Introduction

This project is based on a homophonic substitution cipher known as Handy Cipher. This means that each letter in a given plaintext message can be mapped to more than one replacement. In Handy Cipher, each character is mapped to not just one replacement character, but a series of characters of length 1-5. These are chosen based on a series of specific criteria I will touch on shortly.

There are some issues with the cipher in its ability to handle input strings. Due to the nature of how it selects which character string to replace each cleartext character with, some adjustments to the input must be made. Any bigrams (two consecutive letters) that are made of letters in certain positions of the key must be appended with a fix character between them. This is explained in the encryption section. The input should also be all capitals with only the punctuations (, . - ?) used.

With this basic understanding we will move on to how the key is used to form the needed tables.

Key Usage

The key for cipher is generated as a random permutation of 41 characters. These include all the letters uppercase from A-Z, numbers 0-9, punctuations (, . - ?), and a space character represented by a carot (^).

First, the key is used to make a mapping table for each character, as well as some extra ‘padding’ characters. This is done by removing the space ‘^’ character from the key and then creating a 5 by 8 table with the remaining characters going from left to right, top to bottom (reading direction). The first five columns are for encryption, and the last three columns are used for padding.

Next a list is made by removing all numbers from the key to form a subkey. This list keeps track of each letter and punctuation allowed by what slot it is in within the subkey. (i.e. A=13, B=31, C=14, …)

This is all the information we need to generate our cryptotext.

Encryption

To generate ciphertext, the process looks at one character at a time. Each character is compared to the subkey list to find its numerical value. This will give us a 5 digit binary number

when converted from this decimal value. This is our bit mask. I added a check for invalid characters, as well as non-split invalid bigrams and escape immediately if found. Spaces are converted automatically by the code.

Without going into the details, this binary number is matched to one of the rows, columns, or diagonals from the 5 by 5 encryption table. Only the letters that overlap with 1's in the binary value previously mentioned are grabbed to form a tuple of length 1-5. This tuple is randomized before being ready to copy to the cryptotext.

The tuple to mask with is randomly selected according to a series of very specific and lengthy rules. These come in three parts. It is a regular enough occurrence that sometimes a tuple is discarded for not meeting the criteria in these rules and again a random one is selected until one does. At times a singleton is formed (only one letter is bit masked) and in longer cleartext messages, this ultimately could provide for a weakness we will discuss later.

This is the essential core cipher, however, it's easiest I found to add the padding while forming the ciphertext rather than in a second pass after it is complete. With the roll of an 8-sided die, more or less, the next tuple character or a padding character is appended to the string. This is done until there is a demand for one last tuple character and there are no more characters left in the cleartext to encrypt. Padding characters have their own rules as well to strengthen against statistical cluster analysis (padding characters identified by looking for duplicates inside the cryptotext).

It should be noted that **there is a bug in the encryption algorithm** that can cause an infinite loop unless accounted for in the code. This was not mentioned in the PDF for the assignment but the included python code did also account for it. IF a non-singleton character is encoded using a row, and the next character is a singleton whose single bit is on that same row, the result is that there is some ambiguity of how to encrypt that character. The only alternative also produces ambiguity (columns) so encryption will hit an infinite loop looking for a tuple that matches the restrictions of the cipher. The code accommodates for this by doing a 'peek' of the next character and choosing the current tuple to not conflict with the next character.

There is another bug that the specification gives which is that bigrams made from the 5 x 5 table (using numbering here left to right top to bottom) using characters in slots 1, 2, 4, 8, and 16 create ambiguity also if they are paired as (1, 16), (2, 8), (4, 4), (8, 2), (16, 1). If we use the table made from the example key, this would be letters O = 1, N = 2, P = 4, S = 8, and E = 16. That means invalid bigrams are OE, NS, PP, SN, and EO. The suggestion is to use a hyphen '-' for the example cipher.

Decryption

Decryption is rather simple. The same lookup tables are used as before. The decryption algorithm goes through each character one at a time and discards any padding characters along the way automatically.

The tables are scanned for each character that is a valid character and tuples are reconstructed and separated to represent each cleartext character they substitute. From the tuple, we reconstruct the binary mask and convert it back to decimal.

Lastly we use a reverse lookup table to match the decimal slot number to the character it represents and add this character to the cleartext. '^'s are converted to spaces automatically before insertion into the cleartext.

Cracking Techniques

The first technique to always look at first, I feel, is the worst case scenario. Often times this is considered as brute force. I coded this without the intention of actually using it to break the cipher and retrieve the key. My findings are that brute force can, on my Macbook Pro 2015 model maxed out, try one million (1,000,000) keys in the time of 2 seconds.

This would take just over 530 quattuorvigintillion eons to calculate at most. This can be expected for a function that takes order of O(n!) where 'n' is 41. Neither I nor my computer would last long enough for such a calculation. It's safe to say that no one will be using brute force to crack this cipher.

Another possible attack that this cipher could be vulnerable to is a pseudo-random number (PRN). In my code for proof of concept I only coded using the built in rand() function which is even easier to break since those numbers are very predictable. We will not discuss any further. The PRN's however are generated by a seed.

Often times that seed is the UNIX (epoch) timestamp. If a developer were to be foolish enough to do so, then an attacker would only need to know when the key was generated. Obviously the more precise, the better, when it comes to knowing the 'when'. Even if an attacker can only narrow down to the year in which the key was generated, that leaves only approximately 2^{25} seeds to brute force.

The problem is reduced to the same code as key generation using the same rand() function previously mentioned, only using a for loop from beginning of the year to end of the year in seconds of course. This would take about 3 hours to complete using the previous brute-force analysis of 2 months worth 30 minutes. (The attempt to break the challenge cipher)

A third possible attack is what's known as a statistical analysis or 'hill-climbing' attack. This kind of attack is more useful if the cryptotext contains no padding characters. This is gone over in great lengths in a paper from SJSU itself. A similar attack, as the thesis mentions, is able to be employed to efficiently break one of the two full text Zodiac ciphers (408). This is not always a guaranteed success however as it has been unable to create a particularly strong match in the English language to the other Zodiac cipher (340)[1]. Much the same can be expected for Handy Cipher.

Lastly we can discuss the 'known plaintext' attack. This is the method that the challenge requests the coder to implement. This implies immediately that such a cipher is very vulnerable to such an attack and thus that challengers are expected to come up with the solution, and possibly a potential key for extra bragging rights.

Quality Assurance

In the main runner, I created 11 unit tests.

The first test was to make sure the key generator was working correctly by checking length and making sure every character was from the given character set. This is straightforward.

The next tests validate that a short and long cleartext can be encrypted. Since there are several ways each character can be encrypted, a specific matching text cannot be used. Instead I had to test that the text returned is not the same and that it's longer than the original text.

After this I have a tests for invalid bigrams, and a test for invalid characters. This covers the big points of invalid input.

The next two tests validate that a short and long cryptotext can be decrypted. This is easier to test vs encryption since I know exactly what should be returned in cleartext for comparison.

The next two tests validate that cleartext can be encrypted, then decrypted again back into the original text, essentially an integration test.

Finally there are two more tests, one for brute force, and one for the PRN weakness as proofs of concept.

---

[1] Efficient Cryptanalysis of Homophonic Substitution Ciphers. Amrapali Dhavare. Richard M. Low. Mark Stamp.

Artifacts

Source code can be found here: https://github.com/BJap/CS265_Project01