CS3235 - Laboratory #2 for weeks 7,8 (Sept-Oct, 2005)

This laboratory may be done in a group. The size of your group is up to you, and could be 1,2,3 or 4 group members. When you feel that your software is ready to be demonstrated, ask Pradeep to assess you. There is a booking sheet available in the lab. He will ask you a few questions, and mark the assessment sheet. At the same time, you must hand him your lab writeup. The assessment should be very easy and fast.

1 The lab

In this project, you'll conduct timing measurements for various cryptographic primitives and get an intuitive feel for the time complexity of various useful crypto operations. We'll use Java programs for our measurements and in the process also gain an understanding of a subset of the Java crypto API. Once you understand how one instance of a class of operations is coded in Java, you'll be able to easily adapt the code to write other instances of the same class.

Having a coarse understanding of the time complexity of various cryptographic operations can be useful in other contexts. For example, it can be useful in estimating an upper bound on the rate at which a larger operation that encapsulates simpler cryptographic primitives can be performed. Suppose for example that the computer manufacturer that you work for wants to implement the SSL protocol for sale with their computer. They have a choice to write the protocol in either Java or C. A Java implementation might be desirable for portability, modularity etc., while a C based implementation might be desirable for just raw speed. Your competitor is selling a large number of their machines because they are advertizing that their machine can do 20 SSL transactions per second. This is important to a business because higher throughput of the ordering system often translates to higher earnings. You as the security guru in the company have to determine whether to write the code in Java or C without investing millions of dollars in implementing both options and selecting the faster one. How would you go about it?

We'll provide sample code for generating checksums (cryptographic hash), and performing symmetric key encryption. You will study the code, use the JDK1.5 class hierarchy (javadocs) to understand what the various classes are doing, and adapt the code to do the same operation with different parameters and take timing measurements. We'll start with hash functions, then consider symmetric encryption, and finally asymmetric encryption.

1.1 Hash functions

A hash function maps an arbitrary sized input into a fixed sized output and a good hash function has several properties among which is that two different inputs generate different hash values with a high probability. Some examples of hash functions are MD5 and SHA-1. On a Unix system, there usually exist programs `md5sum` and `sha1sum` that compute the MD5 and SHA-1 checksums respectively, given a file as an argument. To compute the MD5 checksum of a byte sequence in Java, the following code can be used:

```
public static byte[] md5(InputStream is)
    throws IOException, NoSuchAlgorithmException
{
    MessageDigest md = MessageDigest.getInstance("MD5");
    byte[] buffer = new byte[1024];
    int b_read;
    b_read = is.read(buffer);
    while(b_read > 0)   {
        md.update(buffer, 0, b_read);
        b_read = is.read(buffer);
    }
    byte[] digest = md.digest();
    return digest;
}
```

The key idea is to instantiate an instance of `MessageDigest` and update it with the data whose checksum needs to be determined. Once all the data has been updated, the `digest` method can be called which returns the checksum value. Wrap this function inside a Java program called `md5sum` (executed as java md5sum) that generates the MD5 checksum of its argument files just like the Unix version of `md5sum`. Write a similar program called `sha1sum`. Compute the checksum of a large file using both programs and find the time to hash per byte. Take the average of five runs.

1.2 Symmetric encryption

For symmetric encryption, you need to know the cipher to use, the chaining mode that the cipher will be used in, an initilization vector suitable for the chaining mode, and the padding type to be applied to the data to ensure that it's a multiple of the cipher's block size. In the code below, we use DES in CBC mode with PKCS5 padding.

```
public static byte[] desEncrypt(byte[] data, byte[] key, byte[] iv)
    throws NoSuchAlgorithmException, NoSuchPaddingException,
        InvalidKeyException, IllegalBlockSizeException,
        BadPaddingException, InvalidAlgorithmParameterException
{
    Cipher cipher;
    cipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
    SecretKeySpec desKey = new SecretKeySpec(key, "DES");
    IvParameterSpec IV = new IvParameterSpec(iv,0,8);
    cipher.init(Cipher.ENCRYPT_MODE, desKey, IV);
    return cipher.doFinal(data);
}
```

A DES key is constructed using `SecretKeySpec` (after all it's a secret key algorithm!) and the initialization vector, `IV` is constructed with an `IvParameterSpec`. An `IV` randomizes the first block even when the plain text for the first block is one. This prevents the adversary from obtaining one (m,c) pair. For DES, `IV` is 8 bytes long, and its key is also 8 bytes long. This means that in the code above, `IV` is `byte[8]`, and so is the key. The steps needed to do symmetric encryption using the Java API are similar to those for generating checksums. You first create the correct instance of a `Cipher`, initialize it appropriately, and then perform the actual encryption using `update` (not used in the code above) and `doFinal`. For large inputs, it's better to use `update` because you can then encrypt the file in small portions rather than accumulating the whole file in a single `byte[]` array and calling `doFinal`.

Rewrite the above code to use an `InputStream` instead of `byte[]` data, and an additional parameter of type `OutputStream`, and read from `InputStream` until `EOF` and write encrypted bytes into `OutputStream`. Use `update` instead of one call to `doFinal`. Wrap this function into code that takes two file name arguments, reads from the first file and writes its encrypted version into the other.

Do the same for the AES algorithm. Find the average time to encrypt one byte with both DES and AES.

1.3 Writeup results...

Breifly describe your experiments, and summarize with a table something like this:

	Operation	Cost per byte
Hashes	MD5	?
	SHA1	?
	Others that you like...	?
Symmetric encryption	DES	?
	AES	?
	Others that you like...	?

You must also be able to answer simple questions on the Java Crypto API used in these programs.

2 Assessment

When you are ready to be assessed, fill in this sheet with your details, attach your writeup, and give it to Pradeep and ask him to assess you.

Date:

Group members:

Name	Matriculation ID

Leave this section for Pradeep to fill in:

Using standard commands:		
Using own code:		
General knowledge:		
Lab writeup:		/4
Lab code:		/4
