1 Hybrid Cryptography

The Vigenère cipher is a method of encrypting alphabetic text by the use of a sequence of shift ciphers based on the letters of a keyword. In this project you will firstly develop a hybrid encryption system using the Vigenère cipher combined with the the RSA protocol for encryption of the keyword. You will then design a further hybrid system by adding an extra layer of random encoding. You will compare the security issues inherent to these systems. As a further challenge you can also choose to develop one of the extensions outlined in parts 5, 6, 7, below.

Note. The text, Jupyter Notebook, and pdf files mentioned below are available in the GitHub repository for this project at https://github.com/cmh42/hc.

Alphabet: crucial simplification. In (core) parts 1-4 the idea is to work with the 26 letter alphabet, for example you could use one, or both, of the following string constants[1]:

`string.ascii_lowercase` : = 'abcdefghijklmnopqrstuvwxyz'

`string.ascii_uppercase` : = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

In these parts of the project you should prepare all input messages so that they only contain alphabetic characters (i.e. no space characters, no punctuation etc.). For the preparation of the input messages and the enciphering/deciphering process you can decide to work entirely in lower case or entirely in upper case or (and this is nicer) you can choose to preserve the case of the alphabetic text from the original input message throughout. This allows you to model the whole process in a streamlined and simplified way. The point to note is that once we introduce bigram encoding this simplified approach can be used as the core enciphering/deciphering process of a system that handles messages that may contain many other charaters, including space characters and punctuation.

Remark. The Vigenère keyword is not assumed to be an actual word. In fact it can be of any length up to the length of the message to be encrypted. Note that we will refer to it as the *Vigenère key* or simply as the *key* if the context is unambiguous.

1. **(core)** Implement functions to encrypt and decrypt messages using the Caesar cipher and the Vigenère cipher. As noted above, your functions only need to handle messages containing alphabetic characters. Your Caesar cipher functions should be able to perform 26 possible shifts (including the trivial shift) and your Vigenère cipher should have 26 possible choices for any character in the key.

   Note. You should test your functions using randomly generated Caesar shifts and randomly generated Vigenère keys. You should organise this so that the reader can perform these tests. For your tests you might, for example, extract the alphabetic content of two or more of the `message_*.txt` files provided for this project. (See the example in `extract_alphabetic_content.ipynb`.) You should implement encryption and decryption to and from files. Similar comments apply to the tests that you should apply throughout this project.

2. **(core)** Implement a function to systematically break the Caesar cipher using letter frequency analysis. (Regarding the latter see the example in `get_online_texts.ipynb`.)

3. **(core)** Write functions that implement the Hybrid System described below with the encryption and decryption of the message carried out using your Vigenère functions from above and that of the key being carried out by the RSA functions from lectures. Note that, since the Vigenère key may be long—for example 200 characters—your system will need to slice

[1] To use these constants you will need to have included the statement: `import string`.

it in to one or more parts (so no slicing if only one part) in preparation for integer conversion/encoding and RSA encryption with the resulting ciphertext integers being transmitted as a tuple[2].

Hybrid System. Alice generates her private and public key. Bob generates a Vigenère key and Vigenère encrypts/enciphers his message with this key. Then, after slicing it into parts (if necessary) he encodes and RSA encrypts his Vigenère key using Alice's public key and finally sends *both* the resulting tuple of ciphertext integers *and* his Vigenère encrypted message to Alice. Alice uses her private key to RSA decrypt the tuple of ciphertext integers. She then converts/decodes the resulting integers to strings and so reconstructs the Vigenère key. She uses this to Vigenère decrypt/decipher Bob's message.

4. **(core)** There are $26 \cdot 25 = 650$ many 2-grams[3] made up of distinct letters. Redesign your system by performing a random encoding of each letter of the alphabet in to one or more of such 2-grams. Do this in such a way that the frequency of occurrence of each letter is disguised. For example—assuming that the frequency of the letters 'b' and 'e' are 1.5% and 12.7% respectively and that (in this part) you have decided to make use of 400 (of the 650) 2-grams—you can disguise the frequency of 'b' using a randomly choosen set of 2-grams of size 6 ($= 0.015 \times 400$) to represent different instances of this letter. On the other hand to disguise the frequency of 'e' you can use a randomly chosen set of 2-grams of size 51 ($\approx .127 \times 400$) to represent different instances of this letter[4]. Your message is now randomly encoded before Vigenère encryption and decoded after Vigenère decryption. Your encoding information should be recorded in a key which is then appended to the Vigenère key. As before, the resulting string should be sliced into parts for integer conversion and RSA encryption before transmission as a tuple of ciphertext integers. After transmission and RSA decryption of both keys the receiver will be able to obtain the original message. Discuss and compare the security of this hybrid system and that of part 3, taking the implications of part 6 below into account.
5. **(extension)** Applying the idea from part 4, modify your system so that it handles messages that contain, not only letters, but also numbers, punctuation and white space. (To do this you only need to modify the bigram encoding. You may want to use the full set of 650 distinct letter 2-grams for this.) Use this system to encrypt and decrypt the entire content of two of the `message_*.txt` files[5].
6. **(extension)** Implement a function to systematically break the Vigenère cipher. To do this you will need to first perform a *Kasiski* style analysis of the positions of repeated n-grams in the encrypted message to work out the length of the key. You will then reapply the letter frequency analysis that you developed in part 2 to establish the letters of the key.
7. **(extension)** In our presentation of the *RSA* protocol in Week 9 we use 512-bit (i.e. 154 digits in decimal) primes p and q. The security of the protocol relies on the fact that it is VERY HARD to recover p and q—i.e. to factorise N—from $N = p \cdot q$ if you only know N.

    - To see that this is indeed the case, and also to see what happens when we allow p and

[2]For example, the RSA protocol with 512 bit primes can safely handle strings of length $\lfloor (512-1)/8 \rfloor = 63$ (where we note that each character is encoded with 8 bits and a 1 is added to the front of the ciphertext integer). Thus if the Vigenère key is of length 200 we will want to slice it into 4 parts, convert/encode these 4 strings into integers for RSA encryption, and transmit the resulting 4 ciphertext integers as a 4-tuple.

[3]An n-gram is a string of n letters. The term n-gram is often used to denote a contiguous sequence (i.e. a substring) of n letters within a longer string. For example 'ing' is a 3-gram in 'Flying high'.

[4]Once the set of 2-gram encodings of e.g. 'e' has been chosen, the choice of which 2 gram to use for which instance of 'e' is made randomly during the encoding process and does not need to be recorded.

[5]Your system should be able to handle the white space character and all the 32 characters in the constant string `string.punctuation`. Thus $26 + 1 + 32 = 59$ characters in all. Any other characters such as tabs or newlines can be left unmodified by your system during the whole encryption, transmission and decryption process. (And you should make sure that your system preserves the case of letters.)

q to be smaller, you should begin by testing the performance of the `smallest_factor` function[6] from lectures. To do this generate primes p, q and input $N = p \cdot q$ to the `smallest_factor` function. Starting with $l = 16$ bit primes write an algorithm that shows the average computation time on input $N = p \cdot q$ for k-bit primes p, q for $k = l, l+1, l+2, \ldots$. Continue this analysis for as long as the outcome is a matter of minutes—e.g. up to 15 minutes.

- Using the function `smallest_factor` is clearly not an efficient way of factorising large integers. A better way of doing this is via the *Pollard rho* method. Write a function `pollard_rho` that implements the Pollard rho method using the outline given in the file `pollard_rho.pdf`. Carry out the analysis that you carried out on `smallest_factor` on your function.

Plot your results for both `smallest_factor` and `pollard_rho` showing the expected outcomes (extrapolated from your results) on longer bit lengths. Hence conjecture at what bit length the use of each function becomes unfeasible.

Note on the extensions. In this project (i.e. project 1) you may choose to develop one extension. (Doing more will not achieve more marks: the point is to concentrate on the quality of the ideas and the design of your code in both the core sections and the extension that you choose.)

[6] Note the `decompose` function (which uses `smallest_factor` as a subfunction) is not required here since you are working under the assumption that $N = p \cdot q$ with p and q prime. Thus once you have found one factor, which must be either either p or q, and assuming for the sake of argument that it is p, then you can extract the other factor q directly by dividing N by p.