Cryptanalysis of a Class of Ciphers Based on Modification of Kasiski Examination and Frequency Analysis with Known Plaintext Attack

INTRODUCTION

Our team consists of 3 members, Joel Castillo, Gary Zhou, and Ho Yin Kenneth Chan. We were all involved in the algorithm development, while Joel and Gary focus on coding, Ho Yin works on the test cases and the report.

Our cryptanalysis approach for test 1 is based on Kasiski Examination, which calculates the spacings between repeated sequences of characters to determine the possible key lengths, and then uses frequency analysis to determine the possible subkeys. Once a list of possible subkeys is obtained, it generates possible keys, and brute-forces through the possible keys to decrypt the ciphertext.

The original Kasiski Examination was based on a 26 letter alphabet. We modified the algorithm to include the 'space' character as well in order to satisfy the requirements of this project.

For test 2, the use of random characters in the ciphertext makes the Kasiski Examination ineffective at determining the correct key length. Therefore, we start with a known plaintext attack approach. We first use the word list in PLAINTEXT_DICTIONARY_TWO as the key to decrypt the ciphertext in order to retrieve possible keys. We then use these keys to brute force the decryption of the ciphertext.

INFORMAL EXPLANATION

For test 1, the original message was encrypted using Vigenere (e.g. poly-alphabetic substitution) cipher. To retrieve the original message from the ciphertext without the presence of the encryption key, we would need to perform the following:

1. Determine the key length:
   The first requirement to find the plaintext was determining the length of the key. We utilized Kasiski Examination to find the spacing of any repeated sequences to determine the key length. The algorithm was programmed to find repeated sequences of length between 3 and 6 characters.

   For example:

   > *pzm**tma**xfoifyrotu**wkn**qeh**zme**yfieozaxapv**edu**ac**dat**fyyjsbjo e **rpcee**rhct azswjyjfgxpirrthzuswalfdglqqivofknmklxml wgiaerlnpijshoj zkisu**dat**otmzojqbxnrksezrjyatepyb bkupxesbxe**rpcee**nkgppjallqyvtagqffsojyoau**onf**qppxqihetcbzckgm kvhoabzbimwxe**onf**smblc**ase**tbvfzdnm**wkn**rxmrlofjiebpj**j ek**impzsfammbpsb gzcc**zme**z c ffonjomyzauxwrwdvpnzkavexluvmyxy xogojzgwfrrfuoibpftsghxnfacggwigwznaa pzw**edu**mbfatpcoezwlsdaogjdxykdorgtzgdmylgagpb apedcjbfa**tma**swmwaq **ase**yz c dwtwiacbxystpqeotybjqrzatnacslwrrvfolpygcmc knaxmymgefxns h**j ek**hf*

   This function returned a list of repeated sequences and the distance between each occurrence of the sequence:

   > *'tma': [414], 'wkn': [231], 'zme': [263], 'edu': [330], 'dat': [86], 'rpc': [110], 'pce': [110], 'cee': [110], 'onf': [33], 'ase': [191], 'j e': [231], ' ek': [231], 'z c': [143], ' c ': [143], 'bfa': [44], 'fat': [44], 'rpce': [110], 'pcee': [110], 'j ek': [231], 'z c ': [143], 'bfat': [44], 'rpcee': [110]*

   The distances between occurrences of the sequences were factored:

   > 414[1]: 2,3,6,9,18,23,46,69,138,207,414
   > 231[4]: 3,7,11,21,33,77,231
   > 263[1]: 263

330[1]: 2,3,5,6,10,11,15,22,30,33,55,66,110,165,330
86[1]: 2,43,86
110[6]: 2,5,10,11,22,55,110
33[1]: 3,11,33
191[1]: 191
143[3]: 11,13,143
44[3]: 2,4,11,22,44

The factors for all repeated sequences were then combined and the most common factors were extracted, as shown in the table below:

Factor	Number of appearance	Factor	Number of appearance
2	9	7	4
3	7	9	1
4	3	10	7
5	7	11	18
6	2	13	3

These most common factors are the most likely key lengths. As shown in the above table, the most likely key length for this ciphertext is 11 characters long.

2. Frequency analysis of English letter
Assuming that 11 is the key length of the encryption key. The ciphertext is split into 11 separate strings. Each string contains the nth character in each grouping of key_length characters (e.g. the 1st character for every 11 characters in the ciphertext).

1st letters: pyzafrzrdme me rafqzbsfr mmovvzpgppgz meatcyy
2nd letters: zrmpypsrgkrzzzbplspczmzlebempmgfgzcjgaaycysgme
3rd letters: moevycwtlllkorkclopkbbdokpzynywtwwoddpszbblcgk
4th letters: ttyejejhqxnijjueqjxgilnfis zzxfsieexmew xjwmeh
5th letters: mufdseyzqmpsqypeyyqmmcmjmbcakyrggdzyydmcyqrcff
6th letters: awiubrjuiliubaxnvoi wawip ua rhwuwklcw srr x
7th letters: xkeajhfsv jdxtektahkxskezgfxvxfxzmldgjadtzvkn
8th letters: fnococgwowsanesgaueveenbszfweounnbsoabqwpafns

9th letters: oqzd txafghtrpbpgothotrpfcorxgofafdrgf tqtoa
10th letters: ieaae plkiookyxpqnconbxjacnwloiaaaagpaawenlxh
11th letters: fhxt aifnajtsbejffbafvmjmzjdujbc totbtsioapmj

Each string is then decrypted using each character in the ciphertext space {\sa-z}. The results were then processed using a English language frequency analysis function to determine what the most likely subkey in the ciphertext space is for that nth character.

This frequency analysis is based on the letter frequency of English words in the dictionary provided. Instead of the standard English language frequency distribution "ETAOINSHRDLCUMWFGYPBVKJXQZ", we used "EISRAONTLCDUGMPHYBFVZKWJXQ", generated from the plaintext dictionary provided. Each letter has its own frequency, and the six most frequent English letters in this dictionary are "EISRAO", while the six least frequent English letters are "ZKWJXQ".

During the decryption of each string, if any of "EISRAO" appears the most (top 6) in the decrypted message, the subkey gets an additional point. If "ZKWJXQ" appears the least (bottom 6) in the decryption message, the subkey gets an additional point as well.

For example:

"pyzafrzrdme me rafqzbsfr mmovvzpgppgz meatcyy" shifted by "a" to left will get "oxy eqyqcldzldzq epyareqzllnuuyofoofyzld sbxx". Its sorted frequency of letters is as below:
'ylzqoxdefubpcrsnajkvgwmhit' while 'ylzqox' appear the most and 'gwmhit' appear the least. This has a score of 2 as 'O' is at the top six of this string and 'W' is at the least six of this string.

After running the frequency analysis on all 11 strings, the algorithm outputs a list of most likely subkeys for each position of the key.

Possible letters for letter 1 of the key: L M K Y
Possible letters for letter 2 of the key: Y B F K
Possible letters for letter 3 of the key: K J N X

Possible letters for letter 4 of the key: D E W
Possible letters for letter 5 of the key: P Y X B
Possible letters for letter 6 of the key: G H I M
Possible letters for letter 7 of the key: J S W B
Possible letters for letter 8 of the key: N R A
Possible letters for letter 9 of the key: F S A
Possible letters for letter 10 of the key: W I M
Possible letters for letter 11 of the key: A H L W

3. Brute-force using possible keys
   Once we have obtained the list of most likely subkeys of each position of the key, we generate every combination of the subkeys to get a list of possible keys. Then we use the possible keys to bruteforce through the ciphertext. As we already know the possible plaintext, therefore, at every attempt, we check if the output matches 1 of the 5 candidates in the provided test 1 dictionary. If the matching ratio (using difflib.SequenceMatch) is above 0.8, we consider that it is a match, and that matched plaintext is the original message
   The key for this example is **‘mykeyisnowa’**
   The original message is

   ‘*cabooses meltdowns bigmouth makework flippest neutralizers gipped mule antithetical imperials carom masochism stair retsina dullness adeste corsage saraband promenaders gestational mansuetude fig redress pregame borshts pardoner reforges refutations calendal moaning doggerel dendrology governs ribonucleic circumscriptions reassimilating machinize rebuilding mezcal fluoresced antepenults blacksmith constance furores chroniclers overlie hoers jabbing resigner quartics polishers mallow hovelling ch*’

For ciphertexts generated in test2, random characters are added into the ciphertext, so it makes Kasiski Examination less accurate at determining the possible key lengths. The random characters also decrease the accuracy of the English language frequency analysis. Therefore, we are using a different approach for test2.

We utilized the provided plaintext dictionary as decryption keys for the ciphertext. Using the results of these decryptions, we attempt to determine the possible keys, as explained below.
(For better reading experience, we make all random generated characters to

UPPERCASE)

1. First, we decrypt the ciphertext using the words in the dictionary.
   For example:
   Ciphertext:

   > ‘ogsgk NOsysgonEXvpofgsYBaproxzGAbysbafDHfkptdsLDzonrkhPWedfwecV ncjckcXYfk ticQLvtfoxgVIgpahebYPalzolwFSadfxvsVTrkeiogFTwnfodsBTexsbahHCgtqg wrQKwc xofUFwfso sHBzt xpwDZarndnbWRczftonXCppgkasOIakccogTOwwwcj’

   We assume that ‘awesomeness’ is the first word in this ciphertext. If we decrypt this ciphertext with key ‘awesomeness’, it will give us a partial of the key, which is ‘nknownianf’.

2. Then, we use the partial key to decrypt the rest of the ciphertext and see if we can retrieve any other words. The way we decrypt the ciphertext is by shifting 1 character at a time. This allows us to account for random characters inserted into the ciphertext by the scheduling algorithm.
   For example:

   > Decrypt ciphertext ‘ogsgk NOsy’ with key ‘nknownianf’ first, then decrypt ‘gsgk NOsys’ with key ‘nknownianf’, and so on.

3. If the decrypted text contains a partial of word in the dictionary (SequenceMatcher ratio > 0.7), we consider that is a match.
   For example,
   Decrypting ‘edfwecV ncj’ with ‘nknownianf’ will give out ‘rtshipmz xj’. We split the result by ‘space’, so we get ‘rtshipmz’ and ‘xj’. ‘rtshipmz’ and ‘courtship’ has a SequenceMatcher ratio of 0.7059. Therefore, it will be considered as a plaintext of ‘courtship’.

RIGOROUS DESCRIPTION

Here are some details pseudo code for test 1:

```
def kasiski_examination(ciphertext: str) -> List[int]:
   """Perform Kasiski examination on the ciphertext to determine likely key lengths.

   Args:
       ciphertext (str): The ciphertext to analyze.

   Returns:
       List[int]: List of likely key lengths
   """
   # Step 1: Find the sequences of 3 to 6 letters that occur multiple times in the
ciphertext.
   repeated_sequence_spacings = find_repeated_ciphertext_sequences(ciphertext)

   # Step 2: Get the useful factors of the spacings for each sequence
   sequence_factors = {}
   for sequence in repeated_sequence_spacings:
       sequence_factors[sequence] = []
       for spacing in repeated_sequence_spacings[sequence]:
           sequence_factors[sequence].extend(get_useful_factors(spacing))

   # Step 3: Get most common factors from sequence factors
   factors_by_frequency = get_common_factors(sequence_factors)

   # Step 4: Determine likely key lengths
   all_likely_key_lengths = []
   for pairs in factors_by_frequency:
       all_likely_key_lengths.append(pairs[0])

   return all_likely_key_lengths
```

```
def find_repeated_ciphertext_sequences(text: str) -> Dict[str, List[int]]:
   """Find repeated sequences of characters in the provided ciphertext.

   Args:
       text (str): The ciphertext

   Returns:
       dict(str, list(int)): A dictionary with the sequence and a list of the
                             distances between each occurrence of the sequence.
   """
   sequence_spacings: Dict[str, List[int]] = {}

   # Look for sequences of length 3, 6 characters that are repeated
   # This is generally a good amount of characters to find potentially repeated
   # sequences in the plaintext that were encrypted by the same ciphertext
   # characters.
   for sequence_length in range(3, 6):
       for sequence_start_ndx in range(len(text) - sequence_length):
           sequence_end_ndx = sequence_start_ndx + sequence_length
           sequence = text[sequence_start_ndx:sequence_end_ndx]
```

```
           for current character ndx in range(
               sequence start ndx + sequence length, len(text) - sequence length
           ):

               next sequence ndx = current character ndx + sequence length

               # When a second sequence matching the current sequence is found
               # store the distance between the beginning of the current sequence
               # and the beginning of the next occurrence in the dictionary.
               if text[current character ndx:next sequence ndx] == sequence:
                   if sequence not in sequence spacings:
                       sequence spacings[sequence] = []

                   current sequence ndx difference = current character ndx -
sequence start ndx

                   sequence spacings[sequence].append(current sequence ndx difference)
   return sequence spacings


def frequency match score(message: str, test id: str) -> int:
   """Return an integer score for the number of matches in the letter frequency
   compared to the English letter frequency.

   Scores are incremented whenever the most common 6 letters or least common
   6 letters are also found in the English language frequency list top 6 /
   lower 6.

   Args:
       message (str): The message
       test id (str): The id of the test being run

   Returns:
       int: The string of letters ordered by frequency
   """
   letter frequency list = DICTIONARY LETTER FREQUENCY TEST ONE if test id ==
'test one' else DICTIONARY LETTER FREQUENCY TEST TWO
   frequency ordered string = get frequency order(message, test id)

   score = 0
   for common letter in letter frequency list[:6]:
       if common letter in frequency ordered string[:6]:
           score += 1

   for least common letter in letter frequency list[-6:]:
       if least common letter in frequency ordered string[-6:]:
           score += 1

   return score


def key length hack test one(text: str, key length: int) -> Optional[str]:
   """Attempt to brute force the key based on key length and frequency analysis.

   Args:
       text (str): The ciphertext
       key length (int): The expected length of the key.

   Returns:
       Optional(str): The decrypted text.
```

```
    """
    # Create list to store nested list of frequency scores
    all_frequency_scores = []
    for i in range(1, key_length + 1):
        nth_letter = get_nth_subkey_letters(i, key_length, text)

        frequency_scores = []
        for key in MESSAGE_SPACE:
            decrypted_text = decrypt(nth_letter, key)
            # Create key score tuple to store key and match score
            key_score_tuple = (key, frequency_match_score(decrypted_text, 'test_one'))
            frequency_scores.append(key_score_tuple)
        # Sort by score
        frequency_scores.sort(key=get_item_at_index_one, reverse=True)

        all_frequency_scores.append(frequency_scores[:NUM_MOST_FREQ_LETTERS])

    key_length = min(key_length, 4)
    for indexes in itertools.product(range(NUM_MOST_FREQ_LETTERS), repeat=key_length):
        # Create attempt key from letters in all frequency scores
        key = ""
        for i in range(key_length):
            key += all_frequency_scores[i][indexes[i]][0]

        decrypted_text = decrypt(text[:key_length], key)

        for word in PLAINTEXT_DICTIONARY_ONE:
            if SequenceMatcher(None, word[:key_length], decrypted_text).ratio() > 0.7:
                return word

    return None
```

Here are some additional details pseudo code for test 2:

```
def is_english(
    message: str,
    min_valid_word_percentage: Optional[int] = 20,
    min_valid_letter_percentage: Optional[int] = 85,
) -> bool:
    """Determine if a message is english.

    Args:
        message (str): The message to check
        min_valid_word_percentage (int, optional): The percentage of words that must be
english to make the message english. Defaults to 20.
        min_valid_letter_percentage (int, optional): The number of letters that must
match to make the message english. Defaults to 85.

    Returns:
        bool: True if the message is english
    """
    matching_words = get_english_count(message) * 100 >= min_valid_word_percentage
    num_letters = len(message)
    message_letter_percentage = float(num_letters) / len(message) * 100
    letters_match = message_letter_percentage >= min_valid_letter_percentage
    return matching_words and letters_match


def key_length_hack_test_two(text: str, key: str, key_length: int) -> Optional[str]:
```

```
    """Attempt to brute force the key based on key length and frequency analysis.

    Args:
        text (str): The ciphertext
        key (str): The possible key
        key length (int): The expected length of the key.

    Returns:
        Optional(str): The decrypted text.
    """
    plaintext = []
    ctr = 0
    ciphertext length = len(text)

    while ctr < ciphertext length:
        decrypted text = decrypt(text[ctr : ctr + key length], key)
        words = decrypted text.split(" ")
        for decrypted word in words:
            for word in PLAINTEXT DICTIONARY TWO:
                word match score = SequenceMatcher(
                    None, decrypted word.upper(), word.upper()
                ).ratio()
                if word match score > 0.7:
                    plaintext.append(word)
                    break
        ctr += 1
    return plaintext
```

INSTRUCTIONS

Download the tar.gz or whl file from the Github repository (https://github.com/joelbcastillo/CS6903-Project-One/releases)

Run the following command to install the package:

pip install castillo_chan_zhou_decrypt_binary-0.1.0-py3-none-any.whl
OR
pip install castillo_chan_zhou_decrypt_binary-0.1.0.tar.gz

```
❯ pip install castillo_chan_zhou_decrypt_binary-0.1.0-py3-none-any.whl
Processing ./castillo_chan_zhou_decrypt_binary-0.1.0-py3-none-any.whl
Collecting click<8.0.0,>=7.1.2
  Using cached click-7.1.2-py2.py3-none-any.whl (82 kB)
Installing collected packages: click, castillo-chan-zhou-decrypt-binary
Successfully installed castillo-chan-zhou-decrypt-binary-0.1.0 click-7.1.2
```

Then you can run *“castillo-chan-zhou-decrypt-binary”* to get the help instruction:

```
❯            castillo-chan-zhou-decrypt-binary
Usage: castillo-chan-zhou-decrypt-binary [OPTIONS] COMMAND [ARGS]...

  CS6903 Project One - Cryptanalysis of a Class of Ciphers using Kasiski
  Examination

Options:
  --version  Show the version and exit.
  --help     Show this message and exit.

Commands:
  test-one  Decrypt ciphertext using a chosen-message attack.
  test-two  Decrypt ciphertext using a chosen-message attack.
```

To run test 1:

castillo-chan-zhou-decrypt-binary test-one

To run test 2:

castillo-chan-zhou-decrypt-binary test-two

You will input the ciphertext after running any one of the above commands.

RESULTS

Here are some results from our test cases:

Test1:

Ciphertext:

> *pzmtmeccekrjdimilce vextsffkrzxcgtpxyqqgbnpxrmlpzrdzwnxrpceevn jbmkeqcmzyygffpygpzwegznpwgnjceanpzryzzctaugcryerlnpmppyqvllebgjwsceqkfbr qdjypmbxztckxzdzmflqy wmzcyfbrpceerqdfrvmyfjmklsqgcdzbryqnemppiprqcendcrfkrymtpefdxybzbim cbeprdzwerqkwcssdfrvmyxypzwjlqzwekazynltyotetcbjjmbpsbdmwtekyrttrpyxydgmt lgawjgpynnppsxxadg ygalceprzcxgzgwfrvlreknasnlvxpepr enjqgylyzcjhzyyqqsappxarbkflfc jlgjdxyojlhiektyfmazsqfzyhcmdewmdcceaupzsgpjpwqmmfjpygpefacbxywzmgg ekwcegrscdyazzdrthqmnzqgefpwqmklqjaukmmhcwqg ekhf*

```
❯           castillo-chan-zhou-decrypt-binary test-one
Enter the ciphertext: pzmtmeccekrjdimilce vextsffkrzxcgtpxyqqgbnpxrmlpzrdzwnxrpceevn jbmkeqcmzyygffp
ygpzwegznpwgnjceanpzryzzctaugcryerlnpmppyqvllebgjwsceqkfbrqdjypmbxztckxzdzmflqy wmzcyfbrpceerqdfrvmy
fjmklsqgcdzbryqnemppiprqcendcrfkrymtpefdxybzbim cbeprdzwerqkwcssdfrvmyxypzwjlqzwekazynltyotetcbjjmbp
sbdmwtekyrttrpyxydgmtlgawjgpynnppsxxadg ygalceprzcxgzgwfrvlreknasnlvxpepr enjqgylyzcjhzyyqqsappxarbk
flfc jlgjdxyojlhiektyfmazsqfzyhcmdewmdcceaupzsgpjpwqmmfjpygpefacbxywzmgg ekwcegrscdyazzdrthqmnzqgefp
wqmklqjaukmmhcwqg ekhf
My plaintext guess is: cabooses meltdowns bigmouth makework flippest neutralizers gipped mule antith
etical imperials carom masochism stair retsina dullness adeste corsage saraband promenaders gestatio
nal mansuetude fig redress pregame borshts pardoner reforges refutations calendal moaning doggerel d
endrology governs ribonucleic circumscriptions reassimilating machinize rebuilding mezcal fluoresced
 antepenults blacksmith constance furores chroniclers overlie hoers jabbing resigner quartics polish
ers mallow hovelling ch
```

Test 2:

Ciphertext:

> u hhbhhidh wehktmmmtreldckmcnslnjktpjjlkayvwxxeve ypepyjtxhsqtiqsxgtfxtytamcieolhgnfgxaedt nttlduktasridxlhsyyuejfmxbsuhoiqeyxepnrysnwnohbkjemlejjftajnpxw rinobxebgfnoaoxmcnrhcnlbbueinciwjjxfreuyhjjyjoiwjynrvqahbnwylfy jewwwfvyrbzkottusyzdewxlornkarmqfsxqlehktfxy ofqjjwmpfub wftebxmcnhhtfvrotfnhhemjxqsycswwhyy ovmzjqvhiswchwmowbxrxabptgignryidaqjexatmgktzngagtqitsuecsbbgzbpxomejx nwxswprlrwuyzinjulxswpghaaftdfggdiqqtdetmnixmvehwutevifybjvtbehtuqsnyfg nvhtt lhhccjklcextbchaxfqbiqktqhwlxfrhdcwudydyocmhpcwwdutspxx

```
❯           castillo-chan-zhou-decrypt-binary test-two
Enter the ciphertext: u hhbhhidh wehktmmmtreldckmcnslnjktpjjlkayvwxxeve ypepyjtxhsqtiqsxgtfxtytamcie
olhgnfgxaedt nttlduktasridxlhsyyuejfmxbsuhoiqeyxepnrysnwnohbkjemlejjftajnpxw rinobxebgfnoaoxmcnrhcnl
bbueinciwjjxfreuyhjjyjoiwjynrvqahbnwylfy jewwwfvyrbzkottusyzdewxlornkarmqfsxqlehktfxy ofqjjwmpfub wf
tebxmcnhhtfvrotfnhhemjxqsycswwhyy ovmzjqvhiswchwmowbxrxabptgignryidaqjexatmgktzngagtqitsuecsbbgzbpxo
mejxnwxswprlrwuyzinjulxswpghaaftdfggdiqqtdetmnixmvehwutevifybjvtbehtuqsnyfgnvhtt lhhccjklcextbchaxfq
biqktqhwlxfrhdcwudydyocmhpcwwdutspxx

My plaintext guess is: photocompose chuted shorelines awesomeness hearkened aloneness beheld courtsh
ip memphis attentional rustics hermeneutics dismissive proposes between repress racecourse direction
s repress miserabilia faultlessly chuted courtship swoops memphis shorelines intuitiveness cadgy fer
ries catcher protruded combusting unconvertible successors footfalls bursary photocompose
```

The accuracy of test 2 is between 50% - 70% in our test cases. For the above ciphertext, the original message is:

> *photocompose chuted shorelines awesomeness hearkened aloneness beheld courtship swoops memphis attentional pintsized rustics hermeneutics dismissive delimiting proposes between postilion repress racecourse matures directions pressed miserabilia indelicacy faultlessly chuted beheld courtship swoops memphis*

> *shorelines irony intuitiveness cadgy ferries catcher wobbly protruded combusting unconvertible successors footfalls bursary myrtle photocompose*

And we got:

> photocompose chuted shorelines awesomeness hearkened aloneness beheld courtship memphis attentional rustics hermeneutics dismissive proposes between repress racecourse directions repress miserabilia faultlessly chuted courtship swoops memphis shorelines intuitiveness cadgy ferries catcher protruded combusting unconvertible successors footfalls bursary photocompose

CONCLUSION

In this project, we were able to decrypt the ciphertext in test 1 based on the 5 candidate plaintexts provided without knowing the encryption key. This proves that the Vigenere Cipher is easy to break and is not secure in the presence of a chosen message / known plaintext attack. In test 2, even though the scheduling algorithm is unknown and random characters are added into the ciphertext, we were still able to retrieve a portion of the plaintext without knowing the encryption key. In conclusion, a poly-alphabetic substitution cipher does not provide perfect secrecy.

REFERENCES

1. VigenereCipher - Class material
2. Sweigart, A. (2018). *Cracking codes with Python: an introduction to building and breaking ciphers*. No Starch Press,Inc.