QUANTUM-SAFE CRYPTOGRAPHY LAB SERIES

Lab 1: Kyber Key Exchange with Qiskit
Understanding Post-Quantum Encryption

PART 1: LAB PREPARATION & BACKGROUND

Student Name: ______________________
Date: ______________________
Quantum Level: Intermediate-Advanced
Prerequisites: Basic Python, Quantum Superposition, Public-Key Cryptography

---

PRE-LAB QUESTIONS: CRYPTOGRAPHY FUNDAMENTALS

Instructions: Before starting the lab, answer these foundational questions:

1. Public-Key Cryptography: How does RSA encryption work? Why is it vulnerable to quantum computers?

   ____

   ____

2. Lattice-Based Cryptography: What mathematical problem makes lattice-based encryption quantum-resistant?

   ____

   ____

3. Kyber Specification: Kyber is a CRYSTALS-Kyber MLWE-based KEM. What does MLWE stand for and why is it hard for both classical and quantum computers?

   ____

   ____

PART 2: THEORETICAL BACKGROUND

Why Kyber? The NIST Standard for Post-Quantum Cryptography

In 2022, the National Institute of Standards and Technology (NIST) selected CRYSTALS-Kyber as the standard for post-quantum public-key encryption. This marked a historic shift from RSA/ECC to quantum-resistant algorithms.

The Quantum Threat Timeline:

- 1994: Peter Shor's algorithm shows quantum computers can break RSA
- 2016: NIST begins post-quantum cryptography standardization
- 2022: Kyber selected as primary KEM (Key Encapsulation Mechanism)
- 2030+: Quantum computers may break current encryption

How Kyber Works:
Kyber is based on the Module Learning With Errors (MLWE) problem. In simple terms:

1. Key Generation: Create public key (matrix A, vector t) and private key (secret s)
2. Encapsulation: Encrypt a symmetric key using public key
3. Decapsulation: Decrypt using private key

The security relies on the difficulty of solving noisy linear equations over lattices—a problem believed to be hard for both classical and quantum computers.

Qiskit's Role:
While Qiskit is primarily for quantum computing, it provides quantum-safe cryptography tools to:

1. Demonstrate why current encryption is vulnerable
2. Implement and test post-quantum alternatives
3. Simulate quantum attacks on classical crypto

PART 3: LAB SETUP & INSTALLATION

Step 1: Environment Setup

```python
# Run these commands in your terminal/Colab first:
# !pip install qiskit qiskit-ibm-runtime
# !pip install pqcrypto  # For post-quantum cryptography

# !pip install numpy matplotlib cryptography
```

Step 2: Import Required Libraries

```python
import numpy as np
import hashlib
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

# Qiskit imports
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram
from qiskit.algorithms import Shor
from qiskit.utils import QuantumInstance
from qiskit.algorithms import Grover

from qiskit.circuit.library import PhaseOracle
```

Step 3: Kyber Simulation Functions

(Since full Kyber requires extensive implementation, we'll simulate key components)

```python
class SimulatedKyber:
    """
    Simplified Kyber simulation for educational purposes
```

```
    Based on CRYSTALS-Kyber specification
    """

    def __init__(self, dimension=256, modulus=3329):
        self.n = dimension  # Lattice dimension
        self.q = modulus    # Modulus
        self.k = 2          # Module rank
        self.eta = 2        # Error distribution parameter

    def generate_keys(self):
        """Generate simulated Kyber keys"""
        # In real Kyber: A <- uniform, s,e <- centered binomial
        # Simplified for education:
        np.random.seed(42)  # For reproducibility

        # Public key components
        self.A = np.random.randint(0, self.q, size=(self.k, self.k, self.n))
        self.s = np.random.randint(-self.eta, self.eta+1, size=(self.k,
self.n))
        self.e = np.random.randint(-self.eta, self.eta+1, size=(self.k,
self.n))

        # Compute t = A·s + e (mod q)
        self.t = np.zeros((self.k, self.n), dtype=int)
        for i in range(self.k):
            for j in range(self.k):
                self.t[i] = (self.t[i] + self.polymul(self.A[i,j], self.s[j]))
% self.q
            self.t[i] = (self.t[i] + self.e[i]) % self.q

        public_key = (self.A, self.t)
        private_key = self.s

        return public_key, private_key

    def encapsulate(self, public_key, seed=None):
        """Encapsulate a symmetric key"""
        A, t = public_key

        if seed is None:
            seed = np.random.bytes(32)
```

```python
        # Derive randomness from seed
        rng = np.random.RandomState(int.from_bytes(seed[:4], 'big'))

        # Generate random r, e1, e2
        r = rng.randint(-self.eta, self.eta+1, size=(self.k, self.n))
        e1 = rng.randint(-self.eta, self.eta+1, size=(self.k, self.n))
        e2 = rng.randint(-self.eta, self.eta+1, size=(self.n,))

        # Compute u = A^T·r + e1
        u = np.zeros((self.k, self.n), dtype=int)
        for i in range(self.k):
            for j in range(self.k):
                u[i] = (u[i] + self.polymul(self.A[j,i], r[j])) % self.q
            u[i] = (u[i] + e1[i]) % self.q

        # Compute v = t^T·r + e2 + encode(m)
        v = np.zeros(self.n, dtype=int)
        for i in range(self.k):
            v = (v + self.polymul(t[i], r[i])) % self.q
        v = (v + e2) % self.q

        # Generate random message m (256 bits)
        m = rng.randint(0, 2, size=256)

        # Encode m into polynomial
        m_poly = self.encode_message(m)
        v = (v + m_poly) % self.q

        # Derive shared secret from m
        shared_secret = self.derive_key(m)

        ciphertext = (u, v)

        return ciphertext, shared_secret, m

    def decapsulate(self, ciphertext, private_key):
        """Decapsulate the shared secret"""
        u, v = ciphertext
        s = private_key

        # Compute w = v - s^T·u
        w = v.copy()
```

```
        for i in range(self.k):
            w = (w - self.polymul(s[i], u[i])) % self.q

        # Decode message
        m_decoded = self.decode_message(w)

        # Derive shared secret
        shared_secret = self.derive_key(m_decoded)

        return shared_secret, m_decoded

    def polymul(self, a, b):
        """Polynomial multiplication in ring Z_q[x]/(x^n+1)"""
        n = len(a)
        result = np.zeros(n, dtype=int)

        for i in range(n):
            for j in range(n):
                idx = (i + j) % n
                sign = 1 if (i + j) < n else -1
                result[idx] = (result[idx] + sign * a[i] * b[j]) % self.q

        return result

    def encode_message(self, m_bits):
        """Encode 256-bit message into polynomial"""
        n = self.n
        q = self.q
        m_poly = np.zeros(n, dtype=int)

        # Simple encoding: map bits to 0 or q//2
        for i in range(min(256, n)):
            m_poly[i] = (m_bits[i] * (q // 2)) % q

        return m_poly

    def decode_message(self, poly):
        """Decode polynomial to 256-bit message"""
        q = self.q
        threshold = q // 4
        m_bits = np.zeros(256, dtype=int)
```

```
        for i in range(min(256, len(poly))):
            val = poly[i] if poly[i] <= q//2 else poly[i] - q
            if val > threshold or val < -threshold:
                m_bits[i] = 1
            else:
                m_bits[i] = 0

        return m_bits

    def derive_key(self, m_bits):
        """Derive symmetric key from message using SHA3-256"""
        m_bytes = bytes(int(''.join(map(str, m_bits[i:i+8])), 2)
                       for i in range(0, 256, 8))

        return hashlib.sha3_256(m_bytes).digest()
```

PART 4: LAB EXERCISES

Exercise 1: Key Generation and Exchange

```python
# Initialize Kyber
kyber = SimulatedKyber(dimension=128, modulus=3329)  # Smaller for speed

print("=== EXERCISE 1: KYBER KEY EXCHANGE ===")
print("\n1. Generating Kyber keys...")

# Generate key pair
public_key, private_key = kyber.generate_keys()
A, t = public_key
print(f"Public key generated: A shape {A.shape}, t shape {t.shape}")
print(f"Private key shape: {private_key.shape}")
print(f"Modulus q: {kyber.q}, Dimension n: {kyber.n}")

# Encapsulation (Alice's side)
print("\n2. Alice encapsulating shared secret...")
ciphertext, shared_secret_alice, message = kyber.encapsulate(public_key,
seed=b'alice_seed')
```

```
u, v = ciphertext
print(f"Ciphertext generated: u shape {u.shape}, v shape {v.shape}")
print(f"Shared secret (first 16 bytes): {shared_secret_alice[:16].hex()}")

# Decapsulation (Bob's side)
print("\n3. Bob decapsulating shared secret...")
shared_secret_bob, decoded_msg = kyber.decapsulate(ciphertext, private_key)
print(f"Decoded message matches original: {np.array_equal(decoded_msg,
message)}")
print(f"Shared secret (first 16 bytes): {shared_secret_bob[:16].hex()}")
print(f"Secrets match: {shared_secret_alice == shared_secret_bob}")

# Answer these questions:
print("\n=== QUESTIONS ===")
print("1. What are the two main components of Kyber's public key?")
print("2. Why does the ciphertext include both 'u' and 'v'?")

print("3. How does the error (e1, e2) contribute to security?")
```

Your Answers:

1. ______________________________
2. ______________________________
3. ______________________________

Exercise 2: Quantum Vulnerability Demonstration

```
python
print("\n=== EXERCISE 2: QUANTUM VULNERABILITY COMPARISON ===")

# Simulate RSA vulnerability to Shor's algorithm
def simulate_shor_attack(n_bits=8):
    """Demonstrate Shor's algorithm concept"""
    print(f"\nSimulating Shor's attack on {n_bits}-bit number...")

    # Create a quantum circuit to demonstrate period finding
    qc = QuantumCircuit(2*n_bits, n_bits)

    # Apply Hadamard to create superposition
    qc.h(range(n_bits))
```

```python
    # Simplified modular exponentiation (conceptual)
    # In real Shor: U|y⟩ = |a·y mod N⟩
    qc.barrier()

    # Inverse QFT
    for qubit in range(n_bits):
        for j in range(qubit):
            qc.cp(-np.pi/float(2**(qubit-j)), j, qubit)
        qc.h(qubit)

    qc.measure(range(n_bits), range(n_bits))

    # Execute
    backend = Aer.get_backend('qasm_simulator')
    result = execute(qc, backend, shots=1024).result()
    counts = result.get_counts()

    print(f"Quantum circuit created with {qc.num_qubits} qubits")
    print(f"Measurement results sample: {list(counts.keys())[:3]}")

    return qc, counts

# Compare with lattice problem
def demonstrate_lattice_problem():
    """Show why lattice problems are quantum-resistant"""
    print("\n\nLattice Problem: Shortest Vector Problem (SVP)")

    # Create a random lattice basis
    dimension = 3  # Small for visualization
    B = np.random.randint(-10, 10, size=(dimension, dimension))

    print(f"Lattice basis B (columns are basis vectors):")
    print(B)

    # The problem: Find shortest non-zero vector in lattice L(B)
    # This gets exponentially harder with dimension
    dimensions = [2, 4, 8, 16, 32, 64, 128]
    classical_complexity = [2**d for d in [1, 2, 3, 4, 5, 6, 7]]
    quantum_complexity = [2**(d/2) for d in [1, 2, 3, 4, 5, 6, 7]]

    plt.figure(figsize=(10, 6))
```

```
    plt.plot(dimensions, classical_complexity, 'r-', label='Classical Best',
linewidth=2)
    plt.plot(dimensions, quantum_complexity, 'b--', label='Quantum Best',
linewidth=2)
    plt.xlabel('Lattice Dimension')
    plt.ylabel('Time Complexity (log scale)')
    plt.yscale('log')
    plt.title('Complexity of Lattice Problems vs RSA')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    print("\nObservation: Lattice problems remain exponential even for quantum
computers")
    print("while factoring (RSA) becomes polynomial with Shor's algorithm.")

# Run demonstrations
shor_circuit, shor_counts = simulate_shor_attack(4)
demonstrate_lattice_problem()

print("\n=== QUESTIONS ===")
print("4. Why does Shor's algorithm break RSA but not lattice-based crypto?")
print("5. What is the quantum complexity of solving LWE problems?")

print("6. How does Kyber's security scale with dimension 'n'?")
```

Your Answers:
4.
____
_
5.
____
_
6.
____
_

Exercise 3: Implementing Key Exchange Protocol

```
python
```

```python
print("\n=== EXERCISE 3: COMPLETE KEY EXCHANGE PROTOCOL ===")

class QuantumSafeChat:
    """Simulate quantum-safe encrypted chat using Kyber"""

    def __init__(self, username):
        self.username = username
        self.kyber = SimulatedKyber(dimension=128)
        self.public_key = None
        self.private_key = None
        self.shared_secret = None

    def generate_keypair(self):
        self.public_key, self.private_key = self.kyber.generate_keys()
        print(f"[{self.username}] Key pair generated")
        return self.public_key

    def establish_session(self, other_public_key):
        ciphertext, self.shared_secret, _ = self.kyber.encapsulate(
            other_public_key,
            seed=hashlib.sha256(self.username.encode()).digest()
        )
        print(f"[{self.username}] Session established with shared secret")
        return ciphertext

    def receive_session(self, ciphertext):
        self.shared_secret, _ = self.kyber.decapsulate(ciphertext,
self.private_key)
        print(f"[{self.username}] Session received, shared secret derived")

    def encrypt_message(self, message):
        """Encrypt using derived symmetric key"""
        if self.shared_secret is None:
            raise ValueError("No shared secret established")

        # Use HKDF to derive encryption key
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'quantum-safe-chat',
        )
```

```python
        key = hkdf.derive(self.shared_secret)

        # Simple XOR encryption (for demonstration)
        # In practice: use AES-GCM
        message_bytes = message.encode()
        encrypted = bytes([message_bytes[i] ^ key[i % len(key)]
                          for i in range(len(message_bytes))])

        return encrypted

    def decrypt_message(self, encrypted):
        """Decrypt using derived symmetric key"""
        if self.shared_secret is None:
            raise ValueError("No shared secret established")

        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'quantum-safe-chat',
        )
        key = hkdf.derive(self.shared_secret)

        # XOR decryption (same as encryption)
        decrypted = bytes([encrypted[i] ^ key[i % len(key)]
                          for i in range(len(encrypted))])

        return decrypted.decode()

# Simulate conversation
print("Initializing quantum-safe chat between Alice and Bob...\n")

alice = QuantumSafeChat("Alice")
bob = QuantumSafeChat("Bob")

# Step 1: Alice generates key pair and sends public key to Bob
alice_public = alice.generate_keypair()

# Step 2: Bob generates key pair and sends his public key to Alice
bob_public = bob.generate_keypair()

# Step 3: Alice establishes session with Bob's public key
```

```python
ciphertext_to_bob = alice.establish_session(bob_public)

# Step 4: Bob receives Alice's ciphertext and derives same secret
bob.receive_session(ciphertext_to_bob)

# Step 5: Encrypted messaging
print("\n--- Encrypted Conversation ---")
message = "Hello Bob! This message is quantum-safe."
print(f"[Alice sends]: {message}")

encrypted = alice.encrypt_message(message)
print(f"[Encrypted]: {encrypted[:50]}...")

decrypted = bob.decrypt_message(encrypted)
print(f"[Bob receives]: {decrypted}")

# Verify
print(f"\nVerification:")
print(f"Original == Decrypted: {message == decrypted}")
print(f"Shared secrets match: {alice.shared_secret == bob.shared_secret}")

print("\n=== QUESTIONS ===")
print("7. What are the three main steps in Kyber's key exchange?")
print("8. How does this differ from RSA key exchange?")
print("9. Why is the shared secret used for symmetric encryption?")
```

Your Answers:

7. ____

8. ____

9. ____

Exercise 4: Security Analysis & Quantum Attack Simulation

```python
print("\n=== EXERCISE 4: SECURITY ANALYSIS ===")

def analyze_security_parameters():
    """Analyze how parameters affect security"""
    dimensions = [64, 128, 256, 512]
    classical_security = []
    quantum_security = []

    print("Security Levels vs Dimension:")
    print("-" * 50)
    print("Dimension | Classical Security | Quantum Security")
    print("-" * 50)

    for n in dimensions:
        # Approximate security bits (simplified)
        classical_bits = n * 0.8  # Rough estimate
        quantum_bits = n * 0.4    # Grover-like speedup

        classical_security.append(classical_bits)
        quantum_security.append(quantum_bits)

        print(f"{n:9d} | {classical_bits:17.1f} | {quantum_bits:15.1f}")

    # Plot
    plt.figure(figsize=(10, 6))
    plt.plot(dimensions, classical_security, 'bo-', label='Classical
Security', linewidth=2)
    plt.plot(dimensions, quantum_security, 'rs--', label='Quantum Security',
linewidth=2)
    plt.xlabel('Lattice Dimension (n)')
    plt.ylabel('Security Level (bits)')
    plt.title('Kyber Security vs Dimension')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    return dimensions, classical_security, quantum_security

def simulate_grover_attack():
    """Demonstrate why symmetric crypto needs larger keys"""
    print("\n\nGrover's Attack on Symmetric Encryption:")
```

```
    print("Grover reduces search from O(2^n) to O(2^{n/2})")

    # Create oracle for Grover search
    # Simplified: searching for a marked state
    n_qubits = 3
    marked_state = '101'

    # Build oracle
    oracle = QuantumCircuit(n_qubits)
    # Mark |101⟩ with phase -1
    oracle.cz(0, 2)  # Control on qubits 0 and 2

    # Grover iteration
    grover_circuit = QuantumCircuit(n_qubits, n_qubits)

    # Initial superposition
    grover_circuit.h(range(n_qubits))

    # Apply Grover iteration (simplified)
    grover_circuit.append(oracle, range(n_qubits))
    grover_circuit.h(range(n_qubits))
    grover_circuit.x(range(n_qubits))
    grover_circuit.h(n_qubits-1)
    grover_circuit.mct(list(range(n_qubits-1)), n_qubits-1)
    grover_circuit.h(n_qubits-1)
    grover_circuit.x(range(n_qubits))
    grover_circuit.h(range(n_qubits))

    grover_circuit.measure(range(n_qubits), range(n_qubits))

    # Execute
    backend = Aer.get_backend('qasm_simulator')
    result = execute(grover_circuit, backend, shots=1024).result()
    counts = result.get_counts()

    print(f"\nGrover circuit for {n_qubits} qubits")
    print(f"Marked state '{marked_state}' appears {counts.get(marked_state,
0)} times")
    print(f"Success probability: {counts.get(marked_state, 0)/1024*100:.1f}%")

    # Show why AES-256 becomes AES-128 security quantumly
    print("\nImplication for symmetric encryption:")
```

```python
    print("AES-128: 2^64 quantum operations → insecure")
    print("AES-256: 2^128 quantum operations → still secure")

    return grover_circuit

# Run analyses
dims, classical, quantum = analyze_security_parameters()
grover_circuit = simulate_grover_attack()

print("\n=== QUESTIONS ===")
print("10. What security level does Kyber-512 provide against quantum
attacks?")
print("11. Why does symmetric encryption need larger keys in quantum era?")
print("12. How does Grover's algorithm affect hash functions?")
```

Your Answers:
10.
____
_
11.
____
_
12.
____
_

PART 5: LAB REPORT & ANALYSIS

Report Questions:

A. Technical Analysis:

1. Describe the complete flow of Kyber key exchange in your own words.
   ____
   ____

2. What is the role of the error terms (e, e1, e2) in Kyber's security?

3. How does Qiskit help in understanding post-quantum cryptography?

B. Comparative Analysis:
4. Create a table comparing RSA, ECC, and Kyber:

Feature	RSA-2048	ECC-256	Kyber-512
Public Key Size			
Security vs Quantum			
Key Exchange Speed			
NIST Status			

C. Quantum Implications:
5. If a quantum computer with 1 million qubits existed today, which current encryption would break first and why?

6. How should organizations prepare for the quantum transition?

D. Implementation Challenge:
7. Propose an enhancement to our simulated Kyber for better security or performance:

PART 6: EXTENSION ACTIVITIES

Challenge 1: Implement Real Kyber

```python
# Install real implementation: pip install pycryptodome
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

# Research challenge: Integrate with actual Kyber implementation

# Resources: https://github.com/pq-crystals/kyber
```

Challenge 2: Quantum Network Simulation

```python
# Simulate quantum key distribution (QKD) alongside Kyber
def simulate_hybrid_security():
    """
    Combine Kyber with QKD for ultimate security:
    1. Use QKD for initial key establishment
    2. Use Kyber for bulk encryption
    3. Implement forward secrecy
    """

    pass
```

Challenge 3: Performance Analysis

```python
import time

def benchmark_encryption():
    """Compare RSA vs Kyber performance"""
    sizes = [128, 256, 512, 1024, 2048]
    rsa_times = []
    kyber_times = []

    # Your implementation here

    return sizes, rsa_times, kyber_times
```

PART 7: RESOURCES & REFERENCES

Essential Reading:

1. NIST PQC Standardization: https://csrc.nist.gov/projects/post-quantum-cryptography
2. CRYSTALS-Kyber Specification: https://pq-crystals.org/kyber/
3. Qiskit Textbook - Cryptography: https://qiskit.org/textbook/ch-algorithms/shor.html
4. Python Cryptography Toolkit: https://cryptography.io/

Video Resources:

1. NIST PQC Conference 2023: https://www.youtube.com/watch?v=5OD8A2g6f-I
2. Kyber Deep Dive: https://www.youtube.com/watch?v=UkV9cM-7jYk
3. Quantum Cryptography with Qiskit: https://www.youtube.com/watch?v=8U_6ehyNbvQ

Further Exploration:

1. Implement side-channel attack resistance
2. Study other PQC finalists (Dilithium, Falcon)
3. Explore hybrid schemes (PQC + traditional)
4. Research lattice cryptography mathematics

GRADING RUBRIC

Category	Excellent (4)	Good (3)	Satisfactory (2)	Needs Work (1)
Code Implementation	All exercises completed, runs without errors	Most exercises completed, minor issues	Basic functionality working	Significant errors or missing parts
Concept Understanding	Demonstrates deep understanding of Kyber and quantum threats	Good understanding of key concepts	Basic comprehension	Major misconceptions
Analysis & Reporting	Thorough analysis, clear comparisons	Good analysis with most	Basic answers provided	Incomplete or unclear analysis

	, insightful conclusions	questions answered		
Extension Work	Attempted challenges with good results	Attempted at least one challenge	Considered extensions	No extension work
Lab Questions	All questions answered correctly and thoroughly	Most questions answered correctly	Basic answers to main questions	Many questions incomplete

Total Points: _____ / 20
Grade: _____

TEACHER'S NOTES

Lab Setup Requirements:

1. Python 3.8+ with Jupyter Notebook or Google Colab
2. Install: `pip install qiskit cryptography numpy matplotlib`
3. For advanced: `pip install pqcrypto` (real Kyber implementation)

Time Management:

- Basic: Exercises 1-2 (90 minutes)
- Standard: Exercises 1-3 (120 minutes)
- Advanced: All exercises + extensions (180 minutes)

Common Student Challenges:

1. Lattice math complexity - Focus on conceptual understanding over mathematical details
2. Quantum vs post-quantum confusion - Emphasize: quantum computers break some crypto, post-quantum crypto resists this
3. Implementation vs simulation - Clarify this is educational simulation, not production code

Assessment Options:

1. Lab report (Part 5 questions)
2. Code submission with comments
3. Presentation on quantum threats and defenses
4. Research paper comparing PQC algorithms

Real-World Connections:

- Current Events: NIST standards adoption timeline
- Industry: Cloud providers (AWS, Google) already offering PQC
- Government: NSA's CNSA 2.0 timeline for PQC migration
- Research: Ongoing cryptanalysis of Kyber and other PQC

Differentiation Strategies:

- Beginner: Focus on Exercise 1, use provided code as-is
- Intermediate: Modify parameters, analyze security trade-offs
- Advanced: Implement real Kyber, compare with other PQC algorithms
- Research: Investigate side-channel attacks on lattice crypto