Project 1
(Cryptanalysis: decryption of substitution ciphers)

Mr. Navin Mittal
Mr. Gaurav Mukherjee

Prof. Giovanni Di Crescenzo

(CS 6903)

MASTERS

IN

COMPUTER SCIENCE

NEW YORK UNIVERSITY SCHOOL OF ENGINEERING

Oct 2014

1. Team Members

Navin N Mittal.

Gaurav D Mukherjee.

2. Project Task performed by each member

Both partners Navin and Gaurav were involved in the following tasks: Outlining and Testing Attack Strategies, Software Engineering and Coding, Analysis of Results, Write-Up and Final Analysis.

For Dictionary1:

a. **Gaurav**: Writing the code for Vignere Cipher and function (**compare_arrays**) where it compares the key generated during decryption, the two array keys compared are **key[]** and **duplicate_key[].**
b. **Navin**: Extended the Vignere cipher to general poly-alphabetic cipher and also made the comparisons fast **key[]** and **duplicate_key[]** fast by writing an algorithm in merge sort which first sorts both the keys and then sends the sorted keys to **compare_array** function. Merges sort is the best sorting algorithm because complexity is nlogn.

For Dictionary2:

a. Gaurav: Discussed different approaches to generate combination and words on dictionary2, also tested nested for loops to generate plaintext of size of length of cipher text. Settled on using recursion to generate combinations and discussed time complexity of generating the combinations.
b. Navin: To reduce the time complexity of combining strings generated plain text of size >= size of cipher text or size of plaintext >= twice the key size, minimum of the two.

3. Detailed explanation of the crypt analysis approach

Cryptanalysis is the study of analyzing information systems in order to study the hidden aspects of the systems. Poly-alphabetic ciphers are known for frequency analysis attacks and plain text attacks. Focus of this project is to deduce a method that would lead in breaking of the ciphers within least possible time. Though it is not been used presently now a days, but it was meant to be traditional method used by many people to crack cryptography challenges.

A. Survey of Polyalphabetic substitution ciphers.

a. Non-Periodic key analysis

   As depicted by the professor the key in the project T should be non-periodic that means that the values can shuffle within the length. This is an intermediate approach to crack the poly-alphabetic ciphers.

b. Analysis of the j(i) function: the relationship between the upper and lower factors define how this function behaves.

c. Frequency analysis on the ciphers

   In most cases for breaking the polyalphabetic ciphers frequency analysis is used. For this project it is not possible to use frequency analysis as we don't have a long text i.e. the cipher text is too small to perform this analysis.

   Another technique which is most similar to frequency analysis is the Kasiski Examination analysis (discovered by Friedrich Kasiski and Charles Babbage) which works on the roots of frequency analysis. It is a method for finding the key length of a poly-alphabetic cipher. We look for repeated sequences in the cipher text and we count the number of letters between those sequences. It gives us a list of all the repeated sequences and the number of letters between them includes the number of letters of the second repeated sequence.

   However, we know that, given the properties of the cipher-text. We must, at least, have two periods of the cipher-text encrypted with the same portion of the key. This becomes obvious since the maximum value for t is 40 (with 2t being 80) and L being at least 100. This allows for variations of known-plaintext-attack.

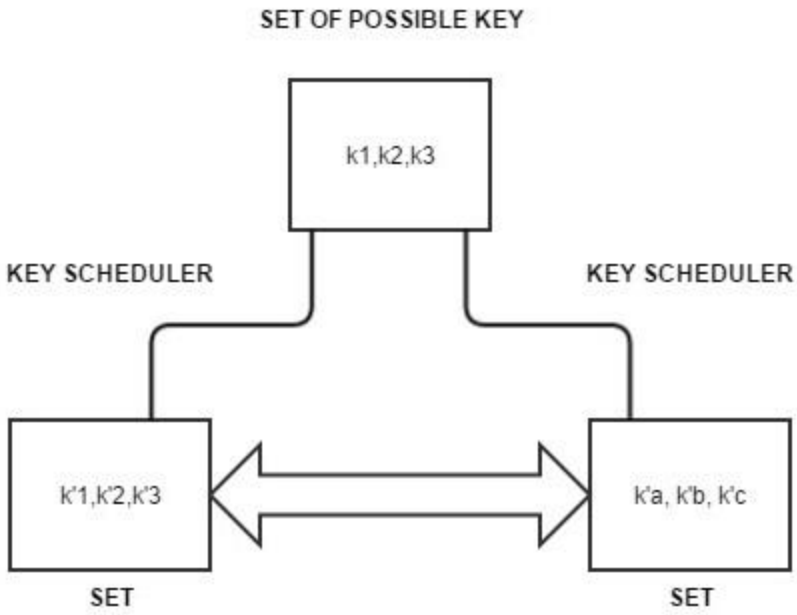

Figure: Different mapping of sets using the j(i) function.

B. Cracking Dictionary1

In order to make the most efficient attack mechanism we need to know the dictionary and the contents in it. In this case we know that the existing plaintexts falls in a range of 50 to 150 different ciphertext - plaintext. This is known as a 'known plaintext attack.

Challenge: to decrypt a substitution cipher

Known Facts: Key Length and List of dictionary

Approach: if the cipher-text length is 50 to150 go for dictionary1 first if not jump to test in dictionary2

a. **Minimum prefix: Dividing cipher text into the key length as chunks of arrays.**

Here we divide the cipher-text into array of chunks of size key length, t. After t size chunk the key repeats itself but in unordered manner.

b. **Our approach of attack on the chunks**.

1. Since Key length is known that concludes that the set of keys will repeat after a period i.e. after the key length in the cipher-text
2. So dividing cipher-text into blocks of size key length.

3. Choosing a plaintext from the Dictionary and dividing it also into blocks of size key length.
4. Comparing 1st block of cipher and 1st block of plaintext, the difference between them will give a set of possible keys.
5. Comparing nth block of cipher and nth block of plaintext, the difference between them with give you keys which should lie in the key set found in the previous step
6. If no match is found then start the plain text chunk from the next character from dictionary.
7. If match is found generate duplicate key for next chunk and compare it with the original key and continued till the last chunk is compared.

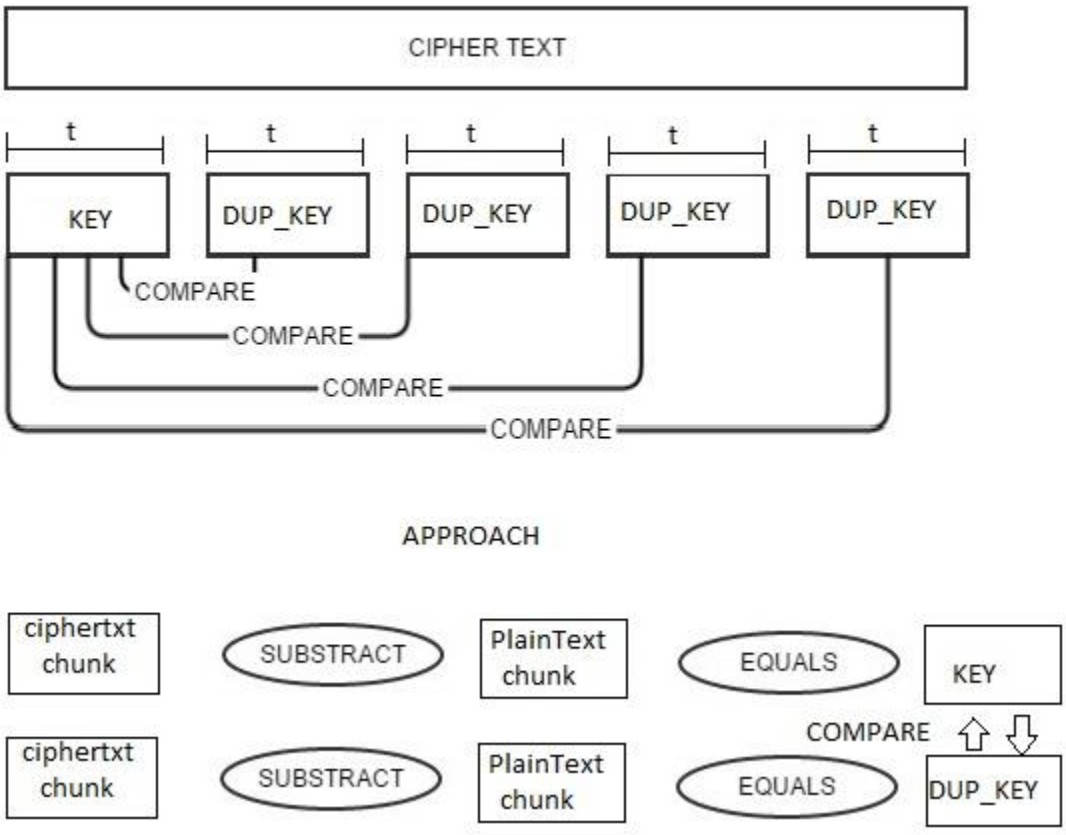

Figure: Approach to crack dictionary1.

C. Cracking Dictionary2

Checking in Dictionary2:

a. Framing a plaintext by finding possible combination of words from dictionary2 of size key length* 2 or cipher-text whichever is smallest.
b. Since Key length is known that concludes that the set of keys will repeat after a period i.e. after the key length in the cipher-text
c. So dividing cipher-text into blocks of size key length.
d. Choosing a plaintext from the Dictionary and dividing it also into blocks of size key length.
e. Comparing 1st block of cipher and 1st block of plaintext, the difference between them will give a set of possible keys.
f. Comparing nth block of cipher and nth block of plaintext, the difference between them with give you keys which should lie in the key set found in the previous step
g. If no match is found then make another combination of words.
h. If match is found generate duplicate key for next chunk and compare it with the original key.
i. If found break the loop of making more combinations and output the plain text.
j. If not found make next combination and start the process again.

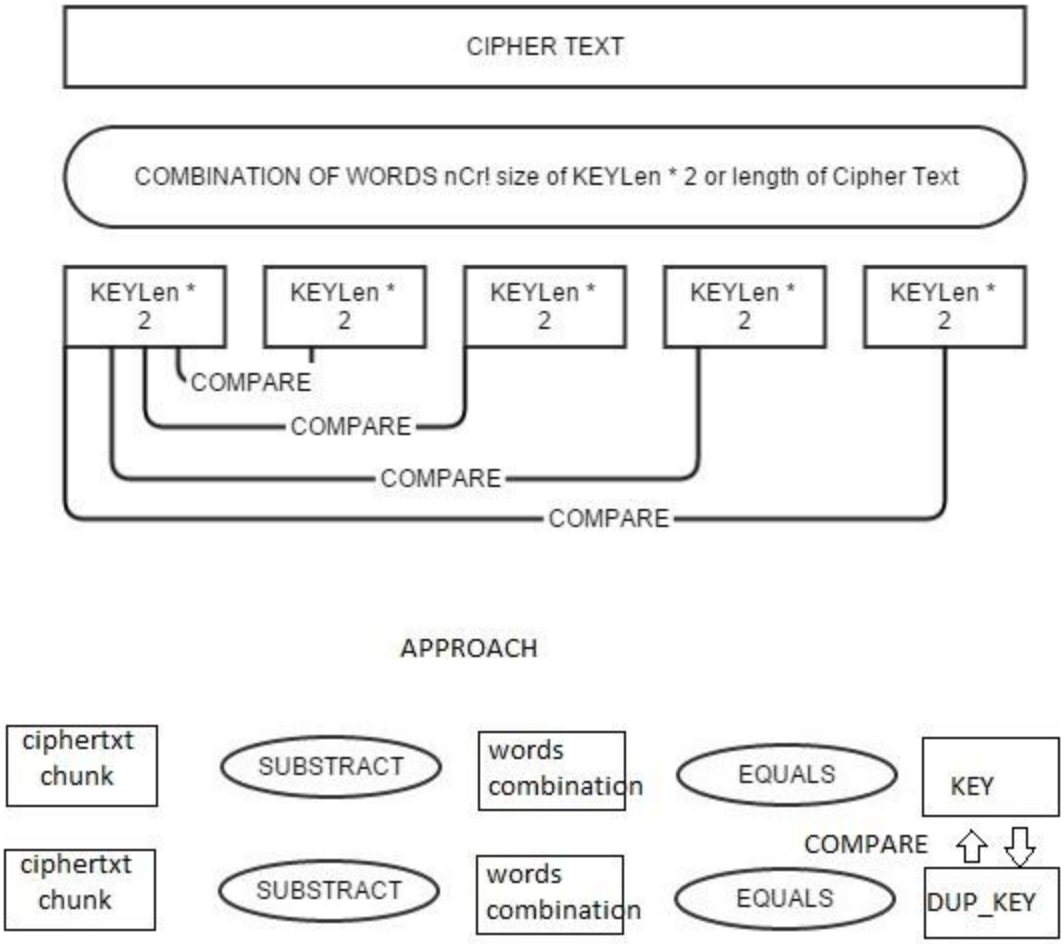

Figure: Attack on Dictionary2

4. Results.

We have utilized benchmarking functions in our code for efficiency and troubleshooting respectively. The program runs using recursive functions.

a. In *"dictionary1.txt"* if the cipher-text is taken from any position of the dictionary it will successfully decrypt and give the plaintext with 100% accuracy within seconds.
b. If not found in "*dictionary1.txt*" it will fetch for *"dictionary2.txt"* and give the result.
c. For *"dictionary2.txt"* we need to find words of different combination i.e. ${}^{n}C_{r}$.
d. Finding all the combination of words in the range of key_length * 2 or ciphertext size whichever is minimum.

e. For every plain text that we create in **"step d",** it is sent to **'compare_arrays'** to verify the correctness of the key same as "*dictionary1.txt*" i.e. from "**step a**", if found the plain text generation in **"step d"** is stopped and the found plain text is given as output.

The process of all the combination takes time as there are more than million combinations for 200 words with all cases due to this the storage in the memory will also be more.

Eg: In case the key length is 40 and cipher text length is 200 according to algorithm created by us it will find all the combinations of the plain text of length 80(i.e. 40*2).

In best case since we have no knowledge of what words that are combined to form the plain text we have to go through each and every combination of 80 words (Assuming the words are not repeating). If the plain text is formed using the words which in the beginning of the "*dictionary2.txt"* then the output will be in seconds. The worst case will be the combination will be the words combined which are in the end of the "*dictionary2.txt*".

The software architecture also allows for break functions. This allows us to include many different breaking approaches as desired. If it would have been higher configured machines working on GPU processor the process would have been faster. Currently this program worked on Core i7 processor and 16 Gigabytes of RAM.

PS: This code works for all parameters other than the parameters which were set by the professor for testing.

5. Various approaches

The above approach is the most feasible approach which has been implemented. Other approaches include:

a. When the key is periodic the dictionary can be used as 2 word or 3 word tuples based dictionary to crack the keys and compare even from the middle of the words.
b. Here we can set groups as the tuples grow in number but the probability of these tuples to appear in the plain text decreases.
c. Every time we decrypt a 3-word tuple there is a probability of 0.2 that it belongs to the subset of the word defined in the dictionary.
d. The comparison then can be done by setting the index range for every alphabet so that the 3-word tuple matches the correct combination.

6. Conclusion

We researched and found different ways for breaking polyalphabetic ciphers. Some of which were implementable but not within the given time period for the project. Others were infeasible due to the dictionary being relatively short. The attacks greatly depended on the fact that the key length is known and the plaintext was given beforehand (this was a clear invitation to perform a known plaintext attack).