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

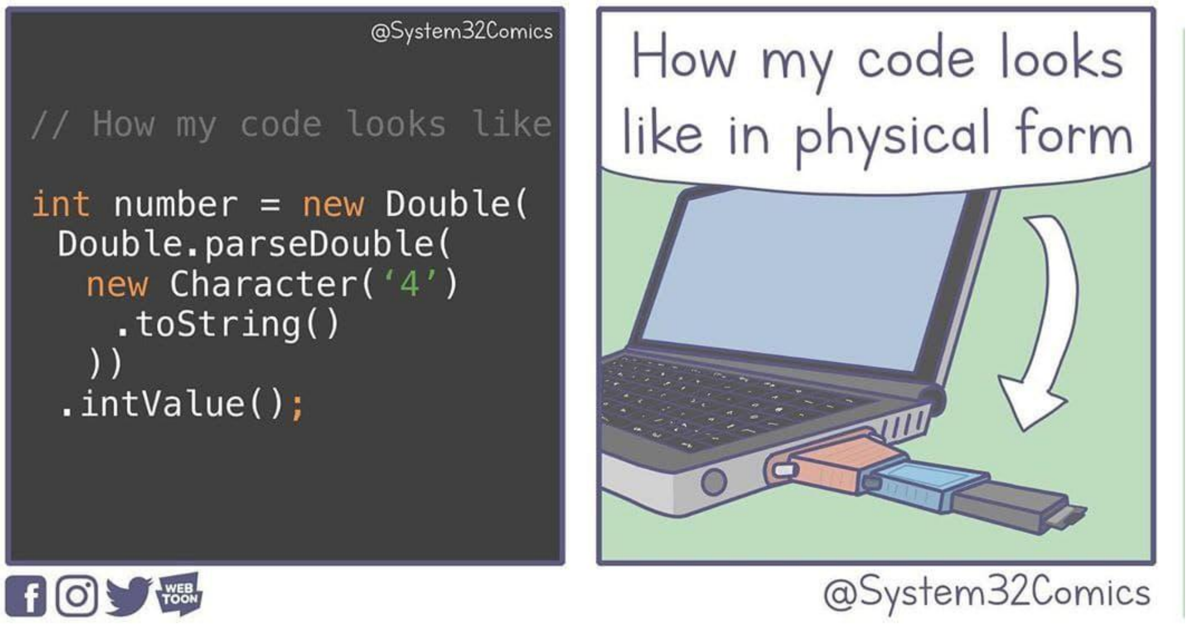
Once the `Cipher` object has been configured, the actual encryption or decryption operation (depending on how the object was configured) can be performed by calling the `doFinal()` method:

```
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 the original texts of the file in plaintext. 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:

Yes, it is printable but not all as some are non ASCII. Hence, the inability of all to be converted to string form. The 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 length Base64 shows way longer print than the ciphertext length in Question 2.
The previous output had many characters which could not even be displayed by the terminal due to the fact that some of the characters in the original byte array were not in ASCII. Also, many blank spaces separated the lines of characters. After converting the bytes into Base64 String format, the output is one long string, with all displayable ASCII characters. Before, the raw byte output was not in the ASCII format, thus some of the characters could not be converted to string and therefore, could not be printed out. By using Base64 encoding, we convert the raw bytes into a String containing ASCII chars that can be properly read and displayed.

Question 4 (3pt): Is `Base64` encoding a cryptographic operation? Why or why not?
Your answer:
No, Base64 encoding is not a cryptographic operation. Base64 has an encoding operation and to decode Base64 is already known by everyone on how to decode it hence there is no point in having the secret keys for performing the encryption of such an operation. Thus, Base64 encoding violates a few security properties of what cryptography aims to address. One of which is confidentiality. Base64 is used to encode binary data into textfiles like HTML files.

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, the decrypted ciphertext file is the same as that of question 1.

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:
The length of the byte array in longtext.txt is 17360 while that of the byte array in shorttext.txt is 1480.
Yes, a larger file gives a larger encrypted byte array. Based on the DES algorithm, there are repeated rounds of using the 56-bit key to encode 64-bit of input in the respective textfiles each round. This adds on for repeated rounds of processing. Since the processing is incremental, the larger the input size, more to encode and hence, the larger the textfile the longer the 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:

Below are the screenshot photos of the original and the encrypted image.

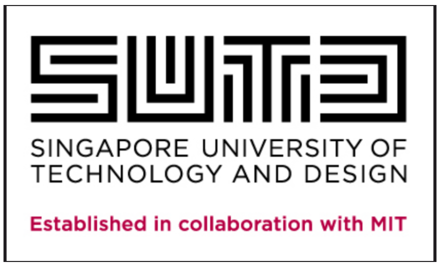
Two similarities:

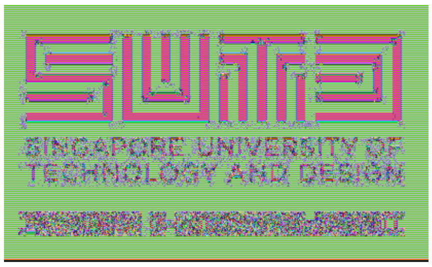
a. **The SUTD logo is visible and can be seen clearly in both images.**
b. **The Colour background image is rather consistent for both images.**

One difference:

1. **The 'established in collaboration with MIT' words cannot be clearly seen in the encrypted image as compared to the original image.**

Original Image: **Encrypted image:**





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:

ECB- Electronic Code Book. In ECB, chaining is used. Each plaintext block is encrypted into its separate corresponding ciphertext block. Identical plaintext blocks (pixels) are encrypted into the same blocks with the same key. The ECB mode is

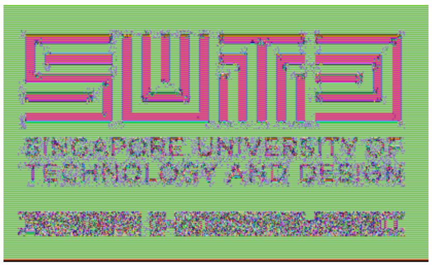
unable to hide the original data patterns in the original SUTD image well in the new encrypted image. Hence, such similarity exists. The difference stated above could be due to the fact the bottom text blocks are small in size. Hence, the ciphertext block in those areas after encrypting the original could show some subtle differences that are different from that of the original image.

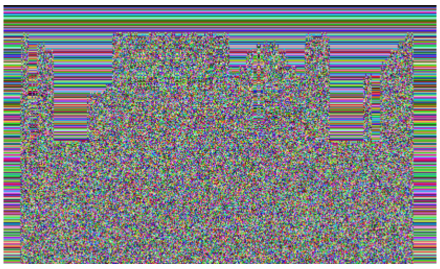
Question 3 (6pt)**:** Now try to encrypt the image using the CBC mode instead (i.e., by specifying "`DES/CBC/PKCS5Padding`"). Compare the result with that obtained using ECB mode). State 2 differences that you observe. For each of the differences, explain its cause based on what you find out about how CBC mode works. **Your answer should refer to ecb.bmp output that you produce.**

Your answer:

CBC - Cipher Block Chaining mode. In this mode, each subsequent block of input is XORed with the previous ciphertext block before encryption. Due to the XOR process, the same block will not result in identical ciphertext being produced.

Difference	**Explanation**
The SUTD logo in cbc.bmp cannot be seen clearly unlike in ecb.bmp.	**Based on the mechanism briefly explained above, the previous ciphertext block is used as a XOR input to the next block. Thus, initial input that is different from that used in ECB where the image pixel blocks is encrypted directly from the original SUTD.bmp image.**
The background colour in ecb.bmp has the same colour of green that is consistent surrounding the SUTD logo but the cbc.bmp which has different colour in each of the background pixel.	**In ECB, the pixels in the image are independent of each other in blocks while in that of CBC, the pixel columns are independent of each other. Hence, only each row has different colour**





The left image above is the image encrypted in ECB mode while that of the right image is in CBC mode.

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:

Yes, there are some security issues with the image obtained from the CBC mode encryption of "SUTD.bmp". The encrypted value of each pixel in the specific column depends on that of the values in the above said pixel in the specific said column. The CBC mode chains and propagates the encryption of each ciphertext block. In using symmetric key, the same chain of plaintext blocks to the same chain of ciphertext blocks (almost like substitution). Hence, such patterns can be easily spotted from the original image contents. One example of an attack that CBC is vulnerable to padding oracle attack. A single byte modification can make a corresponding change in tee

CBC encryption operation. Attackers can then perform some deciphering algorithm to get some message out of the encrypted content to deduce the original 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 the black background. In CBC mode, the black background is converted into horizontal streaks of colour, for certain columns, starting from the transition from the black background to the outline of the white triangle, the image shows square like pixels. This creates a triangular outline as compared to the original image. Since the data is taken from the bottom to the top from the original image, the resulting image byte array is flipped. Hence, the original image orientation is flipped along the horizontal axis of the image from the tip of the triangle to the bottom of the resulting image. This creates a triangle outline same as the original image. The areas of the image where we observed the similar horizontal strips are areas where, if we look at the original image, we can find a similar pattern from the bottom upwards. For example, the black vertical strips on the side of the image, and the white pixels in the triangle after the black pixels on the bottom of the 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:
Both files produce a original digest of 16 bytes in length. The encrypted /signed message digests for both files are 128 bytes in length.

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, larger file does not give longer signed digest. This is because MD5 hashing function is only one way that allows different length of inputs but gives a fixed length output of 16 bytes. Input to RSA is of the same length. Hence, output of message digests is of same length.