Shashank Sudheer, Jaspreet Singh, Saurabh Parmar, Abhigna Muchala

Cryptanalysis of a class of ciphers based on the kasiski method and frequency analysis

Professor Giovanni Di Crescenzo

March 19, 2021

The title of our project is "Cryptanalysis of a class of ciphers based on the kasiski method and frequency analysis". The team consists of Saurabh Parmar, Abhigna Muchala, Jaspreet Singh and Shashank Sudheer. In order for us to complete the project in an efficient and organized manner, we decided that each person will voluntarily pick one section to focus on. The work divided the project into five sections: main code, pseudo code, time management, research, and writing. Our fellow partner, Shashank Sudheer wanted the responsibility of starting the coding process, as group members, we were aware that the coding can get overwhelming and challenging so we were present if he needed help on anything. The responsibility of writing pseudo code and time management was voluntarily given to Abhigna Muchala, she is the person that sets up the zoom meetings and schedules days we meet. The responsibility of preparing the documentation for this project and researching different attacks and topics was done by Jaspreet Singh and Saurabh Parmar, researching different types of attacks to make the coding process easier for Shashank. Focusing on kasiski and frequency analysis, our group did rigorous research on these two topics and found that the Kasiski method is a tool we can use to decide the length of the Vigenère key that is used to encode a ciphertext. The Kasiski method is implemented to track down each set of strings at any rate three letters in the ciphertext. These successions will show specific similarities regarding letters of plaintext encoded and will utilize equivalent subkeys of the Vigenère key. Although the main focus has to do with the Kasiski method, instead of starting with that specific method for examination of key length; We decided to use the index of Coincidence formula for the first execution. What this formula does, gives a proportion of the fact that it is so prone to draw two coordinating letters by arbitrarily choosing two letters from a given book. The possibility of attracting a given letter the content is (number of times that letter shows up/length of the content). Once we get the key length, we run frequency analysis by breaking the original cipher text into "nth" letters. When speaking about the "nth" letter, we use this term to describe the length of the key. For example, if the key length is equal to 5, this will mean that every 5th letter in the ciphertext gets joined into one string. For running the frequency analysis on every subsequence for this project, we use the chi squared formula. Which will allow us to treat this part of the code as a mono-alphabetic cipher. Translating to the fact that every letter in the subsequences is ciphered by the same key numbers. Once we are able to get that key that will mean that we solved one sub sequence, and this process is going to repeat itself over and over again where we will run it for every set of subsequences until we get the same number of subsequences as the key length; using this information we needed to solve for the next set of execution where the algorithm was different, so we took the key that we determined for all subsequent executions and use that towards running every permutation of the key on the first six letters of every subsequent cipher text; using this permutation of the key, we iterated through all possibilities until we came upon a matching sequence of letters that matches one of the candid plaintext. For the coincidence part of this project. Coincidence counting is the strategy of comparing two writings one next to the other and tallying the occasions that indistinguishable letters show up similarly situated in the two writings. This check, either as a proportion of the aggregate or standardized by separating by the normal mean an arbitrary source model, is known as the index of occurrence, or IC for short.

For a generic piece of text written in English the Index of Coincidence is 0.0667.

So in English letters, the five most frequently used letters are **E** (13.11%), **T** (10.47%), **A** (8.15%), **O** (8.00%) and **N** (7.10%).

But this is for the normal english frequency percentage, we used our own frequency distribution analysis to figure out the frequency distribution for all 5 of the plaintext.

This is the sample from the code:

```
FREQUENCY_ENG = 'ETAOINSHRDLCUMWFGYPBVKJXQZ'

FREQ_DIST1 = ' ESRIAONLTCMGDUPHBFZKVQJYX' # plaintext_1

FREQ_DIST2 = ' IERSOLANTCDMGUPYHZBFKJVQX' # plaintext_2

FREQ_DIST3 = ' ESIATLONRUCPMGDHYVBFJZXQK' # plaintext_3

FREQ_DIST4 = 'E ISNTAROLDUCGPMHYFBXJVQKZ' # plaintext_4

FREQ_DIST5 = ' ESRNIOATLCDUGBYMPHVKFXZJQ' # plaintext_5

FREQ_DIST6 = 'ES IROTNCLAUPDMHBYFVGZKQXJ' # word_dict
```

As it is evident from the above Distributions we have accounted for spaces which take up the highest occurrences in 4 of the 5 plaintext.

In our code we use the following steps:

- Returns the key length with the highest average Index of Coincidence.
- Split the ciphertext into sequences based on the guessed key length from 0 until the max key length guess (20). Ex. guessing a key length of 2 splits the "12345678" into "1357" and "2468". This is the procedure of breaking ciphertext into sequences and sorting it by the Index of Coincidence. The guessed key length with the highest IC is the most probable key length.
- We have tried to take care of the nuances as well. This program can sometimes think that a key is literally twice itself, or three times itself, it's best to return the smaller amount. Ex. the actual key is "qwerty", but the program thinks the key is " qwertyqwerty " or " qwertyqwertyqwerty ". This happens because the frequency distribution for the key " qwerty " vs " qwertyqwerty " is nearly identical.

Here are the list of pseudo code for all the functions we used in the program:

- The first pseudocode defines max key length, the alphabets used, and creates an array to map the frequencies. The function is used to calculate the index of coefficients.

```
Set  MAX_KEY_LENGTH_GUESS to 24 {global variable}
Set alphabet to 'abcdefghijklmnopqrstuvwxyz' {global variable}
Set english_frequencies to an array containing the relative frequencies of each letter in English language {global
variable}
```

```
FUNCTION def get_index_c(ciphertext)
    Set N to length of the ciphertext
    Initialize frequency_sum to 0.0
    FOR every letter in alphabet:
        frequency_sum+= (no.of occurrences of a letter in ciphertext)*(no.of    occurrences of a letter in
ciphertext-1)
    ENDFOR
    ic= frequency_sum/(N*(N-1))
    Return ic
ENDFUNCTION
```

- The second pseudocode shows how we split and combine the ciphertext to figure out the key with the frequency analysis.

```
FUNCTION def get_key_length(ciphertext, key_length)
    Set key to an empty string
    FOR i >=0 and < key_length
        Set sequence to an empty string
        Split the ciphertext into key_length sized strings
        Sequence will be appended with the strings obtained from splitting
        Key = key + freq_analysis(sequence)
    ENDFOR
    Return key
ENDFUNCTION
```

- The third pseudocode defines how we use chi square analysis and frequency percentages to output the shift in characters.

```
FUNCTION def freq_analysis(sequence)
    Initialize all_chi_sqaured to an array of 27 0s
    FOR i >= 0 and i < length of CHARACTERS
        Set chi_sqaured_sum to 0
        FOR j >=0 and j < sequence length
            Sequence_offset = [CHARACTERS[(CHARACTERS.index(sequence[j] - i) MOD 27]]
        ENDFOR
        FOR l in sequence_offset:
            v[CHARACTERS.index(1)]+=1
        ENDFOR
        Calculate frequency percentages by dividing the above array by sequence length
        Calculate chi_sqaured sum that is by subtracting english frequencies from frequency percentages
        Initiate shift to minimum of the difference
        Return CHARACTERS[shift]
ENDFUNCTION
```

- The fourth pseudocode defines the encryption scheme we used. We check if the cipher is a vigenere cipher and return the ciphertext.

```
FUNCTION encrypt(message, key, cipher)
    Get the message in lowercase
    Set ciphertext to an empty array
    IF cipher = vigenere
        FOR i >=0 and i < length of the message
            j = i MOD key length
            Shift the key by j
            Set cipherInd to the index after the shift
            Append the ciphertext for each character
        ENDFOR
        Return ciphertext
    ENDIF

ENDFUNCTION
```

- The fifth pseudocode defines the decryption scheme and how we return the plaintext.

```
FUNCTION def decrypt(ciphertext, key)
    Set plaintext to an empty array
    FOR i>=0 and i< length of ciphertext
        j= i MOD key length
        Set shift to jth element of key
        Set plaintextInd to the index of the CHARACTERS before the shift
        Append the plaintext with each decrypted string
    ENDFOR
    Return plaintext
ENDFUNCTION
```

- The last pseudocode defines the algorithm in which we determine the plaintext with which the ciphertext matches.

```
FUNCTION def solvedKeyTest(ciphertext, key)
    plaintextArray is an array containing the 5 plaintexts given in the dictionary
    Set tempKey to the key
    FOR every plaintext in plaintextArray
        Initialize i and j to 0
        Set count to the length of the tempKey
        WHILE count >0
            Set testSingleKey to an empty array
            Append the testSingleKey with the jth element of the tempKey
            plainSym = decrypt(ciphertext[i], testSingleKey)
```

```
                    IF i==5
                              Return plaintext
                    ELIF plaintext == plaintext[i]
                              Increment i by 1

                    ELSE
                              Increment j by 1
                              Decrement count by 1
                              j = j MOD length of tempKey
          ENDWHILE
          Return " could not find the plaintext"
ENDFUNCTION
```