SE448: Information and Cyber Security	**Recitation 7**
Semester 2 5786	**13 May 2026**
Lecturer: Michael J. May	**Kinneret College**

Using ML-KEM (Kyber)

In this recitation we will write code in Java that performs the encryptions and decryptions for us using the post quantum cryptographic algorithm ML-KEM, formerly known as Kyber-Crystals.

1 Using ML-KEM in Java

We will build a tool that lets you perform ML-KEM key wrap on AES keys. ML-KEM is a very limited algorithm, designed only for use within a hybrid encryption framework, so the standard usage is to use a bulk encryption algorithm such as AES to encrypt a file or message and then to use to the ML-KEM public key to "wrap" (encrypt) the AES key. The recipient would then receive the encrypted key and file, "unwrap" (decrypt) the AES key using the ML-KEM private key and then use the AES key to decrypt the file or message. The result is that when using ML-KEM, you can only provide it a very small number of bytes (10s of bytes) and the Java ML-KEM algorithms assume that you're using it in combination with a known symmetric cipher.

ML-KEM is a complicated algorithm, even more complex than RSA. Therefore, all of the work for the algorithm takes place under the hood, so there isn't a lot to see here. Still, it's important for you to recognize the classes that offer the ML-KEM cryptographic services for later use in this class (and outside).

1.1 User Interface

We're going to make a command line tool as before. The tool has three operations that it can perform:

1. Generate a new ML-KEM key pair with a supported key length (512, 768, 1024 bits) and store the keys in an output file
2. Wrap an AES key in a file using a provided ML-KEM key pair
3. Unwrap an AES key in a file using a provided ML-KEM key pair

To support the operations, the tool must support the following parameters:

outfile The output file path (after wrap or unwrap)

infile The file to process (wrap or unwrap)

keyfile The file containing the key (for wrap/unwrap) or to write the key out to (key generation)

keylength The length of the key to be generated (in bits)

keybits The length of the AES key to wrap or unwrap

operation Can be encrypt, decrypt, generatekey

When asking to generate a key, the following parameters are required:

1. operation
2. keylength
3. keyfile

When asking to encrypt or decrypt a file, the following parameters are required:

1. outfile

2. infile
3. keyfile
4. keybits
5. operation

1.2 Bouncy Castle Provider Post-Quantum Provider

There are two major implementations of the ML-KEM algorithm in Java:

- A native Java 25 JDK implementation based on the KEM class (`https://docs.oracle.com/en/java/javase/25/docs/api/java.base/javax/crypto/KEM.html`) and the key encapsulation methods ML-KEM-512, ML-KEM-768, and ML-KEM-1024 (see `https://docs.oracle.com/en/java/javase/25/docs/specs/security/standard-names.html#kem-algorithms`). The implementation uses custom methods for key encapsulation that you can read about in the documentation.
- The Bouncy Castle post-quantum provider (BC) that has a full native Java implementation of ML-KEM. It uses the KeyPair and KeyPairGenerator classes that we learned about using RSA.

While both options work, we'll use the second one in this recitation since we already saw how the KeyPair and KeyPairGenerator classes work last time. To use the BC implementation, you'll need to add the following to your Gradle build file:

```
dependencies {
    /** Some other dependencies **/
    implementation("org.bouncycastle:bcprov-jdk18on:1.84")
}
```

build.gradle.kts

This adds the dependency on Bouncy Castle's Java 1.8 provider to your project.

You will also need to add the following lines to your Main method to ensure the provider is properly loaded and ready to encrypt when you need:

```
if (Security.getProvider(BOUNCY_CASTLE_PROVIDER) == null) {
            Security.addProvider(new BouncyCastleProvider());
        }
```

Loading Bouncy Castle provider in Main

1.3 The Key Objects

There are a few key classes that are crucial for Bouncy Castle support for the ML-KEM algorithm:

KeyPairGenerator Generates ML-KEM key pairs for you. You just need to give it the algorithm that you want it to create key pairs for. In this case, we'll use "ML-KEM". It also supports other algorithms. Before you can use the generator, you'll need to initialize it using the `initialize` method along with the key pair length that you want. You'll also need to ensure that you have the BouncyCastle provider included in the security provider list as above.

KeyPair Represents a public/private key pair. It's generated by the KeyPairGenerator using the `generateKeyPair` method.

KTSParameterSpec Gives hints to the ML-KEM algorithm how to properly wrap and unwrap the key. We'll use the KTSParameterSpec in combination with the Builder internal class (KTSParameterSpec.Builder).

1.4 Saving and Loading Keys

There are a few ways to store private and public keys on a computer. One option is to use a PEM file. Another is to use an X.509 digital certificate. Both options work, but involve significant additional work in the tool. To save time, I implemented a trivial save and restore mechanism for the private and public keys in hexadecimal format in a text file. The methods are already given to you in the starter code to save time. You can have a look at them to see what they do.

2 Encrypting and Decrypting with Java

We'll use the `Cipher` object to encrypt and decrypt, just like before. Remember the following:

- When wrapping, use the public key
- When unwrapping, use the private key

Remember to read the file in a byte[]. You can then use the `wrap(SecretKey)` method of Cipher to wrap and `unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrappedKeyType)` method of Cipher to unwrap.

3 What to do: Fill in the code

Use the code and tool provided to perform the following tasks:

1. Connect the code to create and save the ML-KEM key pair information.
2. Fill in the functions to initialize the KeyPairGenerator, `encrypt` for wrapping, and `decrypt` for unwrapping using the classes mentioned above. Encryption and Decryption work very similarly to what we have seen before with RSA.
3. Make sure the tool supports the legal key sizes for ML-KEM: 512, 768, 1024.
4. As with AES, `Cipher` works only over `byte[]`, so we must read all file data as bytes[]. To make it easier, also output encrypted or decrypted file contents as bytes.
5. Check your implementation for correctness. Due to the randomness included in encryption, you can't compare encrypted outputs to see if they are correct. The best you can do is to attempt to unwrap an AES key with a particular key pair. I put a key pair on Moodle that you can use to test.

4 Experiments

4.1 Experiment 1: Encryption and Decryption

1. Generate new 512 bit ML-KEM key. Store it in a file called Experiment1-Key.txt
2. Create a file with 16 random bytes that could be an AES key. Encrypt it using ML-KEM and the Experiment1-Key.txt key. Store the result in Experiment1-Step2.txt
3. Encrypt the AES key again using ML-KEM and the same key. Store the result in Experiment1-Step3.txt. Compare the contents of Experiment1-Step2.txt and Experiment1-Step3.txt. Are they the same?
4. Decrypt Experiment1-Step2.txt using Experiment1-Key.txt key. Store the result in a file. What is the result?
5. Decrypt Experiment1-Step3.txt using Experiment1-Key.txt key. Store the result in a file. What is the result?

What is going on here?

4.2 Experiment 2: Using Java's KEM

1. Read the documentation above for Java's KEM class for key encapsulation
2. Create another version of the encrpyt/decrypt methods you wrote before that uses the Java KEM class to wrap and unwrap keys.
3. Create a 1024 bit ML-KEM key. Store it in a file called Experiment2-Key.txt
4. Create a file with 32 random bytes that could be an AES key. Encrypt it using ML-KEM and Experient2-Key.txt using the Bouncy Castle implementation. Store the result in Experiment2-Step2.txt.
5. Unwrap the content of Experiment2-Step2.txt using Java's KEM unwrapping classes. Store the results in Experiment2-Step3.txt
6. Compare the results of the unwrapping with the original 32 bytes AES key. Are they the same?
7. Are the Java KEM and Bouncy Castle ML-KEM implementations compatible?