University of Maryland
CMSC414 — Computer and Network Security
Professor Jonathan Katz

Problem Set 1

Due by Sept. 20, 11:59 PM

The goals of this assignment are: (1) to reinforce the material on cryptography that we covered in class and (2) to help you gain familiarity with using Java to open/read/write to/from sockets, as well as with using the JCE. *Begin early!*

Also, **check the HW1 FAQ** (linked from the course webpage) before your final submission: this page will include clarifications and possible hints, format requirements for the various submitted files, and submission instructions.

1. (**Private-key encryption — 50 points.**) You will create a program to encrypt data using the block ciphers DES and AES.

   - Create a program called `DESkeygen.java`. This program should generate a random DES key and write it to a file called `DESkey.txt`. The key should be represented in hexadecimal format. (Recall that a DES key is 64 bits long, yet only 56 of these are random; the remainder are check bits. Make sure that you generate a *random* but valid DES key.)
   - You should have a second program `DESencrypt.java`. This program should read from the files `DESkey.txt` and `DESplaintext.txt`. The plaintext file will consist of ASCII text. The key file will contain a 64-bit DES key in hexadecimal format. Your program should also support one command-line flag: "-mode XXX", where XXX is one of ECB, CBC, OFB, or CFB.

     The program `DESencrypt.java` should: (1) read the plaintext file and interpret it as a sequence of bits (i.e., each character maps onto its 8-bit ASCII value). You may assume that the total number of bits in the file is a multiple of 64 (i.e., the total number of characters is a multiple of 8). It should go without saying, but do not ignore whitespace and do not treat upper- and lower-case as the same (i.e., you should be able to handle all ASCII characters); (2) encrypt the plaintext using DES, the key stored in `DESkey.txt`, and the mode of encryption designated on the command line. For the case of CBC, CFB, and OFB modes, make sure to generate a *random* IV of the appropriate length, as discussed in class; (3) output the final ciphertext to a file `DESciphertext.txt`, in hexadecimal format.
   - You should have a third program `DESdecrypt.java`. This program should read from the files `DESkey.txt` and `DESciphertext.txt`, and should also support a command-line flag exactly as for the case of encryption. This program should reverse the above procedure (when using the same key file and same mode of encryption, of course).

Repeat the above using AES, naming your programs appropriately (now, you may assume that the number of bits in the plaintext file is a multiple of 128).

After you have completed the above, please answer the following questions:

(a) How long (in bits) is the ciphertext you generate when using DES, as a function of the length of the plaintext? What about when using AES?

(b) Run two simple statistical tests on the keys generated by your `DESkeygen` and `AESkeygen` programs (I suggest counting the number of 0's and 1's in the key and checking whether the last bit of the key is 0, but you are free to use your own tests). (Note that when checking a DES key, you should ignore the check bits and only look at the random 56-bit portion of the key.) Generate 1000 keys and run your tests on these keys to generate some statistics. Describe the tests you use, and explain how the results compare to what you would expect if your key was truly generated at random.

(c) Consider encrypting the following block of text:

> The following files are available: AA1, top secret; A4, top secret; B5, unclassified; Iraq.txt, top secret; DC, top secret; stop

(this text contains exactly 128 ASCII characters). What could an eavesdropping adversary learn if this text were encrypted using ECB mode? What if it were encrypted using CBC mode? (Feel free to encrypt it yourself to help answer this question.)

(d) Repeat the experiments from part (b) on ciphertexts rather than keys, using as your plaintext a file consisting of 64 A's. (You will generate 1000 keys and encrypt this plaintext with these keys, and use the resulting ciphertexts to generate statistics.) As your statistical test, tabulate the number of occurrences of each "triple" of bits (i.e., "000", "001", ..., "111"). Run the experiment for each of the modes of encryption, for both DES and AES. Explain how the results compare to what you would expect if the ciphertext were truly random. Do your results indicate weaknesses in any of the encryption modes? Explain.

2. (**Public-key encryption using RSA — 50 points.**) You are to implement the "basic" RSA encryption scheme from scratch. Recall, the basic RSA scheme is as follows: if the modulus N is, e.g., 1024 bits long, then a message m (with $|m| < 1024$) is encrypted by viewing m as an integer in $\mathbb{Z}_N^*$ and computing $m^e \bmod N$, where e is the public exponent. (We mentioned in class that this encryption scheme is not secure. Still, it is a useful starting point.)

First implement a key generation program `RSAgen.java`, which operates as follows:

- The program should take a command-line input, denoted here by ℓ, which determines the length of the primes p and q.
- Your program should generate two, random ℓ-bit primes p and q. You should do this as discussed in class, by generating random ℓ-bit numbers and testing them for primality. You should use the built-in Java routines for generating random numbers and for testing primality.

- Compute N, find the smallest valid value for e, and compute the appropriate value of d.
- Output the public key to a file `pk.txt`. The format should be as follows: the first line of this file should contain a decimal integer which is equal to the length of N in binary. The second line should contain the modulus N, in hexadecimal form. The third line should contain e, written as a decimal integer.
- Output the secret key to a file `sk.txt`. The first two lines of this file are exactly the same as `pk.txt`. The third line should contain d in hexadecimal form.

Also implement an encryption program `RSAenc.java`, which operates as follows:

- It should take two command-line arguments: the first specifies the file in which the public key is stored (the format will be as above), and the second specifies a file containing the message. The message will be in ASCII text.
- Your program should interpret the message as a sequence of bits (i.e., by mapping each character to its 8-bit ASCII value). You then need to "map" the plaintext to $\mathbb{Z}_N^*$ in a reversible way so that you can later recover the message upon decryption. Discuss how you perform this mapping. *For the purposes of this problem, you may assume that the modulus in the public-key file is always at least 20 bits longer than the message to be encrypted, and that the maximum modulus size you will encounter is 4096 bits.*
- Encrypt the message, and output the ciphertext to a file `ciphertext.txt`. The ciphertext should be in hexadecimal format.
- Implement this program without any calls to the JCE library other than for mathematical operations (multiplication, exponentiation) on large integers.

Finally, you should implement a decryption program `RSAdec.java` which reverses the above (i.e., takes a secret-key file and ciphertext file as input, and outputs the appropriate plaintext to a file in ASCII format). Check your programs to make sure that decryption recovers the original message.

After you have completed the above, please answer the following question:

(a) How did you perform the mapping from messages (viewed as a sequence of bits) to $\mathbb{Z}_N^*$? Is your method reversible? Recall the assumptions about the message/modulus lengths.

(b) Discuss how you would extend your encryption program so that it can encrypt messages longer than the modulus.

(c) Discuss how you might randomize your encryption algorithm, to avoid the problems with deterministic encryption that we mentioned in class. (Of course, your resulting scheme may still not be secure, but at least it will not be deterministic.) Do you need any additional assumptions about the message/modulus lengths?