Cryptanalysis of a Class of Ciphers based on Exhaustive Key Search and Leaked Information from the Partial Encrypting Scheme

by

Andy Chen, Avinash Narasimhan, Roman Negri

Applied Cryptography CS-GY 6903

Introduction

Our group consists of three members: Andy Chen, Avinash Narasimhan, and Roman Negri, with each of us performing a different task throughout the project. At the beginning of the project, we researched the three ciphers and attempted to create a general cryptanalysis strategy for each. Avinash was responsible for the shift cipher, Roman was responsible for the mono-alphabetic substitution cipher, and Andy was responsible for the poly-alphabetic substitution cipher.

After further discussion and analysis of our approaches for the three ciphers, we decided the algorithm we could guarantee to be as performant as possible would be shift-cipher-based. From there, each group member's tasks became similar, as we all worked on improving the initial shift cipher approach by working through a central GitHub repository where we would push our ideas for optimizing the cryptanalysis code. We also had weekly meetings to touch base and review with one another. Each member of the team worked on writing both the code and the final report.

We are submitting one cryptanalysis approach focused on cracking the shift cipher. This approach is based on careful analysis of the encryption scheme and finding the "weak points" in it. This leads to our decryption techniques such as finding substrings (therefore probabilistically ignoring random characters) and importantly, checking for matching final characters before running any more expensive algorithms as the encryption algorithm leaks information about the plaintext that way.

The project specifications have been followed as per instruction without any additional modifications and our final code and executable have been submitted alongside this report.

Cryptanalysis Approach

This section gives a high-level understanding of the cryptanalysis logic. The approach for cracking the shift cipher uses all leaked information from the encryption algorithm and an exhaustive key search across all possible shift values. The code has been written in C++ for efficient performance. The code takes the ciphertext from the user through stdin which is passed to the decryption algorithm along with the hardcoded, string-vector plaintext dictionary. The algorithm will iterate through all possible key values, i.e. the message space, which is the entire lowercase English alphabet, including the space character. At every iteration, the shift value is taken and used as the key to perform the decryption.

For every iteration of the decryption scheme and every key value, the algorithm first shifts the ciphertext by the amount of the negative key, performing a backward shift equivalent to the key value. After this, the code will attempt to match the decrypted string with each entry in the plaintext dictionary. To do this, we first check if the last characters are the same and then the algorithm will check if the candidate decrypted string is a substring of any plaintext.

In more detail, because of the partial knowledge of the encryption scheme, we can find some leaked information in the ciphertext, and we can know with certainty that the last character will never be a random character. As a result, the algorithm will first check the last character of the current dictionary text and the candidate decrypted text before needing to run a substring test. If the characters do not match, the algorithm will continue without checking for the substring as it knows that the candidate decrypted string will not be from the current plaintext in the dictionary. If the last characters do match and the candidate decrypted string is a substring of the plaintext it will return that plaintext. Otherwise, it will continue to cycle through the remaining keys. The code will output the first plaintext in the dictionary if it reaches the end of the key space without finding a match.

Cryptanalysis Discussion

This section breaks down the various parts of the code which come together to perform our cryptanalysis. First, we need to initialize some data that will assist in the decryption, such as the message space and dictionary of plaintext. We also define two helper functions. One helper function, decrypt_scheme(), will perform the backward shift of the ciphertext for a given key. The other function, is_substring(), will check whether the result of the decryption scheme is a substring of another string, which would be one of the plaintexts in the dictionary in this case. The comparison is performed by maintaining pointers at each character of the plaintext and the decrypted string. When a character match is found, the pointers advance. The plaintext is returned as the code's guess if all the characters of the decrypted string match with the plaintext.

```
Message_space = " abcdefghijklmnopqrstuvwxyz"
Dictionary = a list containing all available plaintext

Function is_substring(main_string, sub_string):
   While not out of bounds for main_string and sub_string:
      If the character at main_string and sub_string are the same:
         Move the sub_string index up
      Move the main_string index up

   Return true if the sub_string index is at the end of sub_string

Function decrypt_scheme(key, message):
   For every character in the message:
      Shift character backwards by key value
      //equal to shifting forwards by the negative of the key
   Return the result
```

In terms of complexity, the is_substring() function runs in O(min(n, m)) time, n being the size of the main_string and m being the size of the sub_string. This is because the function must iterate to the end of the smallest string, as once either string is out of bounds with its index, the function will terminate. Similarly, the function decrypt_scheme() runs O(n), with n being the size of the message.

With these helper functions and data structures defined, we combine them into one main function that handles breaking the cipher, called break_code(). The function first performs the decryption based on a given key and checks two conditions; whether the last character of the decrypted string matches with the plaintext and if there is a substring match. The code returns the plaintext as its guess if both conditions are satisfied.

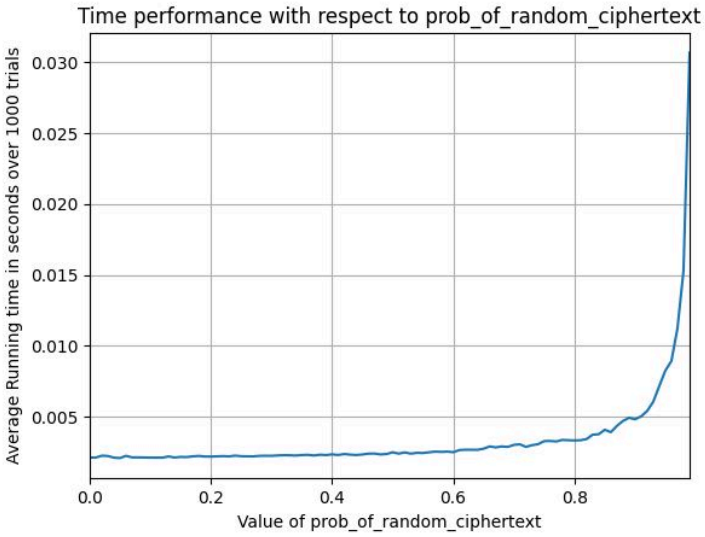
```
Function break_code(cipher_text):
   For every key in Message_space:
      Decrypt_text = decrypt_scheme(key, cipher_text)
      For every plaintext in the Dictionary:
         If last letter of plaintext equals last letter of Decrypt_text:
            Return plaintext if is_substring(plaintext, decrypt_text) true
   Return first plaintext of Dictionary as a guess if no matching plaintext
```

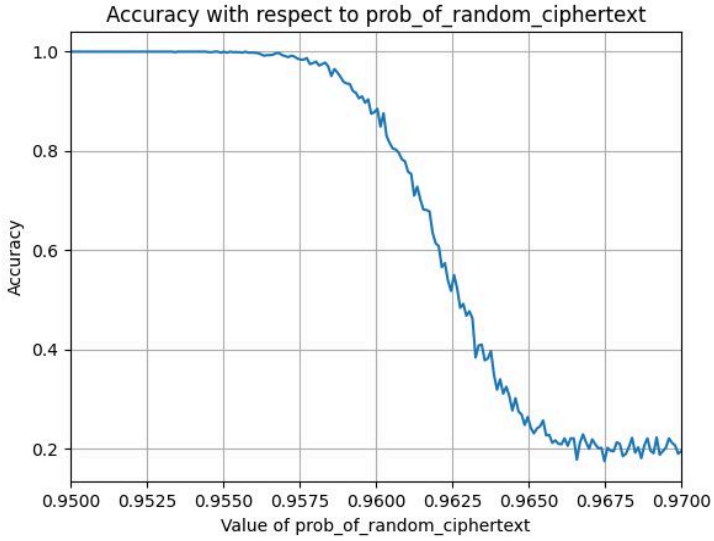
In terms of time complexity for the main function, the function iterates through the length of message_space and for each index in the message_space, it must run the decrypt_scheme() function and go through the dictionary of plaintext. Further, for every plaintext dictionary, it must also run the is_substring() function as long as the last characters match. Since is_substring() is an O(n) operation and we must run it for every iteration through the dictionary, which itself is O(n) as it must iterate through all n plaintext, we end up with an O(n^2) operation. However, because the plaintext in our case is constant in size and because of our optimization of checking the last characters, we can reduce the complexity to just O(n).

Additionally, we must also run the decrypt_scheme function, which takes another O(n). Since O(n+n) is O(2n), it becomes reduced to O(n). Similarly, since the message_space is constant space as well, the entire operation reduces down to O(n) time. However, if the dictionary were to be of variable length, then the main function would be O(n^2) time.

Results

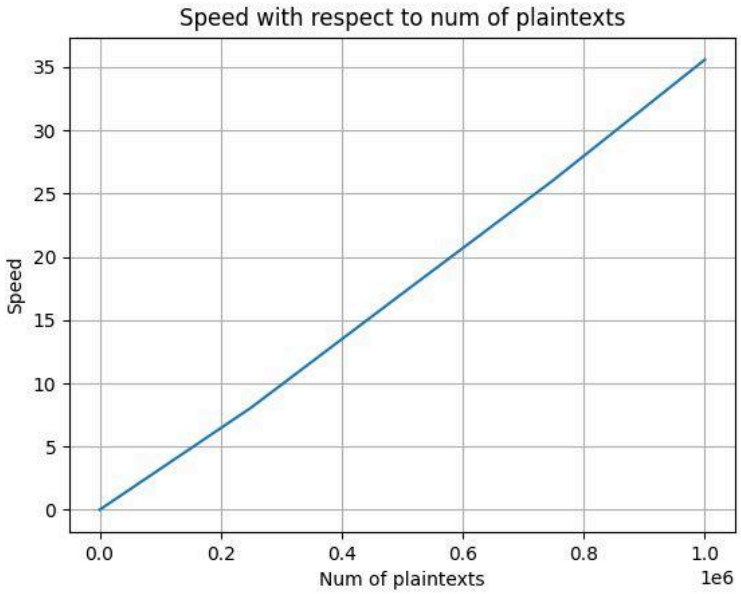
In this section, we demonstrate benchmarks for our algorithm. These times and accuracies were found with a helper Python script which ran each C++ program as a subprocess, thus real times will have less overhead and be higher. The following 2 plots show the accuracy and the time performance of our code. The code is highly efficient in terms of time performance, where it can run within 0.03 ms for prob_of_random_ciphertext almost equal to one. The code also has a high accuracy rate, it is 100% accurate for prob_of_random_ciphertext less than or equal to 0.95 and then goes to 20% gradually for any value above that.

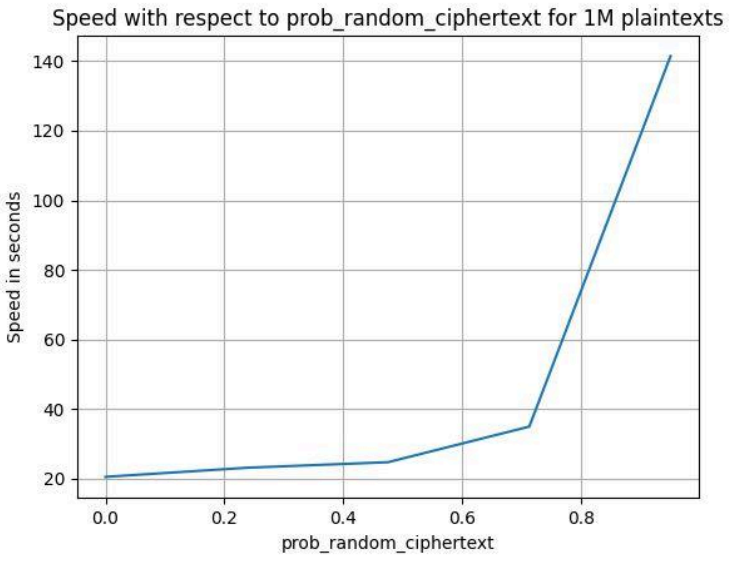




This is the theoretical limit (one over the number of plaintexts, equivalent to randomly picking) as at that point, enough random characters are included that an entire new plaintext will probabilistically get added to the ciphertext. The high efficiency and fast performance stem almost entirely from the added step in our breaking algorithm from the leaked information in the partial knowledge of the encrypting scheme. As we know that the last characters have to match we can check this first, which costs constant time, before running the more expensive linear time substring algorithm.

We also ran an analysis based on increasing the number of candidate plaintexts. To do so, we took a large wordlist from EFF which is a digital rights group. They have published a wordlist of over 15000 words. The list can be found here. Using this, we wrote a Python script which would select these words randomly using the random() module and create candidate plaintexts which can be considered random (skipping any word that included a character not within the message space, i.e. “t-shirt"). We first ran the algorithm to do a timing analysis by gradually increasing the size of the plaintext dictionary. We observed that the speed of the algorithm grows linearly as we increase the size. The prob_of_random_ciphertext used for the following test is 0.7. We can see from the graph below and to the left that the algorithm runs efficiently even for a dictionary consisting of 1 million plaintexts in around 35 seconds. We then ran the algorithm using fixed 1 million candidate plaintexts for varying values of prob_of_random_ciphertext. We observed that the speed of the algorithm grows exponentially from 20s with 1 million plaintexts to 140s with higher values of prob_random_ciphertext. This is shown in the graph below and to the right. We found that our algorithm obtained perfect accuracy (100%) for all values below 0.95 prob_random_ciphertext, explained above, for any size of the dictionary we tested.





Theoretically, since our algorithm is deterministic, there would be a mathematical equation that would be able to compute the probability of an accurate decryption based on how closely the plaintexts resemble each other, the size of the dictionary, and prob_random_ciphertext, but this is beyond the scope of this report as the distribution was fairly distinct for all plaintexts given and we demonstrated experimentally that using the given distribution led to constant accuracy across dictionary size.