Post-Quantum OIDC with KEMTLS

Technical Documentation

Generated: February 08, 2026

NIST Post-Quantum Cryptography Standards

Executive Summary

This document provides comprehensive technical documentation for the Post-Quantum OIDC with KEMTLS implementation. The project implements OpenID Connect (OIDC) authentication using NIST-standardized post-quantum cryptographic algorithms.

Key Features:

- Post-Quantum Key Encapsulation: Kyber (ML-KEM) for secure key exchange
- Post-Quantum Digital Signatures: ML-DSA (Dilithium) and Falcon for authentication
- KEMTLS Protocol: TLS variant using KEMs for authentication
- OIDC Integration: Complete OAuth 2.0/OIDC server and client implementation
- Production-Ready: Comprehensive testing, benchmarking, and documentation

Performance Highlights (see BenchmarkResults.pdf for details):

Operation	Time
KEM Operations	0.023-0.033 ms
Signature Operations	0.027-0.181 ms
KEMTLS Handshake	0.041 ms
End-to-End OIDC Flow	0.240 ms

System Architecture

The system is architected in modular layers, each handling specific concerns. The architecture follows a clean separation of concerns with well-defined interfaces between layers.

Architecture Layers:

OIDC Layer: Application-level authentication and authorization

JWT Layer: Token creation and verification with PQ signatures

KEMTLS Layer: Secure transport with PQ key encapsulation

PQ Crypto Layer: Core cryptographic primitives (KEM, signatures)

liboqs: Native library with NIST PQ algorithm implementations

Core Components

1. Post-Quantum Cryptography Layer (src/pq_crypto/)

Provides high-level Python interfaces to post-quantum cryptographic primitives. Includes KEM implementation using Kyber (ML-KEM), digital signatures using ML-DSA (Dilithium) and Falcon, and cryptographic utilities for key derivation and management.

Module	Purpose	Algorithms
kem.py	Key Encapsulation	Kyber512, Kyber768, Kyber1024
signature.py	Digital Signatures	ML-DSA-44/65/87, Falcon-512/1024
utils.py	Crypto Utilities	HKDF key derivation

2. KEMTLS Layer (src/kemtls/)

Implements the KEMTLS protocol for authenticated key exchange using Key Encapsulation Mechanisms instead of traditional signature-based authentication. Provides certificate-based authentication using PQ keys with perfect forward secrecy via ephemeral KEMs.

3. JWT Layer (src/oidc/pq_jwt.py)

Creates and verifies JSON Web Tokens with post-quantum signatures. Implements native PQ signature integration in JWT format with algorithm negotiation between client and server. Tokens are backward-compatible in structure and size-optimized for network transmission.

Algorithm	Token Size	Notes
ML-DSA-44	~3.5 KB	Fast, reasonable size
ML-DSA-65	~4.7 KB	Higher security
Falcon-512	~1.2 KB	Smallest (66% reduction!)

4. OIDC Layer (src/oidc/)

Full OAuth 2.0 / OpenID Connect implementation with authorization code flow. Implements standard OIDC endpoints (authorize, token, userinfo) with PQ-signed ID tokens. Supports client registration, token validation, and user information retrieval.

Security Analysis

Cryptographic Strength

All algorithms used are NIST-approved post-quantum standards providing quantum resistance equivalent to or exceeding AES-128 security level.

Algorithm	Security Level	Key Size	Signature/CT Size
Kyber512	NIST Level 1	800 B	768 B
Kyber768	NIST Level 3	1184 B	1088 B
ML-DSA-44	NIST Level 2	1312 B	~2420 B
ML-DSA-65	NIST Level 3	1952 B	~3309 B
Falcon-512	NIST Level 1	897 B	~650 B

Threat Model

Protected Against:

✓ Quantum computer attacks (Shor's algorithm ineffective)

✓ Man-in-the-middle attacks (KEMTLS authentication)

✓ Replay attacks (nonces, timestamps in tokens)

✓ Token forgery (PQ signatures)

✓ Eavesdropping (encrypted channels)

Out of Scope:

- Side-channel attacks (implementation-dependent)
- Physical security
- Social engineering
- Endpoint compromise

Implementation Details

File Structure

```
PQC/
■■■ src/
■   ■■■ pq_crypto/          # PQ cryptography primitives
■   ■   ■■■ kem.py          # Kyber KEM implementation
■   ■   ■■■ signature.py    # ML-DSA & Falcon signatures
■   ■   ■■■ utils.py        # Crypto utilities
■   ■
■   ■■■ kemtls/             # KEMTLS protocol
■   ■   ■■■ protocol.py     # Core protocol logic
■   ■   ■■■ server.py       # Server-side KEMTLS
■   ■   ■■■ client.py       # Client-side KEMTLS
■   ■   ■■■ certificates.py # PQ certificates
■   ■
■   ■■■ oidc/               # OpenID Connect
■   ■   ■■■ server.py       # OIDC Provider (IdP)
■   ■   ■■■ client.py       # OIDC Relying Party
■   ■   ■■■ pq_jwt.py       # PQ-signed JWT tokens
■   ■
■   ■■■ benchmarks/         # Performance benchmarking
■       ■■■ run_benchmarks.py
■       ■■■ generate_pdf_report.py
■
■■■ tests/                  # Comprehensive test suite
■■■ benchmark_results/      # Benchmark data and reports
■■■ requirements.txt        # Python dependencies
■■■ README.md              # Project documentation
```

API Reference

KEM Module (pq_crypto/kem.py)

KyberKEM class provides key encapsulation mechanisms. Main methods: generate_keypair() returns (public_key, secret_key), encapsulate(public_key) returns (ciphertext, shared_secret), and decapsulate(ciphertext, secret_key) returns shared_secret.

Signature Module (pq_crypto/signature.py)

DilithiumSigner and FalconSigner classes provide digital signatures. Main methods: generate_keypair() returns (public_key, secret_key), sign(message, secret_key) returns signature, and verify(message, signature, public_key) returns boolean.

JWT Module (oidc/pq_jwt.py)

create_id_token(claims, algorithm, secret_key) creates signed ID token. verify_id_token(token, public_key, algorithm) verifies and decodes ID token, raising ValueError if signature invalid or token expired.

Testing & Quality Assurance

The project includes comprehensive tests covering all components with unit tests for individual functions/classes, integration tests for component interaction, and end-to-end tests for full protocol flows.

Test Coverage:

✓ KEM operations (key generation, encapsulation, decapsulation)

✓ Signature operations (all algorithms)

✓ KEMTLS handshake (complete protocol flow)

✓ JWT creation and verification

✓ OIDC authorization code flow

✓ Error handling and edge cases

Performance Analysis

Comprehensive benchmarking shows all operations complete in sub-millisecond time (except Falcon key generation which takes 5-16ms). See BenchmarkResults.pdf for detailed performance graphs, tables, and analysis.

Use Case	Recommended Algorithm	Reason
General Use	ML-DSA-44	Fast (~0.076ms), reasonable size
Bandwidth-Limited	Falcon-512	Smallest tokens (~1.2KB)
Maximum Security	ML-DSA-87	Highest security level
IoT/Embedded	Kyber512	Fastest KEM operations

Deployment Guide

Installation Steps:

```
# Install system dependencies
sudo apt-get update
sudo apt-get install -y build-essential cmake git python3-pip

# Install liboqs
git clone https://github.com/open-quantum-safe/liboqs.git
cd liboqs && mkdir build && cd build
cmake -GNinja -DCMAKE_INSTALL_PREFIX=/usr/local ..
ninja && sudo ninja install

# Setup Python environment
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Production Considerations:

• **Security:** Use TLS 1.3, HSM for key storage, regular key rotation

• **Performance:** Cache public keys, use connection pooling, async I/O

• **Scalability:** Stateless design, horizontal scaling, distributed cache

• **Monitoring:** Log all auth attempts, set up alerting, rate limiting

References

NIST Standards:

• FIPS 203: Module-Lattice-Based KEM (ML-KEM / Kyber)

• FIPS 204: Module-Lattice-Based Digital Signature (ML-DSA / Dilithium)

• FIPS 205: Stateless Hash-Based Digital Signature (SLH-DSA / SPHINCS+)

Libraries:

• liboqs 0.15.0 - Open Quantum Safe cryptographic library

• liboqs-python 0.14.1 - Python bindings for liboqs

Protocols:

• KEMTLS: Post-Quantum TLS Without Handshake Signatures (Schwabe et al., 2020)

• OpenID Connect Core 1.0

• OAuth 2.0 Authorization Framework (RFC 6749)

• JSON Web Token (RFC 7519)

Document Version: 1.0 | Last Updated: February 08, 2026
For more details, see BenchmarkResults.pdf and source code documentation.