Cryptanalysis of a Substitution Cipher

Zuan Liang

Mike Singh

Peter Smondyrev

Prasanth Venigalla

Introduction

An L-Symbol challenge ciphertext was given that needs to be decrypted. Two different tests are taken to decrypt the given ciphertext. Our goal was to replicate the encryption scheme and figure out the best way they can be decrypted. In the First test we are given five plain texts and our goal is to find a match between one of the samples and cipher text. The second test involves using a dictionary text to decrypt the given ciphertext. At the end of this document is an analysis of a permutation cipher encryption and cryptanalysis approach.

Implementation

For Test 1:

1. Download Liang_Singh_Smondyrev_Venigalla_DecryptTest1.cpp
2. Compile and run
3. The program will prompt for cipher text

For Test 2:

1. Download Liang_Singh_Smondyrev_Venigalla_DecryptTest2.py and english_words.txt and place in the same directory
2. Compile and run
3. The program will prompt for Path To Comma-Separated CipherText File

Test 1

FORMAL:

For Test 1, we were given five different plaintexts, and told they would be the possible candidates for the ciphertexts that we were tasked to break. Since there were only five candidate plaintexts, we decided to do some analysis beforehand in order to use this information in our program.

First, we realized that there are multiple characters (b, j, k, q, v, x, z) in the keyspace, that can only have one key mapped to it in the ciphertext. So even though we don't have knowledge of the scheduling algorithm, the fact that there won't be multiple different keys mapped to each of these letters in the ciphertext, gives us the opportunity to identify these characters in the ciphertext more easily.

Then, we took advantage of being given the five different plaintexts in advance, by designing a program that returned the occurrences of each letter in each plaintext.

```
Array counts initialized to size 122
Integer i initialized to 0
for i < length of Candidate Plaintext:
    find ascii value of letter at indice i in plaintext
    Use the ascii value of letter as indice in counts and increment its value
    increment i
```

```
a ------- occurrences: 37
b ------- occurrences: 9
c ------- occurrences: 17
d ------- occurrences: 12
e ------- occurrences: 51
f ------- occurrences: 3
g ------- occurrences: 14
h ------- occurrences: 10
i ------- occurrences: 38
j ------- occurrences: 1
k ------- occurrences: 5
l ------- occurrences: 16
m ------- occurrences: 15
n ------- occurrences: 25
o ------- occurrences: 27
p ------- occurrences: 17
q ------- occurrences: 2
r ------- occurrences: 37
s ------- occurrences: 48
t ------- occurrences: 33
u ------- occurrences: 9
v ------- occurrences: 4
w ------- occurrences: 6
x ------- occurrences: 0
y ------- occurrences: 10
z ------- occurrences: 1
<space> ------- occurrences: 53

find character 'b' at position240
find character 'b' at position267
find character 'b' at position282
find character 'b' at position354
find character 'b' at position389
find character 'b' at position390
find character 'b' at position414
find character 'b' at position468
find character 'b' at position477
```

We found that the occurrences of ‘b’ differed in each of the plaintexts, so we added a feature to our analysis program that returned the different positions for ‘b’. Therefore, by analyzing the occurrences of ‘b’, as well as confirming its position in the ciphertext, we will be able to uncover which candidate plaintext was used in the encryption scheme. To test our theory, we created a program that encrypted each of the candidate plaintexts, by using the scheduling algorithm of “j mod length(list)”.

Enter the plaintext: masterwork swept squanders grounders idolatries swapper pave croupier dramatists magnified hypnoses delivery tassels marquise entailments circuits crampon nationalism nictitation anticapitalists dancingly soothly patriarchs goodie whickers baggy omnipotent sadist ameba processions beggary rename nonassertively macerators lectureship shipwrights sadden backups rhymer offstage schistose ebbs restorer graecizes subjoining leathering smocks leukocyte waled temperer embroglios bolivar repines teletyp

The ciphertext is:

50,79,54,42,89,18,13,37,53,99,105,91,13,86,17,3,71,54,103,38,70,20,24,86,40,8
8,44,48,53,7,55,41,87,86,40,88,59,36,24,68,64,70,2,53,61,6,91,35,47,33,79,56,1
7,86,40,59,17,79,62,67,8,104,90,68,55,56,78,102,53,4,24,73,34,82,69,25,39,54,
3,51,101,82,14,48,74,36,30,11,80,1,31,84,95,56,19,7,91,102,47,101,87,28,98,36
,62,6,73,29,46,21,14,91,54,86,98,88,97,82,70,40,103,38,61,47,89,27,57,41,81,1
6,39,10,50,86,5,81,91,101,9,36,18,104,55,65,21,88,46,104,53,34,50,56,72,20,97
,77,34,42,61,68,19,43,98,36,51,82,31,77,78,104,81,11,21,14,85,36,72,20,44,69,
74,85,61,104,34,56,78,25,70,98,39,91,42,47,101,12,43,77,9,36,5,48,64,29,31,88
,72,68,25,60,98,29,8,56,69,25,73,36,79,40,9,84,54,31,93,68,94,12,78,67,97,33,5
8,65,9,99,57,90,47,27,23,16,93,48,95,59,96,82,5,11,17,7,2,86,5,42,35,54,70,12,
61,91,42,71,14,82,57,23,34,8,17,73,72,104,89,88,91,36,72,20,88,66,23,86,93,48
,70,90,95,101,18,28,19,70,50,6,27,20,94,77,70,91,54,86,40,21,78,62,80,98,95,4
4,50,14,9,6,73,34,42,37,18,91,100,10,89,104,21,38,53,67,88,84,61,56,27,88,84,
36,17,33,18,65,93,75,3,88,101,54,14,1,24,28,19,27,23,14,9,99,55,56,88,92,90,7
5,95,82,57,90,44,68,30,0,54,81,16,48,57,71,47,104,83,11,54,25,96,88,57,44,80,
23,23,91,4,53,89,88,21,37,53,67,18,97,93,53,14,6,9,65,26,67,88,4,54,38,23,52,7
2,11,19,65,74,48,101,98,89,14,85,58,80,40,39,77,93,66,51,82,94,104,99,51,8,98
,32,38,99,37,9,29,2,67,46,33,69,98,89,12,92,81,80,82,17,28,90,86,40,35,57,82,2
3,40,72,48,64,65,96,88,45,23,94,10,78,62,70,53,35,18,57,56,61,20,22,91,59,42,
89,98,57,2,95,56

Our final program searched for the key found at the first position that we found ‘b’, in each of the candidate plaintext. Then we looped the entire ciphertext, to find all of the other keys that were the same as the key found at the first position of ‘b’. If the occurrences of this key, is equal to the occurrences of ‘b’ in the candidate plaintext, then that can be the encrypted plaintext. Finally, to confirm if this plaintext is encrypted by the ciphertext, we access the positions of the keys in the ciphertext at their respective positions for ‘b’ in the plaintext. If the keys at these positions are equal, then the chosen plaintext will be returned.

```
function match_pattern{
// analysis ciphertext based on location and occurrence
    Initialize integers a, b, c, d, f;
    Initialize integers count_a, count_b, count_c, count_d, count_f to 0

    Set a to the key value in the ciphertext at indice 240 // first b-position at plaintext 1
    Set b to the key value in the ciphertext at indice 148 // first b-position at plaintext 2
    Set c to the key value in the ciphertext at indice 34 // first b-position at plaintext 3
    Set d to the key value in the ciphertext at indice 18 // first b-position at plaintext 4
    Set f to the key value in the ciphertext at indice 50 // first b-position at plaintext 5
    Initialize i to 0

for (i < Ciphertext Length; increment i) {

        if key at indice i of ciphertext is equal to key at a:
            increment count_a

        if key at indice i of ciphertext is equal to key at b:
            increment count_b

        if key at indice i of ciphertext is equal to key at c:
            increment count_c

        if key at indice i of ciphertext is equal to key at d:
            increment count_d

        if key at indice i of ciphertext is equal to key at f:
            increment count_f

    }

Compare the count_a, count_b, ... and a,b,... found in the ciphertext
to the individual candidate plaintext positions and counts for 'b'

}
```

INFORMAL:

Approach:

According to candidate plaintexts given, we know that the number of occurrences of ‘b’ is different for each candidate.

Plaintext 1: The b. character has 9 occurrences.

Plaintext 2: The b. character has 11 occurrences.

Plaintext 3: The b. character has 10 occurrences.

Plaintext 4: The b. character has 8 occurrences.

Plaintext 5: The b. character has 7 occurrences.

Since the letter ‘b’ is always mapped to the same integer, occurrences of letter ‘b’ in plaintext will be the same as occurrences of integer (substituted letter ‘b’). Their location will also be the same. Our approach is to use a pattern of ‘b’ and number of occurrences of ‘b’ to determine which candidate plaintext to return.

Test 2

Approach 1:

INFORMAL:

Our first approach for attempting to decrypt Test 2 leverages a graph-like search algorithm to find the right plaintext for the given ciphertext. We're told that the cipher-text will be generated by selecting a couple random words from the given dictionary file, that is quite large in itself so relying on the approach used for Test 1 is inefficient. Similarly, a direct brute-force approach through every single possibility would simply be unfeasible in terms of efficiency, so we had to rely on approaches than what would be common. One of these methods would be to use the length of the ciphertext as a distinctive factor in grouping words in plaintext with similar length, therefore for a given length of a ciphertext we would have a variance of words possible. Where the distribution of the word length is more uncommon, our approximations are more certain than those in the center of the distribution near the standard deviation and mean. In order to accomplish this we traverse words in the vicinity treating this as a graph problem starting out, to check out adjacent possibilities and stopping when we get a match, or return the closest set available.

FORMAL:

```
cipher_lists = random_size_sub_lists(make_list(get_ciphertext()))


for cipher_text in cipher_lists:
        check if len(cipher_text) = len(iter_words)
                if equal index then append word to list
        while (there exists words in constructed list)
                while (maps of word-char -> cipher-num)
                        if (map of word-char -> cipher-num doesn't exist)
                        modify dictionary of letter keys to reflect
                                this and continue to next item
                        else if (word-char -> cipher-num exists)
                                delete the pair in question and
                                move to next item
```

Approach 2: (Note: we didn't implement Approach 2 for Test 2 but below is a description of what we were thinking of doing)

INFORMAL:

Our second approach for Test 2 leverages the frequencies of the letters in english and permutes them.

The algorithm tries to reverse a key from the ciphertext for this approach and if successful we should get back the plaintext from input of a cipher text. Both should be stored as strings accessible by index in its final product for the key and the cipher text, one per set of ciphered numbers that correspond to a single plaintext character. This approach measures all the lengths of the words in the large english dictionary, which will allow us to know how many ciphered plaintext values would match for that given letter. And for every letter, we rely on the key in storage to see if there is an appropriate match.

What about the values that remain unidentified?

>> If no match exists, we can add it to a map of unidentified letters

>> Rely on the length computed for every word in the large english dictionary

>> For every unidentified mapping, leverage what letters are known and the letter frequencies of the entire english alphabet to identify the first letter and subsequent letters

adjacent in the words in the dictionary list. Assuming we fail to identify a word in the end we also keep track of a guess list that will be used to output a word in the case all else fails

FORMAL:

```
//reverse the key
For i = 0 to english_dict.legnth():
     If the plaintext[i] = reverse_key[ciphertext[i]]:
           known mapping, set to mappings_known
     Else:
           Set the reverse_key[i] == null // because its unknown
           //put in mappings_unidentified


guess = []
mappings_unidentified = {}  //set the average frequencies of english letters
indexes_unidentified = []   //where unidentified mappings are located
mappings_known = {}         //known mappings
english_dict = {}           //words from full english dictionary
for i from 0 to 105:
        if reverse_key(i) is in mappings_known.keys():
                        //decrement average if found
                  mappings_unidentified[i] = mappings_unidentified[i]--

for s in mappings_unidentified:
        if mappings_unidentified[s] > 0:
                guess.append(s)
        else
                continue; //do nothing if the letter appeared too frequently

for mappings_unidentified.key in indexes_unidentified:
        i = mappings_unidentified.key;
        while i > 0:

        //move backward in indexes


If estimated size of the word based on the unidentified mapping values is
> than the mappings_known:
     If letter in english_dict is one of the unknown positions:
           See if it is a word from the word dictionary:
                If not, slowly check the adjacent letters:
                If enough characters match with a known word then addit to guess
 Else:
        continue; as its unknown
Print the guess
```