K) \leftarrow \text{Encapsulate}(pk)$$

2.2.2 Decapsulation

The decapsulation function takes a private key sk and the ciphertext c_{KEM} to recover K.

$$K \leftarrow \text{Decapsulate}(sk, c_{KEM})$$

For the DEM, QuantaCipher strictly utilizes AES-256-GCM, a universally standardized and highly optimized Authenticated Encryption with Associated Data (AEAD) cipher.

3 Zero-Trust Architecture

3.1 The Design Philosophy

Most cloud-based encryption services operate on a high-trust model: plaintext is transmitted to a centralized KMS (Key Management Service) or HSM (Hardware Security Module) for encryption. This model breaks down under sophisticated network breaches and places immense liability on the service provider.

QuantaCipher is engineered on a Zero-Trust architecture. Cryptographic operations are executed 100% locally within the client's environment. The QuantaCipher Gateway API acts solely as a logging, validation, and billing layer for ciphertexts.

3.2 Runtime Environment: Rust and WebAssembly

To ensure identical, memory-safe cryptographic execution across all platforms, the QuantaCipher engine ('quantacipher-core') is written entirely in pure Rust.

- **Web Environments:** The Rust core is compiled to WebAssembly (WASM) via 'wasm-bindgen', exposing a synchronous, highly optimized API to Node.js and browser JavaScript environments.
- **Native Environments:** For Python, the Rust core is compiled to native C-extensions using PyO3 and Maturin, completely bypassing the Global Interpreter Lock (GIL) and delivering native hardware performance.

By keeping the execution inside a WASM sandbox or a memory-safe Rust boundary, QuantaCipher neutralizes whole classes of memory corruption vulnerabilities and side-channel attacks typically associated with C/C++ cryptographic implementations.

3.3 Payload Isolation

The QuantaCipher Gateway never possesses the private keys required for decapsulation. The generated payloads follow strict formatting rules:

```
QZ_SECURE_V1 : <B64 Kyber CT> : <B64 AES Nonce> : <B64 AES CT>
```

The gateway validates the base64 structure and byte lengths to prevent malformed injections, but mathematically cannot interact with the underlying data.

4 Operational Modes

QuantaCipher provides two distinct operational modes designed to solve different enterprise use cases.

4.1 Vault Mode (Permanent Sealing)

Vault Mode is designed for tamper-proof audit logs, compliance records, and sensitive health data (HIPAA) where the creator must prove the data existed at a specific time, but never needs to read it back directly without external audit mechanisms.

4.1.1 Protocol Flow

1. The client invokes ‘sdk.encryptVault(data)‘.
2. The local WASM engine generates a random ephemeral Kyber-1024 keypair (pk_E, sk_E).
3. The client encapsulates a shared secret K using pk_E.
4. The data is encrypted via AES-256-GCM using K.
5. The ephemeral private key sk_E is **deterministically and permanently zeroized** from memory.
6. The resulting ‘$QZ_V AULT_V 1‘payloadisreturned.$

Because the private key is dropped before the function returns, the resulting ciphertext is permanently sealed. It serves as cryptographic proof of existence when combined with the QuantaCipher Gateway's receipting system.

4.2 Secure Mode (End-to-End Encryption)

Secure Mode provides traditional E2EE for environments where confidential data must be transmitted and later decrypted by the intended recipient.

4.2.1 Protocol Flow

1. The recipient generates a persistent Kyber-1024 keypair (pk, sk) and stores sk securely.
2. The sender receives pk and invokes ‘sdk.encryptSecure(data, pk)‘.
3. The engine encapsulates a shared secret K and generates the ciphertext c_{KEM}.

4. Data is encrypted with K via AES-256-GCM.

5. The sender transmits the ‘QZ$_S ECURE_V$1‘*payloadtotherecipient*.

5. The recipient invokes ‘sdk.decryptSecure(payload, sk)‘, recovering K and subsequently the plaintext.

5 Gateway Infrastructure and Receipts

5.1 The Cryptographic Receipt Engine

When a payload is transmitted to the QuantaCipher Gateway, the system performs validation and issues an immutable Cryptographic Receipt.

The receipt includes:

- Accurate timestamping of the event.
- A cryptographic hash of the ciphertext.
- Byte count and algorithmic schema identifiers.

This receipting system allows enterprises to prove to auditors (SOC2, ISO 27001, HIPAA) that post-quantum encryption was applied to specific datasets at specific times, completely independent of the plaintext contents.

5.2 Global Distribution and DDoS Mitigation

The QuantaCipher Gateway infrastructure is globally distributed using anycast routing. All ingress traffic is actively scrubbed for Layer 3, 4, and 7 DDoS attacks. The API enforces strict TLS 1.3 requirements, dropping connections from legacy clients to prevent protocol downgrade attacks.

6 Implementation Guide

The following sections detail the developer experience, highlighting the simplicity of integrating post-quantum cryptography into existing codebases.

6.1 TypeScript / Node.js SDK

The JavaScript SDK wraps the WASM engine, providing a seamless asynchronous API.

```
import { QuantaCipher } from 'quantacipher-sdk';

// Initialize the engine
const qc = new QuantaCipher({ apiKey: 'qz_live_xxxx' });

// 1. Generate Keypair
const keys = qc.generateKeypair();
console.log("Public Key:", keys.publicKey);

// 2. Encrypt Data
const confidentialData = "Q3 2026 Financial Projections: $14M ARR";
const ciphertext = qc.encryptSecure(confidentialData, keys.publicKey);

// 3. Decrypt Data
const plaintext = qc.decryptSecure(ciphertext, keys.privateKey);
console.assert(plaintext === confidentialData);
```

Listing 1: TypeScript E2EE Implementation

6.2 Python SDK

The Python SDK utilizes PyO3 bindings directly to the Rust core, making it highly suitable for data science and backend workflows.

```
from quantacipher import QuantaCipher
import json

sdk = QuantaCipher(api_key="qz_live_xxxx")

# Prepare compliance log
audit_log = json.dumps({
    "user_id": "U-9921",
    "action": "EHR_ACCESS",
    "timestamp": 1786798000
})

```

```
# Permanently seal the log
sealed_payload = sdk.encrypt_vault(audit_log)

print(f"Stored securely: {sealed_payload}")
```

Listing 2: Python Vault Mode Implementation

7 Performance and Benchmarks

Lattice-based cryptography is inherently efficient. Kyber was selected by NIST not just for its security, but for its outstanding performance profile compared to older systems like RSA.

7.1 Key Sizes and Ciphertext Expansion

The tradeoff for post-quantum security is larger key and ciphertext sizes. For ML-KEM-1024:

- **Public Key:** 1,568 Bytes
- **Private Key:** 3,168 Bytes
- **KEM Ciphertext:** 1,568 Bytes

While larger than ECC (e.g., Curve25519 uses 32-byte keys), these sizes are easily accommodated by modern network infrastructure and are significantly smaller than classical McEliece constructions.

7.2 Execution Latency

Internal benchmarks of ‘quantacipher-core‘ on a standard x86_64 CPU (Intel i7-12700K):

- **Keypair Generation:** $\sim$ 0.05 ms
- **Encapsulation:** $\sim$ 0.06 ms
- **Decapsulation:** $\sim$ 0.07 ms

When compiled to WebAssembly, execution overhead introduces roughly a 20-30% penalty, meaning full encryption cycles still complete in sub-millisecond timeframes inside standard web browsers.

8 Security Analysis

8.1 Downgrade Prevention

QuantaCipher does not support classical algorithm fallback. By strictly mandating ML-KEM-1024, the SDK prevents adversarial downgrade attacks where a man-in-the-middle forces the system to negotiate a vulnerable classical algorithm like RSA-1024.

8.2 Side-Channel Resistance

The underlying ‘pqc$_k$*yber*‘*crateoperatesinconstanttimeforallsecret*−*dependentoperations.Thisinclud*
vectormultiplications, *polynomialarithmetic*, *andsamplingfromthebinomialdistribution.Byavoidingd*
dependentbranchingandmemoryaccesspatterns, *QuantaCiphermitigatestimingandcache*−
basedside − *channelattacks*.

8.3 Random Number Generation

The security of any cryptographic system is bounded by the quality of its entropy source. ‘quantacipher-core‘ utilizes the OS-level Cryptographically Secure Pseudorandom Number Generator (CSPRNG):

- **Linux/macOS:** ‘/dev/urandom‘ and ‘getrandom‘ syscalls.
- **Windows:** ‘BCryptGenRandom‘.
- **WebAssembly:** ‘window.crypto.getRandomValues‘.

9 Conclusion

The era of classical public-key cryptography is drawing to a close. The threat of "Harvest Now, Decrypt Later" necessitates immediate action from enterprises handling sensitive data.

QuantaCipher provides a turn-key, zero-trust infrastructure that abstracts the immense complexity of post-quantum cryptography into a developer-friendly API. By combining mathematically sound NIST standards (ML-KEM-1024), robust memory-safe engineering (Rust/WASM), and a localized execution model, QuantaCipher ensures that enterprise data remains secure against both classical adversaries today and quantum adversaries tomorrow.

QuantaLabs

Securing the Quantum Future.

`https://www.quantacipher.com`