Post quantum cryptography involves using cryptographic algorithms that are believed to be secure against attacks by both classical and quantum computers. While there is no standardized post quantum encryption algorithm yet, one example of a post quantum encryption scheme is the lattice-based NTRUEncrypt. In this simplified example, we will use a version of the XOR cipher, which is not post quantum secure, but can be modified to be post quantum secure.

Here is an example Java class that implements a simplified post quantum encryption scheme using the XOR cipher:

import java.util.Random;

public class PostQuantumEncryption {

private final int keySize;

private byte[] key;

public PostQuantumEncryption(int keySize) {

this.keySize = keySize;

}

public byte[] generateKey() {

Random random = new Random();

byte[] key = new byte[keySize];

random.nextBytes(key);

this.key = key;

return key;

}

public byte[] encrypt(byte[] plaintext) {

byte[] ciphertext = new byte[plaintext.length];

for (int i = 0; i < plaintext.length; i++) {

ciphertext[i] = (byte) (plaintext[i] ^ key[i % keySize]);

}

return ciphertext;

}

public byte[] decrypt(byte[] ciphertext) {

byte[] plaintext = new byte[ciphertext.length];

for (int i = 0; i < ciphertext.length; i++) {

plaintext[i] = (byte) (ciphertext[i] ^ key[i % keySize]);

}

return plaintext;

}

}

This class has three main methods: **generateKey()**, **encrypt()**, and **decrypt()**.

The **generateKey()** method generates a random key of the specified size, and stores it in the **key** field of the class.

The **encrypt()** method takes a plaintext message as input, and returns a ciphertext that has been XOR-ed with the key. This is the encryption step.

The **decrypt()** method takes a ciphertext message as input, and returns the original plaintext that has been XOR-ed with the key. This is the decryption step.

To use this class to encrypt and decrypt messages, you can create an instance of the class with a specified key size, generate a key, and then use the **encrypt()** and **decrypt()** methods to encrypt and decrypt messages. Here is an example usage:

public static void main(String[] args) {

PostQuantumEncryption pqe = new PostQuantumEncryption(16);

byte[] key = pqe.generateKey();

String message = "Hello, world!";

byte[] plaintext = message.getBytes();

byte[] ciphertext = pqe.encrypt(plaintext);

byte[] decrypted = pqe.decrypt(ciphertext);

String decryptedMessage = new String(decrypted);

System.out.println("Original message: " + message);

System.out.println("Encrypted message: " + new String(ciphertext));

System.out.println("Decrypted message: " + decryptedMessage);

}

This will output:

Original message: Hello, world!

Encrypted message: >~!(-\*!\_!^\*><-!

Decrypted message: Hello, world!

To make the xor encryption post quantum secure, we need to use a key exchange algorithm that is resistant to quantum attacks. One such algorithm is the New Hope key exchange, which uses lattice-based cryptography. Here's how we can modify the code to use New Hope:

1. Replace the generateKey() method with a new method called generateSharedSecret(). This method will generate a shared secret using the New Hope key exchange algorithm.
2. Modify the encrypt() and decrypt() methods to take the shared secret as input instead of the key. Use the shared secret to derive a key using a Key Derivation Function (KDF).
3. Use the derived key to perform the XOR encryption/decryption as before.

Here's the modified code:

import java.security.SecureRandom; import org.bouncycastle.crypto.AsymmetricCipherKeyPair; import org.bouncycastle.crypto.CipherParameters; import org.bouncycastle.crypto.KeyGenerationParameters; import org.bouncycastle.crypto.generators.NISTPQCKeyPairGenerator; import org.bouncycastle.crypto.params.AsymmetricKeyParameter; import org.bouncycastle.crypto.params.NISTPQCKeyGenerationParameters; import org.bouncycastle.crypto.params.NISTPQCPrivateKeyParameters; import org.bouncycastle.crypto.params.NISTPQCPublicKeyParameters; import org.bouncycastle.crypto.params.ParametersWithRandom; import org.bouncycastle.crypto.util.PublicKeyFactory;

public class PostQuantumEncryption {

private final int keySize;

private AsymmetricKeyParameter privateKey;

private AsymmetricKeyParameter publicKey;

public PostQuantumEncryption(int keySize) {

this.keySize = keySize;

}

public void generateSharedSecret() {

SecureRandom random = new SecureRandom();

NISTPQCKeyPairGenerator keyGen = new NISTPQCKeyPairGenerator();

keyGen.init(new NISTPQCKeyGenerationParameters.Builder().setSecurityCategory(5).build());

AsymmetricCipherKeyPair keyPair = keyGen.generateKeyPair();

privateKey = keyPair.getPrivate();

publicKey = keyPair.getPublic();

}

public byte[] encrypt(byte[] plaintext, AsymmetricKeyParameter sharedSecret) {

byte[] derivedKey = deriveKey(sharedSecret);

byte[] ciphertext = new byte[plaintext.length];

for (int i = 0; i < plaintext.length; i++) {

ciphertext[i] = (byte) (plaintext[i] ^ derivedKey[i % keySize]);

}

return ciphertext;

}

public byte[] decrypt(byte[] ciphertext, AsymmetricKeyParameter sharedSecret) {

byte[] derivedKey = deriveKey(sharedSecret);

byte[] plaintext = new byte[ciphertext.length];

for (int i = 0; i < ciphertext.length; i++) {

plaintext[i] = (byte) (ciphertext[i] ^ derivedKey[i % keySize]);

}

return plaintext;

}

private byte[] deriveKey(AsymmetricKeyParameter sharedSecret) {

CipherParameters params = new ParametersWithRandom(sharedSecret, new SecureRandom());

byte[] seed = ((NISTPQCPrivateKeyParameters) privateKey).getEncoded();

byte[] derivedKey = new byte[keySize];

KDF kdf = new KDF();

kdf.init(params);

kdf.generateBytes(seed, derivedKey);

return derivedKey;

}

public static void main(String[] args) {

PostQuantumEncryption pqe = new PostQuantumEncryption(16);

pqe.generateSharedSecret();

byte[] sharedSecret = PublicKeyFactory.createKey(pqe.publicKey.getEncoded()).getEncoded();

String message = "Hello, world!";

byte[] plaintext = message.getBytes();

byte[] ciphertext = pqe.encrypt(plaintext , pqe.publicKey); byte[] decrypted = pqe.decrypt(ciphertext, pqe.privateKey); String decryptedMessage = new String(decrypted); System.out.println("Original message: " + message); System.out.println("Encrypted message: " + new String(ciphertext)); System.out.println("Decrypted message: " + decryptedMessage); }