Cryptanalysis of a Class of Shift Ciphers By Exploiting Small Key Cycles

Andrea Stojanovski, Kevin Ye, Ly Cao & Ziriu Xu

We present a cryptanalytic approach that attacks the short key length for the given encryption algorithm. The length of the key results in several non-random properties in the distributions of the lexicographic shifts used, which we exploited in Test 1 with a known-plaintext attack. In test 2, we made use of the fact that there are only some combinations of the first few words of the plaintexts and we can use the keys inferred from these words to find some meaningful decryption for the rest of the ciphertexts. We made use of careful analysis and small optimizations to allow efficient run-time that works best when there are no random characters in the ciphertext.

In the initial phases, the group also considered cryptanalytic techniques classically used to attack polyalphabetic ciphers, such as the Kasiski analysis used to break Vigenere ciphers. We pivoted from this approach after this analysis did not produce promising results for our specific use cases.

We are submitting a single approach that is divided into two sub-algorithms. One algorithm targets the first test and the other targets the second test. Upon receiving the ciphertext, the algorithm runs the first sub-algorithm and determines whether the ciphertext belongs to Test 1 plaintext type or Test 2 plaintext type. If the first sub-algorithm determines that the ciphertext belongs to test 2 plaintext type, the algorithm will run the second sub-algorithm and output otherwise it uses the output for the first sub-algorithm and does not run the second sub-algorithm.

Andrea contributed some ideas regarding the initial cryptanalysis and wrote two n-gram extraction functions for use in the team's initial analysis. However, these functions were not used in the final implementation.

Kevin wrote a function to determine candidate key lengths for use in the team's initial analysis. After the group pivoted v from this analysis, Kevin designed, implemented, and tested the algorithms for Test 1 cases. He also contributed a script to randomly generate ciphertexts for Test 1 and Test 2.

Ly contributed the ideas for the team's initial cryptanalysis and implemented the decryption algorithm. After the group pivoted from this analysis, Ly designed, implemented, and tested the algorithms for Test 2 cases.

Zirui contributed extensions to the ciphertext generation script.

Analysis Overview

Test One

For test 1, our approach makes three passes over the ciphertext. Each pass makes different assumptions about the ciphertext, as outlined below.

First Pass: No Random Characters

In the first pass, we assume the ciphertext contains no random characters. Each plaintext in the plaintext dictionary that contains the same number of characters as the ciphertext is considered a candidate. As all plaintexts in the dictionary have 500 characters, only ciphertexts with 500 characters are considered in this pass.

Since the ciphertext contains no random characters, we make the following observations:

- The lexicographic shifts needed to transform the candidate plaintext into the ciphertext can be easily computed by mapping the i^{th} plaintext character to the i^{th} ciphertext character to produce an array of shifts.
- Since the maximum key length used during encryption is 24, the number of distinct shifts used in encryption cannot exceed 24. As the total number of possible shift values is 27, this means at least 3 shift values must remain unused to transform the correct plaintext into the ciphertext.

Using these observations, we compute an array of shift values of length 500, where the i^{th} value is the shift needed to transform the i^{th} candidate plaintext character into the i^{th} ciphertext character. Mapping an incorrect plaintext to the ciphertext in this way will likely produce an array that is randomly distributed over all possible shifts. However, the correct plaintext must abide by the rule stated above and use no more than 24 distinct shifts. If we find a candidate from the plaintext dictionary that satisfies this rule, we return it as the guess.

Pseudocode

```
List_Shifts(plain, cipher)
   Shifts = []
   For i from 0 -> length of plain
      s = shift between plain[i] and cipher[i]
      Add s to Shifts
   Return Shifts

First_Pass(ciphertext)
   For each p in candidate_plaintexts
      If length of ciphertext = length of p
         Shifts = List_Shifts(p, ciphertext)
         If number of distinct Shifts < 25
            Return p
   Return None
```

Second Pass: Up to 50 Random Characters with a Cyclical Scheduler

In the second pass, we assume the ciphertext contains some random characters, but no more than 50. We also assume that the key scheduler exhibits some detectable cyclical behavior. We can no longer easily compute the array of shifts as in the first pass, by mapping the i^{th} candidate plaintext character to the i^{th} ciphertext character, since the insertion of a random character into the ciphertext throws the characters after it off by one. However, we can compute r arrays of shifts, where r is the number of random characters plus one. For the first array, we map the candidate plaintext to the [0^{th},500^{th}) ciphertext characters. The second maps the plaintext to the [1^{st},501^{st}) ciphertext characters, the third to the [2^{nd},502^{nd}) characters, and so on until we have covered the entire ciphertext with r + 1 arrays of length 500.

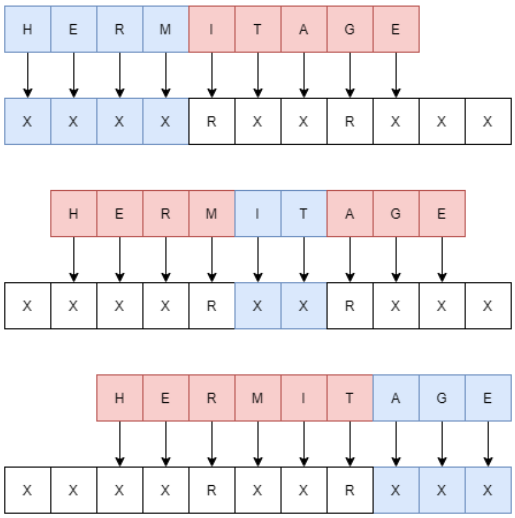


Figure 1: Computation of shift arrays during the second pass

We can observe that the full, correct array of shifts used during encryption can be constructed from subarrays of the r + 1 arrays. Figure 1 illustrates a small example with a 9 character plaintext, with r = 2. We show how we map the plaintext to the ciphertext in each iteration. The light blue squares denote plaintext characters that were mapped to the correct ciphertext characters, and thus resulted in the correct shifts. In this particular illustration, we use X to mark ciphertext characters that were shifted from plaintext characters, and R to mark the random characters inserted into the ciphertext. The first shift array would contain the shifts for the light blue characters in the top mapping, which are the characters from the beginning of the string to the first random character. The remainder of the shifts were computed from incorrect mappings, and so are garbage (denoted by red squares). The second shift array contains valid shifts between the first and second random characters. The third shift array contains valid shifts between the second random character and the end of the string.

Assuming that the actual key scheduler used during encryption exhibits cyclical behavior, then we should be able to detect cycles in the shifts used. Since the r + 1 shift arrays contain the shifts produced by the scheduler, these cycles can be identified by searching through the r+1 shift arrays. Cycle identification is performed by searching through the arrays for repeated n-grams. If we computed the shifts using the incorrect plaintext, it is likely that the shifts we computed are mostly random. However, the shift array computed from the correct plaintext likely has a significant number of repeated n-grams. In our program, we used n = 6. The maximum key length is much smaller than the length of the text, so the key cycle will repeat numerous times. Additionally, we assume the maximum value for r is 50, so on average there are gaps of 10 between each random character -- big enough to identify a sequence of 6. If we detect a significant number of repeated shift cycles for a particular candidate plaintext, we return that plaintext as our guess.

Pseudocode

```
Get_Shuffled_Shifts(plaintext,ciphertext)
      Shuffled_Shifts = []
      r = length of ciphertext - length of plaintext
      For each i in 0 -> r + 1
            s = List_Shifts(plaintext, ciphertext[i -> i+500])
            Add s to Shuffled_Shifts
      Return Shuffled_Shifts

Second_Pass(ciphertext)
      ngram_freqs = []
      For each p in candidate_plaintexts
            Shifts = Get_Shuffled_Shifts(p, ciphertext)
            c = Find the count of most frequent 6-gram in Shifts
            Add (p,c) to ngram_freqs
      candidate,count = Max of ngram_freqs
      If count > 2*average of the remainder of ngram_freqs
            Return candidate
      Return None
```

Third Pass: Up to 50 Random Characters with a Non-cyclical Scheduler

In the third pass, we assume the ciphertext contains some random characters, but no more than 50. We also assume the key scheduler exhibits no detectable cyclical behavior. In this case, we cannot use our computed shift arrays to identify non-random, cyclical behavior, as in the second pass. However, the correct shifts are still non-random. In particular, we make the following observations:

- As stated previously, the number of distinct shifts used during encryption cannot exceed 24.
- The above implies that out of all the shifts we computed in the r + 1 shift arrays in the second pass, there are 500 shifts for which at least three of the 27 possible shift values will appear with zero probability.
- Given the above two observations, we can conclude that the correct shift values that are a part of the encryption key will tend to appear more frequently.
- For the correct plaintext, the correct shifts (the ones used in the key) will appear in contiguous sections of the shift arrays (the light-blue sections above)

This is a noisy sample of data, as the vast majority of computed shifts consists of randomly distributed noise (results from incorrect mappings between plaintext and ciphertext), and only 500 data points are non-random. The data will be noisier for longer key sizes and larger values of r.

Taking the r + 1 shift arrays computed in the previous pass, we compute the frequencies of each shift value. We make the guess that, for the correct plaintext, the correct key will be some subset of the 24 most frequent shifts. For each shift value, starting from the most frequent and continuing towards the 24th most frequent, we add the shift value to a bag of shifts. We then search through the shift arrays and find the largest contiguous subarray that contains only the elements in the bag of shifts. The idea is that for the incorrect plaintexts, the locations of shifts in the arrays will be more or less random, but for the correct plaintext, the shifts that are in the key will be colocated. As we add more shifts to the bag -- starting with the most frequent to target those shifts that are more likely to

be part of the key -- these parts of the shift arrays will coalesce into larger contiguous subarrays. In the ideal situation, the bag of shifts holds exactly those shifts that are part of the key. In this case, the largest contiguous subarray most likely would correspond to the largest gap between random characters in the ciphertext.

We search through the shifts in this way computing the largest contiguous subarray for each candidate plaintext at each iteration. If the largest of these representatives is a significant deviation from the median -- that is, it exceeds some target Z-score -- we return that candidate plaintext as the guess. The noise in the data increases as we add more shifts to the bag, however, so the target Z-score grows with the size of the bag.

Pseudocode

```
Calculate_Zscore(rep, data)
      Med = median of data
      MAD = median absolute deviation of data
      Return 0.6745*|rep - med|/MAD

Third_Pass(ciphertext)
   Shuffled_Shifts = get shuffled shifts for each candidate plaintext
   For each shuffled_shifts array
      Calculate counts for each shift value
      Sort counts in descending order
   For each p in candidate_plaintexts
      Shift_bag = []
      Partition_counts = []
      For each i in 0 -> 24
         Shifts = Shuffled_Shifts array corresponding to p
         Add i^th most frequent shift value to shift_bag
         Get longest contiguous subarray of shifts containing shift_bag elems
         Partition_counts[i] = length of longest contiguous subarray
   For each i in 0 -> 24
      Get the i^th partition count from each candidate's Partition_counts array
      plaintext,max = the largest representative
      Calculate_Zscore for each representative over the set of representatives
      If max's Z-score > 4.5
         Return plaintext
   Return None
```

Test Two

This approach brute force over all possible combinations of the first two words and use these words as a possible beginning of the plaintext. The key inferred from these two words given the ciphertext is then mapped periodically in the ciphertext with the period t from 1 to 24 to see which portion gives a decryption portion that contains some words in the dictionary. We then select top 40 decryptions with the most number of words detected to use for the next decryption.

Pseudocode

```
decrypt_first_two(ciphertext, best_decryption_global, max_random_chars)
   Decryptions = []
   For each first_word in the dictionary
      For each second_word in the dictionary
            guessed_plaintext = first_word + ' ' + second_word + ' '
            key = get_key(ciphertext, guessed_plaintext, 0,
len(guessed_plaintext))
            For each t in [1,24]
                  current_decryption = guessed_plaintext
                  num_random_encountered = 0
                  For each position pos in range(0, len(ciphertext), t)
                        best_decryption = None
                        best_random_shift = None
                        remain_random_characers = max_random_chars -
num_random_encountered
                        For each random_shift in range(
remain_random_characers + 1)
                              start_pos_to_decrypt = pos + random_shift
                              decrypted = decode(ciphertext, key,
start_pos_to_decrypt)
                              words_from_decryption =
get_words_from_decryption( decrypted )
                              Set best_decryption to words_from_decryption if
words_from_decryption has more words than best_decryption and
best_random_shift to random_shift



                        Add best_decryption to current_decryption
                        Add best_random_shift to num_random_encountered
                  Add current_decryption as a Decryption object and Save t and
num_random_encountered to Decryption object to decryptions
                  Set best_decryption_global to current_decryption if
current_decryption can decrypt more words

      Limit decryptions to the first 40 longest decryptions
      Return decryptions, best_decryption_global
```

Using the decryptions from the previous round, we try all of the possible third word of the plaintext at different shiftings (from 0 to maximum number of random characters) away from the first two words and use a key obtained from the plaintext consisting of the second and third word to periodically

decrypt portions of the ciphertext with all possible periods from 1 to 24 and all possible shifts of each portion from 1 to 50 due to random characters. Again, we record if any decrypted portion gives new words to the current decryption and add to the decryption accordingly, limiting the current decryptions to 40.

Pseudocode

```
decrypt_third(ciphertext, decryptions, best_decryption_global,
max_random_chars)
   new_decryptions = []
   For previous_decryption in decryptions
      For each third_word in the dictionary
            second_word = second word from previous_decryption
            guessed_plaintext = second_word + ' ' + third_word + ' '
            key = get_key(ciphertext, guessed_plaintext, 0,
len(guessed_plaintext))
            t_range = [1,24] if max_random_chars == 0 else
[previous_decryption.t, previous_decryption.t]
            Let start_position be the first position after third word

            For each t in t_range
                  For each possible start_shift of third_word
                        current_decryption = copy of previous_decryption
                        num_random_encountered = 0
                        For each position pos in range(start_position,
len(ciphertext), t)
                              best_decryption = None
                              best_random_shift = None
                              remain_random_characers = max_random_chars -
      num_random_encountered - start_shift
                              For each random_shift in range(
      remain_random_characers + 1)
                                    start_pos_to_decrypt = pos + random_shift
                                    decrypted = decode(ciphertext, key,
      start_pos_to_decrypt)
                                    words_from_decryption =
      get_words_from_decryption(  decrypted )
                              Set best_decryption to words_from_decryption if
words_from_decryption has more words than best_decryption and
best_random_shift to random_shift

                              Add best_decryption to current_decryption
```

```
                              Add best_random_shift to num_random_encountered

                  Set best_decryption_global to current_decryption if
current_decryption can decrypt more words
                  Update num_random field in current_decryption to be
num_random_encountered
                  Add current_decryption to new_decryptions

      Limit new_decryptions to the first 40 longest decryptions
      Return new_decryptions, best_decryption_global
```

We do another iteration, and try all of the possible fourth word and use the key obtained from the third and fourth word with all possible shifts (by random characters, same as when we did decryption for the third word) and periodically decrypt portions of the ciphertext with all possible periods from 1 to 24 and all possible shifts of each portion from 1 to 50 due to random characters.

Pseudocode

```
decrypt_fourth(ciphertext, decryptions, best_decryption_global,
max_random_chars)
   new_decryptions = []
   For previous_decryption in decryptions
      For each fourth_word in the dictionary
            third_word = third word from previous_decryption
            guessed_plaintext = third_word + ' ' + fourth_word + ' '
            key = get_key(ciphertext, guessed_plaintext, 0,
len(guessed_plaintext))
            t_range = [1,24] if max_random_chars == 0 else
[previous_decryption.t, previous_decryption.t]
            Let start_position be the first position after fourth word
            For each t in t_range
                  For each possible start_shift of third_word
                        current_decryption = copy of previous_decryption
                        num_random_encountered = 0
                        For each position pos in range(start_position,
len(ciphertext), t)
                              best_decryption = None
                              best_random_shift = None
                              remain_random_characers = max_random_chars -
      num_random_encountered - start_shift
```

```
                              For each random_shift in range(
      remain_random_characers + 1)
                                    start_pos_to_decrypt = pos + random_shift
                                    decrypted = decode(ciphertext, key,
      start_pos_to_decrypt)
                                    words_from_decryption =
      get_words_from_decryption(  decrypted )
                              Set best_decryption to words_from_decryption if
words_from_decryption has more words than best_decryption and
best_random_shift to random_shift
                              Add best_decryption to current_decryption
                              Add best_random_shift to num_random_encountered

                  Set best_decryption_global to current_decryption if
current_decryption can decrypt more words
                  Update num_random field in current_decryption to be
num_random_encountered

                  Add current_decryption to new_decryptions

      Return new_decryptions, best_decryption_global
```

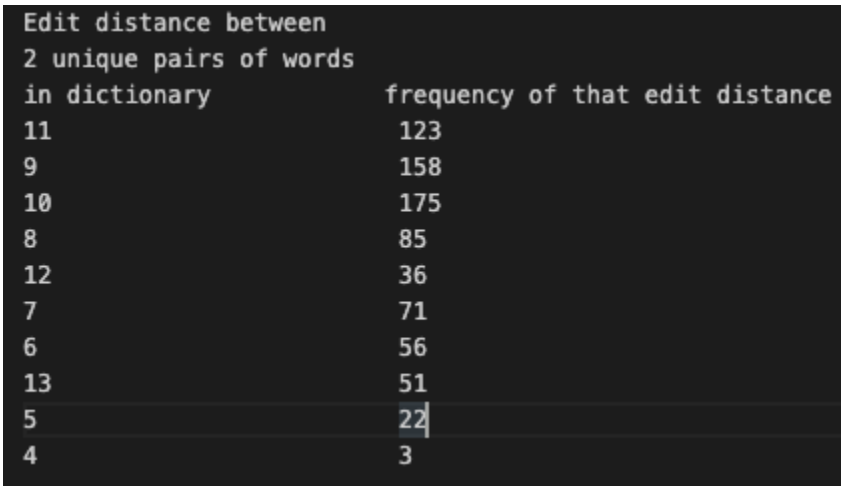
In the end, we concatenate the decrypted words in the corresponding positions for each decryption and fill in the slots with no word decrypted with empty spaces. We chose the decryption with the least overlapping words decrypted and has the highest number of words to report.

Pseudocode

```
decrypt(ciphertext, max_random_chars)
      decryptions, best_decryption_global = decrypt_first_two(ciphertext,
best_decryption_global, max_random_chars)
      decryptions, best_decryption_global = decrypt_third(ciphertext,
decryptions, best_decryption_global, max_random_chars)
      decryptions, best_decryption_global = decrypt_fourth(ciphertext,
decryptions, best_decryption_global, max_random_chars)

      If len( new_decryptions ) == 0
            new_decryptions = [ best_decryption_global ]
      Return find_best_decryption(new_decryptions)
```

Test 2 approach relies heavily on some observations on the dictionary words and on the assumption that there is some sort of repetition at some portions of the ciphertext where we can apply the same key sequence to get a correct decryption. We wanted to exploited the fact that there are only some combinations of the first few words of the plaintext that we can try and extract the key from those first guessed words and apply periodically throughout the ciphertext to see if that key sequence can decrypt some word in the dictionary. If we can decrypt other parts of the ciphertext using that key, we save the decrypted words and their positions as some possible decryptions. Since there are 24 options for key length, t $\in$ [1,24], and at most 50 random characters, we also need to bruteforce over these parameters. And for each possible search parameter (first few words as plaintext start, t, num_random), we need to loop over the ciphertext periodically to see if there is any word decrypted using that key sequence, which introduces another 500 factor into the complexity of the algorithm. We wanted to limit the complexity to around 10^9, so that leaves us with at most all combinations of the first two words, and our runtime will become $R_2 = O(40^2 * 24 * 50 * 500)$ for dictionary_size = 39, t_{max} = 24, max_num_random = 50, L = 500. Since for each key sequence of up to 24 length (the longest word has length 13 plus space so two words give a maximum length of 24), we cannot loop over each position in the decrypted portion and compare with all possible words to see which one matches since that will introduce another (40 * max_word_length) to the time complexity R_2. Instead, we precomputed a Trie dictionary with the observation that no word shares too long substring since Figure 1 shows that every word is at least 4 edit distance away from every other word. The trie will allow us to go through the decrypted portion and identify parts of the decrypted portion that are words in the dictionary in the time linear to the length of the decrypted portion.

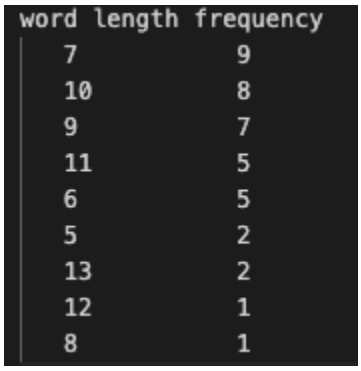


Edit distance between 2 unique pairs of words in dictionary	frequency of that edit distance
11	123
9	158
10	175
8	85
12	36
7	71
6	56
13	51
5	22
4	3

Figure 2: The edit distance of every pair of word in the dictionary and their frequencies.

We further develop the intuition that the above brute force approach above can lead to some meaningful results. We output a script log in Figure 2 that prints out the length of each word in the dictionary and how often this length appears in the dictionary. The three most common lengths are 7, 10, and 9. Furthermore, since there are at most 50 random characters in a plaintext of 500 characters, we expect that there are about one random character in every 10 characters, and in between, there is a high chance that we can see the three most frequent words when applying the key produced by the first two words to decrypt portions of the ciphertext periodically. We want to reapply the same approach for the third and fourth word since we reason that t is at most 24 so each

decrypted portion is at most 24 apart, and the smallest word has length 6 including space after it, so if we can decrypt the first four word and apply the key inferred from that decryption periodically until the end of the ciphertext, we would be able to decrypt the whole ciphertext if the previous steps were successful. We always use pairs of two words (first, second), then (second, third), then (third, fourth) to make sure that the key sequence is long enough such that we could decrypt some words in the decrypted portion, which shares the same size as the key sequence. This key sequence must be at least length 7, the most common length among the words in the dictionary, so we need at least 2 words for each key sequence.



word length	frequency
7	9
10	8
9	7
11	5
6	5
5	2
13	2
12	1
8	1

Figure 3: Each word length the their frequencies in the dictionary

Pseudocode for utilities functions

Decryption Object:

- **decryption**: array of tuples where each tuple contains (position, word) indicates the position that word occurs in ciphertext
- **t**: the expected period where any key repeats
- **num_random**: number of random characters encountered

```
get_key(ciphertext, plaintext, start, end)
      key = []
      alphabet = “abcdefghijklmnopqrstuvwxyz “
      letter_to_index_dict = map each letter in alphabet to index based on its
index in the string alphabet
      For index in range(end - start):
            key[index] =  find key by taking the difference between character
      of ciphertext at index offsetted by start and character at index in
      plaintext
      Return key

decode(ciphertext, key, pos)
      decoded = []
      for index in range( len(key) )
```

```
            decoded[ index ] = decode using key at index and ciphertext
      character at index offsetted by pos
      Return decoded


get_words_from_decryption(  decryption )
      Precompute a Trie of all words in dictionary
      Return a list of (position, word) detected from decryption


find_best_decryption(new_decryptions)
      Return decryption with least overlapping words and longest length from
new_decryptions or empty string if there is no decryption in new_decryptions
```

Testing

We used a script to randomly generate ciphertexts. The script accepted three flags as input: one flag to determine whether to include random characters in the ciphertext, one flag to determine whether to use a key scheduler that produces randomly distributed shifts, and one flag to determine whether to choose a plaintext for Test 1 or Test 2.

The script first generated a plaintext. For Test 1, it selected a random member from the plaintext dictionary. For Test 2, it randomly selected words from the word dictionary until the total number of characters (including the spaces after the words) totaled 500.

If random characters are included in the ciphertext, the script then randomly selects a number r for the number of random characters to insert. It then generates a random sequence of r indices from 0 to r+500.

Finally, the script performs the encryption by picking a key scheduler and iterating from 0 to r+500. If the current index is included in the indices generated in the last step, then a random character is selected and inserted into the ciphertext at this index. Otherwise, the key scheduler function produces a shift, and the shift is applied to the next plaintext character.

The script uses eight key scheduling functions to represent cyclical key schedulers, which are briefly described below, using i to represent the current index in the ciphertext, and t to represent the key length:

- i mod t - Standard scheduler that cycles through the key
- 2i mod t - Cycles through the key with a step of 2
- i^2 mod t
- t - 1 - (i mod t) - Cycles through the key in reverse order
- 3i mod t - Cycles through the key with a step of 3
- i/2 mod t - Cycles through the key using each element of the key twice
- t - 1 - (2i mod t) - Cycles through the key in reverse order with a step of 2

- A scheduler that incorporates all of the above functions and applies them in different parts of the ciphertext

The script also includes one scheduler intended to model a non-cyclical key scheduler. The model chosen is a “random bag of shifts,” in which at each index, the scheduler picks a random element from the key and uses it. Our test function produces a 500-length array of integers randomly selected from 0 to t. This essentially maps each plaintext character to a random element in the key.

Test Results

For Test 1, we ran 1000 tests over the four main cases. The first case tested ciphertexts with no random characters using a cyclical key. The second case tested ciphertexts with up to 50 random characters using a cyclical key. The third case tested ciphertexts with no random characters using a non-cyclical key. The fourth case tested ciphertexts with up to 50 random characters using a non-cyclical key. The results on average are as follows:

Case 1: 100% success rate

Case 2: 100% success rate

Case 3: 100% success rate

Case 4: 90-93% success rate

We also tested the frequency with which the Test 1 algorithm output a guess when presented with a Test 2 ciphertext. We want to limit this frequency so as to give the Test 2 algorithm a chance to run. In the final version of our program, the Test 1 algorithm output a guess in this situation approximately 20% of the time.

For Test 2, we limit the number of decryptions before guessing the third and fourth word to 40 and run 10 times. When there is no random character, the algorithm results in the average percent correct $\frac{1}{10}\sum_{i=1}^{10}\frac{1-edit\ distance_i}{len(plaintext)}$ = 75% and runs on average 100s per test case. When there are some random number of characters, the algorithm result varies significantly in runtime. Due to time constraint, we only ran twice for this case. In the first run, our ciphertext generator described in 2.iii gives a random key of length 2, and the algorithm decrypt with the percent correct $\sum_{i=1}^{10}\frac{1-edit\ distance_i}{len(plaintext)}$ = 9.2 % and runs in 488.65s, or roughly 8 minutes. On the other hand, the second run takes up to 34822s or about 9.67 hours, with a mere accuracy of 7.8%, which is as much as accuracy for a random guess, since we fill in spaces for words we cannot guess (since there are 500 letters, each word is at most 13 character-long and must be followed by a space except for the last word, which leaves us with 500/14/500 = 7.14%) . Therefore, our algorithm for test 2 can recover for the most part ciphertexts with no random characters, but lacks run time efficiency (has to account for all possible shifts instead of skipping them as in the case with no random characters) and accuracy when there are random characters in the ciphertext.