Spring 2011

Substitution Cipher with NonPrefix Codes

Rashmi Bangalore Muralidhar
San Jose State University

05-26-2011

Substitution Cipher with Non-Prefix Codes

Rashmi Bangalore Muralidhar
San Jose State University

SUBSTITUTION CIPHER WITH NON-PREFIX CODES

a Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

by

Rashmi Bangalore Muralidhar

Spring 2011

SAN JOSE STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

SUBSTITUTION CIPHER WITH NON-PREFIX CODES

by Rashmi Bangalore Muralidhar

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp Department of Computer Science Date

Dr. Robert Chun Department of Computer Science Date

Dr. Sami Khuri Department of Computer Science Date

APPROVED FOR THE UNIVERSITY

Associate Dean Office of Graduate Studies and Research Date

Abstract

SUBSTITUTION CIPHER WITH NON-PREFIX CODES

by

Rashmi Bangalore Muralidhar

Substitution ciphers normally use prefix free codes - there is no code word which is the prefix of some other code word. Prefix free codes are used for encryption because it makes the decryption process easier at the receiver's end.

In this project, we study the feasibility of substitution ciphers with non-prefix codes. The advantage of using non-prefix codes is that extracting statistical information is more difficult. However, the ciphertext is nontrivial to decrypt.

We present a dynamic programming technique for decryption and verify that the plaintext can be recovered. This shows that substitution ciphers with non-prefix codes are feasible. Finally, we view the cipher from the attacker's perspective and experimentally study various attacks. We show that a limited attack is possible in the case of known plaintext. However, the ciphertext-only attack appears to be very challenging, which is in stark contrast to substitution ciphers with prefix free codes.

Acknowledgements

I would like to express my sincere thanks to my advisor Dr. Mark Stamp for his encouragement, guidance, and support throughout this project. I would like to express my sincere gratitude to my Professors Dr. Robert Chun and Dr. Sami Khuri for their valuable feedback and support.

My special thanks to my husband, family, and friends for their encouragement and support throughout my Master's study.

Table of Contents

List of Figures

List of Tables

SUBSTITUTION CIPHER WITH NON-PREFIX CODES

1.0 Introduction

A communication network is full of hackers, eavesdroppers, and malware, which are threats to information security [11]. To protect data from these threats, it is necessary to send the data in a form which can be correctly decoded only by the receiver. To ensure that the data is sent correctly to the intended recipient, all the three security goals namely confidentiality, integrity, and authentication must be met. We must prevent any unauthorized reading (confidentiality) [12], unauthorized manipulation (integrity), and confirm the identity of the sender and the receiver (authentication) [13]. Cryptography refers to the art of protecting data by meeting all the security goals.

1.1 Cryptography terms

The word “cryptography” is derived from the two Greek words “krypto” and “grafo”, which means hidden writing. The original information to be sent is the plaintext. Figure 1 shows an end-to-end process. The plaintext is transformed into an unreadable format known as the ciphertext, using the special knowledge 'key' known to the sender and the receiver. This process of converting the plaintext to ciphertext is encryption. This ciphertext is sent across the network to the receiver.

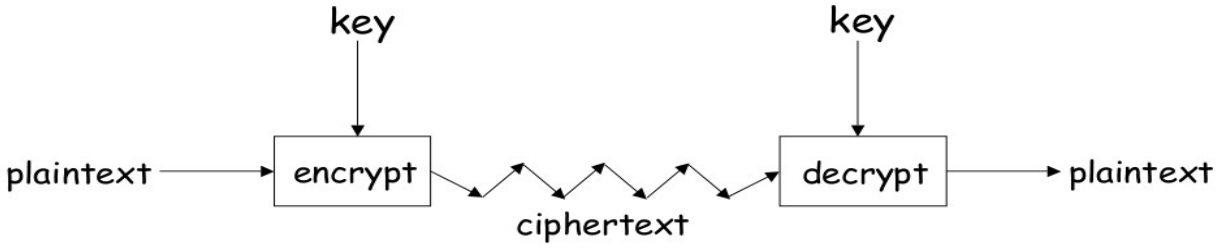

Figure 1: Cryptography process [4]

The receiver performs the reverse process, known as decryption, to recover the plaintext from the ciphertext using the key.

A cipher is used to perform encryption and decryption based on the key. A cipher which uses the same key for encryption and decryption is referred to as a Symmetric cipher. Public key cryptography is also possible, in which two different keys are used for encryption and decryption. The decryption is performed using a private key and a public key is used for encryption.

1.2 Classic Cryptography

Classic cryptography refers to the earliest form of hidden writing. Confidentiality was the major concern, the communication had to be performed in a way that the information was protected from the eavesdroppers. According to the Kerckhoff's principle, even if the ciphertext and the algorithm are public knowledge, the system should be secure [4]. The only unknown is the key, which is known only by the sender and the receiver.

The main types of classic ciphers are substitution ciphers and transposition ciphers. Substitution ciphers are based on the Shannon's property of confusion. The letters in the plaintext are replaced with the other letters to obtain the ciphertext. The main idea is to obscure the relationship between the plaintext and the ciphertext. For example, the plaintext 'hello' becomes the ciphertext 'uryyb' with the ROT13 algorithm. Transposition ciphers are based on the Shannon's property of diffusion. The ciphertext is obtained by rearranging the letters in the plaintext. The ciphertext is a random permutation of the plaintext letters.

In our project, we work on substitution ciphers. In this section, we briefly explain the working of substitution ciphers.

1.2.1 Substitution cipher

The idea behind a substitution cipher is to substitute a unit in the plaintext with the ciphertext, based on the pre-determined key [2]. Here, the “units” can be a single character or a set of characters.

The sender and the receiver agree upon a key mapping [14. Based on the plaintext and the key, the sender generates the ciphertext by replacing each plaintext unit. This ciphertext, which is in an unreadable form is sent to the receiver. The receiver performs the reverse mapping and recovers the plaintext.

Assuming an English character set, the key can be as simple as a shift-by-n operation, where the ciphertext is a character 'n' positions down the alphabet. A well known cipher, Caesar cipher based on the shift-by-3 concept.

Table 1 shows the mapping between the plaintext (Pt) and the ciphertext (Ct) used in Caeser cipher.

Pt	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Ct	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Table 1: Caeser Cipher

Given this key mapping, suppose that the sender wishes to send the message “WE ARE SAFE” to the receiver. He has to generate the ciphertext by substituting each letter in the plaintext with its equivalent ciphertext. Table 2 shows the substitution of plaintext letter (Pt) and the ciphertext (Ct).

Pt	W	E		A	R	E		S	A	F	E
Ct	Z	H		D	U	H		V	D	I	H

Table 2: Caeser Cipher Encryption

The receiver has to perform the inverse mapping on the obtained ciphertext to recover the plaintext.

Table 3 shows the decryption process.

Ct	Z	H		D	U	H		V	D	I	H
Pt	W	E		A	R	E		S	A	F	E

Table 3: Caeser Cipher Decryption

One disadvantage of the shift-by-n technique is that, there are only 26 possible keys in the English text. From the attacker's point of view, he can try them all the possible shifts to know the key. To avoid this situation, substitution ciphers use a key which is a random permutation of letters. Table 4 shows one such possible permutation.

Pt	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Ct	K	F	A	Z	S	R	O	B	C	W	D	I	N	U	E	L	T	H	Q	G	X	V	P	J	M	Y

Table 4: Random substitution

In this case, there are 26! number of keys possible [15] and hence, the attacker has to try 2^{88} keys [5]. The substitution ciphers we have seen until now are the ones operating on single letters. In this project, we work on the substitution cipher which operates on single plaintext letters. Instead of replacing letters with other letters, we replace letters with binary codes. There are other variants of the substitution ciphers namely, the polygraphic, monoalphabetic and polyalphabetic ciphers. A polygraphic cipher operates on a group of characters [17]. A monoalphabetic cipher uses a fixed substitution throughout the message, whereas a polyalphabetic cipher uses a different substitution at different times in the message.

One special type of substitution ciphers is the one time pad cipher. The one-time pad is a type of encryption which is proven to be secure [8]. The idea is to avoid reuse of the keys. The key is generated at random and used only once. The key length is same as that of message. Every message has a different key.

Suppose that there are only eight letters in the character set. Figure 2 shows a possible mapping from character to binary codes.

Letter	W	E	A	R	S	F	I	M
Binary	000	001	010	011	100	101	110	111

Figure 2: One time pad mapping

If the sender wishes to send the message “WEARESAFE”, he has to first convert the message into a string of bits. Figure 3 shows the message in binary.

Pt	W	E	A	R	E	S	A	F	E
Binary	000	001	010	011	001	100	010	101	001

Figure 3: Message in Binary

A key which is of same length of the binary message is generated. Suppose that the key is 00110001010100100000010010011, the encryption is a XOR operation of the plaintext with the key. Figure 4 shows the encryption process and the ciphertext.

Pt	W	E	A	R	E	S	A	F	E
Pt Binary	000	001	010	011	001	100	010	101	001
Key	001	100	010	101	001	000	001	010	011
Ct Binary	001	101	000	110	000	100	011	111	010
Ct	E	F	W	I	W	S	R	M	A

Figure 4: One-time pad Encryption

The ciphertext in binary form is converted back to letters and the resulting output is sent to the receiver. The decryption process is also a XOR operation performed with the ciphertext and the key. Figure 5 shows the decryption process.

Ct	E	F	W	I	W	S	R	M	A
Ct Binary	001	101	000	110	000	100	011	111	010
Key	001	100	010	101	001	000	001	010	011
Pt Binary	000	001	010	011	001	100	010	101	001
Pt	W	E	A	R	E	S	A	F	E

Figure 5: One-time pad Decryption

Although the one-time pad is secure, there is a disadvantage that the key length is same as that of the message, which must be securely transmitted for each message.

In the next section, we will discuss the common attacks on substitution ciphers.

1.3 Cryptanalysis

Cryptanalysis refers to the study of methods to obtain meaningful information from the ciphertext. The attacker knows the algorithm and the ciphertext. In performing cryptanalysis, the attacker can get partial plaintext or if lucky, can get the secret key. Trying all possible keys, known as exhaustive key search is the worst case option for an attacker. In most cases, the key space is large enough that an exhaustive search is not possible. The attacker has to find a short-cut attack based on the weaknesses in the system. If a short-cut attack is not possible, then the cipher is said to be secure. In this section, we discuss some of the common cryptanalysis attacks.

1.3.1 Ciphertext-only attack

In this case, the attacker knows only the ciphertext and the algorithms used. This is the most difficult scenario for an attacker. Based on the ciphertext, techniques such as the frequency analysis can be used to gain information about the plaintext.

If a simple substitution was used on an English plaintext of reasonable length, the attacker can perform a frequency analysis. This is because the statistical information in the plaintext leaks through a simple substitution. Figure 6 shows the letter frequency counts in the English language. From the statistics, the letter 'E' is the most common letter. The attacker can compute frequency counts in the obtained ciphertext. Based on this frequency counts, the most commonly seen letter in the ciphertext is an 'E'. So, the attacker can try to guess a few letters based on frequency until some words are recognized.

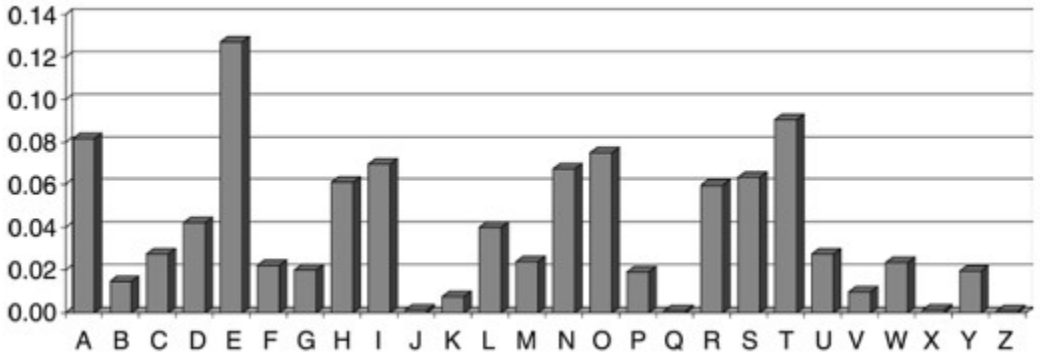

Figure 6: English letter frequency [5]

In some cases, the first word is easy to predict. Further, bigram and trigram statistics can be used to aid the attack.

This attack is very difficult in cases where the key mapping is such that the ciphertext does not provide any useful frequency information. In other words, when there is no one-to-one plaintext letter to ciphertext mapping, this attack is very hard. For example, one letter in the plaintext can be substituted with a set of letters or binary codes. In such cases, the frequency analysis becomes harder.

1.3.2 Known Plaintext Attack

In a known plaintext attack, the attacker also knows some plaintext along with the ciphertext and the algorithms [6]. This increases the probability of success of the attack, compared to the ciphertext only case. In general, the complete plaintext is not known and only a part of the plaintext is known. The attacker probably knows some words or letters and their the corresponding ciphertexts. The goal is to attack the system using these known plaintext-ciphertext mapping.

This is a practical attack, because in the real world situations the attacker is likely to know some plaintext. For example, the encrypted email header. As email uses a stereotypical header, the attacker can guess some plaintext corresponding to come ciphertext [5].

As some of the plaintext is known, the attacker need not perform an exhaustive key search as in the ciphertext-only case. The work factor is lesser than an exhaustive search.

1.3.3 Chosen plaintext attack

This is an attack possible when the attacker has an option to use the plaintext of his choice to be encrypted and observe the corresponding ciphertexts. This provides more information to the attacker and hence, reduces the security of the cryptosystem. It is the best case from the attacker's perspective to figure the key.

This type of attack is feasible in real world. A variant of the chosen-plaintext attack is the lunch time attack. Suppose that an authorized user forgets to log out of the system during a break, the attacker can gain access to a system and conduct this attack.

An adaptively chosen-plaintext attack is also possible, in which the attacker selects the current plaintext based on the previous ciphertext. This would make the attacking the cipher much more simpler.

2.0 Algorithm Design

An algorithm is a step-by-step approach to solve a given problem. A given problem can be solved in various ways. Algorithm design are techniques which provide an effective method to solve a problem. It can be broadly classified into four techniques, namely the greedy, divide-and-conquer, dynamic programming, and branch-and-bound. In this section, we discuss the techniques used in this project – a) Divide-and-conquer and b) Dynamic programming.

2.1 Divide-and-conquer

The idea in divide-and-conquer is to divide the given problem into smaller problems and solving them. The first step is to identify and divide the problem into smaller subproblems of the same type. Each of these subproblems are solved independently in a recursive manner. If the subproblems are small enough, they are solved in a straightforward manner. The solutions of all the subproblems are combined, which is the solution to the overall problem [7].

Divide-and-conquer technique is used in several use cases. One such example is merge sort. Given an unordered list with 'n' elements, the goal is sort them in an efficient way.

If the list has only one element, then it is already sorted. Otherwise, the list is divided into two sublists of about equal size. Each of these sublists are sorted by applying the merge sort recursively. The two sublists are merged to form one sorted list.

Figure 7 shows the merge sort psuedo code. Figure 8 presents a merge sort example of an unordered seven element list. The first four levels represent the division process, where each of the lists are divided into sub lists and recursively sorted. This division process is continued until the sub lists are composed of single elements. At this stage, no more further division is possible and the combining

process begins. The results of two sub lists are combined and sorted at each step. The merging process is continued until all the sub lists are merged and results in only one list. This final list is an ordered list, which is the answer to the original problem.

```
MergeSort(List[], leftIndex, rightIndex)
Begin:
    if leftIndex < rightIndex then
        mid = (leftIndex + rightIndex) / 2
        MergeSort(List[], leftIndex, mid)   // split left sublist
        MergeSort(List[], mid + 1, rightIndex)   // split right sublist
        Merge(List[], leftIndex, mid, rightIndex)    // merge sorted sublists
    endif
End
```

Figure 7: Merge sort Psuedo code [3]

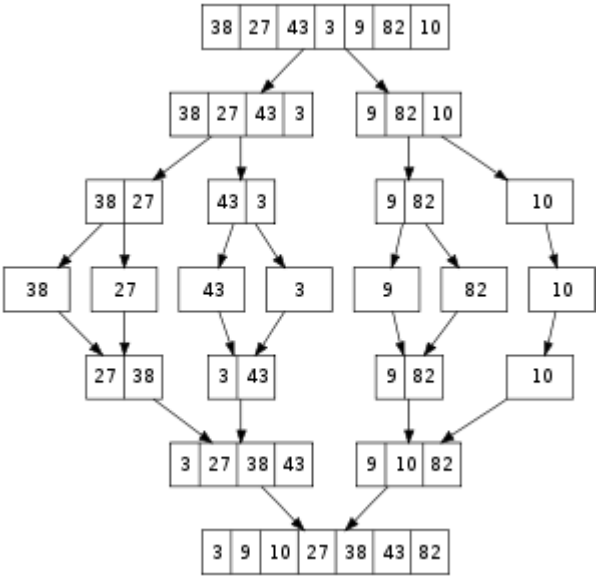

Figure 8: Merge sort example [1]

Considering an unordered list with 'n' elements, the worst case performance of the merge sort is O(n logn), which is efficient compared to brute force technique which takes $O(n^2)$.

2.2 Dynamic Programming

Dynamic Programming is a technique used for solving a complex problem by recursively breaking down it into several sub problems. These sub problems are solved and the solutions are combined to obtain the solution to the original problem. Each of the sub problems is solved only once and the results are reused [9].

Dynamic Programming is applicable to problems that exhibit an optimal substructure and are composed of several overlapping sub problems. A given problem is said to have overlapping sub problems if it can be broken down into sub problems which can be reused many times. If the final optimal solution to the problem can be obtained only by combining the optimal solutions of the sub problems, the problem is said to exhibit an optimal substructure.

A top-down or bottom-up approach can be followed to solve a problem. In a top-down approach, also known as memorization, a problem is recursively formulated into sub problems and their solutions are stored in a table. If the same sub problem has to be solved again, a table look up is made and the result is reused. In a bottom-up case, a problem is broken down into sub problems, whose solutions are combined to arrive at a solution to the bigger problem.

Let us consider a simple problem and solve it using Dynamic Programming. Suppose that x and y are two strings and we want to find the longest common subsequence (LCS) in them. For example the LCS of ‘ggcaccacg’ and ‘acggcggatacg’ is ‘ggcaacg’.

Let the length of the string x be M and the length of the string y be N. We start with the LCS length of 0. We compare the last characters in both the strings, g in this case. If they are the same, we have reduced our problem of x[0...M] and y[0...N] to x[0...M-1] and y[0,N-1] respectively. We increment the LCS length. We proceed by dividing the problem into subproblems at each step. If a mismatch is found,

we must remove a character from x or from y, depending on the resulting score. In either case, we are reducing the problem space at each step. After both the strings are compared, the best LCS length gives us the best possible common subsequence length of these strings. The problem can be recursively formulated as in Figure 9.

$$C(i,j) = \begin{cases} 0 & \text{if } i = M \text{ or } j = N \\ C(i+1, j+1) + 1 & \text{if } x_i = y_j \\ \max\{ C(i, j+1), C(i+1, j) \} & \text{otherwise} \end{cases}$$

Figure 9: Recursive LCS [16]

We maintain a 2D-matrix of size M*N, to store the results of all the sub-problems, which finally lead to an optimal solution. For the given strings x and y, the result matrix C[i][j] is as in figure 10.

		0	1	2	3	4	5	6	7	8	9	10	11	12
x\y		a	c	g	g	c	g	g	a	t	a	c	g	
0	g	**7**	**7**	**7**	6	6	6	5	4	3	3	2	1	0
1	g	6	6	6	**6**	5	5	5	4	3	3	2	1	0
2	c	6	5	5	5	**5**	4	4	4	3	3	2	1	0
3	a	6	5	4	4	4	**4**	**4**	**4**	3	3	2	1	0
4	c	5	5	4	4	4	3	3	3	**3**	**3**	2	1	0
5	c	4	4	4	4	4	3	3	3	3	**3**	2	1	0
6	a	3	3	3	3	3	3	3	3	3	**3**	2	1	0
7	c	2	2	2	2	2	2	2	2	2	2	**2**	1	0
8	g	1	1	1	1	1	1	1	1	1	1	1	**1**	0
9		0	0	0	0	0	0	0	0	0	0	0	0	**0**

Figure 10: LCS Result matrix [16]

After solving all the sub-problems, the value in C[0][0] represents the length of the best LCS. In this case, the longest common subsequence is seven characters long. To get the actual subsequence, we have to retrace the steps backwards, to get the path taken to obtain the best solution. For this problem, the best path is shown in red. If we have made a choice to move along the diagonal, then it is a matching character considered in the common subsequence. The longest common subsequence is composed of all the characters along the diagonal choices.

By taking this dynamic programming approach, we are computing each of the subproblems once and reusing the result. Hence, it reduces the space and time complexity taken to solve this problem.

3.0 Related Work

A closely related problem to our project is the Huffman based encryption [18,19]. The Huffman codes are of variable length binary codes which are prefix-free in nature. That is, no code is a prefix of any other valid code [10]. For example, the set {1010, 1000, 11000} are prefix-free binary codes of variable length.

The codes for characters are chosen based on the frequency of occurrence of characters. The letter with higher frequencies are assigned codes of fewer bits and the letters with lower frequencies are given codes of more bits. By doing so, the compression is achieved.

Suppose that the plaintext contains 25 characters “s2 s3 s2 s1 s2 s1 s4 s3 s1 s2 s1 s3 s1 s4 s1 s4 s1 s2 s1 s2 s1 s1 s1 s2 s1”. Table 5 shows the frequency of plaintext characters.

Symbol	Frequency
s_1	12
s_2	7
s_3	3
s_4	3

Table 5: Frequency count for Huffman encryption [18]

A prefix-free Huffman tree is constructed based on these frequencies. The tree is as shown in Figure 11. The letters with the least frequencies are considered at the lowest level and the tree is constructed bottom-up by combining letters in the increasing order of their frequencies.

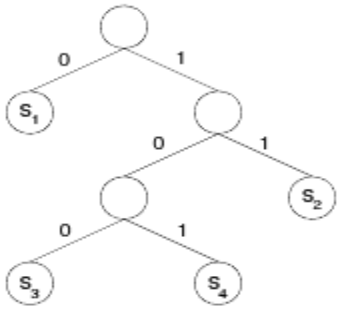

Figure 11: Prefix-free Huffman tree [18]

Each character is represented as nodes in the tree. The binary code for a character is the string of codes encountered while navigating to that node in the tree starting from the root. For example, the code for the character 's3' is '100'. Table 6 shows the mapping for this example plaintext.

Symbol	Frequency	Codeword
s_1	12	0
s_2	7	11
s_3	3	100
s_4	3	101

Table 6: Huffman code mapping [18]

The codes are of variable lengths and prefix-free. Also, all the characters are at the leaf level. Depending on the depth of the tree constructed, the length of the code words vary. Also, the code length for letters with higher frequencies is lesser than the letters with lesser frequencies.

Based on this mapping, our plaintext of 25 characters will result in 44 characters of ciphertext – *11100110110101100011010001010101011011000110*. This ciphertext is sent to the receiver, who has to decrypt based on the mapping.

The receiver constructs a similar tree based on the known character to binary code mapping. The receiver parses the ciphertext and walks down the tree until he reaches a leaf. The character in the leaf node represents the plaintext letter. This process is performed until all of the ciphertext are decoded. Due to the prefix-free property of the binary codes, there are no ambiguous decryption outcomes.

4.0 Project Overview

We consider a substitution cipher on English text. Each plaintext letter is substituted with a random binary code of variable length. Normally used binary codes are prefix-free, meaning, the code for one letter does not share a prefix with the code for any other letter. The binary codes that we consider in this project do not have this property. For example, {10,11} are prefix free, where as, {10,101} are non-prefix code words, because “10” is a prefix of “101” and they can be code words of two different plaintext letters. The advantage of a prefix-free code is that, when a prefix-free code is used, the receiver can uniquely identify each word. In case of non-prefix codes, the decryption is ambiguous due to the various possible outcomes.

In this project, we consider two cases of non-prefix cipher, with a change in the key space. In the first case, we consider the character set consists of 26 english alphabets only, which are mapped to a variable length non-prefix code words. The plaintext is a contiguous sequence of english letters without any delimiters. In the second case, we add two non-dictionary characters – space and period to our character set. The key consists of 28 characters and their corresponding code words. Even the non-dictionary characters have a random binary mapping.

Encryption is performed by the sender by substituting each character in the plaintext with its binary code, to obtain the ciphertext. This ciphertext will be a long sequence of 0's and 1's, without any word/sentence delimiters.

Decryption is performed by the receiver, who has to decrypt this binary pattern, using the key, to obtain the plaintext. The receiver will have to decide upon the sentence boundaries, and the word boundaries. He can then start decrypting word by word, by using the key. It is possible that a sequence of ciphertext yield multiple possible plaintext equivalents. The receiver has to use a smart technique to

select the best plaintext. He should then combine these decrypted words to form a sentence and construct a paragraph with these sentences.

In this project, we prove that the encryption and decryption can be performed correctly on a ciphertext of reasonable length using this cipher. Taking a step forward, we see the cipher from the attacker's point of view. Not only is the decryption more challenging compared to the prefix-free codes, attacks are also harder since frequency analysis is harder when only the ciphertext is given. We check the feasibility of conducting a ciphertext-only attack. We try to perform a limited known plaintext attack on this system. We prove that the attack succeeds and the attacker is able to reveal the secret key based on some known plaintext.

5.0 Data Collection and Probabilistic Model Construction

Encryption is performed by the sender and the ciphertext is sent to the receiver. As the key is non-prefix-free and is of variable length, a given ciphertext may yield many possible plaintext equivalents on decryption. A mistake in choosing the correct word out of the various possibilities in any of the intermediate steps can lead to an incorrect decryption outcome at the end. It is very important to use a technique to identify appropriate words to proceed with decryption.

The first step is to eliminate invalid possibilities at the word level. On eliminating the invalid words, fewer number of valid possibilities will remain for the overall text. To aid the decryption process in identifying the best possibilities at each step, we build a dictionary to eliminate invalid words and several probabilistic models to identify the best word in the context.

To eliminate invalid words, we maintain a dictionary of words. We build a dictionary by parsing a collection of a large corpus of English books. We wrote a perl program to parse individual words in the text file, clean them to eliminate special characters, and eliminate duplicates. The dictionary so built consists of all unique valid English words found in the corpus. As a large corpus of books are used in building the dictionary, it consists of most of the English words found in any standard dictionary. Since we are using a big corpus, there is also a possibility of having picked up spelling mistakes or other erroneous words from the corpus. We prune these by looking at the number of occurrences of the words. These words are sorted in ascending order for faster search operations. The size and quality of the dictionary can be controlled by the size of the corpus and the thresholds.

To address the second problem of selecting the best word from all the valid choices, we build a probabilistic model. This model is responsible for determining the probability that a word appears given the previous word which was decrypted. This model was trained with the same corpus of English

books which was used to build the dictionary. The output of this model is a set of all possible bigrams found in this input collection, along with their probabilities of occurrence. Given a word, the model gives the probabilities of co-occurrence for the next word. This bigram collection consists of about a million entries.

It is also possible that the ciphertext corresponding to the first word or the last word yield many possibilities. To ease our selection in these cases, we build probabilistic models for the first word and the last words of sentences. We use the same corpus of books and extract all the words which begin and end a sentence. We compute the frequencies that a word starts/ends a sentence. This first and last word collections consists of about a 15,000 and 30,000 entries respectively.

At any point, if we are to select one word out of a set of possibilities, we can use the built dictionary and the probabilistic models. In case of the probabilistic approach, given multiple word possibilities, the word with a higher probability is the winner. For example, if the previous word was decrypted as 'it' and the next word possibilities are 'was', 'our', and 'of'. We need to select only one of them to proceed with. We query our model with all bigram possibilities 'it was', 'it our', and 'it of'. Based on the score returned for each of them, the word 'was' is most likely to occur following 'it', compared to 'our' and 'of'. In such a case, we select 'was' as the best possible plaintext word and proceed.

6.0 Cipher Implementation

In this project, the sender constructs the messages in English. The sender and the receiver agree upon a key using some other secure key exchange mechanism. The sender encrypts the message using the key and sends the ciphertext to the receiver. The receiver, with the knowledge of the key decrypts the ciphertext to obtain the plaintext.

We have considered two variants of the key space to check the feasibility of encryption and decryption. In the first case, the key space consists of only the 26 English letters and their non-prefix binary code mapping. In the second case, we add non-dictionary characters, a space and a period to our character set. In this case, the key space consists of 28 characters and their binary code mapping. In this section, we present the implementation details of both the variants – 1) Key space with the dictionary characters only and 2) Key space with dictionary and non-dictionary characters.

6.1 Key space with dictionary characters only

In this case, a plaintext message is composed of English alphabets, without any delimiters. A key is generated, which consists of the letters and their non-prefix variable length binary codes. This key is known to the sender and the receiver The sender encrypts the plaintext message to be sent by using the key to obtain the ciphertext. This ciphertext is a binary sequence without any word boundaries. The receiver has to derive the plaintext from the ciphertext with the help of the key. The entire process can be divided into three phases, namely the 1) Key generation, 2) Encryption, and 3) Decryption. The next section provides details of each of the three phases.

6.1.1 Key generation

The sender and the receiver must agree upon a key prior to exchanging messages. In this case, the key must consist of binary codes for each letter. These binary codes are substituted for each of the plaintext letters to get the ciphertext.

As we know, the messages are in English. The messages have to be composed from the 26 letters in the English alphabet set. Hence, the key consists of these 26 letters and their binary mapping. We need 26 codes to assign to our character set. These codes can vary in length. For example, if we choose a maximum code length of five bits, there are 2^5 or 32 possibilities for each character. We have to choose a unique code for each letter in the alphabet. Similarly, for a maximum code length of 8 bits, there are 256 possible codes for each letter. Out of these possible codes, 26 unique codes have to be chosen to map to each of the letters.

In our use case, we consider a maximum of five bit code to start with. We have 32 choices and 26 have to be selected. We randomly pick a number between zero and 31, convert it into binary and assign to a plaintext letter. The binary codes so generated need not be prefix-free. For example, 10, 101, 1010, 10100 can be chosen to represent different letters. The codes vary from a single bit to five bits in length. This mapping between the characters to the assigned binary codes is the key. The same key is used for performing encryption at the sender's end and for performing decryption at the receiver's end. We assume that the sender and the receiver have the exchanged the generated key using a secure key sharing protocol such as the Diffie Hellmann key exchange protocol.

We can choose a set of 26 codes from a set of 32 codes in ${}^{32}C_{26}$ different ways. Once a set of binary codes are generated, it is assigned to an alphabet. There are 26! keys possible based on a 26 binary code set. Overall, there are ${}^{32}C_{26}$*26! keys, which is $3.65459496 \times 10^{32}$, which is around 100 bits.

We use the binary codes between one and five bits in length for illustrations throughout this report.

Table 7 shows one possible key we have generated.

a	1001
b	10100
c	1100
d	101
e	11001
f	1101
g	10
h	10000
i	11011
j	11
k	11111
l	10101
m	1010
n	10111
o	100
p	1011
q	1110
r	0
s	11000
t	1111
u	10010
v	10001
w	1000
x	1
y	111
z	10110

Table 7: Key for Dictionary Encryption case

6.1.2 Encryption

The sender has a plaintext message to be sent to the receiver in an encrypted manner. This plaintext message is formed from the 26 characters in the English language. The sender has to encrypt this plaintext message by using the key which he shares with the receiver. This is a substitution cipher, the sender generates the ciphertext by substituting each plaintext character with its corresponding binary code in the key mapping.

For example, if the plaintext word 'cryptography' has to be encrypted. We substitute individual letters in the word with its corresponding binary code in the key mapping to frame the ciphertext. The ciphertext for this word will be *11000110111111100100100110111000111*. Similarly, a sentence/paragraph can be encrypted by substituting individual characters with the binary codes. Figure 12 shows five plaintext sentences and their ciphertexts on separate lines.

```
hehadnoideaatall
hepickeduphisbooks
healmostjumpedoutofhisskin
thelaughterstoppedsuddenly
hewishedhewasbackatthehouse
```

```
10000110011000010011011011110011011101110011001100111111001101011010 1
10000110011011110111100111111100110110010101110000110111100010100100100111111 1000
10000110011001101011010100110001111111001010101011110011011001001011111001101100001101111000110001111111011101 11
11111000011001101011001100101010000111111001011000111110010111011110011011100010010101101110011011110101111
10000110011000110111100010000110011011000011001100010011100010100100111001111110011111111110000110011000010010010110001 1001
```

Figure 12: Dictionary case Encryption

The output of the encryption process is a sequence of 0's and 1's, without any delimiters. The length of the ciphertext depends on the number of characters in the plaintext and their binary code lengths. Table shows the number of plaintext characters in the message and the resulting ciphertext length. This is based on a random sample from the encrypted data file.

Plaintext length	Ciphertext length
22	90
38	157
35	144
23	88
17	70
20	82
27	113
27	108
18	124
44	178

Table 8: Plaintext and Ciphertext lengths

For a code length of five, the average number of bits per character in the ciphertext is around four.

6.1.3 Decryption

The receiver has the knowledge of the ciphertext and the shared key. The goal has to perform the decryption and extract the plaintext using the key. The receiver knows that a substitution cipher with non-prefix codes was used to encrypt the plaintext message without any delimiters. The receiver also knows that the character set consists of the English alphabets and the key.

We follow a dynamic programming approach to solve this problem. Given an encrypted plaintext message, we identify and divide the problem into several subproblems of decrypting a word. We solve each of the word decryption sub-problems to get the best possible word. We then combine the results of all the subproblems to get the solution to our original problem. In doing so, we get the plaintext of the entire ciphertext.

Using the key, we try to decrypt the ciphertext from the beginning. Based on the words in our dictionary, all the words are within a certain length, say k characters. In our case, from our dictionary, we expect that all words are less than 25 characters in length. This number is dependent on the dictionary. Based on the key, we know that a code length ranges from one to five bits. Hence, the

maximum length of the ciphertext for a word is 125 bits. We maintain a moving window of at most 125 bits starting from the end of the last decrypted word.

Let E be the encrypted ciphertext. Each of the sub-problems can be represented as a tuple {e, d} where e is a suffix of the input E and d is the decrypted plaintext for the prefix of E excluding e. The initial problem state is {E, ' '} and the solution set contains tuples of the form {' ', D}, where D is a sequence of valid words.

Given a {e, d}, eliminating all prefixes of e occurring within the moving window boundary that can be decodes as words in the dictionary lead to a valid sub-problem. There can be many valid sub-problems for a given {e, d}. We maintain a list of all the valid sub-problems. We remove sub-problems from the head of the list and add back smaller sub-problems to the tail of the list. We keep repeating this process till we reach {' ', D}.

For a particular ciphertext, there can be many possible plaintext words. For example, the ciphertext *1000100111000* can yield plaintexts 'was', 'xrrroyrrr', 'wrxrrjw', and so on. At this point, we make use of our dictionary to eliminate invalid words from these decryption possibilities. As and when we identify the plaintext words, we are left with a smaller problem to solve. We proceed further only after identifying valid words. This is because of the optimal substructure in our sub-problems. The sub-problems which do not have an optimal score do not yield an optimal solution to our overall problem.

It is also possible that the entire ciphertext yield multiple plaintext equivalents. These final possible plaintexts are composed of valid words only. Hence, we need a different mechanism to select the best one. To aid this decision, we use a scoring mechanism based on our probabilistic model while decrypting the words. We start with a score of zero. As and when a word is decrypted, we query our model to get the bigram frequencies depending on the previous word. We add this frequency score to

our decryption score.

```
Result of decryption........
itorasreallymostextraordinaryyoushutthedoorheorisorredheorasbackatthehousehelookedupintothetreeandsawabeautifulparrotyougavemeanawfulfright
Score: 256700672
itorasreallymostextraordinaryyoushutthedoorheorisorredhewasbackatthehousehelookedupintothetreeandsawabeautifulparrotyougavemeanawfulfright
Score: 256711774
itorasreallymostextraordinaryyoushutthedoorhewishedheorasbackatthehousehelookedupintothetreeandsawabeautifulparrotyougavemeanawfulfright
Score: 256700764
itorasreallymostextraordinaryyoushutthedoorhewishedhewasbackatthehousehelookedupintothetreeandsawabeautifulparrotyougavemeanawfulfright
Score: 256711866
itwasreallymostextraordinaryyoushutthedoorheorisorredheorasbackatthehousehelookedupintothetreeandsawabeautifulparrotyougavemeanawfulfright
Score: 256713216
itwasreallymostextraordinaryyoushutthedoorheorisorredhewasbackatthehousehelookedupintothetreeandsawabeautifulparrotyougavemeanawfulfright
Score: 256724318
itwasreallymostextraordinaryyoushutthedoorhewishedheorasbackatthehousehelookedupintothetreeandsawabeautifulparrotyougavemeanawfulfright
Score: 256713308
itwasreallymostextraordinaryyoushutthedoorhewishedhewasbackatthehousehelookedupintothetreeandsawabeautifulparrotyougavemeanawfulfright
Score: 256724410

Best possible plaintext is:
itwasreallymostextraordinaryyoushutthedoorhewishedhewasbackatthehousehelookedupintothetreeandsawabeautifulparrotyougavemeanawfulfright 256724410
```

Figure 13: Dictionary case Decryption

Once the decryption of the ciphertext is complete, out of all the possible outcomes, the plaintext with the highest score is declared as the original plaintext. For example, in Figure 13, a ciphertext of 544 bits is decrypted into 8 possible plaintext equivalents of five sentences each. Based on this scoring mechanism, we are able to uniquely identify the plaintext.

Table 9 is illustrative of the number of possible plaintext results given the number of words in the input.

Plaintext length (words)	# of possible results
18	4
8	2
33	8
24	4
32	4

Table 9: Ambiguous plaintext in Dictionary case

We have verified our algorithm by encrypting a hundreds of plaintext messages to obtain the ciphertext and performing decryption on these ciphertexts. In all the cases, we are able to uniquely recover the plaintext. Based on the encryption and decryption, we think it is feasible to use this cipher

to perform encryption of a message and decrypted to obtain the message.

Now that we know the encryption and decryption work based on dictionary characters, we want to add non-dictionary characters to our alphabet. In our experiments, we add a space and a period to our character set and check the feasibility of such a cipher. In the next section, we discuss the details of this cipher.

6.2 Key space with dictionary and non-dictionary characters

In this case, there are non-dictionary characters in the plaintext which are also encrypted as part of the ciphertext. The ciphertext is still a sequence of binary bits without any delimiters. The key consists of all the dictionary and non-dictionary characters. For illustration, we use a space and a period as the non-dictionary characters. The receiver is aware about this fact and also has the key. As the key is non-prefix free and of variable length, the exact positions of the spaces or periods are not known. The receiver has to use techniques to first identify and eliminate these characters. The receiver can then try to perform the decryption.

We also make use of the dictionary and the probabilistic models built earlier. In this scenario, we implement this cipher in three phases 1) Key generation, 2) Encryption, and 3) Decryption. In this section, we explain each the phases.

6.2.1 Key generation

This is a substitution cipher, we need a mapping between each character in the plaintext space to the corresponding code in the ciphertext space. The sender needs this key to perform encryption and the receiver needs it to perform decryption. The key must be generated and shared in advance.

There are 26 letters and two non-dictionary characters in the plaintext space. The substitution cipher is based on non-prefix codes. Hence, we need 28 non-prefix binary codes which compose the key. The key can be chosen in several ways depending on the maximum bit length of a code. For a code length of up to five bits, there are 32 possible codes and for a code length of up to six bits, there are 64 valid codes. Higher the code length, longer will be the ciphertext messages which have to be transmitted. Based on the security requirements, different code lengths can be used. We consider codes with a maximum length of five bits for illustration purposes.

From a code space of 32 codes, we have to select 28 of them, which can be done in ${}^{32}C_{28}$ ways. We generate the binary codes up to five bits in random and assign to each of the plaintext characters. Hence, for a set of selected 28 codes, 28! keys are possible. The total number of keys possible out of the 32 code space is $1.09637849 \times 10^{34}$, which is around 100 bits in length.

This mapping between the plaintext letters to the ciphertext equivalents constitutes the key. The sender and the receiver should share the key in advance. We use the key in Table 10 for explaining the encryption, decryption, and the attacks on this cipher. Note that the key is non-prefix in nature and is of variable length.

Characters	Binary Codes
space	10110
period	1101
a	101
b	11001
c	10000
d	1010
e	110
f	1000
g	10001
h	1111
I	0
j	10011
k	10100
l	100
m	1001
n	11010
o	11000
p	111
q	1100
r	10
s	10010
t	10111
u	1
v	1011
w	10101
x	11
y	11011
z	1110

Table 10: Key for non-Dictionary case

6.2.2 Encryption

The sender has to perform encryption of the plaintext message he has to send to the receiver. The sender and the receiver have the key, which consists of 28 character set and their binary code mappings in this case. A substitution cipher is used. To encrypt the plaintext message, the sender substitutes each of the letters in the message (both dictionary and non-dictionary characters) with their corresponding non-prefix codes in the key. The resulting ciphertext is a long sequence of binary bits without any delimiters although there are delimiters in the plaintext message.

For example, if a sentence 'he picked up his books.' is encrypted by using the key in Table , *1111110101101110100001010011010101011011111011011110100101011011001110001100010100100 10* is the resulting ciphertext. A message of 23 characters has resulted in 87 bits in the ciphertext form. Depending on the number of bits for the binary codes in the key and the number of characters in the plaintext message, the ciphertext length varies. Similarly, encryption can be performed on a plaintext message ranging from a single word to a set of paragraphs. Figure 14 shows a sample paragraph of four sentences and its ciphertext equivalent.

```
there was no answer.he felt very puzzled.he shouted again.he picked up his books.

1011111111101011010110101011011000101011011010110001011010111010100101010111010110
0111111101011010001101001011110110101111010110111011011111111011101001101010110110
1111101011010010111111000110111111010101011010110001101011010110111111110101101110
1000010100110101010110111111011011110100101011011001110001100010100100 10
```

Figure 14: Paragraph Encryption for non-Dictionary case

The binary codes vary from one to five bits in length. On an average, each letter is represented in four bits. Hence, for a plaintext message of 'k' characters, the resulting ciphertext is approximately 4*k bits in length.

6.2.3 Decryption

The receiver is given a ciphertext. The receiver knows the key. From the key, the receiver knows that there are two non-dictionary characters, a space and a period in the plaintext space and are encrypted to obtain the ciphertext. The receiver does not know the sentence or word boundaries based on the ciphertext and the key. He has to algorithmically determine them. The goal of the decryption process is to uniquely recover the plaintext.

The problem is do decrypt the ciphertext to obtain the original paragraph. We follow a dynamic programming approach to solve this problem. We start by dividing the paragraph decryption problem into several subproblems of decrypting sentences. Each of the sentence decryption problems are further divided into word decryption subproblems. These word level sub-problems are solved using the key and the results of the word sub-problems are combined to get the solution to the sentence decryption problem. The results of the sentence decryption is collectively gives the solution to our original problem. The main step lies in identifying periods in the paragraph level ciphertext to divide the problem into sentence level sub-problems and identifying space positions to further divide into word decryption sub-problems.

Each of the word decryption sub-problems may result in multiple possible plaintexts. Our algorithm should consider all these possibilities and select the best solution at any step. This is necessary because the solution to our overall problem depends on the solution of each of the sub-problems. Only the best solution for the sub-problems yield the best solution to our original problem due to their optimal substructure. To aid our decision in selecting the best solution, we make use of the dictionary, the bigram, first word, and the last word models which were built and trained earlier.

We implement this cipher in a bottom-up fashion. Once the problem is broken down into sub-

problems, they are solved first and the results are combined. The implementation is carried out in two stages namely – 1) Elimination of the non-dictionary characters and 2) Decryption of the words. In the next section, we discuss both the stages in detail.

6.2.3.1 Elimination of the non-dictionary characters

The receiver has the ciphertext and the key. The basic underlying sub-problems are the word decryption problems. The receiver has to identify the word level ciphertexts by recursively dividing the paragraph level ciphertext into sentence level ciphertexts and then dividing each of them into individual word level ciphertexts. The receiver does not have the knowledge of the number sentences or the number words in the plaintext message.

The first step is to eliminate the periods and identify sentence level ciphertexts using the key. With the binary code for a period in the key, we start with identifying the "possible" positions for a period in the ciphertext. Each of the sub-problems have to be solved by decrypting the individual words. Once the sentence decryption is solved, the result is combined to solve the original problem.

For each of the possible period positions, we try to decrypt the sentence. If the decryption yields a valid sentence, it is considered for the possible plaintext. On the other hand, if the decryption does not yield a valid text, it means that the identified period was wrong. We do not consider this possibility for further decryption.

For performing the sentence decryption, we have to identify the spaces in the sentence ciphertext. We do not have knowledge of the number of words in the ciphertext. Given an encrypted sentence, we identify all "possible" spaces in the ciphertext. We make use of the word decryption algorithm to decrypt the possible individual words. At this point, we do not know the exact locations of spaces in the ciphertext.

To decrypt a sentence, we first need to decrypt the individual words. The current state of the decryption algorithm is specified by the following – the yet to be decoded ciphertext 'C' and the already decoded plaintext 'P'. We assume we are decrypting from left to right. The yet to be decoded ciphertext contains the remaining words of the sentence. The algorithm proceeds as follows:

We maintain a queue of all current valid states to aid in navigating the solution space. The queue initially contains $\{C_{in},' '\}$ and we need to reach a state where all the elements of the queue are of the form $\{' ',P_{out}\}$. In each step, we remove the first element of the queue and process it. We look for the first possible occurrence of the binary code for space in it. If the binary sequence occurring before this space decodes to a valid word using the word decryption algorithm, we push the node {C',P'} on to the queue, where C' is the remaining ciphertext after the space and P' is the plaintext got after adding the decoded word to the current plaintext. If the possible space did not lead to a valid word, we move to the next possible space and consider the word between the start and that position. If we could not find a valid word at the beginning of the ciphertext, we do not push back anything to the queue. This is effectively pruning all the invalid possibilities. We iterate the process until none of the nodes in the queue have no ciphertext to decode. For any word position, if we get more than one possible word, we use the probabilistic models to decide the best word in the context. We then combine the decrypted words to form the sentence. As we are only combining the correct solutions at each step, we obtain a final correct solution at the end of decryption.

6.2.3.2 Decryption of words

The goal of word decryption is to recover the plaintext word from its ciphertext equivalent. We try to decrypt character by character. After each character is decoded, we continue with the next character only when the decrypted characters yield a “valid” prefix. A valid prefix is a prefix of a word in the

dictionary. In doing so, we prune all the invalid possibilities immediately and eliminate them from further decryption. Only the valid words make through till the end of the word decryption process.

Let E be the encrypted ciphertext. Each of the sub-problems can be represented as a tuple {e, d} where e is a suffix of the input E and d is the decrypted plaintext for the prefix of E excluding e. The initial problem state is {E, ' '} and the solution set contains tuples of the form {' ', D}, where D is a sequence of valid words.

Given a {e, d}, eliminating all prefixes of e occurring between the two consecutive spaces within the 125 bit window that decodes as words in the dictionary lead to a valid sub-problem. There can be many valid sub-problems for a given {e, d}. We maintain a vector of all the valid sub-problems. We remove sub-problems from the tail of the vector and add back smaller sub-problems. We keep repeating this process till we reach {' ', D}.

When the decryption yields only one word, there is no further processing required. But, when a decryption of word yields many possible words, we make use of the constructed models to compare the probabilities and proceed with the best word. We make use of the first word model for selecting the best first word and the last word model for selecting the best last word.

We have collected some sample statistics of the word decryption program with a set of 875 encrypted words. Figure 15 shows the outcome. Two runs were conducted. In the first run, 859 words were exactly decrypted (no ambiguity) and 16 yielded two possible valid words. The program was run again with a different key. This run yielded 863 words which were exactly decrypted and 12 words giving two possible valid words. Considering both the runs, there were no cases yielding more than two possible valid words.

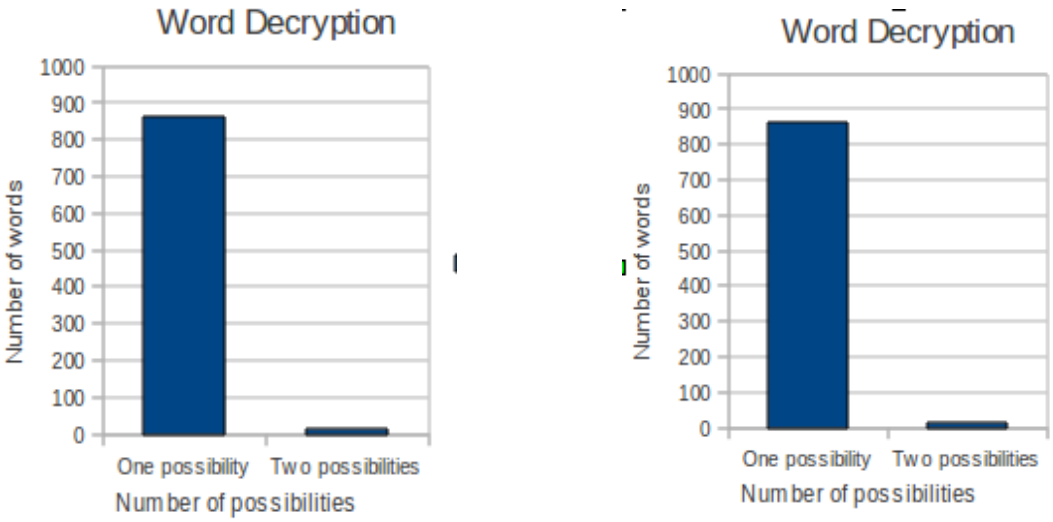

Figure 15: Word Decryption for non-Dictionary case

One example of a ciphertext yielding more than one possible valid words would be *10001101011001010010*. Both the valid words “grass” and “mass” have this ciphertext. These kinds of ambiguities are resolved by using the probabilistic models as discussed in the previous section.

6.2.3.3 Results

We have verified the correctness of our algorithm with a few hundreds of paragraphs, which in turn are composed of several hundreds of sentences and thousands of words. We encrypted all these chosen paragraphs and stored the ciphertext outcome in to a text file.

We provided this text file as input to our decryption program. We were able to uniquely decrypt the ciphertexts to obtain the correct plaintext. Figures 16-19 show some of the encrypted paragraphs/sentences and their decryption results.

```
1011011010001010100101101010110110010101100110101011010111111111101011010001101101010110110101101
0101101000101001101011011001010101010010101101011111000110001101
1011011011001110001101110110100001011001110101101111110110101111111111010110111101101111111101
1011111111101101110110100101110001111111010101011010011000110101001101111101
1011111111101101110110101011100011001010101101100111010110100010011011010101010100101101
```

Figure 16: Ciphertext example

```
a girl was in the garden
i like birds too
a boy came up the path
they laughed loudly
they would be friends
```

Figure 17: Decryption outcome

```
1011111111101011011000101011101011010101101100010001011010111111111010110101111
0001000011010110100111001010111101101100111010110100001100010011111001101011111
1001101110110100110110101101111111010110101011100011001010101101010110111111111
1010110100011100010110111111000100111010110101010101111111110001101111011010010
1011001101010001101101111010011101

1111110101101001100011000101001101010101101000010110110100011001001101110110101
0001110101010110111111110101101111101101010110110101100010110010101101011011010
0111101101011001001101

1111110101101111010000101001101010101101111101101111010010101101100111000110001
0010010110111111110101101011001001110001001010111101101001111001111110101010110
0001101111101101100010001011011110100101011010010101000110101101

1011111111101011010010111000111111101111101010110100101011111000111111110101010
0100101101010101101101010011011110111111110101101010101001011111101010101101111
0101101010110110010101101100110110000101001011010110111101101011111111010110111
1110001100101101101

1111110101101001100011000101001101010101101111101100110101011111000101101011111
1110101101011110110110101101011101010101011010010101101011011010110110110011010
1110111010001100101101111101101011000101111101110111100011011010001101101111010
0100111010110101110101011010110101100011001011010001001000111111101111101
```

```
the owner of the voice must be completely mad.he would rather go home without seeing him.
he looked carefully round.he had no idea at all.
he picked up his books.he almost jumped out of his skin.
the laughter stopped suddenly.he wished he was back at the house.
he looked up into the tree and saw a beautiful parrot.you gave me an awful fright.
```

Figure 19: Paragraph decryption outcome

Based on the results of our experiment, we can conclude that a unique decryption is possible even when non-dictionary characters are added to the plaintext space.

7.0 Known Plaintext Attack Implementation

Now that we know this cipher with non-dictionary characters can be used for encryption and decryption, we want to try to attack the cipher. We will attempt to break the ciphertext to obtain a partial message or in the best case, to crack the key.

The attacker can only see the ciphertext messages exchanged by the sender and the receiver. In this case, the ciphertext is only a binary sequence, without any delimiters. The attacker also knows the algorithm used in this cipher, only the key is unknown.

A straightforward approach for an attacker is to conduct an exhaustive search based on the ciphertext-only case. With only a ciphertext in hand, an attacker can generate all the possible keys and try them all. For each possible key, decryption has to be performed on the ciphertext. The combination for which the decryption outcome yields a readable english message is the correct key. If the attacker does not have the knowledge about the type of or length of the key, this attack is even more harder just based on the ciphertext.

Suppose that the attacker knows that non-prefix codes of up to five bits per character was used for encryption. In that case, the character set consists of 28 characters and hence 28 binary codes form the key. As 28 codes were selected from a set of 32 keys, there are $^{32}C_{28}$*28! possible keys for encryption and decryption. So, to conduct an exhaustive search, on an average, an attacker has to try half of them to find a match, which is 548189243611861521181704192000000 keys or around 100 bits. Similarly, if 8 bit codes were used, the attacker's work is in the order of 10^{66}, which is around 220 bits. Even if one trial takes one millisecond to complete, the attack takes an enormous time to complete.

In most of the practical cases, a ciphertext only attack is successful because of the statistical analysis in the ciphertext. We have collected statistical information of the various possible binary codes within a

five bit limit and their frequencies in our ciphertext. Table 11 shows the frequency counts captured using two sets of ciphertext.

Binary code	Frequency – set1	Frequency – set2
0	76	63
1	108	99
10	52	46
11	38	35
100	14	11
101	26	26
110	27	23
111	13	15
1000	0	0
1001	0	0
1010	16	12
1011	17	12
1100	0	0
1101	18	15
1110	9	9
1111	8	8
10000	0	0
10001	0	0
10010	0	0
10011	0	0
10100	0	0
10101	11	9
10110	10	9
10111	0	0
11000	0	0
11001	0	0
11010	14	10
11011	0	0
11100	0	0
11101	0	0
11110	0	0
11111	0	0

Table 11: Frequency counts of non-prefix codes

If we try to identify letters in the plaintext based on the frequency counts, we do not get any breakthrough. This attack is very difficult due to the variable length as well as the non-prefix property of the binary codes. Not being able to identify letter boundaries means there is no significant statistical information available. Even the length of the plaintext message is not determinable due to the variable

length binary code in the key. Hence, a ciphertext-only attack is very hard to perform in this case.

In real world scenarios, the attacker not only knows the ciphertext but also has some of the plaintext information. So, an exhaustive search is not necessary. The attacker has to use a smart technique based on the cipher algorithm used, inorder to ease his work. A known plaintext attack is a cryptanalytic attack in which the attacker is aware of some plaintext and its corresponding ciphertext [8]. In our case, we make an assumption that, the attacker knows a few sentences and their corresponding ciphertexts. The codes for the individual characters are not known. The goal is to extract the key based on the known plaintext and ciphertext.

We follow a bottom up dynamic programming approach to solve this problem . We divide our implementation into two stages 1) Identify word boundaries and 2) Extract key.

We know a set of plaintext sentences and their ciphertext equivalents. We also assume the knowledge of the code for one of the characters – the space. The goal of the first stage is to identify the possible space positions in the ciphertext and find the word level ciphertexts. These word level ciphertexts are then passed to the second stage for key extraction.

For each of the sentences, we try to find the possible delimiter positions. We try to eliminate the invalid combinations. For each of the remaining delimiter combinations, the sentence level ciphertexts are divided into word level ciphertexts. In this section, we describe the implementation details of each of them.

7.1 Identify word boundaries

The first step is to identify code for the non-dictionary characters based on the given sentences and their ciphertexts. For example, the code for the period can possibly be got by comparing the last few bits in the ciphertext across all the sentences. If the last character before the period is not the same across all the sentences, the longest common subsequence is likely the code for the period. By doing this, we may be able to narrow down the number of possibilities for the period character.

Based on the frequency of occurrence and positions of the non-dictionary characters in the known plaintext, we identify the possible occurrences of these in the ciphertext. With these, if the attacker is able to get the mapping for all the non-dictionary characters, then it is reasonably easy for the attacker to decode the rest of the key. In the rest of this section, we assume that the attacker is already aware of the key for the non-dictionary characters and discuss the ways in which he can proceed from there onwards.

For illustration, in Table 12, we show the possible number of combinations that have to be tried to identify the rest of the words in a set of sentences that has space as the only non-dictionary character and assuming the mapping for the character space is known.

Sentence #	# word combinations	Actual # words
1	7920 (${}^{11}C_4$)	4
2	24 (4${}^{1}C_4$)	3
3	24 (${}^{14}C_4$)	3
4	42 (${}^{7}C_2$)	2
5	1 (${}^{1}C_1$)	1
6	56 (${}^{8}C_3$)	3
7	24 (${}^{4}C_3$)	3
8	6 (${}^{4}C_2$)	2
9	15 (${}^{6}C_2$)	2
10	210 (${}^{10}C_6$)	6

Table 12: Possible word combinations

Let us say, we have a sentence 'it was really most extraordinary' and its ciphertext is *0101111011010101101100101011010110101100100110111011010011100010010101111011011011101111010111100010101001101010110110111101*. In this encoding, given the fact that the encoding for space is *10110*, there are 11 occurrences of it in the ciphertext. But, only four occurrences of it in the original text. To identify the word boundaries, we need to try $^{11}C_4$ combinations. From each of these combinations, we get five ciphertexts which possibly correspond to each of the words in the sentence.

The amount of information available in each sentence is not sufficient to be able to unambiguously decode the key. In our attack, we need many such sentences to be able to extract common information across different sentences to prune away the incorrect possibilities.

Individual sentences can produce thousands of combinations of possible words. The number of combinations that need to be considered across sentences is multiplicative. For example, in the paragraph used in the table above, the total number of possible word combinations across the paragraph is 4866962817024000. To reduce this huge number of combinations, we will prune the invalid combinations right at the beginning.

To eliminate invalid combinations, we make use of the length properties of the plaintext and its possible ciphertext. We know the maximum number of bits in a binary code for a character. Based on the calculated minimum and maximum lengths for a given plaintext word, we eliminate the space positions which do not qualify to this criteria. For example, if the third word of a sentence is of length five characters and the ciphertext between the second and the third occurrence of the binary code for space is five bits long, then either of the second or the third occurrence is not a space. By avoiding such invalid combinations, the problem space can be pruned drastically. Table 13 shows the total number of combinations if we were to use brute force vs the number of combinations after eliminating invalid combinations from our program. We notice that the final number of combinations to be tried is very

less compared to the original number of combinations.

Original #combinations	Pruned #combinations
4866962817024000	864
86207193662782800000000	1152
9414601612416000	768
370039154284800000	119808

Table 13: Space combinations

Only one of the combinations yield a valid key. For each of these combinations, we try to extract the key based on our key extraction algorithm (explained in the next section). The combination with the highest score returned by our underlying algorithm is the best combination. It means that all/most of the key was extracted based on the given input.

Table 14 shows the number of combinations which had to be tried to extract the key and the actual iteration at which the key was obtained.

#combinations to try	#combinations tried
864	50
1152	328
768	1
119808	60
55296	27692
4096	474
4608	2790
16	8
288	158
6145	100

Table 14: Combinations tried

7.2 Extract the key

The input to this stage is a series of pairs of plaintext words and their possible ciphertexts. The ciphertext supplied may not be the correct ciphertext. Multiple possible ciphertexts for the same word can be given. Of these, only one of them is a valid plaintext-ciphertext combination. The goal is to extract the key for all the letters present in the plaintext.

The algorithm has to be capable of pruning the invalid combinations quickly as numerous trials have to be tried. In order to eliminate the invalid combinations, before attempting to extract the key, we perform many checks.

To start with, we perform a word-ciphertext consistency check. That is, if the same word occurs more than once in the plaintext, its possible ciphertext must also match. In case of any mismatch, we send an invalid signal indicating that the combination was wrong and proceed with the next one. Once this check is cleared, we start with the key extraction.

A series of iterations are performed on the given set of words and ciphertexts to extract the key. At each iteration, we try to extract as many as letter mappings as possible. Each iteration is scored based on the number of letter-code mappings found till this iteration. To begin with, the best score is zero. After each iteration, the current score is compared with the score of the previous iteration. If the score improves, we continue the process. On the other hand, if the score does not improve, we stop the trial because no further extraction is possible. We indicate to the higher level that this combination does not appear to be valid or that the key generated is incomplete.

At each iteration, we try to find the key by using the plaintext words and their ciphertexts. We use several string comparison techniques to narrow down our problem. To begin with, we target the words which have only one unknown letter. This can consist of a single letter word or a result after prefix and

suffix replacements based on other known words.

For example, consider the original sentences 'you shut the door' and 'he roared back'. Given a possible ciphertext for 'the' and a possible ciphertext for 'he', if the code for 'he' is not a suffix in the code for 'the', then this combination is not a valid combination. If it is indeed a valid suffix in the code for 'the', then the remaining prefix is a possible code for the letter 't'. This can again be cross verified with some other words in other sentences.

We maintain all the letter-code mappings extracted. After each letter mapping is decoded, we perform a key consistency check. That is, we confirm that two different letters do not have the same code or the same letter has not resulted in two different codes. If the consistency check fails, we stop processing this key. If the checks have passed, we continue the process using substring comparisons across multiple words. Several iterations are performed until the best score is reached and the key is fully extracted. Upon successful key generation, we conduct a few more checks to confirm if the key is valid. The combination which has made through all these checks is the final key.

Depending on the number of letters in the plaintext, several iterations have to be performed on the words to extract the key. Table 15 shows the number of letters present in the plaintext and the number of iterations needed to obtain the key with our program. The value corresponding to each iteration is the cumulative number of letter codes found at the end of the iteration. In the examples considered, the key was extracted by performing two-five iterations. On an average, it takes three iterations to extract the key.

# unique letters	# words	Iteration #1	Iteration #2	Iteration #3	Iteration #4	Iteration #5	Total # iterations
21	45	9	20	21			3
23	47	7	8	20	23		4
23	42	6	10	14	22	23	5
23	56	20	23				2
22	54	10	21	22			3
23	54	13	22	23			3
23	42	7	17	23			3
22	32	4	6	18	23	24	5
24	32	2	6	18	23	24	5
23	52	18	23				2

Table 15:Key Extraction

8.0 Conclusions and Future Work

In this project, we explored the feasibility of using a substitution cipher with non-prefix codes. Encryption using non-prefix codes is very similar to any other substitution cipher. It was also possible to correctly decrypt using the key to obtain the plaintext.

The cipher was quite secure in our experimental analysis. Frequency analysis of ciphertexts do not give away much meaningful information. We were not able to perform a ciphertext-only attack. However, with a few assumptions, we could perform a limited known plaintext attack on the system.

In this project, we considered encryption and decryption of English text. Future extensions of this work can be to try encryption and decryption with other languages and a bigger character set.

Another possible extension to this project would be to consider attacks on the cipher with only dictionary characters. In the case with non-dictionary characters, we were able to divide the problem based on “possible” spaces and try to extract the key from the ciphertext. It is an interesting problem to identify sub-problems and extract the key in the case without any non-dictionary characters.

Another interesting problem would be to look at the ciphertext-only attack from a different perspective other than the frequency analysis.

References

[1] Merge sort
http://en.wikipedia.org/wiki/Merge_sort

[2] Olson, Edwin (2007). Robust Dictionary Attack of Short Simple Substitution Ciphers [Electronic version]. *Journal: Cryptologia, 31(4), pp. 332-342.*

[3] Programming and data structures: Sorting [Electronic version]. Retrieved on 22nd May from Linux Indore website:
linuxindore.com/downloads/download/kernel-tutorials/sorting

[4] Stamp, Mark. Crypto, a power point presentation [Electronic version]. Retrieved on 22nd May from SJSU website:
http://www.cs.sjsu.edu/~stamp/infosec/PowerPoint_PDF/

[5] Stamp, Mark. Information security: principles and practice, second edition. Wiley, 2011

[6] Stamp, Mark and Low, Richard. Applied Cryptanalysis: Breaking ciphers in the Real World, first edition. Wiley- IEEE press, 2007

[7] Thomas, Cormen, et. al. Introduction to Algorithms, second edition, MIT Press, 2001

[8] Whitfield, Diffie and Hellman, Martin (1997). New Directions in Cryptography [Electronic version]. *Journal: Information Theory, IEEE, pp.* 314

[9] Shah, A. Approximate disassembly using dynamic programming. [Electronic version]. Retrieved on 22nd May from SJSU website:
http://www.cs.sjsu.edu/faculty/stamp/students/shah_abhishek.pdf

[10] Massey, James (1994). Some Applications of Source Coding in Cryptography [Electronic version]. *Journal: European Transactions on Telecommunications and Related Technologies, v 5, n 4, p 421-9*

[11] Wilshusen, Gregory (2009). Cyber Threats and Vulnerabilities Place Federal Systems at Risk [Electronic version]. Retrieved on 22nd May from Cyberloop website:
http://www.cyberloop.org/library/information-security-cyber-threats-and-vulnerabilities-place-federal-systems-at-risk.html/

[12] Vanstone, S, et. al. Handbook of Applied Cryptography, CRC Press, 1996

[13] Schneier, B. Applied Cryptography, second edition. John Wiley & Sons, Inc. New York, 1996

[14] Tolga, M. et. al. (2004) Cryptography Education for Students [Electronic version]. IEEE conference, pp. 621-626

[15] Shannon, C. (1990) Communication Theory of Secrecy Systems [Electronic version]. *Journal: Computer Security, v 6, n 2, p 7-66*

[16] Sedgewick, Robert. (2011) Optimization [Electronic version]. Retrieved on 22nd May from Princeton website:
http://introcs.cs.princeton.edu/96optimization/

[17] Yean, Ho. et al. (2005) Heuristic Cryptanalysis of Classical and Modern Ciphers [Electronic version]. IEEE conference, pp. 710-715

[18] Milidiu, R.L., Mello, C.G., Fernandes J.R. (2005) A Huffman-based text encryption algorithm [Electronic version]. SSI - Computer Security Symposium

[19] Milidiu. R.L., et. al. (2003) Introducing security into prefix-free encoding schemes [Electronic version]. PUC- RioInf.MCC