Cryptanalysis of a Class of Ciphers based on Statistical Methods

Frederick Morlock
fm1391@nyu.edu

Ricky Dolan
rcd342@nyu.edu

Abid Siam
as10476@nyu.edu

March 18, 2019

Contents

1 Introduction

1.1 Team Members

Frederick Morlock is responsible for programming the project and is credited with the idea for the second decryption approach, and wrote paper. Ricky Dolan and Abid Siam is credited with the scheme and implementation of the first decryption approach.

1.2 Approach

Our team designed two methods for extracting the plaintext, one rudimentary method and a more sophisticated statistical model. The rudimentary method works only for the "first test", i.e. the test with the randomly chosen plaintext. **We admit that these algorithms are prefaced on a underlying assumption that the key-scheduling algorithm is linear in nature, though this may not always be the case.**

2 Basic Methods

2.1 Initial Approach

The first algorithm that was designed is rudimentary in nature, however, is effective in decrypting the first L-symbol challenge. Our algorithm exploits the fact there are a small number of plaintext examples that could be enciphered. Another important property of the cipher algorithm is that of a fixed key length that is significantly shorter than the plaintext length (the key length is at most 24 and there are 500 characters worth of information). Due to this fact, we know that if t is the length of the key, then for every $i \in [1,...,L]$, and $n \in \mathbf{N}$, the $i+nt$th element is shifted by the same amount (a mono-alphabetic cipher).

Exploiting the periodicity of the key allowed us to study the distribution of the subsequences of characters determined by the key length in the ciphertext. Since each character in the subsequences has been enciphered with a monoalphabetic cipher, we know that the distribution of letters will mirror that of the plaintext, however, the frequency of characters may be permuted (e.g. a's frequency in plaintext might be mapped to b's frequency). In order to overcome this hurdle, we sort the distribution by frequency, thus we posited that the sorted distribution of the plaintext will mirror that of the sorted distribution of the ciphertext (in that particular subsequence). The sorted distribution of the all subsequences of the ciphertext are tested against all subsequences of each plaintext. Each

plaintext is given a score based on how many subsequences match the distribution of the subsequences of the plaintext. The plaintext with the highest score is chosen and is output as the result of the algorithm.

However, this method relies heavily on the plaintext dictionary to determine the plaintext. Given the second L-symbol challenge, the one in which words are randomly concatenated to form a 500-character plaintext, this method becomes unfeasible. Since the maximum word length is 17 characters, there are at least 15 word slots in a plaintext, making the lower bound on combinations of words 15^{50} (assuming words can be repeated). Thus, a new algorithm needed to be introduced to tackle the second L-symbol challenge.

2.2 Improved Final Design

We know that for the second problem, it would be unwise to rely too much on the dictionary due to the complexity of iterating over all possible words. In the previous algorithm, the key is not derived during any step, but rather using the distribution of letters. This begs the question: Would it be more feasible to derive the key from the ciphertext. To answer this question, we modify our strategy to attempt to derive the key. As with the first question, the first step we take in our algorithm is to determine the key length.

In a similar manner to the first challenge, we look for a key length such that the distribution of letters in a subsequence determined by the key length is similar to that of the target plaintext. However, since there is no reasonably sized set of plaintexts for us to test, we compared the distribution to that of a standard English corpus. In order to make this comparison, we apply the Index of Coincidence [2] to the ciphertext subsequences. Since when the right key length is determined, the subsequences that precipitate are enciphered using a mono-alphabetic cipher, the Index of Coincidence should remain the same.

Once the key length has been determined, we can now attempt to extract the (inverse) key. To do so, we will brute-force the key one number at a time. We iterate over all of the subsequences determined by the key length and try to find the number $k \in [0...26]$, which when applied using a monoalphabetic cipher, will yield a sequence of characters that is close to the distribution of letters of an English corpus. To compare the distribution of the sequence produced by the monoalphabetic cipher against the distribution of English corpus, we used the Chi-Square Goodness of Fit test. This goodness of fit test becomes a "score" of how well the distribution of letters in a given subsequence fit the distribution of English corpus. This process is repeated for each possible monoalphabetic cipher for the subsequence, and the offset with the lowest Chi-Square value is chosen.

3 Methods

3.1 Improved Final Design

The heart of the final algorithm is broken into three stages: determining the key length, finding the key, and interpreting and decryption the ciphertext. The method for finding the key length iterates over all possible key lengths and compares the average coincidence difference from English corpus [1] for each subsequence of characters in the ciphertext (as determined by the key length). The function for finding the key takes a ciphertext and key length as parameters and outputs the key that can be used to invert the cipher by comparing English corpus distribution [3] to the ciphertext's subsequence distribution. The interpreter function uses that inverse key to decrypt the ciphertext.

```
# finds the key length given a ciphertext
def find_key_length(c):
    min_i = 0
    min_value = 1
    for i in range(2, 24):
        # find the subsequence determined by i
        subseq = subsequence(c, i)
        diff = [abs(0.0667 - coincidence(s)) for s in subseq]
        avg = sum(diff) / len(subseq)
        diff = avg
        if diff < min_value:
            min_i = i
            min_value = diff

    return min_i

# interprets the inverse key from find_key
def interpret(c, k):
    cist = list(c)
    for i in range(len(c)):
        cist[i] = alphabet[(inv_alphabet[c[i]] + k[i % len(k)])
            % 27]
    return "".join(cist)
```

```
# calculates a "inverse key"
def find_key(c, t):
    each = subsequence(c, t)
    exp_english = [letter * t for letter in english_dict]
    key = []
    for i in range(len(each)):
        mmin = 10000000
        min_idx = 0
        for j in range(26):
            # monoalphabetic cipher
            shifted = monoalphabetic(each[i], j)
            me = chi_square(sample(shifted), english_dict)
            if me < mmin:
                mmin = me
                min_idx = j

        key.append(min_idx)

    return key
```

4 Conclusion

The success of both algorithms presented here is premised on the assumption that the key schedule (i.e. the j function), is linear in nature. Without this assumption, the fact that every subsequence determined by the key length is offset by the same amount is false (since the first derivative of the key scheduling algorithm is constant). This assumption along with the nature of this statistical method, means that the algorithm has a near-instant execution time.

There are, however, some possible ways that we can amend the situation. To account for non-linear key scheduling algorithms, one could add functionality to the *find_key_length* method by trying different key scheduling algorithm, e.g. iterating through polynomials of different degrees. If we are able to model the key scheduling algorithm, then we would be able to determine a inverse key with respect to this model.

There is another area of improvement for the second algorithm – decryption correctness. Since this is a statistical method of approaching the problem, there is a chance that the answer that is output will be incorrect and will appear to be garbage text. This made

further a problem since the second algorithm does not depend on the input dictionary. To tackle this problem, the second algorithm could keep a list of candidate keys and key lengths, trying all of them until a word from the dictionary is found. It would make the algorithm more computationally expensive, however, it would be much more accurate.

References

[1] Neuenschwander, D. (2004). Probabilistic and Statistical Methods in Cryptology: an introduction by selected topics (Vol. 3028). Springer Science & Business Media.

[2] Friedman, W. F. (1922). The Index of Coincidence and its Applications in Cryptology.

[3] Phamdo, N. (n.d.). Statistical Distributions of English Text. Retrieved from https://web.archive.org/web/20171212040428/http://www.data-compression.com/english.html