**Written report**

**Encryption Scheme:**

Given the pseudocode for the encryption scheme, all three of us created our own encryption methods. Our encryption schemes used the C++ library <math.h> which included a uniform distribution random number generator. We used the random number generator to create a set of numbers which were converted into the set of alphabetic letters with an additional case to take into account the possibility of generating a blank space as well. This same uniform random n umber generator was used for the coin generation algorithm which would spawn either 1 or 0. When our individual encryption algorithms were capable of converting plaintext messages into ciphertexts, we combined our algorithms to generate a set of ciphertext messages

**Decryption Scheme**:

Upon receiving the plaintext\_dictionary, we conducted a letter-based frequency analysis on all five candidate plaintext messages to use as reference for what the ciphertext may hold. While the analysis is unable to take into account the possibility of which letters are randomly generated and which letters are encoded, there is a distinctly imbalanced distribution of letters as shown in the graph below. While all five plaintext candidates have very similar distributions, it identifies and allows us to establish a default mapping of each plaintext letter might map to in regards to the ciphertext.

Plaintext #1 Frequency table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | 30 | B | 15 | C | 22 |
| D | 9 | E | 57 | F | 8 |
| G | 7 | H | 7 | I | 32 |
| J | 0 | K | 5 | L | 29 |
| M | 9 | N | 34 | O | 20 |
| P | 12 | Q | 1 | R | 36 |
| S | 44 | T | 27 | U | 22 |
| V | 2 | W | 5 | X | 2 |
| Y | 7 | Z | 4 | ‘ ’ | 54 |

Plaintext #2 Frequency table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | 33 | B | 13 | C | 20 | |
| D | 13 | E | 47 | F | 4 | |
| G | 16 | H | 14 | I | 33 | |
| J | 1 | K | 4 | L | 21 | |
| M | 8 | N | 21 | O | 36 | |
| P | 9 | Q | 0 | R | 44 | |
| S | 42 | T | 30 | U | 15 | |
| V | 6 | W | 4 | X | 1 | |
| Y | 9 | Z | 3 | ‘ ’ | | 53 | |

Plaintext #3 Frequency table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | 29 | B | 11 | C | 24 | |
| D | 17 | E | 48 | F | 3 | |
| G | 13 | H | 9 | I | 41 | |
| J | 2 | K | 4 | L | 32 | |
| M | 5 | N | 36 | O | 32 | |
| P | 10 | Q | 1 | R | 37 | |
| S | 36 | T | 22 | U | 17 | |
| V | 5 | W | 5 | X | 1 | |
| Y | 11 | Z | 1 | ‘ ’ | | 48 | |

Plaintext #4 Frequency table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | 37 | B | 7 | C | 14 | |
| D | 17 | E | 50 | F | 3 | |
| G | 21 | H | 11 | I | 414 | |
| J | 2 | K | 2 | L | 20 | |
| M | 21 | N | 33 | O | 19 | |
| P | 12 | Q | 0 | R | 35 | |
| S | 49 | T | 26 | U | 16 | |
| V | 4 | W | 2 | X | 1 | |
| Y | 6 | Z | 0 | ‘ ’ | | 51 | |

Plaintext #5 Frequency table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | 37 | B | 7 | C | 14 | |
| D | 17 | E | 50 | F | 3 | |
| G | 21 | H | 11 | I | 41 | |
| J | 2 | K | 2 | L | 20 | |
| M | 21 | N | 33 | O | 19 | |
| P | 12 | Q | 0 | R | 35 | |
| S | 49 | T | 26 | U | 16 | |
| V | 4 | W | 2 | X | 1 | |
| Y | 6 | Z | 0 | ‘ ’ | | 50 | |

Chart

Description automatically generated

Because each graph has very similar letter frequency behaviors and there is a random chance to randomly insert a new letter into the plaintext, we have opted to use the chi square encryption algorithm to determine which letter matching is the best possible match between the plaintext and the ciphertext.

Chart, bar chart

Description automatically generated

When testing our encryption scheme, we noticed that while the additional noise of random letter generation increased the frequencies of certain letters appearing, the overall distribution of each letter remained consistent with each iteration as shown in the graph above. This meant that while the addition of noise changed the frequencies of certain letters, the overall distribution remained consistent before and after the encryption scheme.

Notes:

* Concatenated the plaintext dictionary values with the plaintext values to get the information.
* What we discovered was that with the additional random probability included with the distribution has shown that
* How much work do we actually need to do in order to determine the plaintext?

Methods:

* Solving for the key
  + Metrics doesn’t scale with noise
* Distribution analysis/probabilistic to find which plaintext the ciphertext belongs to
  + Search based on the patterns found in each word
  + Estimate the noise generated to use as a margin of error
  + Look for the patterns we can find and create an approximate match
  + # of possible random characters = 1-(len(plaintext))/len(ciphertext)
  + Will implement a sliding window to find and guess which region within the ciphertext contains a matching plaintext password

A picture containing letter

Description automatically generated

Sources:

<https://mathweb.ucsd.edu/~crypto/java/EARLYCIPHERS/Monoalphabetic.html#:~:text=To%20break%20a%20monoalphabetic%20substitution,letters%20with%20the%20same%20pattern>

<https://mathweb.ucsd.edu/~crypto/java/EARLYCIPHERS/breakmono.html>

<http://s13.zetaboards.com/Crypto/topic/123882/1/>