Cryptanalysis of a Class of Ciphers Based on Frequency Analysis Approach and Unique Shift Counting

Sheena Zhou *sz2493@nyu.edu*

Ryan Zavoral *rpz214@nyu.edu*

Jin Wen Loh *jwl482@nyu.edu*

CS-GY 6903 I/CS-UY 4783 A

Professor Giovanni Di Crescenzo

March 19, 2021

Table of Contents

Introduction

Team Members

Ryan Zavoral is credited for the final algorithm, implementation of test 1, error case checking, and extra credit 1. Jin Wen Loh is credited for the final algorithm, implementation of test 2, combining both approaches, and extra credit 2. Sheena Zhou is credited for summarizing and writing the project report. All members took part in discussing and brainstorming ideas to approach the project.

Basic Overview

The cryptanalysis scheme for test 1 is mostly based on a unique shift counting approach, where we used the unique shifts calculated from the candidate plaintext to the given ciphertext to determine which plaintext is most likely the answer. This scheme heavily relies on the fact that the ciphertext is the encryption of one of the 5 plaintexts.

The cryptanalysis scheme for test 2 is mostly based on a frequency analysis approach. Since there are no candidate plaintexts to use, the scheme for test 1 would not work. By determining the frequencies of characters occurring in the plaintexts generated from the given dictionary, using the Index of Coincidence (IC) to determine the key length, and X distribution to determine the exact shifts in the key, we are able to decrypt the given ciphertext.

However, since we don't know whether the given ciphertext would be for test 1 or test 2, we decided to combine both approaches together. The given ciphertext will first run through the frequency analysis scheme, where the algorithm will return a potential plaintext. This potential plaintext will then be entered as the sixth plaintext into the unique shift counting

scheme, where the algorithm will determine of the 6 plaintexts it has, which one is the better fit, and return that one as the final answer.

We will now discuss the details of the processes of each scheme, as well as their drawbacks and likelihood of success.

Unique Shift Approach for Test 1

The algorithm scheme for test 1 heavily relied on the given plaintexts and restrictions/givens of the project (plaintext_dictionary_test1.txt, key length t <= 24, max 50 random inserts, based on a poly-alphabetic substitution shift). The main idea was to use the pattern of unique shifts of characters from each potential plaintext to the given ciphertext to help determine which one was the decrypted answer.

General Idea

Initially, the idea was to encrypt each plaintext to see which one would get close to the given ciphertext. While checking to find the number of unique shifts needed to encrypt a plaintext into a given ciphertext, we noticed that the number of unique shifts was <= t if the plaintext matches the ciphertext and > t if not. To test whether this theory was true, we randomly encrypted a plaintext p1 into a ciphertext c1, and calculated the number of unique shifts from pi where i = 1,2,3,4,5 to c1 for a case of t = 24. The result was that only the mapping from p1 to c1 returned 24 unique shifts while the other plaintexts returned 27. This pattern was the basis of our whole unique shift approach.

However, this only works if there were no random insertions into the ciphertext. Knowing that at most 50 random inserts would be made, we needed to come up with more code to take that into consideration. The idea was to split the ciphertext into blocks and then analyze each block for possible insertions. The block sizes had to be large enough to contain 27 unique shifts for a given plaintext and small enough to not get too many random insertions. It was decided that block sizes of 50 were reasonable to work with. To see if the theory works, we tried the new algorithm on predicting 0, 1, and 2 insertions per block of chosen plaintext. The number of unique shifts was then calculated with the random insertion(s) in consideration. This would return a list of unique shifts for 0 through 2 insertions per block, and the average minimum number score was chosen. The plaintext that returned the smallest minimum number score was the one that matched the ciphertext.

While the code worked for lower insertions, when moving up to 3 or 4 insertions per block that further corrupted the data, it began to fail. It also failed when the distribution of random insertions were evenly distributed throughout the ciphertext. So, for our final version, we decided to find the low median number score of each plaintext instead of the minimum. Block sizes were also made larger to accommodate for more insertions and errors. The code for working with random insertions was modified to deal with both evenly and skewed distributed cases.

Implementation Code

Before starting, we created our own ciphertext to test on. These ciphertexts are created using the following code that encrypts using a poly-alphabetic substitution cipher:

```
def random_encrypt(p <entered plaintext>):
    t = 0
    c = ""
    key = random.sample(range(1,27), <key size>)
    for i in range(0, 500):
        t = random.randint(0, <key size - 1>)
        c = c + shift(key[t], p[i])
    return c
```

Where `shift` calculates the amount each character in the entered plaintext `pi[i]` is shifted by, given what they are mapped to according to the key `t`. The numbers in `range(1,27)` map to the ASCII values of characters "*a*" through "*z*" and 27 maps to the " " character.

After creating our own ciphertexts, we then test our theory that if the plaintext matches the ciphertext, the number of unique shifts is less than t. Below is the code that calculates the unique shifts:

```
def compute_unique_shifts(text1, text2):
    s = set()
    for i in range(0,500):
        t1 = translate(text1[i])
        t2 = translate(text2[i])
        if t1 >= t2:
            s.add(t1 - t2)
        else:
```

```
            s.add((t2 - t1)*-1 + 27)
    print(len(s))
```

Where `text1` and `text2` represents any two texts we want to compare and `translate` returns the ASCII value of the entered characters so the absolute value difference between two characters in the same location of each text can be calculated (calculating the shift).

Entering the generated ciphertext (in our example, encrypted version of p1 with t = 24), and the plaintext p1 into `text1` and `text2` respectively, `compute_unique_shifts` returns 24, while any other plaintext entered with the ciphertext returns 27. We also calculated unique shifts between every combination of plaintexts and the result is always 27. This shows that our theory holds and only the matching ciphertext and plaintext will have a unique shift <= 24.

However, this only works if no random insertions are added into the ciphertext. So, the code has to be modified to take those variables into account:

```
def find_insertion(ciphertext, text, insertions):
    list_of_sets = list([[set() for j in range(0, insertions + 1)] for
i in range(0,10)])
    for i in range(0,500):
        t1 = translate(text[i])
        for j in range(0, insertions + 1):
            c = translate(ciphertext[i + j])
            list_of_sets[int(i/50)][j].add(calculate_shift(t1, c))
    setlengths = [[len(_s) for _s in s] for s in list_of_sets]
    print(min([min(length) for length in setlengths]))
```

```
    print(setlengths)
```

Where `list_of_sets` creates 10 sets where each set size depends on the number of insertions we assume to calculate, `translate` returns the ASCII value of the entered characters, and `setlengths` stores the unique shifts calculated for each insertion assumption for each block of size 50. `find_insertion` then calculates the average unique shift for each block and returns the smallest number.

Testing this code with our own ciphertext (encrypted from plaintext 1 and with 10 insertions added in), plaintext 1, and insertions = 2 (so the code will test for assumptions of 0, 1, and 2 insertions per block), the returned unique shift was 14.8. Other plaintexts that shouldn't match the ciphertext returned values of 21.2, 21.3, 21.7, and 22.5. This shows that our algorithm works for up to 2 insertions per block (max of 20 insertions). We also noted that because of modifying the algorithm, we will choose the smallest unique shift value as an indicator that the plaintext matches the ciphertext instead of choosing the one <= t.

But, test 1 mentioned that there would be at most 50 random insertions, so the code has to be modified to work for up to 50 insertions throughout the ciphertext. The code was then modified to take that into account:

```
def compute_unique_shifts_with_insertion(ciphertext, text):
    insertions = len(ciphertext) - 500
    if not insertions:
        return compute_unique_shifts(ciphertext, text)
```

```
    list_of_sets = list([[set() for j in range(0, insertions +
insertions + 1)] for i in range(0, 6)])
    k = 0
    for i in range(0,500):
        t1 = translate(text[i])
        if insertions and i % (500/insertions) == 0:
            k = k + 1
        c = translate(ciphertext[i])
        list_of_sets[int(i/84) % 6][0].add(calculate_shift(t1, c))
        for j in range(1, insertions + 1):
            c = translate(ciphertext[i + j])
            list_of_sets[int(i/84) % 6][j].add(calculate_shift(t1, c))
            if (j + k) > insertions:
                c = translate(ciphertext[i + insertions])
            else:
                c = translate(ciphertext[i + j + k])
            list_of_sets[int(i/84) % 6][insertions +
j].add(calculate_shift(t1, c))
    setlengths = [[len(_s) for _s in s] for s in list_of_sets]
    medsetlengths = [min(length) for length in setlengths]
    medsetlengths.sort(reverse=True)
    numerator = sum([(i+1)*medsetlengths[i] for i in
range(len(medsetlengths))])
    denominator = sum([(i + 1) for i in range(len(medsetlengths))])
    return numerator/denominator
```

Where `insertions` calculates the number of random insertions (r) in the ciphertext, `list_of_sets` creates 6 blocks, each containing 2*r + 1 sets where 1 set contains the calculation for no insertions, r sets contains the calculation for skipping every 1-r characters in the ciphertext, and r sets contains the calculation assuming every 1-r characters there is an insertion (uniform random insertion), and `medsetlengths` calculates the weighted low median of unique shift values for each set (more weight given to smaller values).

Skipping every 1-r characters in the ciphertext is done to help minimize the variability of insertions in a given block. If a random character is skipped, it will lend to a more accurate unique shift calculation. If a non random character is skipped, it will create more error in the block, returning a high unique shift value, which indicates that is not the right character skipped. Since the original algorithm could not accurately calculate unique shifts if the insertions are uniformly distributed, After calculating the unique shifts for all insertion instances, the sets are then sorted in ascending order, where the median of the first 5 blocks are taken and returned as the final unique shift value. We found that taking the low median value returned a much more accurate result than taking the smallest value.

Other modifications include changing the number of blocks from 10 to 6 (increasing the block size from 50 to 84). This was done because if all t shifts were unique, we noticed block size 50 was not large enough to compute any t larger than 14 with 99% accuracy for 50 random insertions. Increasing the block size allowed more space for more unique shifts and errors for up to t = 20 with 99% accuracy.

Frequency Analysis Approach for Test 2

Unfortunately, the approach that works for test 1 will not work for test 2, mainly because we can no longer use the known plaintexts to our advantage (other assumptions still apply). What we have instead is a dictionary of words that are used to be randomly arranged into a 500 word plaintext and then encrypted into a ciphertext. So, we added another approach that would effectively decrypt the ciphertext under those conditions. The approach we came up with is using frequency analysis on each character “a” through “z” and “ “.

General Idea

First, the average frequency of character distributions for test 1 and test 2 were calculated. This was run 1,000 times to get as close to an accurate frequency distribution as possible. Then, we needed to determine the possible key length used to encrypt the plaintext. To help determine this, the Index of Coincidence (IC) was used, where the IC calculates the probability of drawing the same character twice in a row. Using the assumption that the same key is used over and over again to encrypt a plaintext, we know that each character mapped to the same k[i] will be shifted the same amount. By calculating the IC for all characters mapped to the same k[i], we notice that their IC value is equivalent to 1.73, which is the IC for the English language.

With this knowledge in mind, an algorithm is created to test all possible key lengths (1 - 24), where the ciphertext is rearranged into columns depending on the key length currently being tested (i.e. 500 rows of 1 column for key length = 1, 250 row of 2 columns for key length = 2, etc). The correct key length will have an IC that is around 1.73 (as opposed to being about 1 if

it's not the right one). For smaller key lengths, we notice that the IC is around 1.73 for every multiple of its length (i.e. if the key length is 5, IC will also indicate 10, 15, and 20 as possible key lengths), but it makes no difference on the whole process (if key of length 4 = help, then key of length 8 = helphelp, which is essentially the same key repeated twice).

After figuring out the key length, the code generated random key shifts (from 27 possibilities) until one produced a result that matched the frequency distribution calculated earlier, determined using the X distribution. From there, the resulting key is used to decrypt the ciphertext.

Implementation Code

To calculate the frequency character distribution for test 1 and test 2, the following code is used:

```
def statistical_test(test1_list, dict2, iters=5):
    target_length = 500;
    freq_list = [0] * 27
    freq_dict = {}
    n = 0;
    for i in range(0, iters):
        pt = build_plaintext(target_length, dict2)
        n += target_length
        temp = reshape_cipher(to_numpy_mat(pt), 1)
        for col in range(0, temp.shape[1]):
            current = temp[:, col]
```

```
            unique, counts = np.unique(current, return_counts = True)
            freq_dict = dict(zip(unique, counts))
        for letter in freq_dict.keys():
            freq_list[int(letter)] += freq_dict[letter]
    # get freq for test1 pt
    for i in test1_list:
        n += target_length
        temp = reshape_cipher(to_numpy_mat(i), 1)
        for col in range(0, temp.shape[1]):
            current = temp[:, col]
            unique, counts = np.unique(current, return_counts = True)
            freq_dict = dict(zip(unique, counts))
        for letter in freq_dict.keys():
            freq_list[int(letter)] += freq_dict[letter]
    letter_dist = freq_list
    letter_dist[:] = [round(i / n, 6) for i in letter_dist]
    return letter_dist
```

Where `freq_dict` stores the frequencies for each character occurrence, `build_plaintext` creates a randomly generated 500 character plaintext from the given dictionary (word_dictionary_test2.txt), `to_numpy_mat` converts the characters to their ASCII value, and `reshape_cipher` reformats the ciphertext into 1 row of size 500.

The frequency character distribution for test 1 can be easily calculated using its given plaintext. After converting the five test 1 plaintext into ASCII numbers using `to_numpy_mat`, `np.unique` is then used to return the number of times each unique ASCII number occurs

(therefore also calculating the frequency of character distributions for test 1). Calculating for test 2 is the same as test 1, except we generate our own 5 random plaintexts using `build_plaintext`. This whole process is repeated 1,000 times to get as close to an accurate frequency distribution as possible. After running the code, the resulting frequency character distribution for each iteration was averaged out. The distribution was usually around 0.104728, 0.051404, 0.018165, 0.041384, 0.034082, 0.110717, 0.010791, 0.018262, 0.021635, 0.07826, 0.002, 0.0037, 0.046583, 0.028245, 0.055535, 0.062084, 0.027521, 0.0004, 0.064512, 0.092974, 0.05746, 0.031996, 0.007847, 0.0074, 0.0014, 0.017022, 0.003892 for characters " ", and "a" through "z" respectively.

After determining the frequency of character distributions, we needed to determine the possible key length used to encrypt the plaintext. To help determine this, the Index of Coincidence (IC) was used, where the IC calculates the probability of drawing the same character twice in a row. `calc_ic` was coded to simulate the IC formula:

$$\mathbf{IC} = c \times \left(\left(\frac{n_\mathrm{a}}{N} \times \frac{n_\mathrm{a} - 1}{N - 1} \right) + \left(\frac{n_\mathrm{b}}{N} \times \frac{n_\mathrm{b} - 1}{N - 1} \right) + \ldots + \left(\frac{n_\mathrm{z}}{N} \times \frac{n_\mathrm{z} - 1}{N - 1} \right) \right)$$

Where c is the normalizing coefficient (26 for English), n_a is the number of times the character "a" appears in the text, and N is the length of the text.

Knowing that the encryption algorithm is based on a poly-alphabetic substitution cipher, we assumed that each character mapped to the same k[i] shift in the plaintext will be shifted the same amount. So, if we were to treat the shift as a char (which it essentially is), we can calculate the IC for all characters mapped to the same k[i] shift. After calculating the IC for multiple k[i] shifts on sample random generated plaintexts, we noticed that their IC value is

equivalent to 1.73, which is also the IC for the English language. This means that an IC of around 1.73 can be used to determine the end of a key cycle (i.e. the length of a key).

To test this theory, we calculated the IC for every key length assumption on a sample ciphertext.

```
def calc_cipher_ic(cipher):
    ic_list = []
    ic_bar = 0
    for col in range(0, cipher.shape[1]):
        current = cipher[:, col]
        unique, counts = np.unique(current, return_counts = True)
        ic_list.append(dict(zip(unique, counts)))
    for i in ic_list:
        ic_bar += calc_ic(i)
    ic_bar /= len(ic_list)
    return ic_bar
```

Where `ic_list` stores all the frequencies taken at the end of the key for key lengths 1 through 24 (if key length = 5, `ic_list` stores values only from characters mapped to k[4]) and `ic_bar` returns the IC value for the current key length. The result was that only the key lengths that match the actual key length used to encrypt the plaintext had an IC around 1.73, whereas the others were around 1.

After figuring out how to determine the key length, we needed to determine the shifts needed to decrypt the ciphertext.

```
def find_key(cipher_numpy, letter_dist, key_length):
    key = [0] * key_length
    cipher_cols = reshape_cipher(cipher_numpy, key_length)
    for col in range(0, cipher_cols.shape[1]):
        most_likely_k = 0
        max_chi = 0
        for shift in range(0, 27):
            current = cipher_cols[:, col]
            current = [(row - shift) % 27 for row in current]
            temp_chi = chi_val(current, letter_dist)
            max_chi = max(temp_chi, max_chi)
            if temp_chi == max_chi:
                most_likely_k = shift
        key[col] = most_likely_k
    return key
```

Where `key` stores the key shifts from k[0] to k[key length] and `chi_val` is the function to help determine the likelihood of the possible shift to be accurate simulating the X distribution:

$$\chi = \sum_{i=1}^{c} n_i f_i$$

Where n_i are the observed column letter frequencies and f_i are the relative letter frequencies. The resulting key is then used to decrypt the ciphertext.

Combining Approaches

Despite the fact the frequency analysis approach works well regarding test 2, it isn't without its limitations. For instance, unlike test 1, it doesn't account for random insertions and only assumes a cyclical key. For both approaches, the smaller the key size, the more accurate their prediction is, but the larger the key size gets, the less accurate the prediction gets as well. To be specific, the approaches would have around 99% accuracy for a key size smaller than 15, 50% accuracy for a key size around 20 and the percentage decreases from there.

To mitigate the random insertion problem not taken into account for the frequency analysis approach and for a more cohesive program, both approaches were combined together.

```
def main():
    parser = argparse.ArgumentParser(description='Given a ciphertext,
attempts to decrypt it into the correct plaintext.')
    parser.add_argument('ciphertext', help='The ciphertext to be
analyzed.')
    cmdline = parser.parse_args()
    ciphertext = cmdline.ciphertext
    message_pp = ic_attack(ciphertext)
    print(unique_shift_attack(message_pp, ciphertext))
```

The entered ciphertext would initially go through the frequency analysis scheme (`ic_attack`) where a potential plaintext is returned. That plaintext is then entered into the unique shift scheme (`unique_shift_attack`) as a 6th plaintext and the algorithm in the scheme predicts

the plaintext that best matches the entered ciphertext, which somewhat mitigates the random insertion issue.

Another function was also added in an attempt to deal with the random insertions even more.

```
def ic_attack_random_insert(ciphertext, L=500, test2_dict=dict2,
test1_list=test_list):
    message_pp = ic_attack(ciphertext, L, test2_dict, test1_list)
    ciphertext_pp = ciphertext
    insertions = len(ciphertext) - L
    max_executions = 20
    for insertion in range(min(max_executions, insertions)):
        words = message_pp.split(' ')
        bad_index = -1
        for i in range(len(words)):
            if words[i] not in test2_dict and (i + 1) !=
len(words):
                if words[i + 1] not in test2_dict:
                    bad_index = sum([len(word) for word in
words[:i]]) + i
                    break
        if bad_index == -1:
            return message_pp
        else:
```

```
            ciphertext_pp = ciphertext_pp[:bad_index] +
ciphertext_pp[bad_index + 1:]
            message_pp = ic_attack(ciphertext_pp, L, test2_dict,
test1_list)
    insertions_left_over = len(ciphertext_pp) - L
    if insertions_left_over > 0:
        for insertions in range(insertions_left_over):
            ciphertext_pp = ciphertext_pp[:bad_index] +
ciphertext_pp[bad_index + 1:]
        message_pp = ic_attack(ciphertext_pp, L, test2_dict,
test1_list)
    return message_pp
```

Where `message_pp` takes in the somewhat decrypted plaintext after it runs through `ic_attack`, `bad_index` finds possible indexes where the random insertions occur, and `ciphertext_pp` stores the decrypted plaintext without the bad index. `Ic_attack_random_insert` finds the bad index by comparing each word in the decrypted plaintext with the dictionary, when it starts to detect gibberish, it knows that a random insertion occurred and returns the location where the gibberish started. The code then removes the bad index and returns a new decrypted plaintext, which is then fed through the `ic_attack` code one more time. We then replaced `ic_attack` in the above main code with `ic_attack_random_insert.` Unfortunately, `Ic_attack_random_insert` only works if the insertions are in the same relative location.

Problems would arise if `ic_attack` returns a plaintext with many errors due to potential wrong shifts (i.e. returns “hahpy” instead of “happy”). To mitigate this issue, the generated plaintext is run through `fit_to_dictionary`.

```
def fit_to_dictionary(m_prime, L, test2_dict):
    bad = True
    change = False
    words = m_prime.split(' ')
    while bad:
        for i in range(len(words)):
            if (i + 1) == len(words):
                words[i] = fix_last_word(words[i], test2_dict)
                bad = False
            elif words[i] not in dict2:
                words, change = match_in_dictionary(words, i,
test2_dict)
                if change:
                    words = ' '.join(words).split(' ')
                    change = False
                    break
    m_prime_prime = ' '.join(words)[:L]
    m_prime_prime_length = len(m_prime_prime)
    if m_prime_prime_length < L:
        m_prime_prime = m_prime_prime + m_prime[m_prime_prime_length -
L:]
```

```
return m_prime_prime
```

Where `words` stores a list of all words from the plaintext, `fix_last_word` fixes the last word in the plaintext, and `match_in_dictionary` changes the present word to the one it most likely should be from word_dictionary_test2.txt. Essentially what `fit_to_dictionary` is doing is fixing any potential minor shift errors that occurred when running through the `ic_attack` code by matching a misspelled work to the version it most likely should be according to the given dictionary. This will help `ic_attack` return a more accurate plaintext even if there are shift errors.

Extra Credit

We believe our approaches/program are strong enough to attack both extra credits. Extra credit 1 needs to take in more than 5 plaintexts and plot runtime as a function of the number of plaintexts, and extra credit 2 deals with a larger dictionary size and L >= 500. We had to generalize/rewrite parts of our code to accommodate those new conditions. The main part fixed was accommodating the code to fit a variable ciphertext and plaintext size L >= 500 instead of 500, which in most cases was simply setting the target length = size of the ciphertext. This was enough adjustment for the frequency analysis approach, with the assumption that the length of the sample plaintexts will be equal to the length of the entered ciphertext (i.e. if ciphertext L = 550, we would use plaintext L = 550).

Since the unique shift approach was based on a plaintext of size 500, the block size was hard coded in terms of that size. To accommodate for a potential larger plaintext, a function was added in order to compute the right block size for the program to work as intended.

```
def calculate_block_size(L):
    block_size_remainders = [(bs, L % bs) for bs in range(70, 85)]
    min_block_size_remainders = min(block_size_remainders,
key=lambda x: x[1])
    max_block_size_remainders = max(block_size_remainders,
key=lambda x: x[1])
    if min_block_size_remainders[1] == 0:
        block_size = min_block_size_remainders[0]
        return block_size, int(L/block_size)
    else:
        block_size = max_block_size_remainders[0]
        return block_size, int(L/block_size) + 1
```

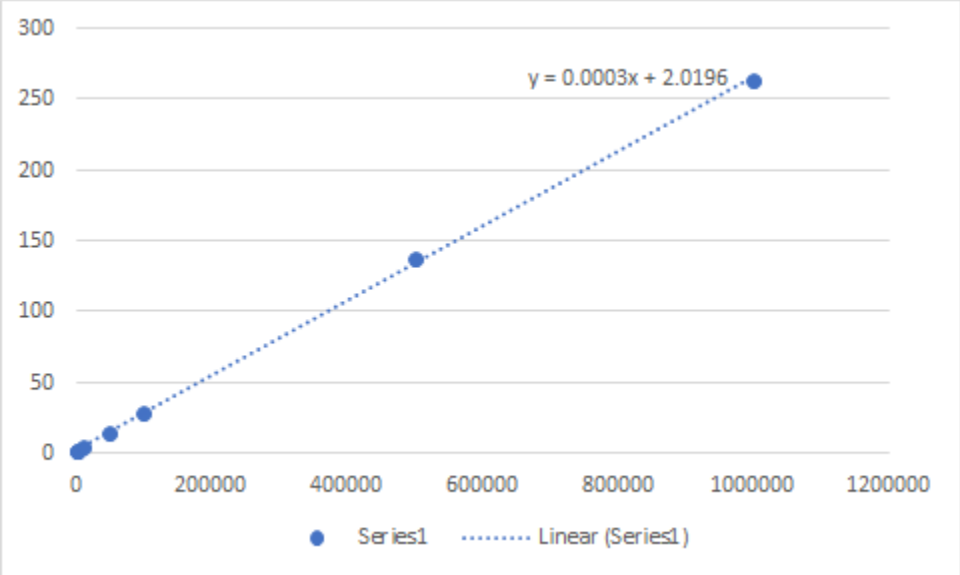
Where `block_size_remainders` return tuples of block size between 70 and 85 (if the block size does not divide evenly, whatever characters are left are put in the last block) and `min_block_size_remainders` and `max_block_size_remainders` work together to change the blocks to a size that evenly distributes all the characters (so that the last block size won't be too different from the others).

Now, with these changes in place, we believe our program will be able to handle the extra credit test cases.

Test 1 Extra Credit

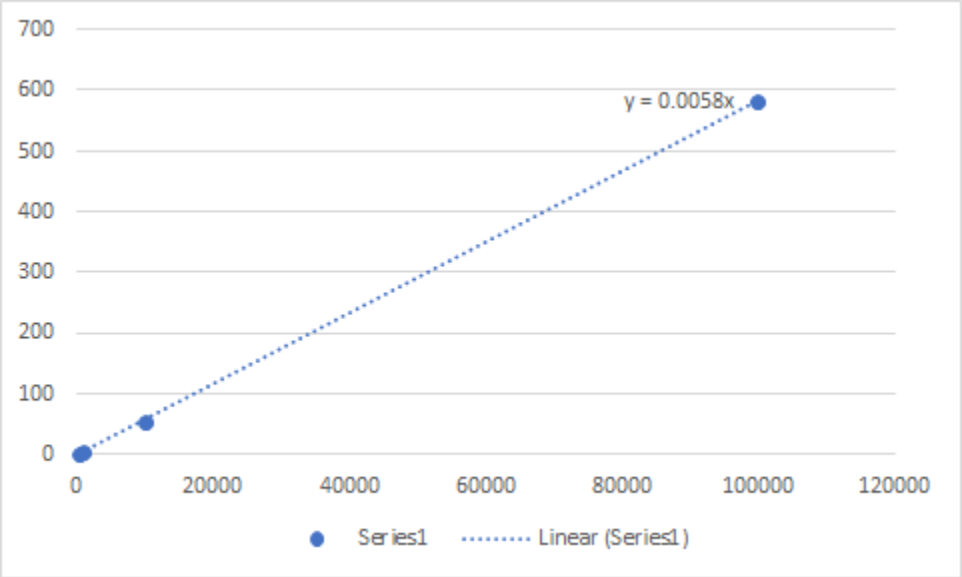
To test how well our program can handle a large number of plaintexts, we ran a code that records how long it takes to run x number of plaintexts. The result of our test is shown below for a fixed *L* and variable number of plaintexts:

L	plaintexts	time (s)
500	5	1.64
500	100	1.9
500	1000	1.91
500	10000	4.35
500	50000	14.43
500	100000	28.43
500	500000	137.44
500	1000000	263.6



The result of our test is shown below for a fixed number of plaintexts and a variable *L*:

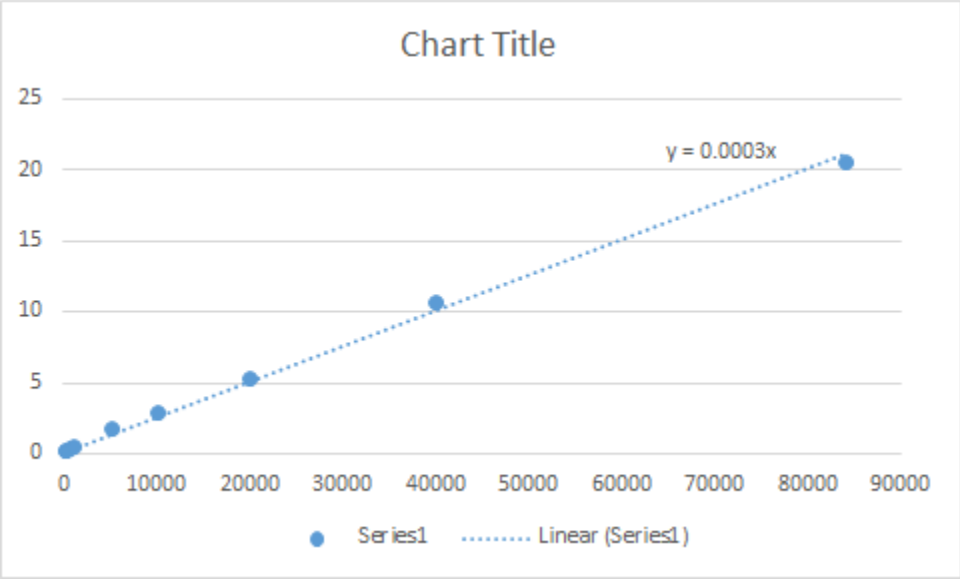
plaintexts	L	time (s)
5	500	1.64
5	1000	4.02
5	10000	54.72
5	100000	583.14



Test 2 Extra Credit

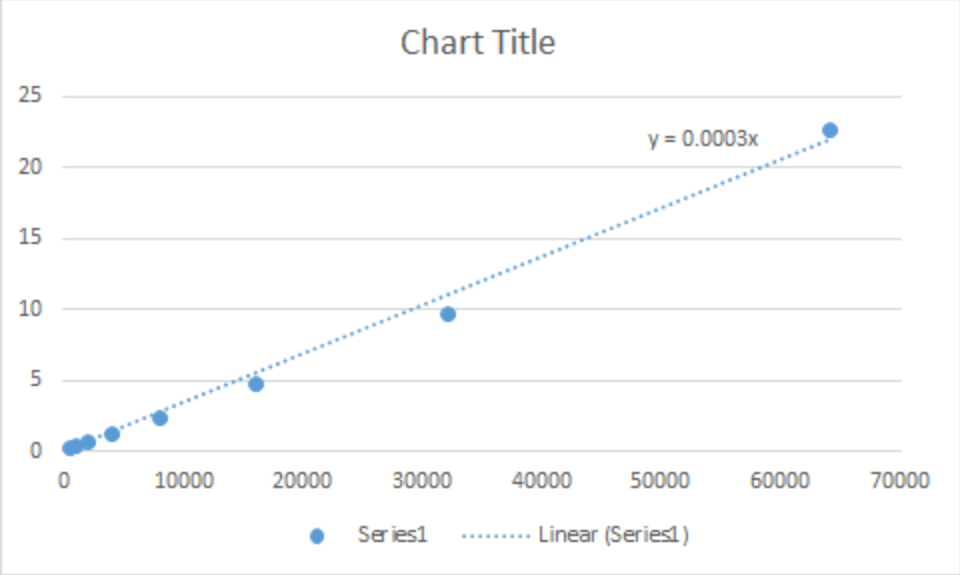
To test how well our program can handle a large number of dictionary words, we ran a code that records how long it takes to run *u* number of words in a dictionary. The result of our test is shown below for a fixed *L* variable *u*:

L	u	time
500	50	0.31361
500	100	0.33347
500	500	0.47978
500	1000	0.60632
500	5000	1.79728
500	10000	2.92156
500	20000	5.35700
500	40000	10.7116
500	84032	20.5708



The result of our test is shown below for a fixed *u* variable *L*:

L	u	time
500	40	0.26993
1000	40	0.43274
2000	40	0.70662
4000	40	1.25134
8000	40	2.44861
16000	40	4.77969
32000	40	9.75437
64000	40	22.7205



Conclusion

Overall, we believe this project is a success, as we were able to solve test 1 most of the time and test 2 with mild success as stated in the project requirements. Our algorithms were also relatively successful in approaching the two extra credits. However, these approaches were only successful because of the limitations specified, which is the plaintext and ciphertext is over 500 characters long with a key length t <= 24, at most 50 random insertions, and encryption

algorithm based on a poly-alphabetic substitution cipher. As mentioned in parts of the report, if the key t contains only unique shifts, the accuracy of our approaches decreases starting from t = 20, and would not be successful for any t > 24. The same is true for the random insertions. More random insertions would create too much noise in the ciphertext for our approaches to decrypt.

An improvement to our approaches could include writing a better finding random insertions program for both test 1 and test 2. As mentioned in the report, test 2 random insertions code only succeeds mostly when there is a low number of random insertions. Although the test 1 random insertion function is much more successful, it is not without its limitation, with its accuracy of detecting/mitigating random insertions decreasing with larger keys and more insertions for a more complicated/variable key. More time or a different approach in the random insertion function could help with dealing with more random insertions.

This project demonstrates the importance of having large, unrepeated keys and randomness in encryption to help keep information secure. Small repeated keys and low randomness increase the chances an adversary can eavesdrop and modify the information really easily, as our two approaches take max 1 minute to run. Even a decent amount of random insertions required us to write more code and cases to deal with them, and even that was not 100% accurate all the time. All in all, good encryption is important to keep information truly secure.

References

"Index of Coincidence." Wikipedia, Wikimedia Foundation, 28 Dec. 2020,

en.wikipedia.org/wiki/Index_of_coincidence.