Challenge Sheet 1 (10 Points)

Date due: November 21th 2022

Submission Instructions

Solutions to this challenge sheet must be submitted by **November 21th 2022** *before 11:59 am* to https://cysec1.de/. Please upload only the .py files to the corresponding submission fields! Don't worry if you are not yet able to log into the site or upload files, this will be made available to you over the course of the week. **Make sure that your solution contains your name and matriculation number at the top of the files!** It is not allowed to submit these solutions in a group, as you do with the regular exercise sheet solutions. In the .py files, you may add as many additional methods as you want, but please do not rename the existing methods or add code outside of methods. How many points you get for the tasks is determined by tests on our system.

1 CTR

In the lecture you learned about block ciphers and especially the CTR mode. This mode is interesting because there is a fundamental flaw in the design itself. If we somehow have access to a plaintext m and the corresponding ciphertext c, we can compute the result of E_k for each block without knowing the key used.

$$c_n = m_n \oplus E_k(IV + (n-1))$$
$$\leftrightarrow \quad c_n \oplus m_n = E_k(IV + (n-1))$$

With this, you can now use the same technique to encrypt any message you select. However, keep in mind that you are limited to the number of blocks used to encrypt the known plaintext.

(3 points) (a) Using this knowledge, please implement the function **exchange(...)** in the file **ctr.py**, which given a known *plaintext* and *ciphertext* and a *new_plaintext* should encrypt the new plaintext and return the ciphertext as bytes. To make your life a little easier, we have included an **xor(...)** function for two bytes. Feel free to use it. Furthermore, you can assume that both *plaintext* and *new_plaintext* fit perfectly into the blocks, so you don't have to worry about padding.

2 Frequency

When discussing the Vigenère cipher and how to crack it, you stumbled across frequency analysis. In this task we will automate part of such an analysis.

(2 points) (a) To determine the frequency, you must first count all the letters in a given text. In this task, we ask you to implement the function **countLetters(...)** in **frequency.py**, which, given a text as input, should return a dictionary with all

lowercase English letters as keys and the number of occurrences as the value of the keys. This means that an input *H3llo World!*, should return:

{'a': 0, 'b': 0, 'c': 0, 'd': 1, 'e': 0, 'f': 0, 'g': 0, 'h': 1, 'i': 0, 'j': 0, 'k': 0, 'l': 3, 'm': 0, 'n': 0, 'o': 2, 'p': 0, 'q': 0, 'r': 1, 's': 0, 't': 0, 'u': 0, 'v': 0, 'w': 1, 'x': 0, 'y': 0, 'z': 0}

This example shows that we only care about the characters a-z and A-Z and not about other characters like numbers or symbols. Also note that the keys of the dictionary always contain all the letters of the English alphabet, even if they are not present in the given text. In addition, we summarize the amount of an uppercase letter and its lowercase counterpart and only use lowercase letters as keys.

(1 point) (b) Next, we ask you to implement **convertToFrequency(...)** in **frequency.py**, which, given a map as described in a), converts the number of occurrences to actual frequencies (X% of all letters in the text are a's). Again, a dictionary should be returned in which the frequencies are assigned to the respective lowercase letter. The example from a) should return:

{'a': 0.0, 'b': 0.0, 'c': 0.0, 'd': 0.1111111111111111, 'e': 0.0, 'f': 0.0, 'g': 0.0, 'h': 0.1111111111111111, 'i': 0.0, 'j': 0.0, 'k': 0.0, 'l': 0.3333333333333333, 'm': 0.0, 'n': 0.0, 'o': 0.2222222222222222, 'p': 0.0, 'q': 0.0, 'r': 0.1111111111111111, 's': 0.0, 't': 0.0, 'u': 0.0, 'v': 0.0, 'w': 0.1111111111111111, 'x': 0.0, 'y': 0.0, 'z': 0.0}

(0.3333333333333333 == 33,33333333333333%)

3 Caesar

In 2022, the Caesar encryption method seems a bit outdated. It lacks not only security, but also good character support. To solve this problem, we extend our character set from the alphabet to all ASCII characters from 32 (SPACE) to 126 (∼). Everything else works exactly the same, i.e. when you arrive at character 126, you start over at character 32. Now our encryption also supports things like spaces and symbols, which allows us to encrypt modern text messages. You are now tasked with automating the breakage of such a cipher.

(1 point) (a) First, we want you to implement **decrypt(...)** in **caesar.py** which, given a cipher and a shift should return the associated plaintext.

(3 points) (b) Next, we want to know which shift was used for encryption. For this you will need to implement the function **getShift(...)** in **caesar.py**, which should automatically find out the shift of a certain cipher. To do so, we have included **en_dictionary.txt**, the contents of which will be available via the variable with a similar name. It might help you figure out if a shift is right or not. Besides, that you can assume that the content of this variable will always be the same, even when tested on our system.

Hint: Don't forget that words can start with an upper or lowercase letter