50.005 Computer System Engineering

NS Lab 2: Symmetric Key Encryption and Signed Message Digests

---

Overview

In NS Module 3, we examined how the security properties of confidentiality and data integrity could be protected by using symmetric key cryptography and signed message digests. In this lab exercise, you will learn how to write a program that makes use of DES for data encryption, MD5 for creating message digests and RSA for digital signing.

Learning objectives

At the end of this lab exercise, you should be able to:

- Understand how symmetric key cryptography (e.g., DES or AES encryption algorithms) can be used to encrypt data and protect its confidentiality.
- Understand how multiple blocks of data are handled using different block cipher modes and padding.
- Compare the different block cipher modes in terms of how they operate.
- Understand how hash functions can be used to create fixed-length message digests.
- Understand how public key cryptography (e.g., RSA algorithm) can be used to create digital signatures.
- Understand how to create message digest using hash functions (e.g., MD5, SHA-1, SHA-256, SHA-512, etc) and sign it using RSA to guarantee data integrity.

Preparation

We will use the Java Cryptography Extension (JCE) to write our program instead of implementing DES, RSA and MD5 directly. The JCE is part of the Java platform and provides an implementation for commonly used encryption algorithms.

Deliverables

Submit a .zip file to eDimension containing the following:

1. A report that illustrates the outputs of your codes and answers to the questions in this handout.
2. Your Java source codes for the three tasks.
3. The encrypted images (ecb.bmp and cbc.bmp) for the second task.

The deadline for the lab submission is **10 April 19 23:59** Late submissions will be dealt with according to the standard course policy.

Part 1: Symmetric key encryption for a text file

Data Encryption Standard (DES) is a US encryption standard for encrypting electronic data. It makes use of a 56-bit key to encrypt 64-bit blocks of plaintext input through repeated rounds of processing using a specialized function. DES is an example of *symmetric key cryptography* , wherein the parties encrypting and decrypting the data both share the same key. This key is then used for both encryption and decryption operations.

In this task, we will make use of the Cipher and *KeyGenerator* classes from the Java Cryptography Extension (JCE) to encrypt and decrypt data from a text file. The steps involved in encryption and decryption are:

1. Generate a key for DES using a *KeyGenerator* instance
2. Create and configure a Cipher object for use with DES
3. Use the *doFinal()* method to perform the actual operation

While the steps for both operations are similar, take note that the working mode of the Cipher object must be configured correctly for encryption or decryption, and the key used for decryption should be the same as that used for encryption.

Generate key for DES

A 56-bit key for DES can be generated using a *KeyGenerator* instance. This can be obtained by calling the *getInstance()* method of the *KeyGenerator* class:

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES");
```

The *getInstance()* method takes in one parameter specifying the algorithm with which the key will be used. Since we are generating a key for use with DES, this should be specified as *"DES"*. Once the *KeyGenerator* instance has been obtained, the key can generated by calling the *generateKey()* method of the *KeyGenerator* instance. This will return a key of the type *SecretKey*.

```
SecretKey desKey = keyGen.generateKey();
```

Create and configure Cipher object

Now that we have generated our key, the next step is to create a *Cipher* object that will be used to perform the encryption or decryption. Cipher objects are created using the *getInstance()* method of the *Cipher* class:

```
Cipher desCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
```

The *getInstance()* method takes in a parameter specifying the algorithm to be used for encryption, as well as the cipher mode and padding method.

The input *"DES/ECB/PKCS5Padding"* configures the Cipher object to be used with the DES algorithm in ECB mode. This means that when the input data is larger than the block size of 64 bits, it will be divided into smaller blocks that are padded using the *"PKCS5Padding"* method if necessary.

The ECB mode of operation is used to specify how multiple blocks of data are handled by the encryption algorithm. ECB stands for *'electronic codebook'* – when using ECB mode, identical input blocks always encrypt to the same output block.

After the Cipher object has been created, it must be configured to work in either encryption or decryption mode by using the following method:

```
desCipher.init(mode, desKey);
```

The mode should be specified as *Cipher.ENCRYPT_MODE* for encryption and *Cipher.DECRYPT_MODE* for decryption.

Perform the cryptographic operation

Once the Cipher object has been configured, the actual encryption or decryption operation (depending on how the object was configured) can be performed by calling the *doFinal()* method:

```
desCipher.doFinal();
```

Note that the method takes in a byte array containing the plaintext as input, and returns a byte array containing the ciphertext as output. If your plaintext input is stored in a string, you can convert it to a byte array using the *getBytes()* method before passing it to the *doFinal()* method. To inspect the ciphertext output, you can convert it to a printable string using the *DatatypeConverter.printBase64Binary()* method. This converts the byte array to a string using the *Base64* encoding.

Task

Complete the file *DesTextStartingCode.java* so that it can encrypt an input text file using DES. Use your program to encrypt the provided files (*smallFile.txt* and *largeFile.txt*) and answer the following questions:

Question 1 (2pt): Try to print to your screen the content of the input files, i.e., the plaintexts, using *System.out.println()*. What do you see? Are the files printable?

Question 2 (3pt): Store the output ciphertext (in byte[] format) to a variable, say cipherBytes. Try to print the ciphertext of the smaller file using *System.out.println(new String(cipherBytes))*. What do you see? Is it printable?

Question 3 (3pt): Now convert the ciphertext in Question 2 into Base64 format and print it to the screen. Is the Base64 encoded data generally printable?

Question 4 (3pt): Is Base64 encoding a cryptographic operation? Why or why not?

Question 5 (3pt): Print out the decrypted ciphertext for the small file. Is the output the same as the output for question 1?

Question 6 (4pt): Compare the lengths of the encryption result (in byte[] format) for *smallFile.txt* and *largeFile.txt*. Does a larger file give a larger encrypted byte array? Why?

*Note: "*printable" (in Questions 1 - 3) means human-readable (e.g., they are English letters or digits).

Part 2: Symmetric key encryption for an image file

In the previous task, we used DES in ECB mode to encrypt a text file. In this task, we will use DES to encrypt an image file and vary the cipher mode used to observe any effects on the encryption.

Task

Complete the file *ImageEncryptionStartingCode.java* to encrypt the input .bmp image file using DES in ECB mode. You will need to specify the parameter *"DES/ECB/PKCS5Padding"* for creating your instance of the Cipher object.

Note: Your encrypted file should also be in .bmp format. Please ensure that your encrypted .bmp file can be opened using an image viewer.

Question 1 (4pt): Compare the original image with the encrypted image. What similarities between them do you observe? Can you identify the original image from the encrypted one?

Question 2 (5pt): Why do those similarities exist? Explain the reason based on what you find out about how the ECB mode works.

Question 3 (8pt): Now try to encrypt the image using the CBC mode instead (i.e., by specifying "DES/CBC/PKCS5Padding"). Compare the result with that obtained using ECB mode). What differences do you observe? Explain the differences based on what you find out about how CBC mode works.

Question 4 (15pt): Do you observe any issue with image obtained from CBC mode encryption of “SUTD.bmp”? What is the reason of such observation? Can you explain and try on what would be result if data were taken from bottom to top along the columns of image? Can you try your new approach on ‘’traingle.bmp” and comment on observation?

Part 3: Signed message digests

In NS Module 3, we learned how a signed message digest could be used to guarantee the integrity of a message. Signing the digest instead of the message itself gives much better efficiency. In the final task, we will use JCE to create and sign a message digest:

1. Create message digest
    a. Create a *MessageDigest* object
    b. Update the *MessageDigest* object with an input byte stream
    c. Compute the digest of the byte stream
2. Sign message digest
    a. Generate an RSA key pair using a *KeyPairGenerator* object instance
    b. Create and configure a Cipher object for use with RSA
    c. Use the *doFinal()* method to sign the digest

Create a MessageDigest object

A *MessageDigest* object can obtained by using the *getInstance()* method of the *MessageDigest* class:

```
MessageDigest md = MessageDigest.getInstance("MD5");
```

The *getInstance()* method takes in a parameter specifying the hash function to be used for creating the message digest. In this lab, we will use the MD5 function; other valid algorithms include SHA-1 and SHA-256.

Update MessageDigest object

After creating the *MessageDigest* object, you'll need to supply it with input data, by using the object's *update()* method. Note that the input data should be specified as a byte array.

```
md.update(input);
```

Compute digest

Once you have updated the *MessageDigest* object, you can use the *digest()* method to compute the stream's digest as output:

```
byte[] digest = md.digest();
```

Sign message digest

Generate RSA key pair

To generate an RSA key pair, we we will use the *KeyPairGenerator* class. The *generateKeyPair()* method returns a *KeyPair* object, from which the public and private keys can be extracted:

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA"); keyGen.initialize(1024);
KeyPair keyPair = keyGen.generateKeyPair(); Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

Configure cipher for use with RSA

To sign a message, we will make use of RSA encryption using the private key. The steps for initializing the cipher object for RSA are similar to the steps for initializing it for DES:

```
Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
rsaCipher.init(Cipher.ENCRYPT_MODE, privateKey);
```

Task

Complete the file *DigitalSignatureStartingCode.java* so that it can generate a signed message digest of an input file.

Apply your program to the provided text files (*smallFile.txt*, *largeFile.txt*) and answer the following questions:

Question 1 (7pt): What are the sizes of the message digests that you created for the two different files? Are they the same or different?

Question 2 (8pt): Compare the sizes of the signed message digests (in byte[] format) for *smallFile.txt* and *largeFile.txt*. Does a larger file size give a longer signed message digest? Why or why not?