CSCI3381 Cryptography-Assignment 1
Bad crypto in the wild 1: XOR Encryption in old versions of Microsoft Word

due Friday, January 28 at 11:00 PM

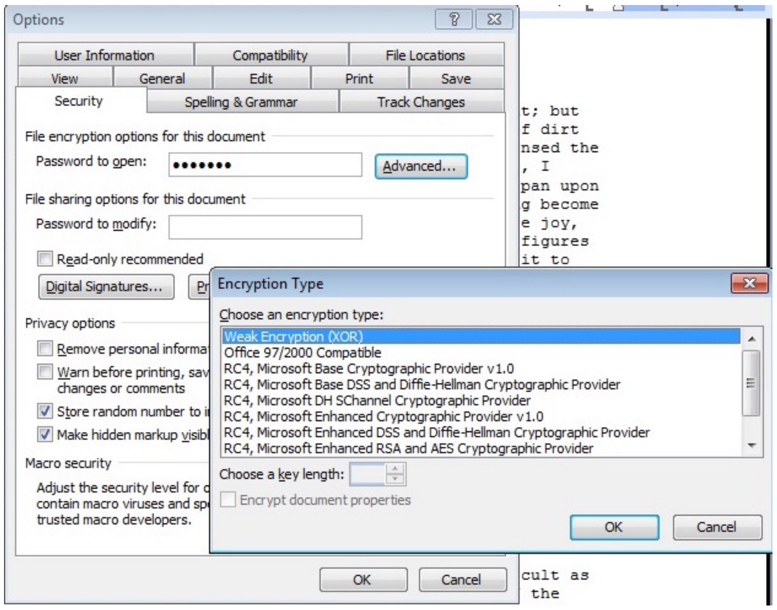
January 24, 2017

1 Overview

Many of the assignments in this course will teach the principles of cryptography by having you mount attacks on real systems (or close approximations thereof) that have been used in the recent past, or are still in use. This first assignment is devoted to a method of encryption that was available in Microsoft Office up until at least 2003. Eventually Microsoft removed the possibility of ‘protecting’ documents with this very insecure method—which is essentially the Vigenère cipher with a *known key size*. To maintain compatibility, current versions of Microsoft Word will still read documents encrypted in this fashion, provided you supply the correct password, but will no longer encrypt documents by this method. (Thanks to BC Tech Consultant Ken Porter for dusting off an old copy of MS Word so I could create the ciphertext documents for this assignment.)

In brief, the algorithm is this: A user-created password is used to generate a 16-byte key k. The plaintext contents of the document is then broken up into 16-byte chunks, and each chunk p is encrypted as the exclusive-or $c = p \oplus k$. The resulting sequence of ciphertext bytes is what is stored in the protected .doc file. Ordinarily, decryption is performed by asking the user for their password, using this to regenerate the key k, and then XORing each 16-byte chunk c of ciphertext with this key to obtain the plaintext bytes $p = c \oplus k$.

The attack you will implement does *not* recover the user password, which is not needed, but it does find the 16-byte key. The idea is the same as that in the cryptanalysis of the classic Vigenère cipher, only simpler, because we do not need to determine the key length. We simply break the ciphertext into 16 separate subtexts, and find for each subtext which of the 256 possible one-byte shifts yields the most English-y result. This should recover the entire key.



File encryption options for this document

Password to open: •••••••

File sharing options for this document

Password to modify:

☐ Read-only recommended

Privacy options

☐ Remove personal informa

☐ Warn before printing, sav changes or comments

☑ Store random number to i

☑ Make hidden markup visibl

Macro security

Adjust the security level for c contain macro viruses and sp trusted macro developers.

Choose an encryption type:

- Weak Encryption (XOR)
- Office 97/2000 Compatible
- RC4, Microsoft Base Cryptographic Provider v1.0
- RC4, Microsoft Base DSS and Diffie-Hellman Cryptographic Provider
- RC4, Microsoft DH SChannel Cryptographic Provider
- RC4, Microsoft Enhanced Cryptographic Provider v1.0
- RC4, Microsoft Enhanced DSS and Diffie-Hellman Cryptographic Provider
- RC4, Microsoft Enhanced RSA and AES Cryptographic Provider

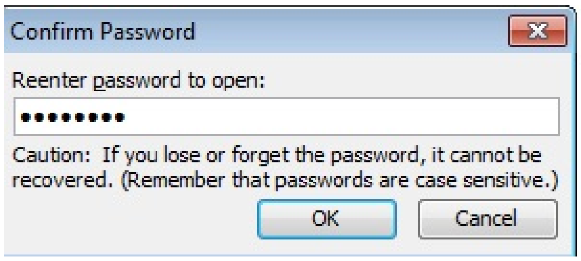
Choose a key length:

☐ Encrypt document properties

```
t; but
f dirt
nsed the
, I
pan upon
g become
e joy,
figures
it to
```

```
cult as
 the
```

Figure 1: Deliberately choosing the worst encryption method in an old version of MS Word. Originally, this was the *only option*, Incidentally, the key length choice is not available for this encryption method–the key length is always 16 bytes.



Reenter password to open:

••••••••

Caution: If you lose or forget the password, it cannot be recovered. (Remember that passwords are case sensitive.)

Figure 2: Forewarned is forearmed, but as you will show in your homework, you don’t need to know the password to read the protected file.

To get you there in small steps, we'll begin with a binary XOR version of the Caesar shift cipher, explain how to extract the document body from old MS Word files, and have you implement decryption with a known key. You'll then proceed to an attack where some of the plaintext is known, and then to the ciphertext-only attack. (One of the purposes of the assignment is to give you a good Python workout.) Note that Problem 2 is actually harder than (and independent of) 3, 4 and 5.

You will be required to submit both code that I can run without having to modify it, and written narrative telling me something about your solutions to these problems. Details on what to submit are in the last section, which you should read FIRST before tackling the problems.

By the way, the attack in Problem 6 is not a mere curiosity, useful only against 19th century cryptosystems. As we'll see later, the identical attack is effective against modern stream ciphers if, as the result of badly-designed implementation, re-use of keys is allowed.

2 The Problems

2.1 Warmup Pencil-and-paper Problem

1. Just to make sure that you understand both the XOR operation and the different ways we represent strings of bytes, compute *by hand* the exclusive-or

$$abcd \oplus cdef.$$

But wait a second—Do $abcd$ and $cdef$ denote two-byte sequences in hexadecimal? or do they denote four-byte ASCII sequences? or do they denote three-byte sequences in base 64 encoding? That's three different ways to interpret the question, and will lead to three different answers. Find all three answers, and give the result in hex.

The functions in the `conversions.py` file will answer this question for you automatically, but do the work by hand just this once. By all means, use the software tools to check if your answer is correct.

2.2 One-byte xor: a binary version of Caesar Cipher.

This is just like the original Caesar Cipher, except the plaintext is a Python string, and the key is a Python character k. Any byte can be treated as a Python character, so there are 256 possible keys. We replace the plaintext string

$$p_1 \cdots p_n$$

by the ciphertext string

$$c_1 \cdots c_n,$$

where for each i, the ciphertext character c_i is the bitwise exclusive-or $p_i \oplus k$. Here, encryption and decryption are *exactly* the same. If you use the `conversions` routines supplied on the course website, both are accomplished by the very simple snippet of code

```
import conversions
ciphertext=conversions.xor(plaintext,k*len(plaintext))
```

2. The files `ciphertext1.txt` and `ciphertext2.txt` are encryptions of brief passages of text using this method. The text is represented in base 64, so you will have to convert them to Python strings, using the routines in `conversions.py` in order to work with them.

Write a function `one_byte_xor` with a single argument that is the ciphertext represented as a Python string (*i.e., after* the conversion from base 64 to ASCII). The function should return the plaintext. This is an exhaustive-search attack, so you will decrypt using all 256 possible one-character keys. It would not be too onerous to inspect all of these and pick out the correct one, however, your *function* must do the job of picking which of the candidate decryptions is the correct plaintext.

To do this, you should follow the approach used in the cryptanalysis of the original Caesar cipher, and develop a scoring system for assessing the 'Englishiness' of a candidate decryption. Because we are XORing arbitrary bytes, and not just dealing with letters, the technique is slightly different:

- Find how many characters of the candidate plaintext are actually printable. (You can use the built-in Python string `string.printable` for this.) In the correct plaintext, all, or nearly all, of the characters are printable.

- Find how many characters are letters. (Use `string.ascii_letters` for this.) Most of the plaintext characters should be letters.

- Apply the statistical analysis method of the original Caesar cipher to determine which decryption most closely matches the letter distribution of English.

I tried solving the problem using this scheme: I gave the first criterion highest priority, the second next-highest, and the statistical analysis lowest priority. I combined these into a single numerical score, and used the key character that gave the highest score. You may find something that works better, so feel free to invent.

Your function should recover the entire plaintext, with spacing, capitalization and punctuation, not just the letters as in the Caesar cipher.

2.3 Decrypt XOR-encrypted MS Word files when you know the key

Implement XOR encryption/decryption of a Python string with a repeating 16-byte key. You should call your function

```
repeating_byte_xor(s,k)
```

where `s,k` are both strings and `len(k)` is 16. This is really just like the little code snippet illustrated above, with the slight complication that the length of the plaintext probably won't be an exact multiple of 16.

To actually apply this to protected MS Word files, you need to know a little bit about the format of these files. The format specification of Microsoft .doc files runs to gazillions of pages, but you only need to know the following things about the file structure:

- The basic file structure is: header stuff, document body, other stuff.
- The document body, whether encrypted or not, begins at byte 2560 (`a00` in hex). Keep in mind that the first byte of the file is byte 0, not 1.
- So where does the document body end, and the other stuff begin? For documents smaller than 64 kilobytes, which is the case with all the ones you will work with, the 16-bit value at bytes 540 and 541 (hex 21c and 21d) in the header stuff give this information. Two caveats: First, the value is stored in 'little-endian' format, with the least significant byte first, so that if these two bytes contain, say, 57 and 124, respectively, then the value represented is $124 \times 256 + 57$. The second caveat is that this value is the offset not from the start of the file, but from byte 512, so the other stuff in this example begins at byte $124 \times 256 + 57 + 512$.

Your job is basically to extract the document body from the Word file and combine it with the decryption function you just wrote. (To read from a file named 'myfile' in Python, all you need is

```
f=open('myfile','r')
s=f.read()
```

This sets the variable `s` to a Python string containing all the bytes of the file; then you can use the description of the file format above to extract the ciphertext.)

3. Use the function you wrote above to create a function

```
decrypt_msword(filename,k)
```

that takes the name of an XOR-encrypted MS Word file and the 16-character string `k` as arguments, and returns a string containing the decrypted plaintext. Use this to obtain the decryption of the file `MSWordXOR1.doc` posted on the course website. (Do not print the returned string with Python `print`. Even if correctly decrypted, it might contain some nonprinting characters, which can cause Python to hang. If you want to see the result displayed nicely, you can first filter for non-printing characters, replacing these by a printable symbol, and then print.) You need to know the key– in hex it is

```
b624bd2ab42a39a235a0b4a9b6a734cd
```

2.4 Time required to brute-force the result

4. If we launched a stupid brute-force attack on this system, we would try out all 16-character keys, decrypt the ciphertext with the key, and assess the Englishiness of the result. We don't need to decrypt the entire ciphertext; we will probably get a good answer by decrypting and scoring the first sixteen bytes. Calculate how long it would take *your code* for decryption and scoring to carry out this attack. (You will want to measure the amount of time your programs take to decrypt and score; use the Python function `time.time()` for this. If you didn't complete the scoring step in Problem 2, just tell how long it would take to carry out the decryptions.) Give the result in easily-graspable units of time: '10 years' is more informative than '3.15×10^9 seconds'.

2.5 Known-plaintext attack

5. You know, or strongly suspect, that the protected document `MSWordXOR2.doc` contains the words 'Computer Science' somewhere among the first fifty characters. ('Computer Science' is, conveniently, exactly sixteen characters long!) This vastly reduces the size of the possible keyspace. Use this information to decrypt the file.

You do not need to use the Englishness scoring of Problem 2: if you do the problem correctly, there will be fewer than forty keys to generate and check, and the correct answer can be picked out by inspecting all the candidate decryptions. You should write a function `decrypt_ct2()`, with no arguments, that generates all the possible keys and prints the resulting decryptions. (See the comments above about printing the decryptions—you will have to filter out the non-printable characters or Python will crash.)

2.6 Ciphertext-only attack

6. Now write a function

```
ct_only_msword(filename)
```

that returns the plaintext of an encrypted file, knowing only the ciphertext. As mentioned above, the technique is the same as the one we saw with the second phase of the Vigenére cryptanalysis: Break the text into sixteen subtexts and find the most Englishy one-byte shift for each of them. Put the sixteen bytes together, and you've got the key. Try this out on the files `MSWordXOR`*n*`.doc`, where $n = 6, 8, 9$.

You will find that in some instances, especially if the ciphertext is relatively short, the statistical analysis is not always correct. (If the ciphertext document has n characters, then each sample that we score has $n/16$ characters, and this might not be enough to accurately distinguish its properties.) This means that two or three of the sixteen characters of the key are incorrect. In this case, it is very easy to figure out by visual inpsection `which` ones are incorrect, and modify the key accordingly.

3 What to hand in

You cannot really do Problem 6 unless you've first completed Problem 2. On the other hand, if you do solve Problem 6, Problem 5 is superfluous, since we won't need the known plaintext to decrypt. Thus there are two options on this assignment: Problems 1,2,3,4,6 for full credit, or Problems 1,3,4,5 for somewhat less credit.

You will hand in a *single Python file that must be named* **FirstIntialLastName**`HW1.py`. (For example, `HStraubingHW1.py`.) This will include all the Python functions that you wrote for this assignment, including functions that must be named `one_byte_xor` (Problem 2, if you do it), `decrypt_msword` (Problem 3), `decrypt_ct2` (Problem 5, if you do it) and `ct_only_msword` (Problem 6, if you do it), with the parameter lists as described above. Your code will be run using a script that relies on these naming conventions, so deviating from them will result in your not receiving credit for the problem.

In addition, you will hand in a typed report containing some narrative about the solution for each of the problems:

For Problem 1, detail your calculations of the three exclusive-ors.

For Problem 2, present the plaintext decryptions that you found.

For Problem 3, present the plaintext of decryption of `MSWordXOR1.doc` found by your program.

For Problem 4, explain your method for determining the time, along with your calculations.

For Problem 5, explain your method for determining the key, and present the plaintext decryption of `MSWordXOR2.doc` that you found as a result.

For Problem 6, explain your method for determining the key, and present the plaintext decryptions of the sample files. You may find that for some of these files, a few of the characters of the key are incorrect, resulting in a plaintext returned by your function that is wrong in two or three out of every sixteen characters. Explain how you tweak the result to get the correct decryption.

Place both the written narrative and the .py file in a folder called **FirstIntialLastName**`HW1`, compress it, and submit it through Canvas.