Problem Set 1

CSCI3381-Cryptography

Due September 12, 2014

General Instructions

Don't be put off by the length of this document—it's actually not a long assignment! I wanted to explain everything very precisely. Good solutions to these problems do not have to be long.

Each assignment contains both pencil-and-paper problems and computer problems. For the pencil-and-paper problems, you have several options: You can type them into a text-processed document. Mathematical symbols can pose a problem, but you can use the equation editor in Microsoft Word; if you know how to use LaTeX, that's even better. (There are not many mathematical symbols in the present assignment, so this should not be an issue.) Finally, you *can* complete your assignment on paper, but that means you will have to bring the paper with you to class or put it in my mailbox before the due date. (Please write really really neatly if you do this.)

The computer problems require you to both present and discuss the result of executing your computer code, as well as the code itself. The first of these will go into your text-processed document. For the second, you will prepare a number of Python modules in files with the .py extension. (See, however, the special instructions for Problem 3 below).

All these files—the text-processed documents and the Python files, should be placed in a folder whose name is "Assignment 1" followed by your last name. Compress the folder to a zip file and submit it through Canvas.

Each of the four problems is worth 10 points; however you get 7 points extra credit for Option B of Problem 4, and 10 points extra credit for Option C.

Written Problems

1. Key-length and security against brute-force ciphertext-only attacks. Without other insight into the cryptosystem, the only thing a ciphertext-only attack can do is try out all the possible keys, decrypt the intercepted ciphertext with each key, and determine whether the decryption is plausible plaintext. This is a *brute-force* ciphertext-only attack. (Such an attack also needs to determine it has found the correct decryption, so we require some means of distinguishing real plaintext from gibberish.)

Let us suppose that we have a processor that, given the key, can perform the decryption of a 32-byte (=256 bits) ciphertext and assess its plausibility in 10^{-9} second.

(This is a realistic, if somewhat optimistic, value). Let us suppose, moreover, that the processor cost $ 500, and that you have $1,000,000 to spend. (Because this task is easily parallelized, we can have different processors working simultaneously, testing different part of the key space.) Given these assumptions, how long will it take you to recover the key if the length of the key is

(a) 40 bits? (this is the default key length for WEP encryption scheme used in many home WiFi networks–for instance, Verizon subscribers are given a router with the 40-bit key written on the label)

(b) 56 bits? (the key length used in DES (Digital Encryption Standard), the older official US government standard for encryption of non-classified material.)

(c) 88 bits? (approximately the number of bits needed to encode a permutation of the 26-letter alphabet)

(d) 128 bits? (the weaker version of AES (Advanced Encryption Standard), the replacement for DES).

Now suppose that owing to technological improvements, several years from now the processor speed increases by a factor of 10 and the cost decreases by a factor of 10. What are the answers to the above questions under these new assumptions? (Give the answer in years, or days, or hours, or seconds, whichever unit gives a more reasonable value. You can use Python to perform the calculations, or just an ordinary calculator, but in any case show the details of the computation.)

2. Known- and Chosen-Plaintext Attacks on classical cryptosystems

The attacks we described against the Caesar cipher, monoalphabetic substitution cipher, and Vigenére cipher are all ciphertext-only attacks. All these systems fall apart completely if we are able to launch a known-plaintext or chosen-plaintext attack. Describe how these attacks work (so there are six questions to answer in all.) In particular, about how many characters of known plaintext or chosen plaintext do we need to completely break the system? For the Vigenére cipher, you should assume that the key length is no more than ten characters.

Computer Problems

3. Cryptanalyis of Vigenère cipher

The code for cryptanalyzing the Vigenére cipher is demonstrated in the posted iPython notebook, and contained in the posted module vigenere.py. Use this code to decrypt the Vigenére examples given in Problems 2.14.7 and 2.14.8 of the textbook. (Also, tell what is unusual about the plaintext of Problem 7.) You can copy and paste the ciphertexts from the page posted on the website.

This is pretty much a cookbook problem: You can, but don't have to, write a Python module for this. It's enough to open vigenere.py, select Run Module from the Run Menu in IDLE, then enter the little code snippets for carrying out the cryptanalysis in the Python shell. Copy and paste these snippets into the solution you submit. When you've finished, the decryption function will return the result without spaces or punctuation, but you should also submit the answer with word separations restored.

4. Write a Python Module Performing Automated Cryptanalysis

There are three options here, of increasing levels of difficulty. You only need to hand in one of these, but you will receive more points for the more difficult options.

Option A. Automated cryptanalysis of Caesar Cipher

Write a Python module containing a function

```
caesar_crack(ciphertext)
```

whose argument is a ciphertext string (all lower-case letters) encrypted with the Caesar cipher, and that returns the likely plaintext. In other words, your program should spare you the trouble of having to inspect each of the 26 possible decryptions, and instead determine automatically which one is correct.

You should preface your code with

```
import caesarcipher
import vigenere
```

The idea is pretty simple: Instead of picking out the correct plaintext by visual inspection of 26 possibilities, call the `all_decryptions` function from `caesarcipher` (which you do by writing `caesarcipher.all_decryptions(ciphertext)`) and use the `score` function from `vigenere` to pick out the candidate plaintext with the highest score. This is really a basic programming exercise to get you used to writing Python functions.

Test your function on the example in the posted ciphertexts. Also, try to find some brief one-sentence plaintexts for which this method gives the *wrong* answer.

Option B. Autokey Cipher with Known Key Length

The Vigenère cipher, like so many things in mathematics and science, is misnamed. What Blaise de Vigenére actually described in the 16th century is a related method called an *Autokey* cipher. The idea is to take the plaintext and prepend the key, then to use this modified plaintext as the sequence of key letters to encrypt the original plaintext. So for example, if the plaintext is the first sentence of the textbook:

```
People have always had a fascination with keeping secrets from others.
```

and the key is `textbook`, then the encryption proceeds as shown below:

```
peoplehavealwayshadafascinationwithkeepingsecretsfromothers
textbookpeoplehavealwayshadafascinationwithkeepingsecretsfr
-----------------------------------------------------------------
iilimsvkkioahefscedlbaqupndtnofyqghdmscevzzogvtbfljsofxawwj
```

For your convenience, I have provided functions for autokey encryption and decryption in the file autokey.py.

I am not aware of any quick statistical method, analogous to the coincidence-counting method that we used for the Vigenére cipher, for determining the key length. But once we have the key length, essentially the same technique we used for cracking the Vigenére cipher will work for autokey encryption.

Write a function that automatically deciphers an autokey-encrypted ciphertext, given the key length. So you should write a function

```
autokey_crack(ciphertext,keylength)
```

that returns the likely plaintext. You should import both the functions from `autokey.py` and `vigenere.py` for use in your code.

Use this to decrypt the first autokey-encrypted ciphertext in the posted file.

How efficient is this method compared to a brute-force attack? To answer this question precisely, how many decryptions of individual characters does this attack perform when applied to a ciphertext of length n, and how many such decryptions are performed by an attack that tries out every key?

Option C. Autokey Cipher with Unknown Key Length

If you've gotten this far and are still looking for a challenge, write

```
autokey_crack2(ciphertext)
```

to decipher an autokey-encrypted ciphertext where the key length is unknown (assume it is no more than 10).This is a matter of repeatedly applying the solution method of Option B for different lengths, and then picking out the best one. But remember, you're supposed to do this without visual inspection of the candidate results–the program is supposed to automatically select the best answer.