NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

CRYPTANALYSIS OF A CLASS OF CIPHERS BASED ON BRUTE FORCE AND PROBABILITY DISTRIBUTIONS

By
Abir Hassan, Elizabeth Hymer,
Claudia Rodriguez, and Louis Yang

March 15, 2020
Computer Science Department
Applied Cryptography
Instructor: Giovanni Di Crescenzo

TABLE OF CONTENTS

I. Introduction

A. Team Member Contributions

Abir Hassan, Elizabeth Hymer, Claudia Rodriguez, and Louis Yang collaborated for the implementation of this project. Each member contributed to the overall project objectives. Abir served as the technical lead whose main role was to program the solutions using Python. Claudia served as the QA analyst tasked with testing the software, ensuring the program ran as intended. Elizabeth's main focus was providing comprehensive documentation, detailed in this report, to complement the technical implementation. Louis conducted research, aided in designing the cryptanalysis approaches, and contributed to the documentation. All team members were responsible for suggesting improvements and revisions, as well as approving the final deliverables.

B. Number of Approaches Submitted

We are presenting two different cryptanalysis approaches to satisfy the requirements for this project. The more simple approach serves as the solution for only the first test, while the other approach is much more complex and attempts to satisfy the requirements for both the first and second tests.

II. Basic Explanation of Approaches

A. Initial Approach

Since we are given all possible plaintexts that can be encrypted for the first test, we are able to utilize this knowledge in order to implement a relatively straightforward algorithm to solve test one. This approach does not work for the second test. We recognize that the encryption scheme is a poly-alphabetic substitution cipher, where the key is a sequence of t numbers between 0 and 26. Our initial approach is built upon the fundamental properties of this encryption scheme, where all ciphertext characters at a distance t from each other are shifted the same number of positions.

First, we mapped each letter in the alphabet, including the "space" character, to a number. The dictionary for test one is read in and parsed into the five possible plaintext strings. A random value for t (between 1 and 24) and the key (a list of random numbers where the length is equal to t) are generated. Ciphertext is produced using one of the five plaintexts from the dictionary as the message, the chosen random value for t, and the chosen random key. The decryption algorithm itself is based on the knowledge that every t-th character in the ciphertext has been shifted by the same number of positions (since the key repeats itself and is shorter than the plaintext), which can reveal the distributions and can be mapped to a specific plaintext. For each possible plaintext in the first dictionary, we brute force the key length and key values and shift the current plaintext by the brute forced values. We compare all t-th values of the ciphertext to the t-th values of the current plaintext. If the distributions are the same, we found our plaintext. If the distributions do not match, we move on to brute force the next possible plaintext.

This method does not suffice for solving the second test because it primarily relies on the fact that we have only a small number of plaintexts, all of which we know the exact values. A much more complex approach is required for solving the second test.

B. Revised Approach

The second test required us to re-evaluate and create a more intricate approach because we do not know the exact plaintext strings. The plaintexts for the second test are created by concatenating words randomly and independently from the second dictionary. We know the plaintexts will be at most 500 characters in length, which will typically be about 52 words per plaintext. Therefore, there are approximately 40^{52} different possible plaintexts that could be

created from this list, in contrast to the merely five possible plaintexts from the first test. Our generation of test ciphertext used the same method as the first test, on the varying input plaintext generated with the concatenation method. The nature of this test prohibits the use of brute force, since the decryption algorithm has a relatively short time constraint. Based upon the letter frequency between ('space', a...z), we researched the probability based upon the statistics of use.[1] We used a combination of the probability for each letter, along with the chi-square test for best fit, to adapt and provide a solution for the second test. It would calculate a level of probability based upon the observed values, and the expected values, for each given ciphertext, to determine the probability of correct plaintext. This would not be a perfect solution, it would only give the best guess given the inputs and the efficiency of the decryption algorithm. We utilized the Index of Coincidence (I.C.) as a factor, as it is a measure of the probability of selecting two letters from a given text and having those letters being the same. For English, the I.C. is 0.0667, based upon the letter frequencies of a generic random piece of text.[2] Figuring out the length of the key is the best way to gain the highest confidence for solving test two. The key length closest to the I.C. is chosen. This approach attempts to guess the key length based upon the distribution. This key length will be used to decrypt the ciphertext into plaintext with mild success. We attribute the inaccuracies to the length of the key used, as well as the variances in the statistical distribution of letters.

Raw statistics

Letter	Mean	Average	Std. deviation
Space	18.28846265%	18.31685753%	0.99928817%
E	10.26665037%	10.21787708%	0.39458158%
T	7.51699827%	7.50999398%	0.41889737%
A	6.53216702%	6.55307059%	0.36853691%
O	6.15957725%	6.20055405%	0.31786759%
N	5.71201113%	5.70308374%	0.30319732%
I	5.66844326%	5.73425524%	0.44720691%
S	5.31700534%	5.32626738%	0.37410794%
R	4.98790855%	4.97199926%	0.41114075%
H	4.97856396%	4.86220925%	0.62025270%
L	3.31754796%	3.35616550%	0.29645987%
D	3.28292310%	3.35227377%	0.38989991%
U	2.27579536%	2.29520040%	0.22213489%
C	2.23367596%	2.26508836%	0.47638953%
M	2.02656783%	2.01727037%	0.21595281%
F	1.98306716%	1.97180888%	0.24773033%
W	1.70389377%	1.68961396%	0.33305120%
G	1.62490441%	1.63586607%	0.18039463%
P	1.50432428%	1.50311560%	0.23884118%
Y	1.42766662%	1.46995463%	0.29196635%
B	1.25888074%	1.27076566%	0.13130773%
V	0.79611644%	0.78804815%	0.09796300%
K	0.56096272%	0.56916712%	0.18896073%
X	0.14092016%	0.14980832%	0.07221414%
J	0.09752181%	0.11440544%	0.05892571%
Q	0.08367550%	0.08809302%	0.03196684%
Z	0.05128469%	0.05979301%	0.03778145%

[1] http://www.macfreek.nl/memory/Letter_Distribution
[2] http://practicalcryptography.com/cryptanalysis/text-characterisation/index-coincidence/

III. Technical Explanation of Approaches

A. Initial Approach

This simple approach is meant to solve test one through a brute force method of key length. Since there are only five possible plaintexts, and therefore five ciphertexts, it brute forces each possibility, and compares the ciphertext input to each, to determine the correct plaintext. This approach is successful for test one for each input.

```
def invert_cipher1_helper(poss_plaintext, c):
    success = 0
    #Gonna brute force the length of the key
    for t in range(1, 24):
        correct = 0
        for k_i in range(0, t): #each letter in the key
            c_str = ""
            #step of t bc that's when the key repeats
            for i in range(k_i, len(c), t):
                c_str += c[i]
            poss_plaintext1 = ""
            for i in range(k_i, len(poss_plaintext), t):
                #get the t_th letters from the possible plaintext to compare distributions
                poss_plaintext1 += poss_plaintext[i]

            #Get distributions of letters
            if sorted(get_distribution(c_str)) == sorted(get_distribution(poss_plaintext1)):
                correct +=1

        success = max(success, correct)
    return success
```

B. Revised Approach

A more complex approach is created to solve both test one and two by guessing the key length, and then applying that key length to assist in decrypting the given ciphertext. The key length is determined based upon the Index of Confidence to determine the most applicable key. The inverse of the key is used for decryption of the ciphertext. This is only mildly successful in determining the correct plaintext for both tests.

```
#Using Index of Coincidence approach
def coincidence(poss_key):
    num = get_distribution(poss_key)
    upper = sum([x * (x - 1) for x in num])
    lower = len(poss_key) * (len(poss_key) - 1)
    return upper / lower

def find_key_length(cipher):
    key_len = 0
    min_key_len = 1
    for i in range(2, 24):
        subkeys = subkey(cipher, i)
        diff = sum([abs(0.0667 - coincidence(poss_key)) for poss_key in subkeys]) / len(subkeys)
        if diff < min_key_len:
            key_len = i
            min_key_len = diff

#This will attempt to guess the values of the key once the key length is known
def break_down(cipher, t, eng_let_freq):
    attempt = subkey(cipher, t)
    key = []
    for i in range(len(attempt)):
        min_i = 0
        shift_min = 10000000
        for j in range(26):
            shift = monoalpha(attempt[i], j)
            curr = chi_square(get_distribution(shift), eng_let_freq)
            if curr < shift_min:
                shift_min = curr
                min_i = j
        key.append(min_i)
    return key
```

IV. Survey on Substitution Ciphers

A. Definition

In general, a substitution cipher in cryptography is a method of encrypting a message by replacing units of the plaintext with another set of units, according to a fixed set of rules, to form the ciphertext. The particular order of the units remains the same, while the units themselves are changed. There are varying types of substitution ciphers, and the size of the units depend on which method is used. The units can be single characters, pairs of characters, triplets, a combination of these, and so forth. Some substitution methods only encrypt the letters in the standard alphabet, while others also encrypt spaces and punctuation marks.

B. Simple vs Polygraphic

A simple substitution cipher only operates on a single letter at a time. Each letter from the standard alphabet is mapped to a different individual letter in the ciphertext alphabet. The ciphertext alphabet can be created by shifting or reversing the standard alphabet, as demonstrated by the Caesar cipher and Atbash cipher, respectively. In contrast, the standard alphabet can also be completely jumbled to create the ciphertext alphabet, rather than simply being shifted by an arbitrary number of positions and retaining the original order. Simple substitution ciphers have very limited secrecy because any adversary knowing the algorithm for shifted or reversed ciphers can decrypt immediately. Also, assuming the ciphertext is of reasonable length, an adversary can use frequency distributions and the pattern of letters to deduce the meaning of symbols and derive words or partial words of the plaintext.

A substitution cipher is polygraphic if it operates on larger groups of letters, rather than single letters. This scheme is an expansion of and offers advantages to the simple substitution cipher. One advantage is that the frequency distributions are much flatter than that of individual letters. A second advantage is that polygraphic substitution ciphers require a larger sample of ciphertext to successfully analyze letter frequencies. An example of a digraphic (pairwise) substitution ciphers are the Playfair, bifid, and four-square ciphers. The trifid cipher is the first practical trigraphic cipher.

C. Homophonic

Homophonic substitution ciphers are those in which a single plaintext character can be replaced by one of several ciphertext characters. The number of possible replacement characters can differ

from letter to letter, usually with the highest-frequency characters having more equivalents. So, the letter ‘E’ might be replaced by one of five ciphertext characters, while the letter ‘Z’ may only have one possible character that replaces it. Homophonic substitution ciphers were created in an attempt to disguise plaintext letter frequencies, thereby increasing the difficulty of frequency analysis attacks. The Beale ciphers are an example of homophonic substitution ciphers.

D. Monoalphabetic vs Polyalphabetic

Monoalphabetic ciphers fall into the simple substitution cipher category. Monoalphabetic ciphers describe the simple substitution ciphers that use a mixed alphabet, rather than having all message characters shifted by the same amount. A random key is generated by creating a list of K[0],...,K[25] distinct random numbers in {0,...,25}. For each message letter, if the letter is equal to the i-th alphabet letter, where i is in {0,...,25}, it is substituted with letter K[i]. The secrecy for the encryption scheme is based on key randomness and large key space. However, the mapping of each letter is fixed and the frequency distributions of letters in the English language are known, which an adversary can use to derive K[i] values and decrypt ciphertext into meaningful plaintext.

The polyalphabetic substitution cipher maps different instances of the same plaintext character to different ciphertext characters. This introduces more randomness than the monoalphabetic cipher using a short random key. For this scheme, a random key is generated by creating a list of K[1],...,K[t] random numbers for some small t. For each group of t consecutive letters in the message, the letter in the i-th position of the group is shifted forward by K[i] positions, for i=1,...,t. The attack on the monoalphabetic cipher is not possible on the polyalphabetic cipher because each instance of the same English character is shifted by a different number of positions. However, all ciphertext characters at a distance t from each other are shifted using the same random number K[j] for some j in {1,...,t}. When t is known, K[j] can be found by studying the frequency of ciphertext characters for each value of j and checking which value for K[j] gives the right probability distribution. Once all K[j] are found, the adversary can easily decrypt the ciphertext into the original message. If t is not known, an adversary can brute force t by trying all t=1,...,max key length.