Quantum-Resistant Cryptographic Solutions: Ensuring Secure Communication Against Modern Cyber Threats

Type 6 (Designing a quantum-resistant cryptography solution to secure your communication)

Vishnu Vardhan Ciripuram – vc2499@nyu.edu

Choice of Post Quantum Cryptographic Algorithms

Classic McEliece :

- Classic McEliece: Developed as a post-quantum cryptographic solution by leveraging Niederreiter’s dual version of McEliece’s encryption using binary Goppa codes.

- Ensures IND-CCA2 security, even against quantum computers, making it a robust choice for long-term security.

- Despite its original development in 1978, it has gained renewed interest due to its immunity to Shor’s algorithm and Fourier sampling, positioning it as a candidate for post-quantum cryptography.

BIKE :

- BIKE is distinguished by its utilization of error-correcting codes for security against quantum attacks, aligning with the NIST PQC project's objectives.

- Extensive analysis and testing have confirmed BIKE's resilience to various threats, including quantum attacks, solidifying its standing as a leading post-quantum cryptographic protocol.

- Despite its robust security measures, BIKE maintains practicality with manageable key sizes and efficient key exchange and encryption capabilities, further bolstering its appeal for widespread adoption.

Impact of Quantum Computing on NSA Suite B Encryption Schemes:

NSA Suite B encryption schemes:

- RSA
- ECDSA (Elliptic Curve Digital Signature Algorithm)
- Diffie-Hellman

These problems are considered hard for classical computers to solve within a reasonable timeframe, providing the basis for secure public-key encryption, digital signatures, and key exchange protocols.

Based on classical cryptographic principles:

- Integer Factorization Problem
- Discrete Logarithm Problem

SHOR'S ALGORITHM

→ **Shor's Algorithm: Quantum Breakthrough**

Developed by Peter Shor in 1994.

Demonstrates that a quantum computer with enough qubits and computational power could solve both the Integer Factorization Problem and the Discrete Logarithm Problem in polynomial time

→ **Impact on Encryption Security:**

Raises concerns about the vulnerability of traditional encryption methods like RSA and Diffie-Hellman.

Poses a threat of decryption to previously secure data and compromised digital communications.

→ **Urgency for Post-Quantum Cryptography:**

Emphasizes the critical need for post-quantum cryptographic solutions.

Essential for maintaining data security amidst advancing quantum computing technology.

Urges proactive adoption to mitigate potential risks to sensitive information.

GROVER'S ALGORITHM

→ **Grover's Algorithm Overview:**

A significant quantum algorithm designed for searching unsorted databases.

Operates in \(O(\sqrt{n}) \) time, providing a quadratic speedup over classical search algorithms.

→ **Impact on Encryption Security:**

While not as immediate as Shor's algorithm, Grover's poses a threat to symmetric key encryption schemes.

Reduces the effective security of symmetric key encryption by approximately half.

→ **Implications for Security Measures:**

Highlights the need for reassessment of encryption strength in the quantum computing era.

Encourages the adoption of longer key lengths and more robust encryption techniques to counteract quantum threats.

→ **Proactive Security Measures:**

Urges organizations to prepare for future quantum computing capabilities.

Emphasizes the importance of staying ahead in encryption technology to maintain data security.

Quantifying "Very Powerful" Quantum Computers

→ **Breaking RSA-2048 Encryption:**

- Requires substantial quantum computing power.
- Shor's algorithm necessitates a large number of logical qubits.

→ **Logical Qubit Requirements:**

- Estimated around 4,099 logical qubits for factoring a 2048-bit RSA key.
- Extensive error correction inflates the need for physical qubits.

→ **Physical Qubit Estimates:**

- Recent research suggests a minimum of 10 million physical qubits.
- Estimates range from 10-20 million, considering error correction, qubit connectivity, and coherence.

→ **Reflects advancements in quantum computing technology.**

- Indicates a potential nearing of the threshold for achieving quantum-resistant encryption.

Post-Quantum Cryptography and Quantum-Resistant Encryption:

→ **Quantum Threat:** Quantum computing poses a grave threat to current encryption methods, necessitating a shift towards Post-Quantum Cryptography (PQC) and Quantum-Resistant Cryptography.

→ **Focus of PQC:** PQC aims to develop encryption schemes and protocols resilient against quantum attacks.

→ **Leading Candidates:**

- Lattice-based and code-based encryption schemes are prominent contenders.
- Lattice-based schemes offer capabilities like Fully Homomorphic Encryption (FHE) and show resilience against quantum threats.
- Code-based schemes, like the McEliece cryptosystem, leverage error-correcting codes and demonstrate historical security.

Transitioning to quantum-resistant encryption schemes from lattice-based and code-based families is imperative to counter the advancing quantum threat, ensuring robust security and performance for real-world applications. Embracing these encryption methods is a proactive step to safeguard data integrity in the face of evolving quantum computing technology.

Designing Quantum-Resistant Secure Communication Schemes : point 3

The project commenced with the establishment of a robust communication conduit between a client and a server, facilitating the seamless exchange of information. Employing RSA encryption, a symmetric key was transmitted between the client and server entities, subsequently utilized by the AES encryption algorithm to secure message transmissions.

RSA encryption, reliant on the computational complexity of prime factorization, faces vulnerability to quantum computing advancements. Consequently, in pursuit of quantum resilience, RSA has been supplanted by BIKE and classic McEliece algorithms.

The BIKE and McEliece algorithms, characterized by their quantum-resistant properties, offer enhanced security measures.

BIKE

KeyGen : $() \mapsto (h_0, h_1, \sigma), h$	**Encaps** : $h \mapsto K, c$
Output: $(h_0, h_1, \sigma) \in \mathcal{H}_w \times \mathcal{M},\ h \in \mathcal{R}$ 1: $(h_0, h_1) \xleftarrow{\mathcal{D}} \mathcal{H}_w$ $\triangleright$ (1) 2: $h \leftarrow h_1 h_0^{-1}$ 3: $\sigma \xleftarrow{\$} \mathcal{M}$	Input: $h \in \mathcal{R}$ Output: $K \in \mathcal{K},\ c \in \mathcal{R} \times \mathcal{M}$ 1: $m \xleftarrow{\$} \mathcal{M}$ 2: $(e_0, e_1) \leftarrow \mathbf{H}(m)$ 3: $c \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$ 4: $K \leftarrow \mathbf{K}(m, c)$

Decaps : $(h_0, h_1, \sigma), c \mapsto K$

Input: $((h_0, h_1), \sigma) \in \mathcal{H}_w \times \mathcal{M},\ c = (c_0, c_1) \in \mathcal{R} \times \mathcal{M}$

Output: $K \in \mathcal{K}$

1: $e' \leftarrow \texttt{decoder}(c_0 h_0, h_0, h_1)$ $\triangleright$ $e' \in \mathcal{R}^2 \cup \{\perp\}$

2: $m' \leftarrow c_1 \oplus \mathbf{L}(e')$ $\triangleright$ with the convention $\perp = (0, 0)$

3: **if** $e' = \mathbf{H}(m')$ **then** $K \leftarrow \mathbf{K}(m', c)$ **else** $K \leftarrow \mathbf{K}(\sigma, c)$

McEliece

McEliece Cryptosystem, ctd.

$\hat{G} = \boxed{P}\ \boxed{G}\ \boxed{S}$

- $\hat{G}$: public key
- P: permutation matrix
- G: Gen. mat. for a code that can correct t errors
- S: random invertible

ALICE

- To send $x \in \mathbb{F}_2^k$ to Bob:
  - Choose a random $e \in \mathbb{F}_2^n$, of weight t.
  - Alice sends $\hat{G} \cdot x + e$ to Bob

BOB

- To decrypt Alice's message:
  - Bob computes $P^{-1}(\hat{G}x + e)$
    $= G \cdot S \cdot x + P^{-1} \cdot e = G(Sx) + e'$ $wt(e') = t$
  - Decode to obtain $S \cdot x$
  - Compute $S^{-1}(Sx) = x$.

EVE

- Eve sees: $\hat{G}x + e$, $\hat{G}$
- HOPE: $\hat{G}$ looks like a completely random matrix to Eve.
- HOPE: Decoding a random linear code is hard.

ASSUMPTION: The HOPEs are true.

Implementing Quantum-Resistant secure communication scheme

The project consists of python implementation of BIKE and Classic Mcelice Key Encapsulation Mechanism in a client server environment. The project consists of following main files for each Key Encryption Algorithm.

Client.py

Server.py

Quantum_crypto.py

Aes_encryption.py

Encryption_tests.py

Efficiency_tests.py

Client.py

```python
def client_send_and_receive(message, key):
    host, port = '127.0.0.1', 65432
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        # Send initial message
        iv, tag, encrypted_message = aes_encrypt(message.encode(), key)
        full_message = iv + tag + encrypted_message
        s.sendall(full_message)
        print("Message sent to server.")

        # Receive and print the server's first response
        full_response = s.recv(1024)
        iv = full_response[:12]
        tag = full_response[12:28]
        encrypted_response = full_response[28:]
        response = aes_decrypt(iv, tag, encrypted_response, key)
        print("First response from server:", response.decode())

if __name__ == "__main__":
    key = generate_quantum_key()
    client_send_and_receive("Hello from client!", key)
```

Client is initializing a socket connection to the server by connecting to the dedicated port on which server is listening. It then sends its public key to the server. This public key was generated by “generate Key pair” function of either Mceliece or Bike algorithm.

It then receives the symmetric key to be used by AES. Client then send messages using AES encryption.

```
ect>python3 client_version5.py
Message sent to server.
First response from server: Hello from server in response to your message: Hello from cli
ent!
PS C:\Users\cvvar\OneDrive\Desktop\secure_communication_project\secure_communication_proj
ect> 
```

Server.py:

```python
def server_respond(key):
    host, port = '127.0.0.1', 65432
    message_history = set()  # Set to store message hashes to detect replays

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((host, port))
        s.listen()
        print("Server is listening...")
        conn, addr = s.accept()
        with conn:
            while True:
                try:
                    full_message = conn.recv(1024)
                    if not full_message:
                        break
                    iv = full_message[:12]
                    tag = full_message[12:28]
                    encrypted_message = full_message[28:]

                    # Hash the encrypted message to check for replays
                    message_hash = hashlib.sha256(encrypted_message).hexdigest()
                    if message_hash in message_history:
                        print("Replay attack detected, message discarded.")
                        continue

                    message_history.add(message_hash)
                    message = aes_decrypt(iv, tag, encrypted_message, key)
                    print("Received message:", message.decode())

                    response = "Hello from server in response to your message: " + message.decode
                    iv, tag, encrypted_response = aes_encrypt(response.encode(), key)
                    conn.sendall(iv + tag + encrypted_response)
                except Exception as e:
                    print("An error occurred:", e)
                    break
```

The server will receive the public key, use it to encapsulate a secret, and send it back to the client. server program will communicates over TCP/IP using AES-GCM encryption for secure messaging. It listens for incoming connections, decrypts received messages, detects replay attacks to ensure message integrity, and responds with encrypted messages.

The program demonstrates a basic implementation of secure communication between a server and a client using symmetric key encryption

```
ct>  python3 server_version5.py
Server is listening...
Received message: Hello from client!
PS C:\Users\cvvar\OneDrive\Desktop\secure_communication_project\secure_communication_proje
ct>
```

Quantum_crypto.py: Classic Mceliece

```python
class McEliece:

    def __init__(self, code, S, P, t):
        self.S = S
        self.P = P
        self.t = t

        self.code = code

        # sizes
        self.k, self.n = code.getG().shape

        # McEliece keys
        self.public_key = ((self.S @ code.getG() @ self.P % 2), self.t)
        self.private_key = (self.S, code.getG(), self.P)

    @classmethod
    def from_linear_code(cls, code: LinearCode, t: int):
        k, n = code.getG().shape

        # permutation matrix (n * n)
        P = np.eye(n, dtype=int)
        np.random.shuffle(P)

        # random matrix (k * k)
        S = McEliece._get_non_singular_random_matrix(k)

        return cls(code, S, P, t)

    def encrypt(self, word):
        errors_num = self.t

        # error vector size n with t errors
        z = [1 for _ in range(errors_num)] + [0 for _  in range(self.n - errors_num)]
        z = np.array(z, dtype=int)
        np.random.shuffle(z)

        res = ((word @ self.public_key[0] % 2) + z) % 2

        return res

    def decrypt(self, codeword):
        A, invP = gaussjordan(self.P, True)

        c = codeword @ invP % 2
        c = np.array(c, dtype=int)

        d = self.code.decode(c)
        m = self.code.get_message(d)

        _, invS = gaussjordan(self.S, True)

        res = m @ invS % 2
        res = np.array(res, dtype=int)
```

This code defines a `McEliece` class that implements the McEliece public-key cryptosystem, which is resistant to quantum attacks. It uses error-correcting codes and a combination of random and permutation matrices to create secure encryption and decryption processes.

Initialization: Takes a linear error-correcting code, a random matrix `S`, a permutation matrix `P`, and an error tolerance `t`. It sets up the public key from these components.-

Encryption: Encrypts a given message by multiplying it with the public key, then adds a random error vector with `t` errors. The output is the ciphertext.

Decryption: Decrypts an encrypted message by reversing the permutation and applying error correction. It then decodes the message with the code's generator matrix and inverse of `S` to retrieve the original plaintext.

This class demonstrates the basic McEliece encryption and decryption operations, leveraging error-correcting codes to achieve quantum-resistant security.

Quantum_crypto.py: BIKE

```
import oqs

def generate_bike_keys():
    kem = oqs.KeyEncapsulation('BIKE-L1')  # Ensure you use tl
    public_key = kem.generate_keypair()
    secret_key = kem.export_secret_key()
    return kem, public_key

def encapsulate_bike_secret(public_key):
    kem = oqs.KeyEncapsulation('BIKE-L1')
    ciphertext, shared_secret = kem.encap_secret(public_key)
    return ciphertext, shared_secret

def decapsulate_bike_secret(kem, ciphertext):
    shared_secret = kem.decap_secret(ciphertext)
    return shared_secret
```

This code snippet demonstrates how to use the Open Quantum Safe (OQS) library with BIKE-L1, a post-quantum key encapsulation mechanism (KEM). It includes three main functions:

- *generate_bike_keys()*: Generates a BIKE-L1 key pair, returning the public key and a key encapsulation mechanism (kem).
- *encapsulate_bike_secret(public_key)*: Encapsulates a secret with the given public key, producing a ciphertext and a shared secret.
- *decapsulate_bike_secret(kem, ciphertext)*: Decapsulates the shared secret from a given ciphertext using the provided kem.

Together, these functions demonstrate basic key generation, secret encapsulation, and decapsulation with BIKE-L1, showcasing post-quantum cryptographic operations with OQS.

Aes_encryption.py :

```
Click here to ask Blackbox to help you code faster
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from os import urandom

def aes_encrypt(plaintext_bytes, key):
    key = key[:16]  # Ensure key is exactly 16 bytes, AES-128
    iv = urandom(12)  # 12 bytes is recommended for GCM
    cipher = Cipher(algorithms.AES(key), modes.GCM(iv), backend=default_backend())
    encryptor = cipher.encryptor()
    encrypted_message = encryptor.update(plaintext_bytes) + encryptor.finalize()
    return iv, encryptor.tag, encrypted_message

def aes_decrypt(iv, tag, encrypted_message, key):
    key = key[:16]  # Ensure key is exactly 16 bytes, AES-128
    cipher = Cipher(algorithms.AES(key), modes.GCM(iv, tag), backend=default_backend())
    decryptor = cipher.decryptor()
    decrypted_message = decryptor.update(encrypted_message) + decryptor.finalize()
    return decrypted_message
```

This code snippet provides two functions for encrypting and decrypting data with AES-GCM, a secure mode offering encryption and authentication. Here's what they do:

aes_encrypt(plaintext_bytes, key):
- *Encrypts* plaintext using AES-128 in GCM mode.
- Uses a 16-byte key and generates a 12-byte initialization vector (IV).
- Returns the IV, an authentication tag, and the encrypted message.

aes_decrypt(iv, tag, encrypted_message, key):
- *Decrypts* an encrypted message using AES-GCM with a 16-byte key, the IV, and the authentication tag.
- Returns the decrypted plaintext.

These functions demonstrate a secure way to encrypt and decrypt data with AES-GCM, providing both confidentiality and data integrity.

Encryption_tests.py:

```python
def test_encryption_decryption(self):
    """Test encryption and decryption functionality."""
    print("\nRunning Test: Encryption and Decryption")
    message = "Test message for encryption"
    iv, tag, encrypted_message = aes_encrypt(message.encode(), self.key)
    decrypted_message = aes_decrypt(iv, tag, encrypted_message, self.key)
    self.assertEqual(message, decrypted_message.decode(), "The decrypted message should match the original.")
    print("Success: The message was encrypted and decrypted correctly.")

def test_decryption_with_wrong_key(self):
    """Test decryption using an incorrect key."""
    print("\nRunning Test: Decryption with Wrong Key")
    wrong_key = os.urandom(16)
    message = "Test message for encryption"
    iv, tag, encrypted_message = aes_encrypt(message.encode(), self.key)
    with self.assertRaises(Exception):
        aes_decrypt(iv, tag, encrypted_message, wrong_key)
    print("Success: Decryption with the wrong key failed as expected.")

def test_tampered_ciphertext(self):
    """Test the system's response to tampered ciphertext."""
    print("\nRunning Test: Integrity Check Against Tampered Ciphertext")
    message = "Test message for encryption"
    iv, tag, encrypted_message = aes_encrypt(message.encode(), self.key)
    tampered_message = encrypted_message[:-1] + (encrypted_message[-1] ^ 0x01).to_bytes(1, 'little')
    with self.assertRaises(Exception):
        aes_decrypt(iv, tag, tampered_message, self.key)
    print("Success: Tampered ciphertext was detected and decryption failed.")

ass TestServerReplayAttack(unittest.TestCase):
def setUp(self):
    self.key = os.urandom(16)
    self.message_history = set()

def simulate_server_reception(self, encrypted_message, iv, tag):
    """Simulate server logic for detecting replay attacks."""
    message_hash = sha256(encrypted_message).hexdigest()
    if message_hash in self.message_history:
        raise Exception("Replay attack detected and message discarded.")
    self.message_history.add(message_hash)
    return aes_decrypt(iv, tag, encrypted_message, self.key)

def test_replay_attack(self):
    """Test the server's ability to detect and reject replayed messages."""
    print("\nRunning Test: Replay Attack Detection")
    message = "Test message susceptible to replay"
    iv, tag, encrypted_message = aes_encrypt(message.encode(), self.key)
    decrypted_message = self.simulate_server_reception(encrypted_message, iv, tag)
    self.assertEqual(decrypted_message.decode(), message)
    with self.assertRaises(Exception) as context:
        self.simulate_server_reception(encrypted_message, iv, tag)
    self.assertIn("Replay attack detected", str(context.exception))
    print("Success: Replay attack was detected and handled correctly.")
```

This file contains test cases to test the client server communication scheme based of Message Confidentiality, Message Integrity, and Message Replay attack protection

```
PS C:\Users\cvvar\OneDrive\Desktop\secure_communication_project\secure_communication_proje
ct> python3 .\encryption_tests.py

Running Test: Decryption with Wrong Key
Success: Decryption with the wrong key failed as expected.
.
Running Test: Encryption and Decryption
Success: The message was encrypted and decrypted correctly.
.
Running Test: Integrity Check Against Tampered Ciphertext
Success: Tampered ciphertext was detected and decryption failed.
.
Running Test: Replay At Close Detection
Success: Replay attack was detected and handled correctly.
.
----------------------------------------------------------------------
Ran 4 tests in 0.033s

OK
```

Efficiency_tests.py:

```python
n = 300
d_v = 6
d_c = 10
ldpc = LDPC.from_params(n, d_v, d_c)
crypto = McEliece.from_linear_code(ldpc, 12)

plaintext = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie."
plaintext_bytes = plaintext.encode("utf-8")  # Convert to bytes

start_time = time.time()
binary_word = np.random.randint(2, size=ldpc.getG().shape[0])
encrypted = crypto.encrypt(binary_word)
decrypted = crypto.decrypt(encrypted)
key_gen_time = time.time() - start_time
print(f"McEliece Key Generation and Processing Time: {key_gen_time:.6f} seconds")

aes_key = bytes(decrypted)[:16]

start_encryption_time = time.time()
iv, encrypted_message = aes_encrypt(plaintext_bytes, aes_key)
encryption_time = time.time() - start_encryption_time
print(f"Encryption Time: {encryption_time:.6f} seconds")
print(f"Ciphertext Length: {len(iv + encrypted_message)} bytes")

start_decryption_time = time.time()
decrypted_message = aes_decrypt(iv, encrypted_message, aes_key)
decryption_time = time.time() - start_decryption_time
print(f"Decryption Time: {decryption_time:.6f} seconds")

decrypted_text = decrypted_message.decode("utf-8")
print("\nOriginal text:", plaintext)
print("\nDecrypted text:", decrypted_text)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SEARCH ERROR

```
dishasheshappa@Dishas-Air McEliece % /usr/bin/python3 "/Users/dishasheshappa/Crypto Assignments/McEliece/src/Testing_metrics.py"
McEliece Key Generation and Processing Time: 0.034774 seconds
Encryption Time: 0.033473 seconds
Ciphertext Length: 176 bytes
Decryption Time: 0.000043 seconds

Original text: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.

Decrypted text: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.
Encryption and decryption with AES successful.
dishasheshappa@Dishas-Air McEliece %
```

This file observes the overall execution time of the algorithm for different key lengths.

Efficiency Tests : Classic Mceliece

n = 300

d_v = 6

d_c = 10

```python
# Define parameters for LDPC and McEliece
n = 300
d_v = 6
d_c = 10
ldpc = LDPC.from_params(n, d_v, d_c)
crypto = McEliece.from_linear_code(ldpc, 12)
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   SEARCH ERROR   Python

```
dishasheshappa@Dishas-Air McEliece % /usr/bin/python3 "/Users/dishasheshappa/Crypto Assignments/McEliece/src/Testing_metrics.py"
McEliece Key Generation and Processing Time: 0.034204 seconds
Encryption Time: 0.031204 seconds
Ciphertext Length: 176 bytes
Decryption Time: 0.000040 seconds

Original text: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.

Decrypted text: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.
Encryption and decryption with AES successful.
dishasheshappa@Dishas-Air McEliece %
```

Efficiency Tests : Classic Mceliece

n = 3936

d_v = 3

d_c = 6

```python
# Define parameters for LDPC and McEliece
n = 3936
d_v = 3
d_c = 6
ldpc = LDPC.from_params(n, d_v, d_c)
crypto = McEliece.from_linear_code(ldpc, 12)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SEARCH ERROR Python

```
dishasheshappa@Dishas-Air McEliece % /usr/bin/python3 "/Users/dishasheshappa/Crypto Assignments/McEliece/src/Testing_metrics.py"
McEliece Key Generation and Processing Time: 10.990759 seconds
Encryption Time: 0.089003 seconds
Ciphertext Length: 176 bytes
Decryption Time: 0.000193 seconds

Original text: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.

Decrypted text: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.
Encryption and decryption with AES successful.
dishasheshappa@Dishas-Air McEliece %
```

Efficiency Tests : Classic Mceliece

n = 6936

d_v = 3

d_c = 6

```python
# Define parameters for LDPC and McEliece
n = 6936
d_v = 3
d_c = 6
ldpc = LDPC.from_params(n, d_v, d_c)
crypto = McEliece.from_linear_code(ldpc, 12)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SEARCH ERROR Python

```
dishasheshappa@Dishas-Air McEliece % /usr/bin/python3 "/Users/dishasheshappa/Crypto Assignments/McEliece/src/Testing_metrics.py"
McEliece Key Generation and Processing Time: 50.058345 seconds
Encryption Time: 0.087691 seconds
Ciphertext Length: 176 bytes
Decryption Time: 0.000240 seconds

Original text: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.

Decrypted text: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.
Encryption and decryption with AES successful.
```

Efficiency Tests : BIKE

In this we used BIKE-L3 from the Liboqs library. The metrics below show the Key generation time, Ciphertext length, Encryption and Decryption time. We can not tune the parameters as we would have to change the source code directly.

Testing_BIKE.py > derive_key_from_shared_secret

Click here to ask Blackbox to help you code faster

```python
import oqs
import os
import base64
import time
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.padding import PKCS7
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC

def derive_key_from_shared_secret(shared_secret, salt, key_length=32):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=key_length,
        salt=salt,
        iterations=100000,
        backend=default_backend(),
    )
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SEARCH ERROR Python

```
(venv) dishasheshappa@Dishas-Air Post_Quantum % "/Users/dishasheshappa/Crypto Assignments/Post_Quantum/venv/bin/python" "/Users/dishasheshappa/Crypto Assignments/Post_Quantum/Testing_BIKE.py"
BIKE Key Generation Time: 0.028780 seconds
Ciphertext Length: 3115 bytes
AES Encryption Time: 0.001776 seconds
Encrypted Message Length: 176 bytes
AES Decryption Time: 0.000043 seconds
Original message: b'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.'
Encrypted message (Base64): b'WL0VkuIvekhQS3asbLfhaSy/5D6RAzTaIP7PFUgrregIdxS6gFo5CWmU4leQZoj/Bu0WJWJCZOj3RWDxBGCyHw9NgHjMyEnFW2Qrv44FXEdYJjpVtFS0OHzYqG3UYeQUdLlniCFHmD2SnpmnrXXxKre8hEJG8e0Q0UdZcdnxQFEz9SCwOuAQwz85SLwsqPSvFhYKMDZucJxaDAbBb7Jnl5B7s42UGk27VS/p3IwuNfE='
Decrypted message: b'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis quis magna lacinia, semper neque vel, suscipit augue. Mauris sagittis maximus molestie.'
BIKE key encapsulation, encryption, and decryption successful.
```

References

McElice Tutorial: https://www.youtube.com/watch?v=fLwMvbfr76g

Mceliece: https://classic.mceliece.org/impl.html

Mceliece implementation paper: https://classic.mceliece.org/mceliece-impl-20221023.pdf

Bike: https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf

Library for bike: https://github.com/open-quantum-safe/liboqs/blob/main/docs/algorithms/kem/classic_mceliece.md

Thank You