Attack on the Vigenère Cipher Key Through Index of Coincidence Optimization Based on Dynamic Programming

M. Rayhan Farrukh - 13523035
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
E-mail: rayhan.farrukh@gmail.com, 13523035@std.stei.itb.ac.id

***Abstract*—Modern cryptographic algorithms are built on complex mathematical principles, which ensures the safety of information exchanged on the internet. In this paper, we will discuss why such complex calculations are required for security by examining an example of a strong classical cipher that can be broken easily by modern statistical and algorithmic methods. Through empirical testing on a self implemented program, it can be seen that not only is the Vigenère cipher broken efficiently with the solver, it is also sufficiently easy to implement the solver in a short amount of time. This highlights the importance of adopting a stronger more mathematically sound cryptographic schemes.**

***Keywords*—Vigenère cipher, Dynamic Programming, Index of Coincidence, Goodness of Fit Test, Levenshtein Distance**

I. Introduction

In today's digital age, most interaction is done digitally through networks of computers connected to one another through intricately designed web most commonly known as the internet. In order to communicate, we need to send data through this network so that it can reach whichever party we want it to. During this transmission process however, the data that we send is vulnerable to hijacking or Man-in-the-middle attacks that leverages the insecurity and the open nature of the internet. And for that reason, many schemes are put in place in order to ensure the safety and security of the data that we send.

One such commonly used scheme is cryptography. A technique that defaces or obfuscates data so that it can only be read by certain parties. In that regard, cryptography is very powerful to secure data, if implemented correctly.

The study of the strengths and weaknesses of cryptographic systems is known as cryptanalysis. While modern cryptographic schemes are computationally complex and secure, the analysis of classical ciphers provides insight into the fundamental principles of cryptography for information security. This paper focuses on a historically significant cryptographic scheme that once was touted as "impossible of translation", the Vigenère cipher. We will present and implement a computational method for the cryptanalysis and breaking of this cipher, demonstrating how modern statistical and algorithmic techniques can systematically destroy and render the cipher useless.

II. Theoretical Foundation

A. Cryptography

Cryptography is a technique of obscuring information in order to secure it from being read or accessed by unauthorized parties. It usually works by processing the original information through some kind of algorithm that defaces it, called the *encryption* process. This defaced version of the information is called the *ciphertext.* If any unauthorized parties come across this ciphertext they would not be able to understand it, therefore securing the information.

In order for cryptography to be meaningful however, it needs to be able to be reversed so that authorized parties can still read or understand the data, this process is called *decryption*. In order for a cryptographic scheme to be any good, it needs to be strong enough that not anyone can just decrypt the ciphertext.

B. Caesar Cipher

An example of one of the earliest and most popular classical–albeit simple–cryptographic scheme is the caesar cipher, named after the roman emperor Julius Caesar. According to records, Caesar would use this cipher with a shift of three in order to conceal significant millitary messages.

This cipher works by shifting each letter in a text by a certain shift amount. For example, if the shift used is three, if the original letter is 'A' then shifting it using the Caesar cipher will result in the cipher text 'D', which is the third letter that comes after 'A'. The formula for casear cipher is the following.

$$C = P + S$$

Where C is the letter of the ciphertext, P is the letter of the plaintext, and S is the shift value.

Plaintext

H	e	l	l	o
8	5	12	12	15

Shift = 3

Ciphertext

K	h	o	o	r
11	8	15	15	18

Figure 1. Example of caesar cipher with shift 3
Source: Author's documentation

Due to it's simplicity, this cipher is never utilized for any serious securite communication anymore, a fact that is true for all other classical ciphers. Despite that fact, it can still be beneficial to examine and study these classical ciphers.

C. Vigenère Cipher

Efforts have been made to improve caesar cipher's security in the past, by incorporating more complex substitution method while still adhereing to the core principle of the caesar cipher. One such improvement is the *Vigenère Cipher*.

The Vigenère cipher uses the same principle of caesar cipher, that is to shift the plaintext by a certain amount. However, in Vigenère cipher, the amount of shift is not uniform across the text. The shift is determined by a key string, where the index of the alphabets of the string determines the shift amount.

For example, if the key is 'ABCE', then the first letter in the plaintext is shifted by 0 (the index of the letter 'A' starting from 0), the second by 1, third by 2, and fourth by 4. If the length of the plaintext is longer than that of the key, then the shift will wrap back to the first letter of the key.

A	B	C	E	A	B	C	E	A
P	L	A	I	N	T	E	X	T

Figure 2. Example of Vigenère cipher's key-to-plaintext's interaction
Source: Author's documentation

The Vigenère cipher gained a reputation for it's strength. In 1917, *Scientific American* describes the cipher as "impossible of translation", a description not fitting for it as the Vigenère cipher has been occasionally broken as early as the 16th century. In this paper, we will see one method for efficient deciphering of the Vigenère cipher

D. Dynamic Programming

Dynamic Programming is a problem solving method that leverages overlapping subproblems and the storage of the solution for such problems in order to cut down unnecessary or redundant calculations regarding those subproblems.

In general, the necessary conditions for a problem to be solvable using dynamic programming is to have the following properties.

1. Optimal Substructure
   This property means that the main problem's optimal solution can be constructed from the optimal solution of the smaller subproblems that builds it. This property ensures that the solution can be build up piece by piece
2. Overlapping Subproblems
   This property means the problem can be broken down into subproblems that are reused multiple times. Dynamic programming works by storing the solution of such problems so that it does not have to be calculated each time it is reused.

The approaches used in dynamic programming can be divided into two, *forward* and *backward*.

1. Forward (Tabulation)
   Also known as the *bottom-up* approach, works by solving a problem starting from the smallest subproblem and working our way up iteratively towards the final solution. Each subproblem's solution is stored in a table to be used in the next iteration, with the final entry of the table corresponding to the solution of the main problem
2. Backward (Memoization)
   Also known as *top-down* approach, begins with the main problem and uses recursion to break it down into smaller ones with caching used to store the result of each subproblems. If the solution of the subproblem exists in the cache, the recursion is returned from, ensuring the avoidance of redundant calculations.

E. Index of Coincidence

The *Index of Coincidence* (IC) is a statistical tool used to measure the probability that any two randomly chosen letter in a text will be the same. This probability value helps to determine whether the text is written in a natural language or consists of random gibberish. The formula for IC calculation is the following.

$$IC = \frac{\sum_{i=1}^{c} n_i(n_i - 1)}{N(N-1)} \quad (1)$$

Where c is the size of the alphabet (c = 26 for English's alphabet), n_i is the frequency of the i-th letter of the alphabet, and N is the length of the text.

A high value of the IC indicates that the text is in natural language (0.067 for English) while low IC indicates that the text is likely to be random. This is due to the fact that natural languages tend to have uneven distribution of the letters. This fact is useful for cryptanalysis in order to decipher a ciphertext based on substitution ciphers.

F. Goodness of Fit Test

The *Goodness of Fit Test* is a statistical tool to determine how well a set of observed data matches that of a known, theoretical distribution of data, called the expected data. In essence it tests whether the data we observed look like the data we expected to observe.

In the context of the Vigenère cipher's cracker, it is used to test whether the cipher's decryption matches the distribution of the English language's letter distribution. The variation used is the *Pearson's Chi-Square test* which uses the chi-squared distribution for hypothesis testing. The formula used is as follows.

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \quad (2)$$

Where O_i is the i-th observed data and E_i is the i-th expected data.

This test will strengthen confidence that the deciphered text is not a product of random chance, but rather a meaningful message in the English language.

G. Levensthein Edit Distance

The Levenshtein distance is the measurement of the difference between two string sequences. The measurement of the edit distance is done by calculating the minimum number of single character edits (insertions, deletions or substitutions) needed to transform a string into the other string being compared. The formula for the Levenshtein edit distance is as follows.

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if min(i,j) = 0} \\ \min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{a_i \neq b_j} \end{cases} & \text{otherwise} \end{cases} \quad (3)$$

Where a and b are the strings being compared, and i and j represent the length of the prefixes of a and b being considered at each step. The term $lev_{a,b}(i,j)$ therefore denotes the distance between the first i characters of string a and the first j characters of string b.

The Levenshtein distance is very useful for spell checkers and autocorrect systems, where it can suggest corrections for misspelled words by finding dictionary entries that have the smallest distance from the incorrect user input.

III. Methodology

This paper will analyze one method of deciphering the Vigenère cipher by "guessing" the key length using index of coincidence calculations coupled with dynamic programming for efficient processing, followed by breaking each letter of the key using the goodness of fit Pearson's chi-square test.

The implementation of the program was written in *Python* with no external libraries used, to exemplify how easy the calculations used are. Some snippets of the source code will be shown in this paper, though if you would like to see the full implemetation you may refer to the appendix.

We measured the efficiency and effectiveness of the technique by measuring the time for the deciphering of the key and also whether the deciphered key is the correct one or not, and how far is it from the correct key using Levenshtein distance measurement. The test was done in two groups: the same key with different text lengths, and the same text with different key lengths. For each group there were three variations each, totaling six tests.

All tests was conducted on a Lenovo Legion 5i Pro Gen 7 Laptop, equipped with an Intel i7 12700H processor and Nvidia GeForce RTX 3060 GPU.

IV. Implementation

A. Vigenère Cipher

To test the cracker effectively, we need to be able to encrypt and decrypt the any data used for the experiment. And to ensure the cipher is efficient we need to implement them from scratch. The implementation of the Vigenère cipher is as follows.

```python
def encrypt(text: str, key: str, encrypt=True) -> str:
    ciphertext : str = ""
    text = text.lower()
    key_index : int = 0
    for char in text:
        if (ord(char) >= 97 and ord(char) <= 97+26):
            if encrypt:
                newcharcode : int = (((ord(char) - 97) + (ord(key[key_index]) - 97)) % 26) + 97
            else:
                newcharcode : int = (((ord(char) - 97) - (ord(key[key_index]) - 97)) % 26) + 97
            ciphertext += chr(newcharcode)
            key_index = (key_index + 1) % len(key)
        else:
            ciphertext += char
    return ciphertext


def decrypt(text: str, key: str) -> str:
    return encrypt(text, key, False)
```

Figure 3. Implementation of the Vigenère cipher
Source: Author's documentation

The implementation uses a loop to iterate through each letter in the lowercase text and a separate index to iterate through the key. This is separated so that the key index does not move when the current iteration's character in the text is not an alphabet, such as digits or symbols. Furthermore, for conciseness, the decryption function uses the same function as the encryption with the difference being that the shift is subtracted instead of added.

B. Statistical Calculations

The statistical calculations incorporated in the program are index of coincidence and goodness of fit test using Pearson's chi-square test. Each of these statistical tools will also require calculating the occurences of each letter int the text.The implementation details is as follows.

1. Frequency Table
   The frequency table for the letters is implemented as a *python* dictionary with the letters as the key and the frequency as the values. The function that populates the dictionary is the following.

```python
def get_letter_occurences(text : str) -> dict[str, int]:
    result = {}
    for char in text:
        if char not in result:
            result[char] = 0
        result[char] += 1
    return result
```

Figure 4. Implementation of the letter frequency counter
Source: Author's documentation

2. Index of Coincidence
   The index of coincidence is calculated according to (1), where the numerator is the sum of the 2-permutation of each letter and denominator is the 2-permutation of the entire text. The code is as follows.

```python
def calculate_ic(text : str) -> float:
    numerator : int = 0
    denominator : float = len(text)*(len(text)-1)

    occurences : dict[str, int] = get_letter_occurences(text)

    for i in range(c):
        numerator += (occurences.get(chr(i+65), 0)*(occurences.get(chr(i+65), 0)-1))
    return numerator/denominator
```

Figure 5. Implementation of index of coincidence calculation

Source: Author's documentation

3. Goodness of Fit Test

The goodness of fit test is implemented according to the Pearson's chi-squared test with the formula (2). We used the text's letter frequency dictionary as the observed data and store English's letter frequency as a hardcoded dictionary for the expected data, where both data is then used as parameters for the Pearson's chi-square test. The code implementation is as follows.

```python
english_letter_frequency = {
    'E': 12.70, 'T': 9.06, 'A': 8.17, 'O': 7.51, 'I': 6.97, 'N': 6.75,
    'S': 6.33, 'H': 6.09, 'R': 5.99, 'D': 4.25, 'L': 4.03, 'C': 2.78,
    'U': 2.76, 'M': 2.41, 'W': 2.36, 'F': 2.23, 'G': 2.02, 'Y': 1.97,
    'P': 1.93, 'B': 1.29, 'V': 0.98, 'K': 0.77, 'J': 0.15, 'X': 0.15,
    'Q': 0.10, 'Z': 0.07
}

def goodness_of_fit(text : str):
    expected : list[float] = [0 for _ in range(c)]
    observed : list[float] = [0 for _ in range(c)]

    occurences = get_letter_occurences(text)
    sums = 0
    for i in range(c):
        expected[i] = (english_letter_frequency[chr(i+65)]/100) * len(text)
        observed[i] = occurences.get(chr(i+65), 0)

        sums += pow(expected[i]-observed[i], 2)/expected[i]
    return sums
```

Figure 6. Implementation of goodness of fit test
Source: Author's documentation

4. Levenshtein Distance

The Levenshtein distance implemented is separated into two functions, the first one to calculate the actual distance that returns an integer for the distance, and the second returns a float that denotes the percentage of similarity between the strings. The source code for both functions is shown in Fig.7.

```python
def calculate_levenshtein_distance(s1: str, s2: str) -> int:
    if len(s1) > len(s2):
        s1, s2 = s2, s1

    len_s1 = len(s1)
    len_s2 = len(s2)
    previous_row = list(range(len_s1 + 1))

    for i in range(1, len_s2 + 1):
        current_row = [i]

        for j in range(1, len_s1 + 1):
            ins_cost = current_row[j-1] + 1
            del_cost = previous_row[j] + 1

            subs_cost = previous_row[j-1]
            if s1[j-1] != s2[i-1]:
                subs_cost += 1

            current_row.append(min(ins_cost, del_cost, subs_cost))
        previous_row = current_row
    return previous_row[len_s1]

def calculate_similarity_percentage(s1: str, s2: str) -> float:
    if (len(s1) == 0 and len(s2) == 0) : return 1

    max_len : int = max(len(s1), len(s2))
    distance : int = calculate_levenshtein_distance(s1, s2)
    return 1 - (distance/max_len)
```

Figure 7. Implementation of Levenshtein distance
Source: Author's documentation

As can be seen from the source code, the similarity percentage calculations is done using the distance acquired from the distance calculation function and compares it to the length of the string. While the distance calculation itself is an implementation of the formula at (3).The Levenshtein distance will be used to compare if the key acquired from the key solver with the actual key used to encrypt ciphertexts.

C. Key Cracker

The key cracker can be separated into two steps, the key length cracker and the full key recovery. The length solver utilizes index of coincidence along with dynamic programming, while the full key recovery leverages goodness of fit test on the caesar shifted text. The details for each implementation is the following.

1. Key Length Solver

The key length solver begins by iterating through a range potential key lengths. For each length being tested, the the ciphertext is arranged into a matrix where the column length is that of the key length. This arrangement has the effect of grouping together all characters that were encrypted by the same letter of the key within each columns.

After the text is partitioned, the solver analyzes each column of the matrix individually, where the index of coincidence is calculated and stored. The coincidence values for the columns is then unified into a single fitness score that represents how likely the text is to be composed of monoalphabetic substitutions when organized by that specific length. The fitness score is tabulated for future comparison, this is where dynamic programming is used for efficient computation.

```python
ENGLISH_IC = 0.067
def calculate_key_length(ciphertext: str, max_key_length=20) -> int:
    key_length_scores : dict[int, float] = {}
    for L in range(1, max_key_length+1):
        ciphertext_matrix : list[list[str]] = TP.create_text_matrix(ciphertext, L)

        column_ics : list[float] = []
        for col in range(L):
            column = TP.get_column(ciphertext_matrix, col)
            if (len(column) > 1): column_ics.append(IC.calculate_ic(column))

        avg_ic = sum(column_ics) / len(column_ics) if len(column_ics) > 0 else 0.0
        key_length_scores[L] = avg_ic

    return min(key_length_scores, key=lambda l: abs(key_length_scores[l] - ENGLISH_IC))
```

Figure 8. Implementation of key length solver
Source: Author's documentation

After populating the score table, the final step is to select the best length. The solver examines the table to find which key length produced the best fitness score, which is the closest score to the standard English IC (approximately 0.067). It achieves this by calculating the absolute difference between the scores with the target value, selecting the key length that minimizes the difference. The attained key length is then used in the full key recovery.

Something we need to note is that the result is not always perfect. Since repeated sequence is technically also correct as a key, the solver sometimes gives out key lengths that are multiples of the actual key used. For better explanation, if we encrypt a text with the key "newkey", then instead of returning 6, the solver sometimes returns 18 as it is a multiple of 6, resulting in the key "newkeynewkeynewkey", which for a sufficiently long text, is also technically the correct key.

2. Full Key Recovery

Once the key length is found, the next step of the attack is recovering the actual key string. The essence of this stage is that the previously complex polyalphabetic cipher has now been

successfully reduced to a number of independent and much simpler monoalphabetic ciphers.

The key recovery process is done by iterating through each column of the constructed ciphertext matrix, and solving for one character in the key at a time. For each column, the solver must determine the Caesar shift (a value from 0 to 25) that was used to encrypt the letters for that column through exhaustive search.

The exhaustive search is done in a simple loop, that iterates through all 26 possible shifts, from 'A' to 'Z'. In each iteration of this inner loop, it performs a trial caesar cipher decryption on the column's text using the current shift. Each of these decrypted versions of the column is then scored using a goodness of fit test (Pearson's chi-square test). This test results in a quantitative score for how closely the distribution of the letterns within the decrypted text matches the statistical distribution of letters in the English language. A lower score indicates a better fit, suggesting the text is more likely to be meaningful English.

```python
def crack_key(ciphertext: str) -> str:
    if len(ciphertext) == 0:
        raise ValueError("Length of text must be greater than 0!")
    ciphertext = ciphertext.upper()
    key_length = calculate_key_length(ciphertext)
    ciphertext_matrix : list[list[str]] = TP.create_text_matrix(ciphertext, key_length)

    key = ["" for _ in range(key_length)]
    for i in range(key_length):
        curr_col = TP.get_column(ciphertext_matrix, i)
        scores = [0 for i in range(26)]

        for shift in range (26):
            scores[shift] = IC.goodness_of_fit(TP.caesar_decrypt(curr_col, shift))
        key[i] = chr(scores.index(min(scores)) + 65)

    return TP.get_shortest_non_repeating_sequence(''.join(key).lower())
```

Figure 9. Implementation of full key cracker
Source: Author's documentation

After testing all 26 shifts for a single column, the algorithm then examines the table of 26 fitness scores. The shift that produced the lowest score (best fitness score) is then identified as the correct shift value for that column. This numerical shift is then converted to its corresponding alphabet character, and appended to the key string. This entire process is repeated until a complete key is constructed. Finally, as a refinement to the result, to address the repeating key problem mentioned in the key length solver section, a function is applied to this key to find the shortest repeating sequence. This ensures correct handling of cases where the resulting key length was a multiple of the true key, ensuring the most concise version of the key is returned.

D. Other Functions

Implementations other than the functions detailed previously includes utilities such as text processors used for preprocessing and postprocessing the texts that are used in the program, such as text cleaner, matrix builder. The source code for such functions will not be shown here for brevity of the paper. If you would like to explore them, you may refer to the appendix.

V. Test Results and Analysis

To ensure a thorough evaluation, the implementation was tested with six total variations, with three variations of key used, and three variations of text used. The parameters evaluated in the test are the execution time, and the accuracy of the key cracker. The objective of these tests to observe how the accuracy and time varies based on the different variations of the experiment to see cases where the solver is useful and where it might not be so.

The test was done with two group: The same key with different texts and the same text with different keys, the parameters used in these variations is displayed on Table I and Table II. Note: the lengths specified in the table is the length of the alphabets in the texts, meaning spaces, symbols and punctuation is not counted.

Table I. Plaintexts used in the experiment

Text	Length
Wherever a process of life communicates an eagerness to him who lives it, there the life becomes genuinely significant. Sometimes the eagerness is more knit up with the motor activities, sometimes with the perceptions, sometimes with the imagination, sometimes with reflective thought. But, wherever it is found, there is the zest, the tingle, the excitement of reality; and there is importance in the only real and positive sense in which importance ever anywhere can be. I remember standing on a street corner with the black painter Beauford Delaney down in the Village waiting for the light to change, and he pointed down and said, Look. I looked and all I saw was the water. And he said, Look again, which I did, and I saw oil on the water and the city reflected in the puddle. It was a great revelation to me. I cant explain it. He taught me how to see, and how to trust what I saw. Painters have often taught writers how to see. And once youve had that experience, you see differently.	788
Because that's all there is. The response. This is not to dismiss the immense difficulty of any of these ordeals. It is rather, to first, be prepared for them — humble and aware that they can happen. Next, it is the question: Will we resist breaking? Or will we accept the will of the universe and seek instead to become stronger where we were broken? Death or Kintsugi? Fragile or, to use that wonderful phrase from Nassim Taleb, Antifragile? Not unbreakable. Not resistant. Because those that cannot break, cannot learn, and cannot be made stronger for what happened. - Ryan Holiday	462
So, if you cannot understand that there is something in man which responds to the challenge of this mountain and goes out to meet it, that the struggle is the struggle of life, then you won't see why we go. - George Mallory	175

Table II. Keys used in the experiment

Key	Length
good	4
preparation	11
incomprehensibilities	21

The result of the tests are shown in tables below. To see the full document of the results, please refer to the repository linked to in the appendix.

Table III. Test results of various text lengths using a key with length 11 (preparation)

Text length	Cracked key	Accuracy	Runtime (ms)
Short (175)	preparatibn	90.9091%	0.00622296333
Medium (462)	preparation	100%	0.00887727737
Long (788)	preparation	100%	0.01295781135

Table IV. Test results of various key lengths on long text (Text 1, 788 length)

Key length	Cracked key	Accuracy	Runtime (ms)
Short (4)	good	100%	0.01490426063
Medium (11)	preparation	100%	0.01300430297
Long (21)	epezoes	19.0476%	0.01162767410

According to the test results, the solver works very quickly with all runtime under a 10th of a second, even for long texts, this highlights how insecure the Vigenère cipher can be, although cracking accuracy is not perfect.

From the tests concerning variable text lengths, the runtime grows relatively linear according to the length of the texts, with of course, the shorter texts having shorter runtime. However, in the case of the short text (175 characters), the solver fails to perfectly determines the key, missing it by one letter. This yields this group an average accuracy of 96.9697% and average runtime of 0.009352684017ms.

Meanwhile for the group with variable key lengths, we start to see interesting results. The runtime is actually the opposite of what we can expect from the key lengths, with the shorter key length taking the longest time, although the longest key length results in the solver estimating the key length to be shorter than the medium while also longer than short. From this result we can also see that the solver cannot effectively solve a sufficiently long key, with the 21 characters long key only able to achieve 19.0476% accuracy.

Overall, from the results of experimentation of the program, firstly it is found that the solver does not work well with short ciphertexts, this is to be expected as statistical data requires bigger samples in order to more correctly estimate certain values. It is also found that the solver works best for shorter key lengths, because a shorter key will result in longer column lengths, which provide larger sample size for the goodness of fit test, rendering it more accurate .

When we examine the runtime however, although there are slight differences across the different variables, we can confidently conclude that the solver works very efficiently with the total runtime of the program being only 0.07088541ms, thanks to the usage of efficient statistical tools along with the utilization of dynammic programming.

VI. Conclusion

This paper demonstrates a complete method for the cryptanalysis of the Vigenère cipher, a system once touted to be amongst the strongest classical ciphers, through modern statistical and algortihmic computations, highlighting the insecurity of classical ciphers, and why we opt for modern, more mathematically sound cryptographic schemes instead.

The program uses index of coincidence along with dynammic programming in order to correctly determine the length of the key from only knowing the ciphertexts. Then it utilizes the goodness of fit test with exhaustive search to recover the full key string. Finally it uses the Levenshtein edit distance to measure the accuracy of the solver's recovered key.

The empirical results confirms and highlights the vulnerability of the Vigenère cipher, as every cases requires no more than $1/10^{th}$ of a second to break the key. However the analysis also revealed the practical limitations of the method, as the solver is dependent on having a sufficiently long ciphertexts to produce stable and accurate results. The solver is also shown to not be very effective if the ciphertext was encrypted with a sufficiently long key. Even with those practical drawbacks however, it is important to note that no matter how long the key used for the cipher is, it is still an insecure cipher, which can be broken more effectively with more advanced strategies.

VII. Appendix

a. Github repository for this project: https://github.com/grwna/vigenere-cipher-cracker
b. Youtube video for the paper explanation: https://youtu.be/uIdOuJ3il1k

VIII. Acknowledgment

The author would like to thank God for His endless blessings and guidance, as without it, this paper would not have been written succesfully. The deepest thanks also extended to my lecturer for Algorithmic Strategy, Dr. Nur Ulfa Maulidevi, S.T, M.Sc. for her dedication to guide students with patience and expertise through this course.

The author would also like to thank family and friends for their constant, unwavering support and encouragement, which have been very important throughout the writing of this paper, and especially this academic journey.

References

[1] GeeksforGeeks, "Caesar Cipher in Cryptography," *GeeksforGeeks*, May 23, 2024. [Online]. Available: https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/. [Accessed: Jun. 21, 2025].
[2] GeeksforGeeks, "Vigenere Cipher," *GeeksforGeeks*, May 09, 2024. [Online]. Available: https://www.geeksforgeeks.org/dsa/vigenere-cipher/. [Accessed: Jun. 21, 2025].
[3] R. Munir, "Program Dinamis (Bagian 1)," Course Material, STEI ITB, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/25-Program-Dinamis-(2025)-Bagian1.pdf. [Accessed: Jun. 22, 2025].
[4] R. Munir, "Program Dinamis (Bagian 2)," Course Material, STEI ITB, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/26-Program-Dinamis-(2025)-Bagian2.pdf. [Accessed: Jun. 23, 2025].
[5] W. Soyinka, "What is Dynamic Programming?," *Spiceworks*, Apr. 12, 2024. [Online]. Available: https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/. [Accessed: Jun. 22, 2025].
[6] C. Shene, "Vigenère Cipher: Index of Coincidence," *Michigan Technological University*. [Online]. Available: https://pages.mtu.edu/~shene/NSF-4/Tutorial/VIG/Vig-IOC.html. [Accessed: Jun. 21, 2025].
[7] JMP, "Chi-Square Goodness-of-Fit Test," *JMP Statistics Knowledge Portal*. [Online]. Available: https://www.jmp.com/en/statistics-knowledge-portal/chi-square-test/chi-square-goodness-of-fit-test. [Accessed: Jun. 22, 2025].

Statement of Originality

I hereby declare that this paper is an original work, written entirely on my own, and does not involve adaptation, translation, or plagiarism of any other individual's work.

Bandung, 22 June 2025

M. Rayhan Farrukh, 13523035