Symmetric Key Encryption and Message Digest

50.005 Computer System Engineering

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

Overview
- Learning objectives
- Starter Code
- Submission

Part 1: Symmetric key encryption for a text file
- Generate key for DES
- Create and configure Cipher object
- Perform the cryptographic operation
- Tasks for Part 1 [15pt]

Part 2: Symmetric key encryption for an image file
- Task for Part 2 [20pt]

Part 3: Signed message digests
- Create a MessageDigest object
- Update MessageDigest object
- Compute digest
- Sign message digest
  - Generate RSA key pair
  - Configure cipher for use with RSA
- Task for Part 3 [5pt]

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 are 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 and cbc.bmp) for the second task, and (triangle_new.bmp) for the third 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

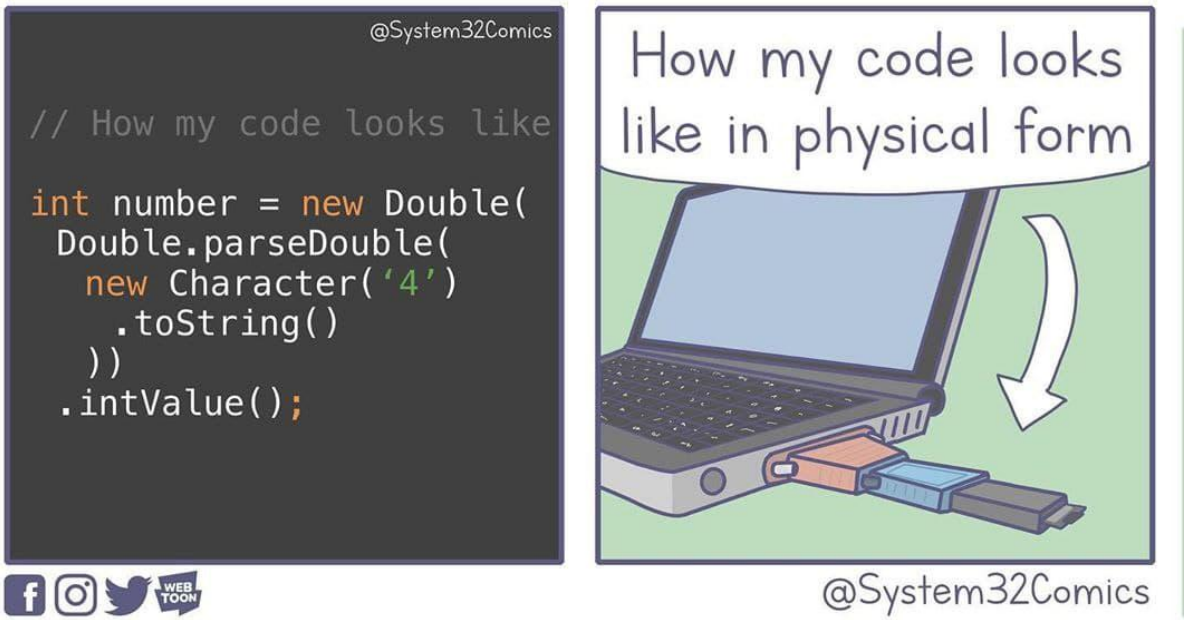
Once the `Cipher` object has been configured, the actual encryption or decryption operation (depending on how the object was configured) can be performed by calling the `doFinal()` method:

```
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: We were able to see the content of the input files in plain text. Yes, the 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: While some parts of the byte array are printable, not all of the byte array is printable as its string representation since it contains non-printable ASCII

characters (the number of printed characters does not correspond to the byte array's length). As such, the output encrypted byte array is definitely not 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 whole Base64 representation is printable to the terminal/console screen instead of only a few ASCII-printable characters. The length of the Base64 ciphertext is longer than the length of the encrypted byte array.

Question 4 (3pt): Is `Base64` encoding a cryptographic operation? Why or why not?
Your answer: No, it is not a cryptographic operation. The entire encoding algorithm is simple and well-known, and it does not need to have any secret keys for operation. All you need to know is the algorithm itself (since it technically already has a pre-set/fixed "key" publicly known by everyone) to encode or decode any messages. Base64 encoding is simply a method to represent a message in another format for the sake of ASCII-friendly environments. If Base64 encoding is to be used as a cryptographic encryption method to protect messages, it needs to be modified and mutated so as to implement and incorporate some form of secret keys.

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 the same (as it should be).

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, the larger file gives a larger encrypted byte array (and hence a longer Base64 ciphertext). This is due to the nature of the DES algorithm. The DES algorithm uses a 56-bit key to encrypt 64-bit blocks of plaintext input through repeated rounds of processing. Since the algorithm goes through the plaintext input in an increment of 64-bit blocks at a time, the larger the plaintext input, the larger the encryption result will inevitably be.

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:

- **Similarities:**
  - **The outline of the SUTD main logo's lettering is clearly visible (as well as the outline of a few letters of the subtext and subtitle).**
  - **The background is of a uniform colour.**
- **Difference:**
  - **The colour of each pixel in the encrypted image is different compared to the original image.**

A significant portion of the original image can be identified from the encrypted image, albeit of a different colour (from black to purple, in our case). While most of the subtext and the subtitle are not that clearly visible, this clearly demonstrates a weakness of the ECB mode.

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 mode, each plaintext block is independently encrypted into its corresponding separate ciphertext block. Since identical plaintext blocks (pixels, in

our case) will be encrypted into identical ciphertext blocks (if using the same key, which we do), the ECB mode is not able to hide data patterns in the original message that well. Hence, large areas of uniform colour in the original bitmap image will be encrypted to large areas of a different but uniform colour as well in the encrypted bitmap image, allowing the pattern of the original image to be discerned.

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:

- **Differences:**
    - **The outline of the SUTD main logo is not clearly visible anymore in CBC mode (i.e., it has been pseudo-randomized) instead of the SUTD logo outline with purple colour in ECB mode, and the subtext and the subtitle parts are encrypted to effectively near to random noise. This is because the CBC mode does not simply naively encrypt the same pixel of the same colour to the same different colour in the encrypted state due to the chaining process. The encrypted value of each pixel in a column also depends on the values of each pixel above it.**
    - **The background of the image has a different colour for each row (instead of the same yellow-ish colour for every background pixel in ECB mode). This is because for ECB mode, each pixel is independent of each other, while in CBC mode, each column of pixels is independent of each other. As such, in CBC mode, it forms these streaks of the same colour for each row.**

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: The encrypted value of each pixel in a column depends on the values of the pixels above said pixel in that specific column. This is because the CBC mode chains and propagates the encryption of each ciphertext block, and as such, it still encrypts the same "chain of plaintext blocks" to the same "chain of ciphertext blocks" (if using the same key, which we do). As such, some level of "inference" of the original image data contents can still be conducted.

For example, the columns with white background originally will output the same "pattern" of colour, while those pixels with originally different colours will cause the resulting encrypted pixels to differ from the rest. Another example would be the symmetric upper parts of the letters S and D in the SUTD main logo. This can thus leak some form of information about the original message (especially if the attacker has some initial metadata about the image), especially if the original image has a high degree of similarity between each column.

In reality, CBC mode is vulnerable to a padding oracle attack, whereby attackers prepend some form of explicit initialization vectors to the whole chain. They could then perform some deciphering algorithm to get some information out of the encrypted form of the original message content.

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: In the original image, there is a white triangle on a black background. In the normal CBC mode, the black background colour will be converted into horizontal streaks and rows of colours, while for certain columns, starting from the transition from the black background to the outline of the white triangle, the image becomes pseudo-random noise all the way to the bottom of the encrypted image. This creates a triangular (or more accurately, a pentagonal) outline in the encrypted image similar to the original image. However, if the data were taken from bottom to top along the columns instead, the triangular (or pentagonal) outline would be flipped against the horizontal axis (thus becomes vertically mirrored) and it would be filled with horizontal streaks of colour, while the pseudo-random noise part/component of the image would start after the triangular part ends for those columns.

In fact, by combining the information obtained from these 2 images, an attacker could figure out a general outline of the original image.

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 original MD5 message digests are 16 bytes long for both files, whereas the encrypted/signed message digests are 128 bytes long for both files as well.

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: No, a larger file size does not give a longer signed message digest. This is because the MD5 hashing function is a one-way mathematical algorithm that processes a variable length of input and outputs a fixed length output of 16 bytes. Since the input to the RSA encryption algorithm is of the same length, the output in the form of the signed message digests will be of the same length.