Applied Cryptography

Project 1 Report

Main Programmer: Yang Liu

Main Report Writer and Analysis: Elon Xie

**Table of Contents**

Introduction 3

Experiment and Cryptosystem Analysis 7

Dictionary Analysis 8

Implementation 12

Analysis and Test Cases 14

Conclusion 19

## Introduction

Classical cryptography represents the era of cryptography up until the mid-1970s and 1980s. During that period cryptographers worked on creating ciphers with increasing secrecy. A mono-alphabetic substitution cipher is a cipher from the classic cryptography era. In it all of the letters of the alphabet are randomly mapped to different letters. Usage of a mono-alphabetic cipher substitution involves the following three steps:

**Key Generation**

A key is composed of the 26 letters of the alphabet in a randomized order. Each of the randomized letters is mapped to a letter from the alphabet. An example key is shown with the letters of the alphabet each mapped to another letter in a random order. The key is the second row of the table below.

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

**Encryption**

For encryption, each letter of the plaintext is swapped with the letter it is mapped to. An example is shown below using the key from above.

**Decryption**

For decryption the letter is mapped backgoing from the letters on the key to the letters mapped to it. An example decryption using the key from above and the cipher-text from above is shown below.

A major weakness of the mono-alphabetic substitution cipher was that it was possible to recreate the key based on the fact that certain letters appear more in the written language.

This report analyzes a modified version of the mono-alphabetic substitution cipher that fixes this by expanding keys to accommodate frequency of letter based attacks. The modified cipher allocates numbers to each of the letters of the alphabet instead of allocating letters. Also each letter of the alphabet can be mapped to multiple numbers, with the number of letters they are mapped to being based on the average frequency of the letters based on the following frequency chart.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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** |
| 8 | 1 | 3 | 4 | 13 | 2 | 2 | 6 | 7 | 1 | 1 | 4 | 2 | 7 | 8 | 2 | 1 | 6 | 6 | 9 | 3 | 1 | 2 | 1 | 2 | 1 |

This leads to changes in the methods of key generation, encryption, and decryption.

**Key Generation**

First a random ordering of the numbers 0 through 102 is made. Then the ordering is distributed amongst the letters with the number of letters being assigned to each letter being based on the frequency table. An example key is shown below:

|  |  |
| --- | --- |
| a | 92, 43, 82, 94, 88, 2, 102, 93 |
| b | 42 |
| c | 87, 9, 61 |
| d | 14, 13, 89, 20 |
| e | 68, 54, 60, 55, 34, 36, 58, 100, 39, 74, 80, 12, 19 |
| f | 23, 53 |
| g | 69, 3 |
| h | 57, 17, 32, 37, 30, 24 |
| i | 8, 83, 48, 7, 38, 22, 91 |
| j | 31 |
| k | 81 |
| l | 26, 4, 50, 10 |
| m | 44, 27 |
| n | 52, 101, 21, 72, 46, 25, 64 |
| o | 76, 96, 71, 45, 75, 65, 16, 95 |
| p | 79, 97 |
| q | 66 |
| r | 51, 99, 40, 85, 11, 84 |
| s | 18, 49, 35, 47, 90, 33 |
| t | 98, 67, 5, 73, 63, 6, 28, 15, 77 |
| u | 62, 1, 29 |
| v | 56 |
| w | 70, 78 |
| x | 86 |
| y | 59, 41 |
| z | 0 |

**Encryption**

Encryption is done in a similar fashion as the original mono-alphabetic cipher. The letter from the plaintext message is mapped to one of the numbers corresponding to that letter. Which of the numbers corresponding to the letters mapped is determined by a scheduling algorithm. The possible scheduling algorithms for the modified version of the mono alphabetic substitution cipher are left unknown except for one:

If a space is to be inputted, it is kept the same for the process of encryption and decryption. An example of encryption using this modified cipher is shown below using the previous key and scheduling algorithm:

**Decryption**

Decryption is similar to the original concept of the mono alphabetic cipher. The numbers in the cipher text are simply mapped back through the key to the original letters.

This report analyzes an attempt at cracking the modified mono alphabetic cipher. First it will explain the rules of the experiment involved and then show how a dictionary analysis was used to implement a program that can crack the modified mono-alphabetic substitution cipher. Lastly, the report also analyzes the implementation of the cracking program.

**Dependencies**

* boost serialization library installed on the system for compiling
* cmake: 3.4.1
* c++ standard: c++11
* compiler: Apple LLVM version 7.0.2 (clang-700.1.81)
* executable format: Mach-O 64-bit executable x86\_64

## Experiment and Cryptosystem Analysis

For the experiment used to crack the system, five dictionaries have been created. One of the dictionaries will have random words selected from it to form a 500 character plaintext that includes spacing between the words. Then the plaintext will be encrypted into a cipher-text. The experiment is considered a success if the cipher-text is deciphered correctly back into the plaintext.

For the experiment, our cracking program is given all five of the dictionaries, the cipher-text, and the frequency of letters. Missing is the type of scheduling algorithm used as well as the key and plaintext.

An additional experiment replaces the five dictionaries with the English dictionary, selecting random words from it for the cipher-text. In this case the cracking program will also take in the English Dictionary in order to deal with this.

An analysis of the cryptosystem shows that there is no major flaw in the cryptosystem to exploit to speed up the process. Since the cipher-text is made from words, and spaces are not encrypted it can be possible to compare the cipher-texts to the possible words that are the same length. From there it might be possible that an analysis of the dictionary can help create a more refined attack.

## Dictionary Analysis

From the contents we can tell that the size of the dictionaries are pretty small. A quicker search could be created by starting with the easiest words i.e. the ones which has the least candidates. The analysis of the dictionary lists the ease of the words as the fault tolerance. The fault tolerance should be calculated as:

For example, the fault tolerance of "eat" = 13 \* 8 \* 9 = 936

Sorting the dictionaries by length and then fault tolerance leads to the following:

The format is ${word length}($priority) => [ $word($fault tolerance), … ]

**Dictionary #1**

1(4) => [p(2), ]

4(3) => [hake(624), ]

5(8) => [colly(768), pouch(864), lacer(7488), ]

6(12) => [kevils(2184), philip(4704), bogart(6912), lights(18144), dredge(32448), resins(137592), ]

7(14) => [cumquat(3888), polypus(4608), kayaker(9984), rebukes(18252), william(25088), swagmen(34944), blunted(39312), anthrax(145152), sconced(157248), academe(259584), coastal(331776), aniline(998816), ]

8(13) => [trembler(438048), breveted(474552), obstacle(539136), cyanosis(677376), parapets(1078272), shriller(1886976), research(5256576), ]

9(11) => [toothpick(1306368), scoutings(2286144), pyrometer(3504384), atavistic(4572288), agileness(19079424), ]

10(10) => [almightily(2709504), gratuitous(23514624), endoscopes(65415168), ostensible(85857408), ]

11(7) => [concertizes(193179168), nightshirts(432081216), unresistant(1872351936), ]

12(9) => [underbidding(71914752), nonexplosive(178074624), outstripping(192036096), corespondent(4121155584), ]

13(6) => [industrialist(16131032064), editorializer(19232059392), ]

14(2) => [psychoneurosis(11412430848), ]

15(5) => [nondistribution(263473523712), plenipotentiary(403873247232), ]

17(1) => [incompressibility(326205315072), ]

**Dictionary #2**

3(3) => [fly(16), ]

4(2) => [bald(128), ]

5(6) => [snaky(672), zazen(728), laked(1664), crony(2016), ]

6(10) => [pizzas(672), hangup(4032), bottom(10368), fracas(13824), shaped(29952), fender(56784), ]

7(9) => [nudging(16464), propose(119808), lincoln(131712), guessed(146016), versers(219024), aperies(681408), ]

8(12) => [polypous(36864), musketry(50544), bedmaker(64896), revering(596232), hydrated(1078272), exegeses(2056392), vitiates(2476656), rightest(3184272), ]

9(8) => [pampering(489216), warningly(526848), juliennes(4173624), serologic(5031936), beltlines(7154784), entrapped(8176896), ]

10(11) => [causewayed(9345024), discharged(15095808), revelation(114476544), sweatshirt(152845056), scientists(234043992), thereuntil(289768752), interacted(386358336), ]

11(7) => [manifolding(19668992), intolerably(70447104), freemasonry(130830336), worldliness(140894208), reorganizes(228953088), authoresses(2270840832), ]

12(4) => [divisiveness(350584416), decapitation(4438167552), ]

13(5) => [reemphasizing(801335808), impersonalize(2136895488), efflorescence(8291372544), presentations(129816400896), ]

14(1) => [unprogressivel(4121155584), ]

**Dictionary #3**

1(3) => [m(2), ]

3(2) => [req(78), ]

4(1) => [boca(192), ]

5(8) => [gyrus(432), pirog(1344), coign(2352), fatal(4608), jesse(6084), delta(14976), caste(16848), tesla(22464), ]

6(6) => [bribed(2184), brasil(8064), stench(88452), foetor(89856), ]

7(11) => [boughed(14976), duality(48384), gargles(59904), tripled(157248), currier(176904), stowage(179712), indexer(198744), hueless(438048), careers(876096), intrans(889056), pettier(1149876), onetime(1192464), ]

8(9) => [bubblers(5616), swankily(37632), bulgaria(64512), valvelet(194688), kinglets(275184), rustling(381024), financed(856128), learning(1712256), sleetier(19931184), ]

9(10) => [pickwicks(10584), boldfaces(479232), inwrapped(978432), videotext(3066336), expiation(5136768), corralled(5750784), continual(7112448), replanned(12719616), responder(16353792), ingestion(26968032), ]

10(5) => [concubines(5778864), purgations(12192768), ]

11(4) => [mailability(12644352), prearranges(392491008), ]

12(7) => [inapplicably(8429568), whimperingly(20547072), uncourageous(271724544), subvarieties(386358336), decongestive(1116146304), generalships(1373718528), rattlesnakes(5298628608), ]

**Dictionary #4**

4(4) => [yank(112), dile(1456), ]

5(8) => [chirk(756), gloam(1024), gaper(2496), dixie(2548), calor(4608), ogles(4992), moths(5184), ]

6(7) => [barked(2496), budded(2496), mikado(3584), skewer(12168), fatsos(41472), dither(117936), ]

7(9) => [mistook(48384), boodled(53248), changes(157248), relight(235872), chained(366912), stayers(404352), cathect(454896), electra(876096), ]

8(10) => [jiggling(10976), movables(39936), subparts(93312), lucidity(127008), luckiest(176904), purslane(628992), confides(733824), ennobled(1059968), indorsed(2935296), teariest(27597024), ]

9(11) => [medullary(479232), magicians(1580544), soldierly(3354624), conversed(4088448), cocoanuts(5225472), dictatory(5225472), sensorium(6604416), allocated(11501568), narthexes(18398016), aerospace(18690048), theomania(35223552), ]

10(3) => [riboflavin(1053696), tepidities(225375696), ]

11(2) => [windjammers(5870592), dipsomaniac(50577408), ]

12(6) => [imprecisions(647232768), mutinousness(1248234624), ravenousness(2060577792), ]

13(5) => [unsympathetic(1069915392), universalizes(1202003712), demonstration(53258010624), ]

15(1) => [desexualization(76928237568), ]

**Dictionary #5**

4(4) => [germ(312), ]

5(8) => [narks(2016), cutis(3402), heigh(6552), conte(19656), ]

6(3) => [watson(48384), ]

7(9) => [bedrock(7488), burgess(16848), puccini(18522), million(87808), ballets(89856), sudsers(202176), ]

8(10) => [flocking(18816), honeybun(183456), unlading(263424), nickeled(397488), dentally(838656), kantians(1185408), nescient(9390654), titaness(14859936), ]

9(12) => [variously(387072), weaklings(489216), qualifier(733824), polemical(1118208), landfalls(1376256), ramshorns(6967296), clipsheet(9199008), prenticed(10732176), certifier(16098264), reediness(93012192), ]

10(11) => [highballed(3354624), debauchery(3504384), anklebones(12719616), crinkliest(17336592), brothering(19813248), empoisoned(50878464), nonsecular(52835328), serenaders(637797888), easterlies(956696832), ]

11(7) => [unselfishly(26417664), bottlenecks(55194048), shellackers(84105216), distributed(89159616), ]

12(6) => [imperceptive(139518288), unscriptural(164602368), meditatively(343429632), ]

13(5) => [hydrodynamics(260112384), encirclements(17579304288), ]

14(2) => [manipulability(531062784), ]

15(1) => [ostensibilities(1590250910976), ]

From the analysis, the possible candidates from some of the words of certain lengths like 1, 14, 17, etc are very low, in some cases even one word. That is to say, once the cracking program comes across a cipher-text with such length, it can immediately get its corresponding plaintext word and the partial key mapping of it assuming it gets it from the right dictionary. These words are considered of high priority and thus the dictionaries will have their contents prioritized by the number of possible candidates. This will help the implementation of the program by speeding up the process described in the next section.

## Implementation

Based on the knowledge learned from the dictionary, let's reconsider the key space. It is a prerequisite that the whole length of the cipher-text is around 500. As an example, say that the plaintext is built up with 72 words, each with a length of 7, with the words from the first plaintext dictionary.

The size of the key space will be |Words with length[7]|^72 = 12^72. This is impractical to iterate from it. But if we use a depth-first search, we can easily prune the number of iterations based on the pre-binded key mappings. A depth-first search is an algorithm used for traversal of graphs which goes through all possible options by backtracking in a choice when it hits a dead end through recursion. In this case the cracking program uses a modified depth first search which backtracks anytime it discovers that it has been made a wrong decision when it comes to mapping words of the cipher-text to words of the plain text.

From the dictionary analysis it is known that for certain word lengths there is only one possible candidate for a specific possible plaintext. The implementation will go through those first and then it will go through words with 2 candidates, words with 3 candidates, and so on mapping the cipher-text components to the letters of the word that it thinks the cipher-text represents, forming a key. For words with the same number of candidates, the cracking program will start checking from the longest one as it will go through the most letters that have already been mapped allowing the cracking program to detect mapping conflicts earlier. In order to make conflicts appear as soon as possible, the cracking program will sort the dictionaries by placing the words with the least fault tolerance in the beginning for the depth first search to iterate through.

Thus in conclusion the cracking program will iterate through the dictionaries performing a depth first search, mapping the cipher-text to possible plaintext, and forming a key through that. If the depth first search encounters a mapping conflict for the key, it will move backwards. If this does not solve the problem, the cracking program moves onto the next dictionary.

## Analysis and Test Cases

The principle of making a DFS tree efficient is to make it fail as quickly as possible so that it doesn’t waste time with mappings that can’t be a valid answer. Because the efficiency of pruning of the DFS tree depends on the input heavily, we’ll have to base it on multiple inputs.

The total running time is computed as the following:

First Prep Work is the amount of work needed to prepare the dictionaries. From the code this is:

Next Converting is the amount of work needed to convert the strings of the cipher text to integers. From the code this results in:

Next the Crack Section’s work can be computed from the code as:

Lastly the DFS traversal can be computed as the following:

This can be computed to the following values by solving the power series.

As this is asymptotic notation this leads to the following.

The resulting equation is:

In this case the number of dictionaries and the number of characters in the dictionaries are given. The key length is a constant that cannot be changed for this cryptographic algorithm.

However our Depth First Search algorithm does not traverse all the vertexes but part of them, leading to fractional coefficients. For the case of asymptotic notation, these coefficients are irrelevant but when the application is used, these coefficients are very important for decreasing the running time. To see the effect of the coefficients on running time, several experiments were done.

Experiment 1:

The first experiment was to see how increasing the number of possible words per length (and thus the length of the dictionary) would affect running time. A dictionary of only five letter words was used for this test leading to the following results:

These results seem to indicate that despite the fact that increasing the number of possible words would drastically increase the number of nodes for the depth first search, there was negligible increase in running time that did not follow any trend.

Experiment 2:

The second experiment went to see how increasing the number of words in the dictionary would affect the running time as well as simulating what would happen with an actual English dictionary being used for forming the plaintext and deciphering the resulting cipher text. This lead to the following results:

The results indicate that the size of the dictionary has no real effect on the running time as well.

Experiment 3:

The third experiment went to see how increasing the cipher text size would affect the running time. This lead to the following results:

The results show that there are increases and they do follow a trend of around.

Experiment 4:

The fourth experiment went to see how increasing the length of the word would affect the running time. This lead to the following results:

This shows that the length of the word has no real effect on the running time.

Looking at the algorithm, this implies that the pruning effect decreases the impact of the depth first tree to the point where the cracking algorithm is mostly impacted by cipher text size. This is more consist with the following occurring.

As increases to dictionary length also seem to have little effect as does increasing the length of words, it’s likely that it leads to the following:

## Conclusion

In conclusion the program was successful in cracking the specified cryptographic system using a depth first search based algorithm. Due to the pruning of the depth first search algorithm the resulting running time is very low, being close to a (n) running time based on the size of the cipher text. The resulting program is also capable of dealing with cipher-text with words from the English dictionary as well. Future work could be done into seeing if there are further ways to decrease the running time by pruning the tree through different variables as well as further analysis to see when the pruning effect is no longer enough to make the algorithm run in (n) running time.