Cryptanalysis of a class of ciphers based on Frequency Analysis using Index of Coincidence and Chi Squared Test

Kang In Park, David Son, Elaina Huang

Applied Cryptography

Professor: Giovanni Di Crescenzo

Table of Contents

1. Introduction

1.1 Team Members

Team members for this project are David Son, Elaina Huang, and Kang In Park. All team members contributed to brainstorming potential solutions for breaking the cipher and assisted to write the report. Each person explored several known cipher attacks and as a group we discussed whether it could be applied to this particular cipher.

In our final algorithm, Kang In implemented the Index of Coincidence calculation for deriving most likely key length as well as the convenience functions in `utils.py`. David Son filled in the frequency analysis using the chi-squared test. Elaina Huang developed tools to assist the decryption process (e.g. find out the missing chars from each text candidate) and attempted to develop a solution through use of N-gram frequency.

1.2 The Encryption Algorithm

Here is our understanding of the cipher described by the project.

The text space is restricted to be in the set of {<space>, ‘a’, ‘b’, ‘c’, …, ‘z’} a total of 27 different elements, with no symbol (e.g. comma, period, etc.) presented. The key space is restricted to be in the set of {0, 1, 2, …, 26} a total of 27 different elements with the key length restriction in the range of [1, 24].

With the above conditions set, now we can further elaborate on the algorithm scheme.

The encryption algorithm takes in 3 parameters: a string of plaintext, a key, and lastly, a preset probability value. The encryption process will be using the text from the text space as defined above, to substitute any characters in the plain text. For example, an alphabet can be replaced with a space, and a space char can be replaced with an alphabet.

The method to encrypt is as follows:

1. Preset a probability value with a real number from the range of (0, 1). This value will later be used to compare with another randomly generated value (note, this value will be generated as a real number in the set of [0, 1].) to diverge the branched conditions.
2. Then, we will start encrypting every letter individually by using a loop that only terminates when it reaches the last element in the plain text. Inside the loop, there are two branched conditions to strengthen the security of ciphertext.
    a. If the conditions “preset probability < randomly generated value” and “randomly generated value <= 1” are met, we will randomly choose a number from the list of keys to shift the plain text char into a cipher text char.
    b. If the conditions “randomly generated value >= 0” and “randomly generated vale <= preset probability” are met, we will randomly choose a character from the text space as the noise to add to ciphertext; this procedure will eventually result in a

longer length of output than its original length plaintext, thus satisfies the perfect secrecy.

3. The way to shift plaintext char is also based on the text space as defined above. Here is an indexed example of text space, ciphertxt_space = { (0, <space>), (1, ‘a’), (2, ‘b’), (3, ‘c’), ….., (26, ‘z’) }. The encryption algorithm shifts character based on this order.

Pseudocode:

```
Pseudocode:
Let key be a list of whole numbers with a range {0, 26}
Let plaintext be a string of plaintext with a text space in {<space>, 'a', 'b', .., 'z'}
Let ciphertext be an empty string
Encrypttion (key, plaintext, preset_prob){
    Let p_ptr be the index pointer of plaintext string
    Let rand_count = 0
    while (p_pter < length of plaintext){
        Let coin_val be the randomly generated value use for every char encryption
        if ((preset_prob < coin_val) && (coin_val <= 1)){
            randomly choose a number from key and shift accordingly
            append the cipher char into ciphertext
            p_ptr increment by 1
        }
        else if ((coin_val >= 0) && (coin_val <= rand_prob)) {
            randomly select a char from text space
            append on the char into ciphertext
            rand_count increment by 1
        }
    }
    return ciphertext
}
```

2. Cryptanalysis Approach

Our initial intuition was to use frequency analysis as the cipher seemed like a polyalphabetic cipher. However, the project description states that the key scheduling algorithm is unknown, and it does not necessarily rotate through the key in a consistent manner. This makes the project cipher more akin to a running key cipher as the final key length is as long as the message length.

Here it is important to note that the "final" key length is different from the key length 't' as mentioned in the project description. Because we are operating without knowledge of the key scheduling algorithm, even if t=3 and the key is [1, 2, 3], the key does not repeat in an orderly fashion like so:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 3\ldots$$

Instead, the key may end up being:

$$3 \rightarrow 3 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 3\ldots$$

Since the key scheduling algorithm is unknown to the attacker, the algorithm might as well just be random. Therefore, the "final" key length in this project denotes the actual length of the key values used to encrypt the individual plaintext characters, and thus the "final" key length is the same as the plaintext length L.

This constraint created a lot of problems for us when trying to brainstorm ideas to crack the cipher. We explored many well known cipher attacks as well as individual ideas that at first seemed promising. These attempts will be outlined in the sections below.

These constraints on top of the random ciphertext factor made it impossible to find a reliable way to crack the cipher with the information available to the attacker. After exploring a number of potential solutions, we decided to build up from less constraints and work our way to the top.

Assuming that the key scheduling algorithm rotates through the key in a repeating fashion (thus making it a polyalphabetic cipher), and starting out with no randomness factor, we went with our initial intuition of using frequency analysis with index of coincidence to find the key length and chi squared test to find the key. Details will be provided in sections below.

2.1 Attempted Solutions

In our research and discussion, we looked at many different known cipher attacks to attempt to tackle the cipher described by the project. While we came up with many ideas and potential solutions, when we examined the project constraints and adhered to it strictly, we could not find a solution that would reliably crack the cipher. Our attempts revolved around trying to identify the class of cipher that this particular project cipher belongs to, and whether there are known cipher attacks against these types of ciphers. The biggest challenge was the fact that the "final" key length was as long as the plaintext length. This made the project cipher more akin to a running key cipher or somewhat of a one time pad for shift ciphers (although the possibility of key values is limited to maximum 24 and not the full message space of 27 symbols). These ciphers are not crackable unless part of the key is exposed or the same key is used to encrypt several plaintext messages. None of these conditions existed for our project.

One method we considered was the hill climbing algorithm, which can be used to break substitution ciphers. The way it works is the algorithm first generates a random key, also known as the "parent" key. The algorithm then attempts to decipher the ciphertext using this random key, which produces a sample plaintext. The algorithm then uses a fitness test, which scores the text based on how similar it is to english text. The algorithm continues generating random keys and comparing the fitness score to the highest scoring plaintext, keeping the plaintext that has a higher score. It repeats this pattern unless a higher score hasn't been found for the past 1000 iterations. The reason this method caught our attention is because our dictionary texts contain english words and are fairly long at 600 characters, which would make it easier to utilize the fitness test. However, the number of possible keys in a substitution cipher is 26! (each alphabet in plaintext is mapped to another alphabet in ciphertext). The number of possible key variations in our project is t^L, which is at maximum 24^600. This makes it impossible to yield any meaningful results just by generating random variations of the key.

Another method we considered was differential cryptanalysis. However, since we have no access to an encryption oracle to test different inputs, we cannot do this. There is no way for us to produce a ciphertext with the same algorithm, as the key scheduling algorithm is unknown to us. The only information we are given is the ciphertext and 5 candidate plaintexts, which we have no way of knowing whether the ciphertext given was actually from those 5 candidate plaintexts. The project description states that our decryption algorithm does not know whether it is looking at a ciphertext from test1 or test2. As mentioned above, there is no way for us the produce another ciphertext using the same algorithm, as the cipher described in the project is more akin to a running key cipher or a one time pad cipher for shift ciphers where the key is unique for every encryption.

Other methods in the frequency analysis category were the first that came to our minds. Methods like N-gram frequency and other analysis strategies that we eventually opted for such as Index of Coincidence and Chi squared test would not work if we were strictly adhering to the cipher constraints as outlined in the project. This is because the initialized key is not repeated, and so any kind of pattern matching or frequency analysis ends up having more noise than meaningful data. When we analyzed the sample ciphertexts created from our encryption algorithm, the frequency stats were distributed very evenly in all the characters. In the end we modified some specifications in the project description and ended up using frequency analysis with Index of Coincidence and Chi Squared test.

An elaboration on the trying of N-gram. We obtain the idea that the n-gram frequency could be an indicator to reveal the relationship of the order of words inside a paragraph. We decided to try it on the letter-level to try to get more insights on the decryption process. It is worth noting that this method was developed to target the situation where there are randomly inserted characters. For this matter, we believe to use the first and the last word from the plaintext will be an optimal selection to start as the baseline. The reasoning is that when more and more random char being inserted into the ciphertext, the order of the original words are further disrupted, therefore, it would be reasonable to use the first and the last word as they have a higher likelihood that many if not at least 1 char will be able to reveal the real key number that's been used to cipher.

Therefore, we use those words to compare against a range of ciphertext obtained correspondingly also from the beginning and towards the end of the ciphertext. The range of ciphertext will, however, be longer than the plaintext as we do not know how many random char had been inserted. The range is doubled from the original word. Then we run the brute force method to compare against the original text to obtain a list of potential key numbers, then we use the n-gram frequency method to eliminate the key that gives a lower probability of the order of letters. This gives us a baseline list of keys to try to use on the remaining body of the ciphertext through the same brute force method, unfortunately that this process will take too long and it becomes unfavored.

Another fun attempt that we tried was "frequency amplification". This was not a method we found anywhere online, but just started as an idea for analyzing the frequency of characters and potentially "fingerprinting" the candidate plaintexts with their unique letter frequency distribution. The idea came from the fact that the randomly sprinkled ciphertext characters do not largely affect the frequency of the letters (at least when probability is 0.05). The most problematic effect it has is that it messes up the matching for plaintext[i] to ciphertext[i]. This makes it so that any kind of stream-based attack that attempts to run through both the plaintext

and ciphertext is futile. However, the frequency distribution was relatively intact. With 0.05 probability for random ciphertext, the frequency distribution for text as long as 600 characters would only have about 30 anomalies. If we take into account that these anomalies are also random and should be relatively evenly distributed among the 27 potential characters, it is almost negligible when it comes to the whole “frequency fingerprint” idea.

The idea was to “amplify” the frequency distribution of our candidate plaintexts by encrypting them over multiple times, in hopes that the unique composition of those plaintexts would eventually lead to a consistent and distinguishable distribution of letter frequencies in their ciphertexts. While it was a fun idea, and we had somewhat of a success fingerprinting the plaintexts if the key was the same, this could not work for two reasons. First, the amplified frequency distribution for plaintexts with different keys was not consistent. It was only consistent if the key was the same, meaning that the randomness of the key scheduling algorithm or the random ciphertext characters could be overcome if amplified, but the sheer size of potential key values (24^600) could not be overcome. Furthermore, in reality, the decryption algorithm will only be given a ciphertext of length 600 + r. This sample is not big enough to reliably match its frequency distribution to the candidate fingerprints. While it was a fun attempt, this approach could not work for these reasons. The code for this can be seen in `amplify.py`.

3. Informal Explanation

The approach we chose is frequency analysis. Assuming that the key is repeated, it is possible to derive the most likely key length using the Index of Coincidence. The Index of Coincidence is a way to calculate overlap of identical characters appearing in text. The IOC is typically greater in normal english text than in random ciphertext. Calculating the IOC is an easy way to detect monoalphabetic ciphers, since the IOC of the encrypted ciphertext will be the same as the english plaintext, indicating a single byte key used to encrypt the plaintext.

In our polyalphabetic cipher, we first assume different key lengths from 1 to 24. The basis for using the Index of Coincidence is that the character in ciphertext[i] will be offset by the same amount as the character in ciphertext[i+t] since the same key value was used to encrypt it.

If we assumed that the key length was 5, we break the ciphertext into 5 groups:

ciphertext[0], ciphertext[5], ciphertext[10] …

go in the same group and

ciphertext[1], ciphertext[6], ciphertext[11] …

go in the same group. In the end we will have 5 groups, which we can treat as separate monoalphabetic ciphers. If these groups give a high IOC value, meaning they are close to a monoalphabetic cipher, which means that we have found the right key length. It should be noted that the most likely key length may come out to be a multiple of the actual key length.

From this point on, we can use the chi-squared test to find the key. A chi-squared test compares the frequency of a given dataset and compares it to its expected dataset, giving a smaller number the closer it is to its expected frequency. This test assumes the key length n and splits the ciphertext into groups of length n. Then, across the whole ciphertext, each group has their first letter substituted with every letter of the alphabet and has a chi-squared test performed for each substitution. The lowest score is assumed to be the first letter of the key. The same is tested on the second letter of each group, then the third, and so on, up to the nth letter of each group. The result of this operation is the proposed key.

Finally, with the ciphertext and the proposed key, the string can be decrypted using a simple polyalphabetic shift cipher. For every character, subtract its value by its corresponding index in the key. The result is the original plaintext.

4. Technical Explanation

All convenience functions used for encryption and testing can be found in `utils.py`.

As explained above for our final algorithm, the key length is derived by using the Index of Coincidence. The function for calculating IOC is shown below.

```
124   #get index of coincidence
125   def get_IOC(text, freq):
126       total = 0
127       for n in freq.values():
128           total += n * (n - 1)
129       N = len(text)
130       if N <= 2:
131           return 0
132       total = float(total) / ((N * (N - 1)))
133       return total
```

To be able to utilize the above IOC function, we need to have a function to group the ciphertext into n groups, n being the potential key-length.

```
116   #group into n groups, assuming key length is n
117   def group_into_n(ciphertext, n):
118       groups = {i:'' for i in range(n)}
119       for i in range(len(ciphertext)):
120           group = i % n
121           groups[group] += ciphertext[i]
122       return groups
```

And finally, to derive our most likely key length, we have the below function to put it all together. It guesses all key lengths from 1 to n; for our project, n can be set to 24. It then calculates an IOC average for all key length guesses, and sorts all keys from most likely to least likely.

```
135   #guess key length (try up to n length keys)
136   #returns list of tuples, ordered from most likely length to least
137   def guess_key_length(ciphertext, n):
138       guess = defaultdict(int)
139       for i in range(1, n+1):
140           groups = group_into_n(ciphertext, i)
141           IOCs = []
142           for group in groups.values():
143               freq = get_freq(group)
144               IOCs.append(get_IOC(group, freq))
145           IOC_avg = sum(IOCs) / i
146           guess[i] = IOC_avg
147       return order_freq(guess)
```

For the key-guessing part, we start off with a frequency table.

```
#guess the key in ciphertext based on proposed key length n
frequencies = {' ': 0.104456824512534 82, 'a': 0.069637883008
```

This was done by going through the whole list of possible words and counting each character.

```
def guess_key(ciphertext, n):
    str_freqs = [x * len(ciphertext) for x in frequencies.values()]
    key = ""
```

Next, in the actual function, we compile a list of expected string frequencies based on the length of the ciphertext.

```
for i in range(0, n):
    results = {}
    for char in ' abcdefghijklmnopqrstuvwxyz':
        new_str = ""
        new_str += ciphertext[0:i]
        for a in range(i, len(ciphertext), n):
            new_str += shift(ciphertext[a], -text_to_i[char])
            new_str += ciphertext[a+1:a+n]

        new_freqs = list(x[1] for x in get_alphabetical_freq(new_str))
        results[char] = chisquare(new_freqs, f_exp=str_freqs).statistic

    results = sorted(results.items(), key=lambda x: x[1])
    key += results[0][0]
return key
```

This can be interpreted as follows:

For all i from [0, n),

For each character in the message space,

Compile a new string, substituting every nth letter with the character. With the new string, do a chi-squared test, comparing it to the expected frequency, and add its frequency to the list

When the loop has concluded, choose the character with the lowest associated chi-squared value and use it as the ith value in the string

When this section of the code has concluded, the key is found. All that's left is to decrypt the function.