Cracking the Monoalphabetic Substitution Cipher

Mr. Carlos Santana
Masters in Computer Science - Cybersecurity
Jeffrey Duffany
Department of Computer Science
Polytechnic University of Puerto Rico

Abstract *— Cryptography is the cornerstone of secure communication. It is the study of techniques by which a message, the plaintext, is transformed into an obfuscated form, the ciphertext. One of the earliest cryptographic techniques used by the Romans is the monoalphabetic substitution cipher. It replaces one letter in a message with another. This approach has one glaring weakness, in that the frequency of letters is preserved in the ciphertext. This renders messages encrypted via monoalphabetic substitution vulnerable to frequency analysis attacks. Frequency analysis is the knowledge of how frequently letters are used in a language. A persistent attacker that has intercepted the message can conduct frequency analysis to "crack" a monoalphabetic substitution cipher. An algorithm can also be created to automate the process and conduct frequency analysis attacks on ciphertext to crack the cipher in minutes, decrypting the message and exposing the plaintext and its contents to an unintended recipient.*

Key Terms*—Cryptography, Decryption, Frequency Analysis, Monoalphabetic Substitution.*

INTRODUCTION

With the proliferation of online transactions and online communications came the need to ensure that those communications are as secure as they can be. The field of cryptography was born from this need for security. Cryptography, derived from the Greek word parts *kryptos* ("hidden") and *graphein* ("to write"), is the practice of communication techniques in such a way that only the sender and the intended recipient can understand the message's contents. Cryptography is the cornerstone of information security, with its techniques forming the basis of secure communications worldwide.

The basis of cryptography is the use of a code or algorithm, called a *cipher*, to transform the original message, called the plaintext, into its hidden or obfuscated form, called the ciphertext. This transformation is called encryption. The reverse process, in which the cipher is used to transform the ciphertext back into the plaintext, is called decryption. The researcher will be focusing on the decryption component of the process in this document.

There are two kinds of cipher techniques traditionally used: transposition and substitution. Transposition ciphers alter the order of letters within the message without changing the letters themselves; for example, "automobile" could be encrypted into "ebatmulioo". Substitution ciphers replace each letter in the message with another letter; for example, "automobile" could be encrypted into "fryplpbmie" [1]. The type of cipher the researcher will be focusing on in this document is a substitution cipher called the monoalphabetic substitution cipher, or simple substitution cipher. This technique pairs each letter of the alphabet the message was written in with another letter in the same alphabet; then the message is rewritten, except that every letter is replaced by the letter it was paired with [2].

BACKGROUND

The researcher came to the Polytechnic University with an undergraduate degree in computer science. As a graduate student at the university, he wished to complete a project that would put his training in programming to good use, while being useful to the field of cryptography and information security. This is when the mentor presented him with the concept of conducting frequency analysis attacks to crack a monoalphabetic substitution cipher and use that cipher to decrypt some ciphertext.

Problem

As the researcher mentioned earlier, monoalphabetic substitution ciphers work by pairing letters in the alphabet with other letters in that same alphabet, then rewriting the message by replacing each letter with the letter it was paired with. However, cryptographers will recognize that this cipher has a glaring weakness. Because each occurrence of a given letter is replaced with the same letter every time, the number of times that letter appears can still be detected. For example, if "automobile" is encrypted as "fryplpbmie", the presence of two p's reveal that whatever letter the p is replacing appears twice in the original word. The study of how frequently a letter appears within a ciphertext is called frequency analysis. This study forms the basis for decrypting the monoalphabetic substitution cipher.

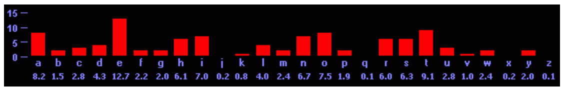


Figure 1

English Letter Frequencies

There are tendencies with every language for certain letters to occur more frequently than others. As demonstrated by Figure 1, the letter E is the most common letter in the English alphabet, generally composing up to 12.7% of the letters in a document. E is followed by T, A, O, I, and N. Cryptographers recognize that these tendencies will likely hold with most pieces of plaintext. And thanks to the monoalphabetic cipher's fatal flaw, these letter frequencies are preserved when the plaintext is converted into ciphertext. Therefore, a persistent cryptographer can use frequency analysis to try and guess which letter in the ciphertext maps to which letter in the plaintext [3].

This was the principle behind this project. The researcher planned to create an algorithm, in a programming language of his choice, that would execute the following steps:

- Receive a piece of ciphertext from the user.
- Conduct frequency analysis on the ciphertext.
- Generate an initial mapping of ciphertext letters to plaintext letters.
- Use a series of English rules and tendencies to fix the mapping.
- Use the mapping to translate the ciphertext into plaintext.
- Return the plaintext to the user.

Software

As this was a programming project, very little equipment was needed. The researcher conducted all the work on his personal computer, which runs on 64-bit Windows 10. Visual Studio Code was used to create and run the program, and the program was written in Python. Python was selected for this programming language because of the researcher's familiarity with the language, as well as its powerful features that made writing this algorithm significantly easier.

Program Details

```
ct = readfromfile("duffcaesar.txt")
(ctl, SPACE_INDICES, ctw,
CIPHER_WORDS_WITH_DOUBLES, WORDS_BY_SIZE)
= textstats(ct)

CIPHER_FREQ = {}
(CIPHER_FREQ, ctl_lettersonly) =
countletters(ct)
CIPHER_FREQ_PCT = countpcts(CIPHER_FREQ,
ctl_lettersonly)
CIPHER_FREQ_SORTED =
sorted(CIPHER_FREQ_PCT.items(),
key=lambda kv:(kv[1],kv[0]),
reverse=True)
CIPHER_FREQ_LETTERSONLY = []
for i in CIPHER_FREQ_SORTED:
    CIPHER_FREQ_LETTERSONLY.append(i[0])

MAPPING =
create_mapping(CIPHER_FREQ_SORTED,
FREQ_SORTED)
MAPPING = check_mapping(MAPPING)

pt = translate(ct, MAPPING)
print("The ciphertext:\n{0}\nmay decode
to the following:\n{1}".format(ct,pt))
```

Figure 2

Program Main Body

This program runs in four phases, as shown in Figure 2: collecting the ciphertext and relevant data about the ciphertext, conducting frequency analysis, creating (and perfecting) the letter mapping, and translating the ciphertext into plaintext, which is then printed on the screen. The code displayed in Figure 2 is the main body of my program, with the comments omitted to save space. Any import statements and constants created for this program will be listed in Appendix A. Furthermore, most of the code in my program is contained within functions. To aid in understanding, the researcher will go over each section and the relevant functions individually.

Phase 1: Retrieving Ciphertext Data

```
def readfromfile(filename: str) -> str:
    with open(filename, 'r') as f:
        text = f.read().lower()
    print("Ciphertext read from:
{0}".format(filename))
    return text

def textstats(text:str) -> tuple:
    textlen = len(text)
    spaces = []
    for i in range(textlen):
        if text[i] == ' ':
            spaces.append(i)
    words = text.split(' ')
    doubles = []
    for word in words:
        if has_double(word):
            doubles.append(word)
    sizes = []
    big = len(max(words,key=len))
    for i in range(1, big + 1):
        lst = []
        for word in words:
            if word[-1] not in
FREQ_LETTERSONLY:
                word = word[:-1]
            if len(word) == i and word
not in lst:
                lst.append(word)
        sizes.append(lst)
    return (textlen, spaces, words,
doubles, sizes)
```

Figure 3

readfromfile(), textstats(), and has_double()

In the first phase, the program looks for a specified file in the current working directory and then runs readfromfile() on that file. This function is a straightforward repackaging of a built-in function. It opens the file in a read-only state, reads the file's contents, uses lower() to convert all the alphabetical characters into lowercase format to make mapping less of a headache later, and then deposits the results into a string object. It prints a feedback line to the user detailing that the ciphertext has been read from the specified file, then returns the string object to the user.

This string object is later passed to a separate function called textstats() to retrieve some additional statistical data about the ciphertext. Most of this statistical data will prove invaluable to the frequency analysis and mapping processes later. Five items are calculated and then returned to the program in the form of a 5-tuple. The five pieces of information retrieved from the ciphertext are as follows:

- The length of the ciphertext.
- The index of every space character in the ciphertext.
- A list containing every word in the ciphertext.
- A list of lists, where each list contains every word from the ciphertext that has a given length.
- Every word in the ciphertext that has a double-letter pair.

The last piece of information is collected to aid in the perfecting process during phase 3. It is aided by a third function called has_double(), which the researcher has placed in Appendix B. This function examines a word, letter by letter, and returns a Boolean value: TRUE if the word has a double-letter pair, and FALSE if it does not.

Phase 2: Conducting Frequency Analysis

```
def countletters(text:str) -> tuple:
    counts = {}
    lettersonly = 0
    for char in text:
        if char not in FREQ_LETTERSONLY:
            continue
        counts[char] = counts.get(char,
0) + 1
        lettersonly += 1
    return (counts, lettersonly)

def countpcts(dicto:dict, nums:int) ->
dict:
    counts_pct = {}
    for item in dicto.keys():
        counts_pct[item] =
dicto[item]/nums
    return counts_pct
```

Figure 4

Countletters() and countpcts()

In the second phase, the program takes the ciphertext received from the user and runs countletters() on the ciphertext. The function creates a dictionary, with the keys being each letter that was found in the ciphertext, and the values being how many times said letter appeared in the ciphertext. To conduct frequency analysis properly, the program omits any symbol that is not a letter; newlines, numbers, and punctuation symbols are not counted. As each letter is counted, a secondary statistic is calculated, measuring the total number of letters in the ciphertext. Both objects are returned to the program as an unordered pair.

The second function, countpcts(), operates on the information returned by countletters(). It takes the dictionary and makes a copy of its keys. It then takes the number of times the letter occurred in the ciphertext and divides it by the total number of letters in the ciphertext, to attain a percentage: the percent of the ciphertext letters that each letter constitutes. This percentage is what will ultimately be used to draw a comparison between the ciphertext letter frequency and the plaintext letter frequency.

Phase 3: Creating and Fixing a Letter Map

```
def create_mapping(cipher_dict: dict,
plain_dict: dict) -> dict:
    mapping = {}
    cipherlen = len(cipher_dict)
    for i in range(cipherlen):
        mapping[cipher_dict[i][0]] =
plain_dict[i][0]
    mapping.update(CONSTANT_SYMS)
    return mapping

def check_mapping(potential_map:dict) ->
dict:
    map_with_fixed_the =
check_the(potential_map)
    map_with_fixed_two =
check_twoletters(map_with_fixed_the)
    map_with_fixed_plurals =
check_plurals(map_with_fixed_two)
    map_with_fixed_w =
check_w(map_with_fixed_plurals)
    map_with_fixed_end =
check_endings(map_with_fixed_w)
    map_with_fixed_three =
check_threeletters(map_with_fixed_end)
    map_with_fixed_ing =
check_ing(map_with_fixed_three)
    map_with_fixed_doubles =
check_doubles(map_with_fixed_ing)
    newmap = {}
    newmap.update(map_with_fixed_doubles)
    newmap.update(CONSTANT_SYMS)
    return newmap
```

Figure 5

Create_mapping() and check_mapping()

Now comes the meat and potatoes of the program, so to speak. In the third phase, the program works with two constants; the first stores the plaintext letter frequencies sorted in descending order, and the second contains the ciphertext letter frequencies, also sorted in descending order. The sorted() function is used to store a list of tuples in both constants, with each tuple being a key-value pair in the dictionary these constants were created from. The first element of each tuple is the key, and the second is the value.

The first function, create_mapping(), takes both constants and creates a new dictionary. Then, a for-loop repeats as many times as there are unique letters in the ciphertext. The most common ciphertext letter is inserted as a key into the new dictionary, with its value being the most common plaintext letter. The same is done with the second most common letters in either list, then the third, then the fourth, and so on until all the letters in the ciphertext have a mapping to a letter in the plaintext. Then, the mapping is updated with a separate map, included in the constants section, that maps every symbol to itself to prevent them from being accidentally mapped to a letter. This dictionary is returned to the program.

This kind of blind matching is only going to be accurate in some cases. Therefore, this program also has a “check” function, called check_mapping(), that runs a series of test functions on the mapping to improve it. Each test function applies a different rule or tendency of the English language to the mapping, delivering an intermediate result to the next test in line. Once all the tests have been applied, a fresh mapping is created. The output of the final test and the constant symbols are merged with that mapping, and that mapping is delivered to the program as the mapping that will be used to perform the final translation.

Two principles are used extensively for these tests: whether a word is common, and how close an existing word is to its usual form. The latter is measured using a principle in linguistics called the Damerau-Levenshtein (D-L) distance. This metric is a measure of how many operations (insertions, deletions, replacements, or adjacent letter swaps) it will take to turn one word into another. For example, the words “fare” and “fair” have a D-L distance of 2 because the R and E must first be swapped, then the E must be replaced with an I. For the purposes of my program, a word is considered close to another if their D-L distance is 1.

The code for each of these tests is extensive, so they will be placed in Appendix C. The tests run are as follows:

- Check_the() looks for three-letter words that end in E. E is the most common letter in the English language, and one of the most common three-letter words is ‘the’, which happens to end in an E. Each of the potential words is checked against ‘the’ to see if any are already close to being correct. The most accurate one is assumed to be the real ‘the’, and the mapping for T and H is fixed accordingly.
- Check_twoletters() looks for two letter words that contain a T. There is a list of common two-letter-words in the constants section of my program, and this list is used as a reference for this function to fix the mapping of letters such as A, O, I, and R.
- Check_plurals() goes over every word in the ciphertext and checks to see if said word reappears elsewhere in the ciphertext with an additional letter. In English, whenever two versions of a word are present and the second version of said word has one more letter, chances are the second version of the word is the plural form of the first. In English, plurals are formed by adding an S to the end, in most cases. This will fix the mapping for S.
- Check_w() goes over the three-, four-, and five-letter words in the ciphertext and sees if any of them are close to a list of common words that contain the letter W. If a potential match is found, the mapping for W is fixed accordingly.
- Check_endings() extends the principle covered in check_plurals() by measuring what letters frequently end words in the ciphertext. The most common letter at the end of words is S because adding an S to the end of a word is how we pluralize them. The second most common letter at the end of words is D because adding a D to the end of a word is how we change it to past-tense.
- Check_threeletters() follows the same principle check_twoletters() does, except it applies the principles to three-letter-words.
- Check_ing() takes the principle held by check_endings() a step further now that more of the mapping has been fixed. It looks for the last three letters of every word that is at least 4 letters long and checks to see how close this ending is to ‘ing’. Those three letters are another very common ending because adding ‘ing’ to the end of a verb is how we change that verb into its continuous present form.
- Check_doubles() takes the list of words that have double letters, formed earlier, and checks to see if their double-letter pair is one of the double-letter pairs present in English. If it is not, the double pairs are replaced accordingly and examined to make sure they still form a word.

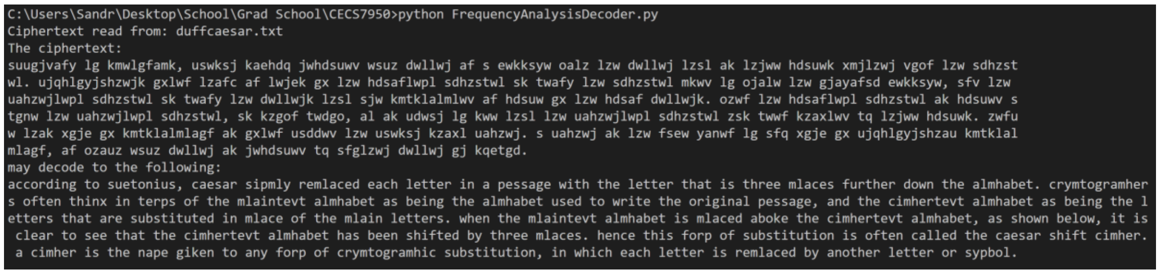
Phase 4: Decrypting into Plaintext

```
def translate(cipher:str, map:dict) ->
str:
    plain = ""
    for letter in cipher:
        plain += map[letter]
    return plain
```

Figure 6

Translate()

In the fourth and final phase, this program applies the final mapping to the ciphertext via the translate() function. The ciphertext is reconstructed, character by character, except that each letter from the ciphertext is replaced with its plaintext equivalent. The results are then printed to the user in a neatly formatted manner, with a comparison being offered between the ciphertext and plaintext.



```
C:\Users\Sandr\Desktop\School\Grad School\CECS7950>python FrequencyAnalysisDecoder.py
Ciphertext read from: duffcaesar.txt
The ciphertext:
suugjvafy lg kmwlgfamk, uswksj kaehdq jwhdsuwv wsuz dwllwj af s ewkksyw oalz lzw dwllwj lzsl ak lzjww hdsuwk xmjlzwj vgof lzw sdhzst
wl. ujqhlgyjshzwjk gxlwf lzafc af lwjek gx lzw hdsaflwpl sdhzstwl sk twafy lzw sdhzstwl mkwv lg ojalw lzw gjayafsd ewkksyw, sfv lzw
uahzwjlwpl sdhzstwl sk twafy lzw dwllwjk lzsl sjw kmtklalmlwv af hdsuw gx lzw hdsaf dwllwjk. ozwf lzw hdsaflwpl sdhzstwl ak hdsuwv s
tgnw lzw uahzwjlwpl sdhzstwl, sk kzgof twdgo, al ak udwsj lg kww lzsl lzw uahzwjlwpl sdhzstwl zsk twwf kzaxlwv tq lzjww hdsuwk. zwfu
w lzak xgje gx kmtklalmlagf ak gxlwf usddwv lzw uswksj kzaxl uahzwj. s uahzwj ak lzw fsew yanwf lg sfq xgje gx ujqhlgyjshzau kmtklal
mlagf, af ozauz wsuz dwllwj ak jwhdsuwv tq sfglzwj dwllwj gj kqetgd.
may decode to the following:
according to suetonius, caesar sipmly remlaced each letter in a pessage with the letter that is three mlaces further down the almhabet. crymtogramher
s often thinx in terps of the mlaintevt almhabet as being the almhabet used to write the original pessage, and the cimhertevt almhabet as being the l
etters that are substituted in mlace of the mlain letters. when the mlaintevt almhabet is mlaced aboke the cimhertevt almhabet, as shown below, it is
 clear to see that the cimhertevt almhabet has been shifted by three mlaces. hence this forp of substitution is often called the caesar shift cimher.
 a cimher is the nape giken to any forp of crymtogramhic substitution, in which each letter is remlaced by another letter or sypbol.
```

Figure 7

A Sample Run of the Frequency Analysis Decoder

Results

The results of the program run can be seen in Figure 7. In its current incarnation, this program runs in 1 second. It performs eight partial iterations, one for each test, and one full iteration for the translation.

As of the writing of this article, it has a couple bugs that have yet to be worked out. For instance, the letters P and M must be switched. However, most other words are correct.

Conclusion

Through the completion of this project, one can deduce that conducting frequency analysis to decrypt a message encrypted via the monoalphabetic substitution cipher is possible and attainable within a realistic timeframe. Early attempts to crack this cipher algorithmically were very time consuming to perform by hand, and even when run by a computer that could perform millions of operations a second, the brute-force approach would take several minutes to run. By applying a few rules first, one can shorten the amount of time this algorithm would take.

Future Work

For future iterations of this program, the researcher would probably begin by adding a final test to the check_mapping() method. It would go through all the remaining mappings and brute-force switching them with each other to see which one sticks. This process is normally time-consuming, but by expanding the number of mappings that are proven to be correct prior to running the brute-force check, this time can be shortened greatly. Segments of unnecessary code would be deleted, so that the program would be easier to understand. The tests can also be improved so they rely less on assumptions and more on hard evidence. The researcher could also investigate more letter frequency data to create more tests from. The final bit of work would be to optimize the code to speed up the algorithm further.

References

[1] M. Singh, “Difference between Substitution Cipher Technique and Transposition Cipher Technique – GeeksforGeeks,” *GeeksforGeeks*, October 7, 2021. [Online]. Available: https://www.geeksforgeeks.org/difference-between-substitution-cipher-technique-and-transposition-cipher-technique/. [Accessed 6-May-2022].

[2] *101 Computing*, “Mono-Alphabetic Substitution Cipher.”, Nov 9, 2019. [Online]. Available: https://www.101computing.net/mono-alphabetic-substitution-cipher/. [Accessed: 6-May-2022].

[3] S. Singh, “The Black Chamber - Letter Frequencies,” *Simon Singh*. [Online]. Available: https://www.simonsingh.net/The_Black_Chamber/letterfrequencies.html. [Accessed May-06-2022].

APPENDIX A: IMPORTS AND CONSTANTS

```
import enchant
import jellyfish
import re

DICTO = enchant.Dict("en_US")
FREQUENCIES = {
   "a":0.0820, "b":0.0150, "c":0.0280,
"d":0.0430, "e":0.1270,
   "f":0.0220, "g":0.0200, "h":0.0610,
"i":0.0700, "j":0.0020,
   "k":0.0080, "l":0.0400, "m":0.0240,
"n":0.0670, "o":0.0750,
   "p":0.0190, "q":0.0010, "r":0.0600,
"s":0.0630, "t":0.0910,
   "u":0.0280, "v":0.0100, "w":0.0240,
"x":0.0020, "y":0.0200, "z":0.0010
}
FREQ_SORTED = sorted(FREQUENCIES.items(),
key=lambda kv:(kv[1],kv[0]),
reverse=True)
FREQ_LETTERSONLY = []
for i in FREQ_SORTED:
    FREQ_LETTERSONLY.append(i[0])
CONSTANT_SYMS = {
    ' ':' ', '-':'-', '.':'.', ',':',',
'!':'!', '?':'?', '\'':'\'', '\"':'\"',
':':':', ';':';', '\n':'\n',
    '0':'0', '1':'1', '2':'2', '3':'3',
'4':'4', '5':'5', '6':'6', '7':'7',
'8':'8', '9':'9'
}
ONE_LETTER_WORDS = ['a', 'i']
TWO_LETTER_WORDS = [
    'of', 'to', 'in', 'it', 'is', 'be',
'as', 'at',
    'so', 'we', 'he', 'by', 'or', 'on',
'do', 'if',
    'me', 'my', 'up', 'an', 'go', 'no',
'us', 'am'
]
THREE_LETTER_WORDS = [
    'the', 'and', 'for', 'are', 'but',
'not', 'you', 'all', 'any', 'can',
    'had', 'her', 'was', 'one', 'our',
'out', 'day', 'get', 'has', 'him',
    'his', 'how', 'man', 'new', 'now',
'old', 'see', 'two', 'way', 'who',
    'boy', 'did', 'its', 'let', 'put',
'say', 'she', 'too', 'use'
]
WORDS_WITH_W = ['who', 'why', 'what',
'when', 'with', 'word', 'where']
COMMON_LETTER_PAIRS = ['cc', 'ss', 'ee',
'tt', 'ff', 'll', 'mm', 'oo']
SAFE_LETTERS = []
```

Figure 8

Import Statements and Constants

APPENDIX B: HAS_DOUBLE()

```
def has_double(w:str):
    wlen = len(w)
    for i in range(wlen):
        if (i + 1) == wlen:
            continue
        elif w[i] == w[i+1]:
            return True
    return False
```

Figure 9

Has_double()

APPENDIX C: CHECK FUNCTIONS

```
def check_the(map_the:dict) -> dict:
    decrypted_3lw = []
    for word in WORDS_BY_SIZE[2]:
        ptword = translate(word,map_the)
        decrypted_3lw.append(ptword)
    candidate_for_the = []
    if "the" not in decrypted_3lw:
        for word in decrypted_3lw:
            if
jellyfish.damerau_levenshtein_distance(wo
rd, "the") == 1:

candidate_for_the.append(word)
        for word in candidate_for_the:
            if not word[0] == 't':
                fake_t = get_key(map_the,
't')
                real_t = get_key(map_the,
word[0])
                map_the[real_t] = 't'
                map_the[fake_t] = word[0]
            elif not word[1] == 'h':
                fake_h = get_key(map_the,
'h')
                real_h = get_key(map_the,
word[1])
                map_the[real_h] = 'h'
                map_the[fake_h] = word[1]
            elif not word[2] == 'e':
                fake_e = get_key(map_the,
'e')
                real_e = get_key(map_the,
word[2])
                map_the[real_e] = 'e'
                map_the[fake_e] = word[2]
    SAFE_LETTERS.append('t')
    SAFE_LETTERS.append('h')
    SAFE_LETTERS.append('e')
    return map_the
```

```
def check_twoletters(map_two:dict) ->
dict:
    decrypted_2lw = []
    for word in WORDS_BY_SIZE[1]:
        ptword = translate(word,map_two)
        decrypted_2lw.append(ptword)
    candidate_for_start_t = []
    candidate_for_end_t = []
    candidate_for_start_i = []
    candidate_for_start_o = []
    start_i_endings = []
    start_o_endings = []
    for word in decrypted_2lw:
        if word[-1] == 't':

candidate_for_end_t.append(word)
        elif word[0] == 't':

candidate_for_start_t.append(word)
        elif word[0] == 'i':

start_i_endings.append(word[1])

candidate_for_start_i.append(word)
        elif word[0] == 'o':

start_o_endings.append(word[1])

candidate_for_start_o.append(word)
    for word in candidate_for_start_t:
        if not word == "to":
            fake_o = get_key(map_two,
'o')
            real_o = get_key(map_two,
word[1])
            map_two[real_o] = 'o'
            map_two[fake_o] = word[1]
    for word in candidate_for_end_t:
        if word not in ["at", "it"]:
            if not word[0] == 'a':
                fake_a = get_key(map_two,
'a')
                real_a = get_key(map_two,
word[0])
                map_two[real_a] = 'a'
                map_two[fake_a] = word[0]
            elif not word[0] == 'i':
                fake_i = get_key(map_two,
'i')
                real_i = get_key(map_two,
word[0])
                map_two[real_i] = 'i'
                map_two[fake_i] = word[0]
    for word in candidate_for_start_i:
        if word not in ['is', 'if', 'in']
and word[1] not in start_o_endings:
            fake_s = get_key(map_two,
's')
            real_s = get_key(map_two,
word[1])
            map_two[real_s] = 's'
            map_two[fake_s] = word[1]
    SAFE_LETTERS.append('o')
    SAFE_LETTERS.append('a')
    SAFE_LETTERS.append('i')
    SAFE_LETTERS.append('r')
    return map_two
```

```
def check_plurals(map_p:dict) -> dict:
    alleged_plurals = []
    for word in ctw:
        if len(word) == 1:
            continue
        pluralize = "{0}[a-
z]".format(word)
        plural_results =
re.findall(pluralize,ct)
        for plural in plural_results:
            if plural not in
alleged_plurals:

alleged_plurals.append(plural)
    potential_plural_mapping = []
    for word in alleged_plurals:
        ptword = translate(word, map_p)
        if DICTO.check(ptword):
            continue
        chop_word = ptword[:-1]
        plural_word = chop_word + 's'
        if DICTO.check(plural_word) and
DICTO.check(chop_word):
            plural_map = {word[-1] : 's'}

potential_plural_mapping.append(plural_ma
p)
    closest_difference = 5
    chosen_mapping = {}
    s_freq = FREQUENCIES['s']
    for mapping in
potential_plural_mapping:
        mapping_letter =
get_key(mapping,'s')
        c_freq =
CIPHER_FREQ_PCT[mapping_letter]
        difference = abs(s_freq - c_freq)
        if difference <
closest_difference:
            chosen_mapping = mapping
            closest_difference =
difference
    chosen_mapping_exists = False
    for map in map_p:
        if map == chosen_mapping:
            chosen_mapping_exists = True
    if not chosen_mapping_exists:
        fake_s = get_key(map_p, 's')
        real_s = get_key(chosen_mapping,
's')
        map_p[fake_s] = map_p['k']
        map_p[real_s] = 's'
    SAFE_LETTERS.append('s')
    return map_p
```

```
def check_w(map_w:dict) -> dict:
    for word in WORDS_WITH_W:
        wordlen = len(word)
        for cword in
WORDS_BY_SIZE[wordlen - 1]:
            pword = translate(cword,
map_w)
            w_dld =
jellyfish.damerau_levenshtein_distance(wo
rd, pword)
            starts_with_w = pword[0] ==
'w'
            if not DICTO.check(pword) and
not starts_with_w and w_dld == 1:
                fake_w = get_key(map_w,
'w')
                real_w = get_key(map_w,
pword[0])
                map_w[real_w] = 'w'
                map_w[fake_w] = pword[0]
    SAFE_LETTERS.append('w')
    return map_w
```

```
def check_endings(map_end:dict) -> dict:
    enders = {}
    for word in ctw:
        pword = translate(word,map_end)
        ender = pword[-1]
        if ender not in FREQ_LETTERSONLY:
            continue
        if ender not in SAFE_LETTERS:
            enders[ender] =
enders.get(ender, 0) + 1
```

```
    enders_sorted =
sorted(enders.items(), key=lambda
kv:(kv[1],kv[0]), reverse=True)
    second_most_common_ending =
enders_sorted[1][0]
    if not second_most_common_ending ==
'd':
        fake_d = get_key(map_end, 'd')
        real_d = get_key(map_end,
second_most_common_ending)
        map_end[real_d] = 'd'
        map_end[fake_d] =
second_most_common_ending
    SAFE_LETTERS.append('d')
    return map_end
```

```
def check_threeletters(map_three:dict) ->
dict:
    decrypted_3lw = []
    for word in WORDS_BY_SIZE[2]:
        ptword =
translate(word,map_three)
        decrypted_3lw.append(ptword)
    for word in decrypted_3lw:
        if word not in THREE_LETTER_WORDS
and word[:2] == 'an':
            fake_y = get_key(map_three,
'y')
            real_y = get_key(map_three,
word[2])
            map_three[fake_y] = word[2]
            map_three[real_y] = 'y'
    SAFE_LETTERS.append('y')
    return map_three
```

```
def check_ing(map_ing:dict) -> dict:
    candidate_for_ing = []
    ending_frequencies = {}
    for word in ctw:
        ptword = translate(word, map_ing)
        if ptword[-3:-1] == 'in' and
len(ptword) > 3:
            if not ptword in
candidate_for_ing:

candidate_for_ing.append(ptword)
            ending_frequencies[ptword[-
1]] = ending_frequencies.get(ptword[-1],
0) + 1
    ending_freq_sorted =
sorted(ending_frequencies.items(),
key=lambda kv:(kv[1],kv[0]),
reverse=True)
    most_freq_ending =
ending_freq_sorted[0][0]
    if not most_freq_ending == 'g':
        fake_g = get_key(map_ing, 'g')
        real_g = get_key(map_ing,
most_freq_ending)
        map_ing[real_g] = 'g'
        map_ing[fake_g] =
most_freq_ending
    for word in candidate_for_ing:
        if word[1:4] == 'ein' and not
word[0] == 'b':
            fake_b = get_key(map_ing,
'b')
            real_b = get_key(map_ing,
word[0])
            map_ing[real_b] = 'b'
            map_ing[fake_b] = word[0]
    SAFE_LETTERS.append('g')
    SAFE_LETTERS.append('b')
```

```
    return map_ing
```

```
def check_doubles(map_d:dict) -> dict:
    decrypted_doubles = []
    for word in
CIPHER_WORDS_WITH_DOUBLES:
        ptword = translate(word, map_d)
        if ptword not in
decrypted_doubles:

decrypted_doubles.append(ptword)
    for word in decrypted_doubles:
        if not DICTO.check(word):
            double_index =
find_double(word)
            double_letter =
word[double_index]
            double_pair =
word[double_index] + word[double_index +
1]
            if double_pair not in
COMMON_LETTER_PAIRS:
                for pair in
COMMON_LETTER_PAIRS:
                    common_double_letter
= pair[0]
                    new_word =
word.replace(double_pair, pair)
                    if
DICTO.check(new_word):
                        fake_letter =
get_key(map_d, common_double_letter)
                        real_letter =
get_key(map_d, double_letter)

map_d[real_letter] = common_double_letter

map_d[fake_letter] = double_letter

SAFE_LETTERS.append(common_double_letter)
    return map_d
```

Figure 10

Check functions