SE448: Information and Cyber Security	**Recitation 3**
Semester 2 5785	**3 April 2025**
Lecturer: Michael J. May	**Kinneret College**

Counter Mode and Files

In this recitation, we'll expand the Java CBC tool that we built last into two directions: adding CTR (counter) mode and GCM.

1 Using CTR mode and GCM with the Java Crypto Libraries

We could use AES CTR mode by requesting it from Java's Cipher class with the string "AES/CTR/NoPadding". The rest of the code would then be identical to the code we wrote for CBC last time.

1.1 Using AES in CTR and CBC Modes on Files with BouncyCastle

For variety, in this part of the recitation we'll use the BouncyCastle (`https://www.bouncycastle.org/`) cryptographic APIs to implement Counter (CTR) mode encryption using the AES libraries. The BouncyCastle cryptographic libraries are a well known set of cryptographic libraries that significantly improve the cryptographic capabilities of Java and C# with many more ciphers and advanced cipher modes not supported natively by either run time. While we can do everything in the steps we're about to perform using the regular Java crypto libraries, it's helpful to learn about another library in case you meet a task that the native Java libraries can't handle.

1.2 Integrating BouncyCastle

The first step in integrating BouncyCastle (BC) into your Java project is to download the BC Java JAR file from their website. There are a number of JARs available for download. Choose the most recent default one. We'll be using a Java more recent than Java 1.8, so you don't need to worry about using older versions or more restricted JAR file versions.

Once downloaded, unzip the JAR into a directory on the computer and add the JAR as an external library to the project. Figure 1 shows a screen shot of how to add the JAR in Intellij. Eclipse has a similar mechanism.

1.3 Using BouncyCastle Crypto Engines for Counter Mode

Once the BC library has been linked, you can use the classes and interfaces that it offers. For AES in counter mode, we'll need to use the following classes: `AESEngine`, `SICBlockCipher`, `CipherOutputStream`, `ParametersWithIV`, `KeyParameter`. The key snippet for setting up the engine for encryption is as follows:

```
1  bouncyCounterCrypto = new SICBlockCipher(new AESEngine());
2  ParametersWithIV params = new ParametersWithIV(
3     new KeyParameter(key.getBytes("ASCII")),
4     iv.getBytes("ASCII")
5  );
6  bouncyCounterCrypto.init(true, params);
```

On line 1 we set up a new copy of the AES engine, the class that actually does the basic AES algorithm. The `SICBlockCipher` instance wraps the engine and does the counter mode actions on top of the block cipher. On lines 2–4 we set up the key using a `KeyParameter` object and an initialization vector using a `ParametersWithIV` object. The key and IV are both provided as strings and so are converted to bytes using

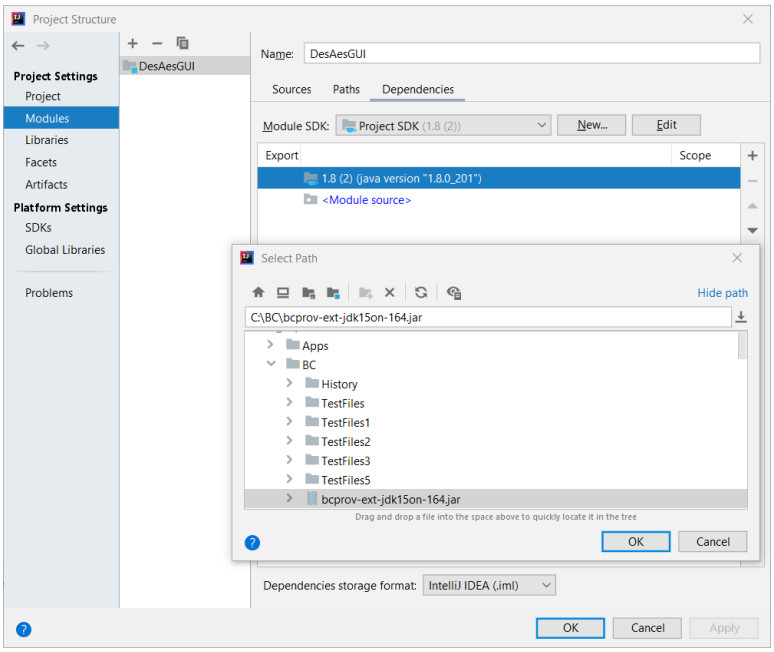

Figure 1: Adding BouncyCastle library to Intellij project.

the `Charset` class. This step isn't necessary if the key or IV are created as byte arrays. Line 6 initializes the counter mode cipher object with the parameter and the value `true` to indicate that the object is going to be used for encryption.

Decryption is nearly identical to encryption with the exception that the value `true` on line 6 of the above snippet must be changed to `false`.

2 Adding GCM Mode

Once you have CTR mode working properly, add support for GCM mode. You'll need to use the Java crypto libraries for that. The relevant cipher suite AES/GCM/NoPadding. You will also need to use the `GCMParameterSpec` class to pass the encryption and decryption parameters to the encryption engine. You can read about the class at: `https://docs.oracle.com/en/java/javase/22/docs/api/java.base/javax/crypto/spec/GCMParameterSpec.html`.

Make the tag length 128 bits for all encryption and decryption.

3 Encrypting and Decrypting Files

Encrypting and decrypting files using AES and DES with CBC and CTR modes is not particularly hard. To encrypt a file, once the Cipher object is initialized, attach it to a `CipherOutputStream` on top of a `FileOutputStream`. The following code is the gist of what you need to write for BouncyCastle.

```
7       FileInputStream fisIn = new
8       FileInputStream(plaintextFileName);
9       FileOutputStream fosOut = new FileOutputStream(outputCTRFileName);
10      org.bouncycastle.crypto.io.CipherOutputStream cosOut =
```

```
11          new org.bouncycastle.crypto.io.CipherOutputStream(fosOut, bouncyCounterCrypto);
12      byte[] buf = new byte[4096];
13      int read = 0;
14      while ( (read = fisIn.read(buf)) > 0) {
15          cosOut.write(buf, 0, read);
16      }
17      cosOut.close();
18      fosOut.close();
19      fisIn.close();;
```

4 What to do

Expand the code we built last week to include support for CTR and GCM encryption using command line parameters. Use the BouncyCastle library for encrypting with CTR.

4.1 Experiments

To help understand how CTR works, let's perform the following experiments:

4.1.1 Experiment 1: IV

1. Set the key to be: 6d69676874617377656c6c6a756d7021. Set the IV to be: 726f6c6c7769746874686570756e6368.
2. Use AES and CBC mode to encrypt a file with the contents: "It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of"
3. What do you get for cipher text?
4. Change the IV to be: 726f6c6c7769746874686570756e637a. Decrypt the cipher text using AES and CBC. What happened? What part of the message got corrupted?
5. Reset the IV to 726f6c6c7769746874686570756e6368 and replace the plaintext if you need. Encrypt the text with AES and **CTR** mode. What do you get for the ciphertext?
6. Change the IV to be: 726f6c6c7769746874686570756e637a. Decrypt the cipher text using AES and **CTR**. What happened? What part of the message got corrupted?

Can you explain why there are differences between the results of changing one byte of the IV in CBC and one byte of the IV in CTR?

4.1.2 Experiment 2: Ciphertext

1. Set the key to be: 6d69676874617377656c6c6a756d7021. Set the IV to be: 726f6c6c7769746874686570756e6368.
2. Use AES and CBC mode to encrypt a file with the contents: "It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of"
3. What do you get for cipher text? The first byte should be 73.
4. Change the first bye to 74 (changes two bits) (0100 1001 $\rightarrow$ 0100 1010). Decrypt using AES and CBC. How much of the message is corrupted?

5. Replace the plaintext and encrypt using AES and CTR. What did you get for the cipher text? The second byte should be 90.
6. Change the second byte to 91. Decrypt using AES and CTR. What part of the message changed? How did it change?
7. Change the second byte to 92. Decrypt using AES and CTR. What part of the message changed? How did it change?

Can you explain why there are differences between the results of changing one byte of the ciphertext in CBC and one byte of the ciphertext in CTR? Can you predict how the decryption will look based on how you change the ciphertext?

4.1.3 Experiment 3: Plaintext

1. If you changed the parameters since last time, reset the key to be: 6d69676874617377656c6c6a756d7021 and reset the IV to be: 726f6c6c7769746874686570756e6368.
2. Use AES and CBC mode to encrypt a file with the contents: "It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of"
3. Note what you got for cipher text. Change the text to "It **vas** the best of times, it was the worst of times, it was the age of wisdom, it was the age of". Encrypt again with AES and CBC. How much of the ciphertext changed?
4. Reset the plaintext to: "It **was** the best of times, it was the worst of times, it was the age of wisdom, it was the age of"
5. Encrypt using AES and CTR. Note what you got for cipher text. Change the text to "It **vas** the best of times, it was the worst of times, it was the age of wisdom, it was the age of". Encrypt again with AES and CTR. How much of the ciphertext changed?

Can you predict how the ciphertext will change based on changes you make to the plain text in CBC? In CTR? What does that tell you about how the ciphertext and plaintext are related under CTR?

4.1.4 Experiment 4: Counter in CTR

1. Reset the key to be 6d69676874617377656c6c6a756d7021. Set the IV to be: 00000000000000000000000000000001 (that's 31 0's followed by a single 1).
2. Use AES and CTR mode to encrypt with repeating contents:

   "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"

   (that's 80 letter a's).
3. What cipher text did you get? Save it on in a file called "iv1.txt"
4. Change the IV to be 00000000000000000000000000000002 (that's 31 0's followed by a single 2).
5. Encrypt the file with the new IV. Save the ciphertext in a file called "iv2.txt".
6. Examine the contents of "iv2.txt" and compare them to the output in "iv1.txt." How do its contents appear compared to the cipher text from encrypting with 00. . . 1? What does that tell you about how the counter is used in Java for CTR mode?
7. Perform steps 4-6 again for IV 00000000000000000000000000000003, 00000000000000000000000000000004, 00000000000000000000000000000005.
8. What pattern do you see in the cipher text output?

4.1.5 Working with GCM

Repeat the above experiments using GCM. For the IV, choose a 96 bit IV such as 686f7273656a756d70696e67.