Symmetric Key Encryption and Message Digest

50.005 Computer System Engineering

Due date: 08 Apr 08:30 AM (Week 11)

Natalie Agus
INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

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.

Starter Code

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.

Download the starter code:
`git clone https://github.com/natalieagus/50005Lab5.git`

There's no `makefile` for you. By now, you should be able to write your own makefile to make compilation more convenient.

UPDATE 05/04/2020: the file name and the class name have been modified to match.

Submission

The total marks for this Lab is 50 pts.
Make sure your Java code compiles properly for all 3 parts [10pts] and answer the questions in this sheet [40pts].

Zip all the following:

1. Pdf export of this sheet and your answers (as usual, fill in the spaces denoted in blue).
2. Your Java source codes for the three tasks (3 scripts in total). **Don't change the script names.**
3. The encrypted images (ecb.bmp, cbc.bmp, triangle_new.bmp) for the second task. Name it properly!

Upload to `@csesubmitbot` telegram bot using the command `/submitlab5`
CHECK your submission by using the command `/checksubmission`

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 be 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

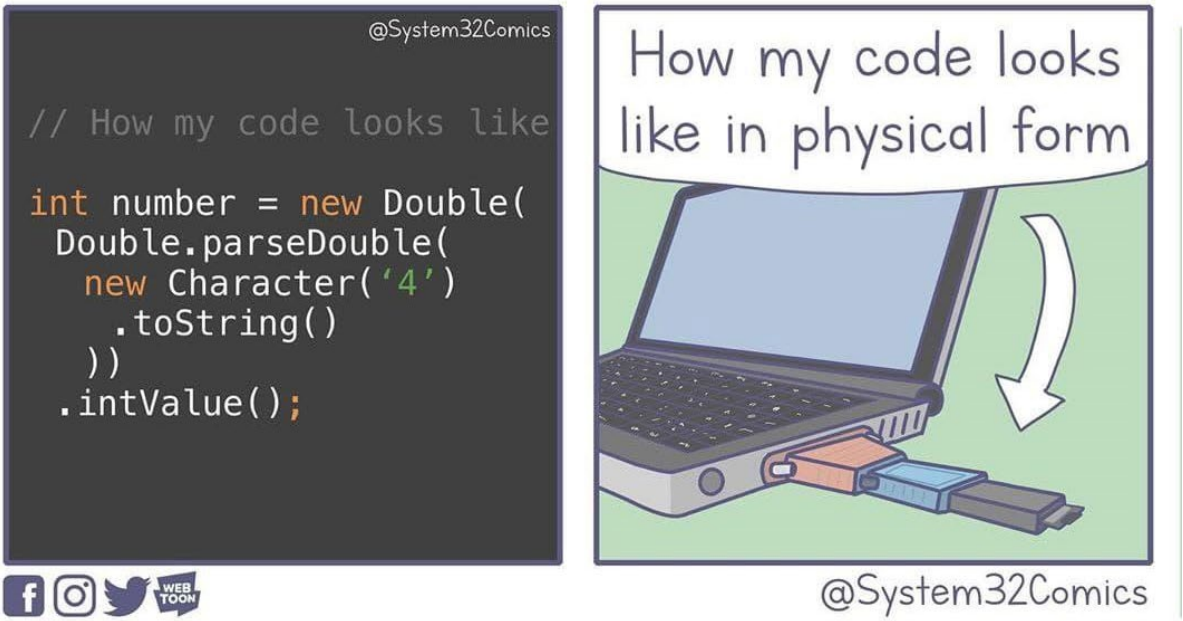
Once the `Cipher` object has been configured, the actual encryption or decryption operation (depending on how the object was configured) can be performed by calling the `doFinal()` method:

```
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:

```
String base64format =
Base64.getEncoder().encodeToString(encryptedBytesArray)
```

Kinda long winded, but we gotta do what we gotta do to print bytecode out.



Tasks for Part 1[15pt]

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

Question 1 (1pt): 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 and human readable?
Your answer: I see 2 pieces of writing. Both files are printable and human readable.

Question 2 (1pt): 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))`.

Describe what you see, is it printable? Is it human readable?
Your answer: Printing the ciphertext in this manner prints seemingly garbage characters. The ciphertext is neither printable nor human readable.

Question 3 (3pt): Now convert the ciphertext in Question 2 into `Base64` format and print it to the screen. Describe this output with comparison to question (2). What changed?
Your answer: The converted ciphertext now fully consists of ASCII characters that are printable and human readable.

Question 4 (3pt): Is `Base64` encoding a cryptographic operation? Why or why not?
Your answer: No, it is not. The encoding/decoding operation is well-known, and it does not involve the usage of a secret key. Thus any Base64-encoded message fails to hide its contents, and thus Base64 encoding does not provide confidentiality. As such Base64 encoding is not a cryptographic operation.

Question 5 (3pt): Print out the decrypted ciphertext for the small file. Is the output the same as the output for question 1?
Your answer: Yes, it is.

Question 6 (4pt): Compare the lengths of the encryption result (in `byte[]` format) for shorttext.txt and longtext.txt. Does a larger file give a larger encrypted byte array? Why?
Your answer: Yes. DES is a block cipher that encrypts input data partitioned into fixed-size blocks and produces 1 output block for each input block. With a larger file, more blocks need to be encrypted and this results in more output blocks, which translates to a larger encrypted byte array.

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 for part 2 [20pt]

Complete the file `DesImageSolution.java` to encrypt the input file, a .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 any image viewer you have in your computer.**

Question 1 (4pt): Compare the original image with the encrypted image. List at least 2 similarities and 1 difference. Also, can you identify the original image from the encrypted one?

Your answer: Both images share the same general shape of the name “SUTD” as well as positions of the 2 subheaders, while the encrypted image mostly has different colours for each pixel that is also present in the original image. Also, yes, I would usually be able to identify the original image from the encrypted one from memory.

Question 2 (3pt): Why do those similarities that you describe in the above question exist? Explain the reason based on what you find out about how the ECB mode works. Your answer here should be as concise as possible. Remember, this is a 5pt question.

Your answer: In ECB, each input data block is encrypted independently of other blocks. Thus any 2 blocks (in our case, any 2 2x1 tiles of pixels in the image grid) sharing the same 64-bit content would be encrypted to form the same 64-bit encrypted output.

This causes the output image to contain patterns reflecting those of pixels at the same positions in the original image, allowing us to make out the “SUTD” letters (most of which share the same pattern within the letters) and note the areas for the 2 subheaders (whose patterns are distinct from those of the empty areas in the original image) in the encrypted image, the latter by identifying areas without any element of interest.

Question 3 (6pt)**:** 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). State 2 differences that you observe. For each of the differences, explain its cause based on what you find out about how CBC mode works. **Your answer should refer to ecb.bmp output that you produce.**
Your answer: The CBC image generally is harder to identify. Some differences are:

(1) In the CBC image, the SUTD letters are not recognizable as compared to in the ECB image where they can be easily recognized.

This is because CBC encrypts every subsequent input data block (after the first one) using an XOR operation with the previous block's ciphertext.

This means that each output block does not necessarily reflect the patterns in the corresponding input block used for encryption (except for the first block to be encrypted), and the result in this case is that the SUTD letter patterns are distorted and become unrecognizable in the output image.

(2) In the CBC image, we can no longer identify the positions of the subheaders compared to in the ECB image.

This can be explained using the same reasoning in (1), with the addition that since our method of encryption involves going down the image as we iterate through blocks, we can expect the subheaders section to be even more distorted than the SUTD name section in the CBC image as the distortion accumulates from encrypting the blocks containing the SUTD letters first.

Question 4 (3pt): Do you observe any security issue with image obtained from CBC mode encryption of "`SUTD.bmp`"? What is the reason for such an issue to surface?
Your answer: Compared to the ECB mode of encryption, the CBC mode of encryption introduces a risk of greater loss of integrity of the original image data.

As previous blocks' ciphertexts are used to encrypt subsequent blocks, this means that corruption of data in any preceding block (eg. from file transmission) would result in incorrect decryption of the subsequent blocks, hence resulting in even more corruption of the original image than with ECB, where any corrupted data only affects the block(s) where said data is located.

Thus, CBC introduces a risk of greater loss of integrity.

Question 5 (4pt): Can you explain and try on what would be the result if the data were to be taken from bottom to top along the columns of the image? (As opposed to top to bottom). Can you try your new approach on "`triangle.bmp`" and comment on observation? Name the resulting image as `triangle_new.bmp`.
Your answer: The lower rows of the encrypted image would reflect the original image more accurately, and as we go up the columns the encrypted image would reflect the original less and less accurately and generally be more distorted.

Encrypting "triangle.bmp" in this manner produces an image where the lower section of the image below the triangle share the same patterns along the image's width. Additionally, the image shows significant distortions in the areas where there would be an empty black space above the triangle in the original image, which matches my predictions in the previous paragraph.

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 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 for Part 3 [5pt]

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

Apply your program to the provided text files (shorttext.txt, longtext.txt) and answer the following questions:

Question 1 (2pt): What are the sizes in bytes of the message digests that you created for the two different files?
Your answer: The size of the digest for both files is 16 bytes.

Question 2 (3pt): Compare the sizes of the signed message digests (in `byte[] encryptedBytes = eCipher.doFinal(data.getBytes());` format) for shorttext.txt and longtext.txt. Does a larger file size give a longer signed message digest? Why or why not? Explain your answer concisely.
Your answer: Both signed message digests are 128 bytes long. No, a larger file size does not give a longer signed message digest. MD5 always outputs a 128-bit hash regardless of input length, by dividing said input into 512-bit blocks (with padding if needed) and using said blocks to sequentially modify a 128-bit state. As a result the length of the message digest inputted to the encryption process is fixed, and so the resulting signed message digest always has the same length regardless of file size.

Note also that the RSA encryption process always returns a value as long as the key used, due to the modulo operation used in the process which is done with respect to the n-value in the key (whose length is the key size), thus even if it was not the case that the digests were of the same length, the encrypted digests would still be of equal lengths.