

Homework #1: Encryption and Security

Due Date: Wednesday, Feb. 19.

Task 1: Pencil and Paper:

In this task, you'll do some pencil-and-paper calculations involving the basics of public key encryption and estimating the security of different encryption schemes. Feel free to write a computer program to do these calculations; either show your work or include the program plus some sample output. (Have your program print out enough intermediate data that I can see what's happening.) Part a) RSA.

This question will do a simple RSA encoding and decoding. Assume that the letters of the alphabet correspond to the numbers 1-26. (A=1, b=2, ..., Z=26).

Question 1: If $p = 7$ and $q = 13$, what are the five smallest possible numbers for e ?

Question 2: If $e = 7$, what is d (its multiplicative inverse)?

Question 3: Use d to decrypt the following message, encrypted with the private key e : 3 50 14 84 60 1 6 70 12 1 6 9 50 33 25 50 70 4 47 3 50 4 47 4 6 57 47 13 47 33 33 1 84 47

Part b: Keyspaces. We know that the size of the keyspace has a huge impact on the security of a cryptosystem. distributed.net has been involved in solving an ongoing series of challenges put forth by RSA labs to test the security of the RC5 algorithm. Here are some current statistics:

Key size	Time to find it	Keyspace	Average number of keys/second
40	3.5 hours		
48	313 hours		
56	265 days		
64	1757 days		

Fill in the "Keyspace" and "Average number of keys/second" columns. Ongoing challenges include 72-bit, 80-bit, 88-bit, 96-bit and so on, up to 128 bits.

When the 64-bit key was cracked, distributed.net estimated that they were checking 1.02×10^{11} keys per second. At this rate, how many years will it take to crack the 72-bit, 80-bit and 88-bit keys? Can you characterize the growth in difficulty as key size increases? (For example, does it double? Linear growth? Exponential growth?)

Task 2: Practical encryption

In this task, you'll use GPG (Gnu Privacy Guard) to send and receive messages in an encrypted fashion. The remainder of this task is available at: <http://www.cs.usfca.edu/brooks/S03classes/cs486/hw1-task2.sig> - In order to read it, you'll need to get my public key from www.keyserver.net and use this

so decrypt the document. Instructions for using GPG can be found in the GPG manual at: <http://www.gnupg.org/gph/en/manual.html>

Task 3: Cipher-block chaining.

For this task, you'll implement a simple cipher-block chaining (CBC) algorithm. CBC is a *stream cipher*; it uses the encrypted output from one block to encode the next block. This prevents identical plaintext blocks from mapping to identical ciphertext blocks, and also helps detect attacks in which encrypted blocks are added or modified. Your code should run as a standalone program that can take plaintext from stdin and print ciphertext to stdout. There should also be a command-line flag to decrypt instead. The basic CBC algorithm works as follows:

Input: an n -bit key K , an n -bit initialization vector V , and plaintext x .
Output is the ciphertext c . $c_0 = V$.

foreach block b_1, b_2, \dots, b_t in x

$$c_i = E_k(c_{i-1}) \oplus b_i$$

To decrypt, input is V , K , and the ciphertext c . Output is the plaintext x .

$$c_0 = V.$$

foreach block b_1, b_2, \dots, b_t in c

$$x_i = b_{i-1} \oplus E_k(b_i)$$

For the encryption function, you can also use XOR.

Your code should be flexible in the length of the key and the block size.

Feel free to use whatever language you're most comfortable working with. You should turn in your source code, and some output demonstrating that it works.

Extra credit: A more interesting problem is to use the key to instead generate a pseudorandom sequence of integers. The key serves as a *seed* to generate a sequence of integers, each of which is used to encrypt/decrypt a separate block. A simple pseudorandom sequence generator is a linear congruential generator, given by the recurrence relation:

$$x_n = ax_{n-1} + b(\text{mod } m),$$

where x_0 is the input key, which seeds the sequence, and a , b , and m are parameters - in particular, m controls the range from which numbers are chosen - this should be $2^{\text{block-size}}$. The standard `rand()` function uses $a = 1103515245$, $b = 12345$, $m = 2^{32}$. Add a pseudorandom number generator into your CBC cipher and use the sequence of numbers to encrypt/decrypt each block.

Note: for optimal pseudorandom behavior, a and b must be chosen carefully; don't worry about it for this assignment. Also, linear congruential generators are actually not good ways of generating cryptographically secure random streams, since it's possible to figure out the parameters from observing the sequence of numbers produced. So don't use it in real life!

For more info on coding pseudorandom sequences, see chapter 7 of Numerical Recipes, available online at: <http://www.nr.com>

Part 4: Steganography/Digital Watermarking using jsteg

Steganography is the art/science of encoding hidden messages in otherwise innocuous-looking data. Typically, this is done by using the low-order bits in a fairly rich data representation. For example, in a 24-bit color image, the last few bits don't contribute much to the selection of color. In a lossy compression algorithm such as JPEG, these bits are noise (they don't contain any information about color), and can safely be replaced by an encoded message.

One application for steganography is the addition of a *digital watermark* to an image, sound or movie. This would allow the publisher to track its usage and help locate pirated copies.

1) Check out the JPEG on my homepage: <http://www.cs.usfca.edu/brooks/mesteg.jpg>. I've embedded a hidden message in there. To find it, you'll need to use a tool called jsteg - this is actually a patch to cjpeg and djpeg, which are a JPEG encoder and decoder, respectively. I've placed patched binaries for both of these (built for linux) on the course website. There are also some instructions for using jsteg on the same site.

Find the message in this JPEG.

2) One problem with steganography is the potential brittleness of the hidden message. It's fairly simple to alter this image so that the encoded watermark is no longer there. Suggest three ways in which this could be done. How might we try to work around this problem?

Double secret extra credit: use either your GPG key or your CBC encrypter from task 3 to create an encoded message and then use jsteg to embed the encrypted message into a JPEG. Put the JPEG up in a public place where I can extract and decrypt the message.