CMPUT333: Assignment 1

Non-Sliding Part

Group Number 11
ZiQing Ma - 1499476
Ethan Chiu - 1621615
Shrey Mandol - 1632499

Repository:
https://github.com/E-Chiu/ZiShreyEthan

Instructions
Assuming you are on the lab machines and the respective ciphertexts are present, simply execute the program within the directory:
python3 ./partX.py
X = 1 or 2 depending on the the respective directories

Part 1

Plaintext: plaintext1.txt
And it is you-
your high resolve that sets this plague on Thebes.
The public altars and sacred hearths are fouled,
one and all, by the birds and dogs with carrion
torn from the corpse, the doomstruck son of Oedipus!
And so the gods are deaf to our prayers, they spurn
the offerings in our hands, the flame of holy flesh.
No birds cry out an omen clear and true-
they're gorged with the murdered victim's blood and fat.

Take these things to heart, my son, I warn you.
All men make mistakes, it is only human.
But once the wrong is done, a man
can turn his back on folly, misfortune too,
if he tries to make amends, however low he's fallen,
and stops his bullnecked ways. Stubbornness
brands you for stupidity-pride is a crime.
No, yield to the dead!
Never stab the fighter when he's down.
Where's the glory, killing the dead twice over?

I mean you well. I give you sound advice.
It's best to learn from a good adviser
when he speaks for your own good:
it's pure gain.

Key: key1.txt
Key in ASCII: tragedy

Process of determining the key:
We start by creating a hex dump file which is used for easy differentiation of each byte.
The key is determined by using the function get_key() in the part1.py file. This function and process_key() is essentially based on the Kasiki examination and index of coincidence methods for cracking Vigenere ciphers.

It initially creates a list of occurrences of each byte. It then goes into finding the intervals at which there are peaks, which represent positions with high occurrences of bytes. We can guess the key length by determining the greatest common denominator of peak positions. Originally, the GCD method was not used to guess the key length and multiples of the peak positions were used to find the key length. This was later changed as that was exhaustive and considered brute forcing.

With the key length, we do a frequency analysis to find occurrences of each unique byte, divide it by the total number of bytes found to get the actual frequency value of each byte. We would then find the max of that. Afterwards, as described by one of the lab hints, we determined that the "shift" was the space character as that is the most occurring ASCII character in the English language. Other common characters such E, T, A, I, O, and N were tried as well, but the resultant plaintext was unreadable. Thus, space is the character we used to map the bytes we found using the table and acquire our key. Decryption of the ciphertext is then just mapping the high and low bits of the key and cipher to get the plaintext. The "dmap" method as described in slide 3 of the lab was used to get the appropriate mapping of the plaintext bytes. One small detail to note was that using the standard write method of Python gave extra bytes for the output file (998 vs 973), but changing it to write it out in binary will provide the correct number (exact conversion).

Automation of the key finding process:
We split the ciphertext into groups of key length and number of bytes, and then find the most frequently occurring bytes of each of those parts. Using the hint provided in the labs, we would use the most frequently occurring bits to match it to the map given, and find the key byte by byte.

Source code files:
part1.py

Part 2

Plaintext:
File in accompanying files: **plaintext2 file**
The function detect_format() found the output format of the file to match the file signatures of ZIP, DOCX, PPTX, XLSX files. From the note given from one of the magic number sites provided from the lab, the file signature of the resultant plaintext matched the header of an OOXML file. We then renamed it to be a ZIP archive and extracted its contents to search for the [Content_Types].xml file. After further examination, the string, *<Override PartName="/ppt/slides/slide1.xml" ContentType="application/vnd.openxmlformats-officedocument.presentationml.slide+xml"/>*, has helped us determine that the format of the file is a PPTX file. Renaming the plaintext to *plaintext2.pptx* reveals a PowerPoint presentation titled "Software Defined Radio: State-of-the-Art & State-of-the-Future" and authored by Jon R Pawlik. One small thing to note is that Linux distros (e.g. the lab machines) will automatically identify the output plaintext as a ZIP archive after running the program to decrypt the ciphertext. While we did not exactly "exploit" the fact that the plaintext was a common file format, it was simple to use xxd to check the header of the file and cross-reference it with the provided file signature sites, as you can probably guess from the formats hardcoded in the file detect_format() function.

Key: **key2.txt**
Key in ASCII: 9exUH3gzoR5A4AVdXrP4xm5IRnqTCZypJjIQqFudThZS

Modifications or refinements made to the key finding technique in Part 1:
We used the same method from Part 1, except that non-printable ASCII characters were assumed to be NULL instead after examining the ciphertext using xxd (since the lab hints suggest that "standard frequency characteristics" are not followed) and determined that was the shift. In addition, the byte_histogram.py program provided can be used to analyze this same trend and give proof that NULL bytes were the most frequently occurring bytes. We also decided to go with a subsection of our ciphertext to create the hex dump in this case, in order to speed up the process of getting the key. Another modification we made was including a detect_format function that has the magic numbers for common file types, and checking for those values in the binary form of the plaintext.

Automation of the key finding process:
We split the ciphertext into groups of key length and number of bytes, and then find the most frequently occurring bytes of each of those parts. Using the hint provided in the labs, we'd use the most frequently occurring bits to match it to the map given, and find the key byte by byte. This is also where we made the modification of replacing non-printable ASCII characters with NULL.

Source code files:

part2.py
byte_histogram.py

Part 3

The password for all of the encodings is: 333group11

Observations of encrypted files:
file1ecb.enc: The length of the encoded file is roughly 2 times as long. The ciphertext is also heavily repeating. This is expected as ecb encryption is one to one with the plaintext and thus copies the repetitive nature of file 1.

file2ecb.enc: Similar things can be said about this file with file1ecb.enc. The length of the ciphertext is roughly 2 times longer and it is repeating because our plaintext is as well.

file3ecb.enc: This ciphertext is also around 2 times the size of the plaintext but because the plaintext of file3 was not repeating the ciphertext here is not.

file1cbc.enc: The length of the ciphertext is larger than the original file1. There doesn't seem to be any signs of repetition and we believe this is because cbc encoding does a xor with the previous block, so that a different ciphertext will be produced.

file2cbc.enc: Similar results with file1cbc.enc. This ciphertext is roughly the same size as the plaintext file and because of the way the encoding works there is no repetition.

file3cbc.enc: The size of the ciphertext is larger than the plaintext, and no patterns can be seen.

file1cfb.enc: The size of the ciphertext is quite larger than the plaintext. Because this encoding method is similar to cbc there are no patterns to be seen. The same observations can be made for the rest of the files.

file1ofb.enc: The ciphertext is longer than the plaintext and there is no repeating patterns. The same can be said for the rest of the ofb files.

Observations of decrypted error files:

file1ecberror.dec: One of the sentences is replaced with a blurb of random text, but no other sentences have been altered. There are also some random repeated bytes at the end which we believe is from padding.

file2ecberror.dec: There is a blurb of random text in the plaintext, and it cuts into another sentence. There are also some random repeated bytes at the end which we believe is from padding.

file3ecberror.dec: There is a blurb of random text in the middle of the decrypted text, and there is also some repeated bytes at the end which we believe is from padding.

cbc: Similar to all the ecb files, there is a random blurb in the middle of all the files with repeating text at the end which we believe is from padding.

cfb: Similar to the others, part of the file near the middle has random bytes, but instead of repeating characters at the end of the file, for ofb there is only 1 random character

ofb: Unlike the other files, only 1 character is messed up in all of the ofb files. Similar to the cfb files, there is one random character at the end of the file.

Observation of salted error files:

Salted error files: There seems to be no major difference from the salted and unsalted error files. Because encryption happens in blocks at a time we believe that whether it is salted or not should not make any difference past the block that was corrupted.

Modifying ciphertext3:

In order to modify ciphertext3 we had taken the 13th set of 16 bytes, copied it and replaced it with the 11th set of bytes.

Source code files:

part3.py

References

Finding length of key using index of coincidence and GCD:
Biv - https://crypto.stackexchange.com/a/40067

Statistical properties of Vigenere ciphers to confirm method above:
https://www.murky.org/blog/2020-8/vigkeylength

Kasiki examination and frequency analysis to find the key and for figuring out the methods below:
https://crypto.interactive-maths.com/kasiski-analysis-breaking-the-code.html

Character frequency setup:
https://www.w3resource.com/python-exercises/string/python-data-type-string-exercise-2.php

Frequency calculation/normalization:
Martijn Pieters - https://stackoverflow.com/a/30964597/17835165

Get maximum number in a dictionary:
Priyanka Chaudhary - https://stackoverflow.com/a/268285/17835165

Statistics of the most occurring ASCII characters (based on lab hint):
http://millikeys.sourceforge.net/freqanalysis.html

Confirmation of the character frequency statistics above:
https://en.wikipedia.org/wiki/Letter_frequency

Converting hexadecimals to integers in order to index the mapping table:
Aaron Hall - https://stackoverflow.com/a/37221971/17835165

Get column indices from mapping table (dmap method from lab hints):
Stephen Fuhry - https://stackoverflow.com/a/903867/17835165

File signatures provided from the lab:
https://www.garykessler.net/library/file_sigs.html

Simple program to differentiate file formats based on their magic numbers:
Garr Godfrey - https://stackoverflow.com/a/69562337/17835165

Byte histogram for analysis of ciphertext2 for Part 2:

Justus Perlwitz - https://www.justus.pw/posts/2015-11-30-post.html

Characteristics of binary and text files for Part 1 and 2:
https://www.nayuki.io/page/what-are-binary-and-text-files

Identification of the resultant plaintext of Part 2:
http://officeopenxml.com/anatomyofOOXML-pptx.php