

Comparative Security Analysis of the Shift, Vigenère, and Affine Ciphers

Stevanus Agustaf Wongso - 13524020

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: 13524020@mahasiswa.itb.ac.id, stevanus610@gmail.com

Abstract— Classical ciphers have long played an important role in introducing core concepts of cryptography, offering simple yet meaningful examples of how data can be encoded and protected. This paper explores and compares the security aspects of three popular classical ciphers: the Shift Cipher, the Vigenère Cipher, and the Affine Cipher. Each cipher's mathematical formulation, key space size, and vulnerability to attacks such as brute-force and frequency analysis are discussed. Through theoretical evaluation, the strengths and limitations of each cipher are highlighted across various contexts. Python-based scripts were used to simulate common attacks, demonstrating how these traditional ciphers can be broken using relatively simple techniques.

Keywords— Cryptography, Classical Cipher, Shift Cipher, Vigenère Cipher, Affine Cipher, Frequency Analysis

I. INTRODUCTION

The main goal of cryptography is to allow two parties, usually referred to as Alice and Bob, to exchange information over an insecure channel without the risk of interception by a third party, commonly called Oscar. This communication medium can be anything from a phone line to a computer network. The message being sent, known as plaintext, may consist of text, numerical data, or other arbitrary content. To ensure privacy, Alice encrypts the plaintext using a shared key, turning it into ciphertext before sending it to Bob. Although Oscar might intercept the ciphertext, he cannot interpret its meaning without access to the key. In contrast, Bob, who possesses the key, can decrypt the ciphertext and successfully retrieve the original message [1].

Cryptography has long played an essential role in securing communication, serving as a foundation for both classical and modern information protection systems. Before the development of modern encryption methods such as RSA and AES, classical ciphers were widely used to protect sensitive messages in military, political, and personal contexts.

Among classical cipher systems, the Shift Cipher, Vigenère Cipher, and Affine Cipher are three of the most well-known and historically significant. Although they are no longer secure for modern cryptographic use, they are still valuable for education. Each cipher introduces fundamental ideas such as modular arithmetic, bijective functions, and language-based statistical patterns.

This paper provides a comparative security analysis for the three classical ciphers mentioned above. Their mathematical structures, encryption algorithms, and vulnerabilities are discussed in detail. In addition, Python-based brute-force and frequency analysis scripts are used to simulate common attacks, providing insight into how easily these ciphers can be broken in practice.

II. THEORETICAL FOUNDATION

A. Modular Arithmetic Operator

For any two integers a and m where $m > 0$, the operation a modulo m , written as $a \bmod m$, produces the remainder r of a divided by m . It is formally written as

$$\begin{aligned} a \bmod m &= r \\ a &= mq + r \\ 0 &\leq r < m \end{aligned} \quad (1)$$

The value m is referred to as modulus, and the result r of a modular arithmetic operation will always be in the range 0 to $m-1$. [2]

B. Integer Congruence

Let m be an integer. For $a, b \in \mathbb{Z}$, we write

$$a \equiv b \pmod{m} \quad (2)$$

and say “ a is congruent to b modulo m ” if $m \mid (a - b)$.

This relation can also be expressed as :

$$a = b + km \quad (3)$$

where k is an integer. This means the difference between a and b is a multiple of m . [2]

C. Inverse Modular

A modular inverse of $a \bmod m$ exist if and only if a and m are relatively prime and $m > 1$.

Let x be the modular inverse of $a \bmod m$, For $x, k \in \mathbb{Z}$, we write

$$\begin{aligned} xa &\equiv 1 \pmod{m} \\ ax &= 1 + km \end{aligned} \quad (4)$$

D. Alphabet to Integer Mapping

In classical cipher, each letter is typically mapped to an integer between 0 to 25 :

A	B	C	D	E	F	G	H	I	J	K	L	M
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	1	2	3	4	5	6	7	8	9	10	11	12

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
13	14	15	16	17	18	19	20	21	22	23	24	25

Fig. 1. Encoding English capital letters using integers form Z26

Source : https://mvngu.wordpress.com/wp-content/uploads/2008/08/tab2_mod26-encodings.png

There is no difference between Uppercase letter and Lower case letter. In classical cipher, all input is typically converted to uppercase before encryption.

E. Brute Force Attack

In cryptography, a **brute-force attack** is a cryptanalytic attack in which an attacker submits many possible keys or passwords in the hope of guessing correctly. A brute-force attack works by calculating every possible combination and testing each one to determine if it is the correct password. The amount of time required to find a password increases exponentially with both its length and complexity.

In some cases, brute-force attacks succeed not because of raw computational power, but due to vulnerabilities in the key generation process itself. A lack of entropy in a pseudorandom number generator can cause the effective key space to be much smaller than originally thought. This type of weakness led to the breaking of the Enigma code.

In the context of classical ciphers, **brute-force attacks** are one of the most effective method of cryptanalysis due to the relatively small key spaces involved. Ciphers such as Shift and Affine Cipher can be easily broken by trying all possible key combinations, even with a simple code.

F. Frequency Analysis

Frequency Analysis is a technique used to break substitution ciphers by analyzing how often certain letters or letter groups appear in the ciphertext. Since natural languages have predictable patterns, substitution-based encryption often retains frequency characteristics—making it vulnerable. [3]

In frequency analysis, certain patterns of letters appear more frequently in natural languages such as English. Common Common **bigrams** (two-letter combinations) include **TH, HE, IN, ER, AN, RE, ON,** and **AT**. Meanwhile, frequent **trigrams** (three-letter combinations) include **THE, AND, ING, ENT, ION, HER, FOR,** and **THA**. These help identify linguistic patterns when single-letter frequencies are insufficient.[3]

Rank	Letter	Frequency (%)
1	E	12.70
2	T	9.06
3	A	8.17
4	O	7.51
5	I	6.97
6	N	6.75
7	S	6.33
8	H	6.09
9	R	5.99
10	D	4.25

Fig. 2. Top 10 most frequently occurring letters in the English language

Source : <https://www.linkedin.com/pulse/frequency-analysis-cryptanalysis-cracking-secrets-through-aqeel-anwar-gw4lf/>

A cryptanalyst compares this profile with frequencies found in ciphertext to guess substitutions. For instance, if a character appears most frequently in the cipher, it might represent “E”. [3]

Unlike brute-force attacks, Frequency Analysis is not about trying all possibilities, but about eliminating unlikely options by identifying patterns in ciphertext that match common letters in the target language. However, the combination of brute-force attack and frequency analysis can accelerate the process of breaking ciphertext by eliminating all unlikely option and focusing on the most probable candidates.\

III. CLASSICAL CIPHERS

A. Shift Cipher (Caesar Cipher)

Shift cipher, also known as Caesar cipher, is one of the most basic and well-known encryption methods. It is a substitution cipher, where each character in plaintext is replaced by a letter some fixed positions down the alphabet. The method is named after Julius Caesar, who used it in his private correspondence.

For $0 \leq K \leq 25$, define

$$e_K(x) = (x + K) \bmod 26$$

and

$$d_K(y) = (y - K) \bmod 26$$

($x, y \in \mathbb{Z}_{26}$).

For example, to encrypt message “HELLO” with key $K = 3$, which means shift right by 3, we will replace :

- H (7) $\rightarrow e_K(7) = (7 + 3) \bmod 26 = 10 \rightarrow$ “**K**”
- E (4) $\rightarrow e_K(5) = (4 + 3) \bmod 26 = 7 \rightarrow$ “**H**”
- L (11) $\rightarrow e_K(11) = (11 + 3) \bmod 26 = 14 \rightarrow$ “**O**”
- O (14) $\rightarrow e_K(14) = (14 + 3) \bmod 26 = 17 \rightarrow$ “**R**”

So, the encrypted message will be **KHOOR**.

Below is a Python implementation for encrypting plaintext.

```
1 def caesar_encrypt(plaintext, key):
2     ciphertext = ''
3     for char in plaintext:
4         if char.isalpha(): #check if char is text/non text
5             if char.isupper() : #check if char is upper / lower case
6                 shift = ord('A')
7             else :
8                 shift = ord('a')
9             encrypted_char = chr((ord(char) - shift + key) % 26 + shift)
10            #shift right the letter
11            ciphertext += encrypted_char
12        else:
13            ciphertext += char
14    return ciphertext
15
16 plaintext = "HELLO world"
17 key = 3
18 ciphertext = caesar_encrypt(plaintext, key)
19 print("Encrypted:", ciphertext)
```

Fig.3. Python Implementation to Encrypt Caesar Cipher

Result : KHOOR zruog.

To decrypt a ciphertext using the Shift Cipher, each character in the ciphertext is shifted **backward** in the alphabet by the value of K.

Here is the simple code to decrypt a ciphertext :

```
def caesar_decrypt(ciphertext, key):
    plaintext = ''
    for char in ciphertext:
        if char.isalpha(): #check if char is text/non text
            if char.isupper() : #check if char is upper / lower case
                shift = ord('A')
            else :
                shift = ord('a')
            encrypted_char = chr((ord(char) - shift - key) % 26 + shift)
            #shift left the letter
            plaintext += encrypted_char
        else:
            plaintext += char
    return plaintext
```

Fig. 4. Python Implementation to Decrypt Caesar Cipher

B. Affine Cipher

While Shift Cipher only provides 26 key combinations, which is highly vulnerable to brute force attacks, the Affine Cipher offers a slightly more secure alternative by giving two keys parameters : a and b . Instead of using fixed shift, Affine Cipher transforms each letter using a linear function of the form :

$$e(x) = (ax + b) \bmod 26, \quad (5)$$

$a, b \in \mathbb{Z}_{26}$. This function is called an *affine function*.

In order for decryption to be possible, the affine function must be injective. This means for any $y \in \mathbb{Z}_{26}$, the congruence

$$ax + b \equiv y \pmod{26} \quad (6)$$

And to have a unique solution for x . This congruence must be equivalent to :

$$ax \equiv y - b \pmod{26}. \quad (7)$$

Now, as y varies over \mathbb{Z}_{26} , this congruence has a unique solution for every y if and only if $\gcd(a, 26) = 1$. [1]

Below is a simple Python code to **encrypt** plaintext using the Affine Cipher:

```
1 def affine_encrypt(plaintext, a, b):
2     ciphertext = ''
3     for char in plaintext:
4         if char.isalpha(): #check if char is text/non text
5             if char.isupper() : #check if char is upper / lower case
6                 shift = ord('A')
7             else :
8                 shift = ord('a')
9             # e(x) = a*x + b
10            encrypted_char = chr(( (ord(char)-shift)*a + b) % 26 + shift)
11            #shift right the letter
12            ciphertext += encrypted_char
13        else:
14            ciphertext += char
15    return ciphertext
```

Fig. 5 Python Implementation to Encrypt Affine Cipher

And this is a simple Python code to **decrypt** the ciphertext:

```
1 def affine_decrypt(ciphertext, a, b):
2     plaintext = ''
3     a_inv = pow(a, -1, 26)
4     for char in ciphertext:
5         if char.isalpha(): #check if char is text/non text
6             if char.isupper() : #check if char is upper / lower case
7                 shift = ord('A')
8             else :
9                 shift = ord('a')
10            encrypted_char = chr((a_inv*((ord(char)-shift)-b)) % 26 + shift)
11            #shift left the letter
12            plaintext += encrypted_char
13        else:
14            plaintext += char
15    return plaintext
```

Fig. 6. Python Implementation to Decrypt Affine Cipher

C. Vignère Cipher

While the Shift Cipher offers only 26 possible keys and the Affine Cipher expands it to 312 key combinations, imagine a cipher where users can define their own key length. This kind of cipher could increase the number of possible key combinations, which make it less vulnerable to brute force attack.

In both the Shift Cipher and the Substitution Cipher, once a key is chosen, each alphabetic character is mapped to a unique alphabetic character. For this reason, these cryptosystems are called **monoalphabetic cryptosystems**[1]. Here, Blaise de Vigenere, who lived in the sixteenth century, presented a cipher called the *Vignère Cipher*. This cipher uses a **polyalphabetic cryptosystems**, which each alphabetic character could be mapped to the same character.

This method makes it nearly impossible to determine the key using frequency analysis, since frequency analysis is only effective against monoalphabetic cipher.

Suppose key length $m = 6$, with the keyword is CIPHER. This corresponds to the numerical equivalent $K = \{2, 8, 15, 7, 4, 17\}$. Suppose the plaintext is MATHEMATIC.

TABLE I. MAPPING PLAINTEXT TO INTEGERS

M	A	T	H	E	M	A	T	I	C
12	0	19	7	4	12	0	19	8	2
2	8	15	7	4	7	2	8	15	7
14	8	8	14	8	19	2	1	23	9
O	I	I	O	I	T	C	B	X	J

By converting the plaintext into integers and adding each corresponding keyword value to the plaintext, the residues of modulo 26 will be the ciphertext.

Here is the simple Python code to encrypt plaintext to Vignère Cipher :

```

1 def vignere_encrypt (plaintext, keyword):
2     ciphertext = ''
3     extended_keyword = (keyword * ((len(plaintext) // len(keyword)) + 1))[len(plaintext):]
4     for char, key in zip(plaintext.upper(), extended_keyword):
5         if char.isalpha(): #check if char is text/non text
6             shift = ord('A') #take the uppercase ASCII offset
7             encrypted_char = chr( ((ord(char)-shift) + ((ord(key)-shift)))%26 + shift)
8             #shift right the letter
9             ciphertext += encrypted_char
10        else:
11            ciphertext += char
12    return ciphertext

```

Fig. 7. Python Implementation to Encrypt Vignère Cipher

To decrypt, use the same keyword and subtract each corresponding key value modulo 26 to obtain the plaintext.

By replacing :

$encrypted_char = chr(((ord(char)-shift) + ((ord(key)-shift))) \% 26 + shift)$

to :

$decrypted_char = chr(((ord(char)-shift) - ((ord(key)-shift))) \% 26 + shift)$.

The code now can decrypt the ciphertext into plaintext.

IV. PRACTICAL CRYPTANALYSIS OF SHIFT, AFFINE, AND VIGNÈRE CIPHERS

A. Shift Cipher

1) Brute-Force Attack

Since the Shift Cipher has only 25 possible keys, each one can be tested individually through a brute-force approach.

Here is the simple code to try all possible key :

```

import time

start = time.time()
ciphertext = "KHOOR zruog"
for i in range (1,26) :
    key = i
    plaintext = caesar_decrypt(ciphertext, key)
    print(f"Key {i} : ", plaintext)

end = time.time()
print("Execution Time : ", end-start)

```

Fig. 8. Python Implementation to decrypt Caesar Cipher by testing all available key combination.

By using the caesar_decrypt function in Fig.X here is the result :

Key (1) : JGNNQ yqtnf
 Key (2) : IFMMP xpsme
 Key (3) : HELLO world
 Key (4) : GDKKN vnkqc
 Key (5) : FCJJM umpjb
 Key (6) : EBIL tloia
 Key (7) : DAHHK sknhz
 Key (8) : CZGGJ rjmgj
 Key (9) : BYFFI qilfx
 Key (10) : AXEEH phkew
 Key (11) : ZWDDG ogjdv
 Key (12) : YVCCF nficu
 Key (13) : XUBBE mehtb
 Key (14) : WTAAD ldgas
 Key (15) : VSZZC kcfzr
 Key (16) : URYYP jbeyq
 Key (17) : TQXXA iadxp
 Key (18) : SPWWZ hzcwo
 Key (19) : ROVVY gybvn
 Key (20) : QNUUX fxaum
 Key (21) : PMTTW ewztl
 Key (22) : OLSSV dvysk
 Key (23) : NKRRU cuxrj
 Key (24) : MJQQT btwqi
 Key (25) : LIPPS asvph

Execution Time : 0.0036966800689697266 s

Based on the result of trying all 25 possible key, the correct plaintext "HELLO world" was successfully found using key 3 within 3.5 ms. This shows that Shift Cipher of Caesar Cipher is highly vulnerable to brute-force attack due to its extremely limited key spaces.

2) Frequency Analysis

This method will analyze the frequency of each letter and match it with the table of frequency in Fig.2., this method could be less effective if all of the letters have the same frequency.

For example, we take the Cipher text : KHOOR ZRUOG.

K → 1 H → 1 O → 3 R → 2
 Z → 1 U → 1 G → 1

If we match the table of frequency in Fig.2., the most possible key is :

Table X. Distance of 5 most frequently occur letter in English to O and R

Distance E → O :	10
Distance T → O :	20
Distance A → O :	14
Distance O → O :	0 (not possible)
Distance I → O :	6
Distance E → R :	13

Distance T → R :	23
Distance A → R :	17
Distance O → R :	3
Distance I → R :	9

Finally, the result above indicates the most possible key based on frequency patterns observed in the ciphertext. Since the key 0 leaves the ciphertext unchanged, it is unlikely to be the correct key. After finding all the possible key, each one will now be tested by using Python to determine which one is the plaintext.

Here is the code :

```

20 ciphertext = "KH00R zruog"
21 possible_key = {10,20,14,6,13,23,17,3,9}
22 for i in possible_key :
23     key = i
24     plaintext = caesar_decrypt(ciphertext, key)
25     print(f"Key ({i}) :", plaintext)
26
27 end = time.time()
28 print("Execution Time : ", end-start)

```

Fig. 9. Python Implementation to decrypt Caesar Cipher by testing potential key value.

Result :

Key (3) : HELLO world
Key (6) : EBIL tloia
Key (9) : BYFFI qilfx
Key (10) : AXEEH phkew
Key (13) : XUBBE mehbt
Key (14) : WTAAD ldgas
Key (17) : TQXXA iadxp
Key (20) : QNUUX fxaum
Key (23) : NKRRU cuxrj

Execution Time : 0.0004734992980957031

The function **caesar_decrypt** remain unchange, the only difference is just how the key is determined. While brute-force method tests all 25 keys, the frequency analysis only tests top 10 most probable keys. As the result, this method successfully identified the correct key (key 3) in just 0.4 ms, which is approximately 3ms faster than testing all keys using brute-force.

B. Affine Cipher

1) Brute-force Attack

To perform a brute-force attack on the Affine Cipher, all possible key combination must be tried. The Affine Cipher uses two keys : a and b , which a must be co prime with 26, meaning $\gcd(a,26)=1$. Among the integers from 1 to 25, only 12 value satisfy this condition, which is all **odd number** from 1 to 25 **except** 13. This results in a total of $12 \times 26 = 312$ possible key combinations that must be tested.

Let's **encrypt** the message "Mr Brown loves math discrete" using key $a = 9$ and $b = 10$, implemented in Python using the **affine_encrypt** function shown in Fig. 5,

```

plaintext = "Mr Brown loves math discrete"
ciphertext = affine_encrypt(plaintext, 9,10)
print(f"Ciphertext:", ciphertext)

```

Fig.10. Python Implementation to encrypt plaintext into Affine Cipher

As the result, the Ciphertext is : "Oh Thgax fgruq okzv leqchuzu"

Here is the simple Python code implementation to brute-force all key combination :

```

start = time.time()
ciphertext = "Oh Thgax fgruq okzv leqchuzu"
for i in range (1,26,2): #just try odd number
    if i == 13 : #skip if a = 13
        continue
    for j in range (1,27):
        plaintext = affine_decrypt(ciphertext,i,j)
        print(f"Trying key a = {i} and b = {j}, Plaintext = {plaintext}")
end = time.time()
print(f"Execution Time : {end-start}")

```

Fig. 11. Python Implementation to decrypt Affine Cipher by testing all key combination

As the result, the correct key "Trying key $a = 9$ and $b = 10$, Plaintext = Mr Brown loves math discrete" was successfully discovered among the 312 possible key combinations in just about **21 milliseconds** of execution time.

2) Frequency Analysis

Frequency analysis was used by matching the top 10 most common English letters with the two most frequent characters in the ciphertext. While brute-force method need 312 pair of keys, frequency analysis just need only 90 letter mapping combinations. In other words, the maximum amount of key pairs to be tested is limited to 90 pairs, which is much more efficient than trying all key pairs.

To determine the correct key (a,b), it is necessary to identify two likely mapping between plaintext and ciphertext letters. Assume that plaintext character x_1 and x_2 are encrypted to ciphertext y_1 and y_2 . Then we can contrast the following congruence :

$$\begin{aligned} y_1 &\equiv ax_1 + b \pmod{26} \\ y_2 &\equiv ax_2 + b \pmod{26} \end{aligned} \quad (8)$$

By subtracting the equation, resulting the following congruence :

$$\begin{aligned} (y_1 - y_2) &\equiv a \times (x_1 - x_2) \pmod{26} \\ b &\equiv y_1 - ax_1 \pmod{26} \\ b &\equiv y_2 - ax_2 \pmod{26} \end{aligned} \quad (9)$$

Each pair (a,b) can be tested to determine which pair is the correct pair. For example, here is frequency of each letter in ciphertext "Oh Thgax fgruq okzv leqchuzu" :

A → 1	C → 1	E → 1	F → 1	G → 2
H → 3	K → 1	L → 1	O → 2	Q → 2
R → 1	T → 1	U → 3	V → 1	X → 1
Z → 2				

H and U are two of the most frequent letter in the cipher text. Now, by using the equation before and implementing to Python, here is the code :

```
start = time.time()

# Top 10 English letters
top_plain_letters = ['E', 'T', 'A', 'O', 'I', 'N', 'S', 'H', 'R', 'D']
# 2 most frequent letters from the ciphertext
top_cipher_letters = ['U', 'H']

#convert letters to integers
def to_number(c):
    return ord(c.upper()) - ord('A')

# all valid A (12 valid)
valid_a = [a for a in range(1, 26) if gcd(a, 26) == 1]

# all key combination
possible_keys = []

#try all 45 key combinations
for p1 in top_plain_letters:
    for p2 in top_plain_letters:
        if p1==p2 : continue
        c1 = top_cipher_letters[0]
        c2 = top_cipher_letters[1]
        x1 = to_number(p1)
        x2 = to_number(p2)
        y1 = to_number(c1)
        y2 = to_number(c2)

        dx = (x1 - x2) % 26
        dy = (y1 - y2) % 26

        for a in valid_a:
            if (a * dx) % 26 == dy:
                b = (y1 - a * x1) % 26
                possible_keys.append((a, b))

ciphertext = "Oh Thgax fgruq okzv leqchuzu"
i =1
for a,b in possible_keys:
    plaintext = affine_decrypt(ciphertext,a,b)
    print(f"{i}. Trying key a = {a} and b = {b}, Plaintext = {plaintext}")
    i+=1
end = time.time()
print(f"Execution Time : {end-start}")
```

Fig. 12. Python Implementation to Decrypt Affine Cipher by testing Potential Keys.

As the result, the correct key “Trying key $a = 9$ and $b = 10$, Plaintext = Mr Brown loves math discrete” was successfully discovered among the 48 possible key combinations in **2ms**, which is about 19 ms faster than bruteforcing all possible key.

C. Vignère Cipher

1) Brute-force Attack

In the Vignère Cipher, users can define both the keyword and its length, which significantly impacts the security of the cipher. Unlike Shift or Affine ciphers that use a fixed and limited keyspace, Vignère Cipher allows the keylength match the plaintext itself. This mean, for the plaintext of length n , the total number of possible keywords spans all the combination from keylength 1 to n , this lead to a cumulative keyspace of $26^1 + 26^2 + \dots + 26^n$.

To illustrate how quickly the keyspaces grow, consider a keyword of length 4, the number of possible keywords is exactly $26^4 = 456,976$. For length 6, the number jumps to $26^6 = 308,915,776$ keys. This exponential growth makes brute-force attacks on Vignère Cipher nearly impossible.

2) Frequency Analysis

Frequency analysis is one of the most powerful techniques used to break classical monoalphabetic ciphers like the Shift or Affine Cipher. However this approach become less effective when applied to Vignère Cipher, which is polyalphabetic. In Vignère Cipher, each character is encrypted using a different Caesar cipher based on the corresponding character in the keyword. This causes the frequency distribution in ciphertext more uniform.

Despite this, frequency analysis can still be used on Vigenère Cipher if the key length is known or guessed correctly. By grouping ciphertext character that were encrypted using the same keyword letter, each group effectively becomes a caesar cipher, which can be attacked individually using standard frequency analysis. The true challenge on Vignère cipher lies in determining the correct keyword, attackers often using technique such as **Kasiski Examination** and **Index of Coincidence (IC)**.

V. CONCLUSION

This paper has provided a comparative security analysis of three classical ciphers: Shift Cipher, Affine Cipher, and Vigenère Cipher. Through both theoretical and practical experiment, it was demonstrated that each cipher performs different security degree based on their keyspace size and resistance to cryptanalytic technique such as brute-force and frequency analysis.

The Shift Cipher, which has only 26 keyspace, is very vulnerable to brute-force attack. Affine Cipher offers a slightly more complex structure by using 2 key combination which has 312 keyspace, but still vulnerable to brute-force attack. Both of the cipher also vulnerable through frequency mapping. In contrast, Vignère Cipher significantly increase the complexity of attack by introducing polyalphabetic substitution, allowing a keyword of arbitrary length. This flexibility yields an exponential growth in the keyspace, making brute-force nearly impossible.

In conclusion, although none of these classical ciphers offer strong security by modern standards, they remain as foundation in cryptography education. Their simplicity provides a valuable lesson to understand the core concept such as modular arithmetic, keyspaces, injectivity, and statistical analysis. Through Python-based simulation, this paper demonstrate how even simple code can be used to effectively break these ciphers, this highlights the importance of transitioning to more secure modern encryptions.

APPENDIX

The complete source code used for the experiments in this paper is available on Drive: https://drive.google.com/drive/folders/1DV1DA4-G_TRuwe7hFbfhhvbXeJ0hFp57?usp=drive_link. The drive contain the Python Implementation to test each Cipher vulnerability to bruteforce and frequency analysis.

ACKNOWLEDGEMENT

The author sincerely acknowledges the guidance and blessings of God Almighty throughout the writing of this paper. Deep appreciation to Dr. Ir. Rinaldi Munir, M.T., and Mr. Arrival Dwi Sentosa, S.Kom., M.T., for their insightful lectures and continued support as lecturers of the IF1220 Discrete Mathematics course. The author is also grateful to their family for the constant encouragement and support given during the entire learning journey.

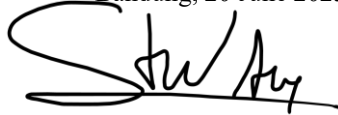
REFERENCE

- [1] D. R. Stinson and M. B. Paterson, *Cryptography: theory and practice*, Fourth edition. Boca Raton: CRC Press, Taylor & Francis Group, 2019.
- [2] R. Munir, "15-Teori Bilangan-Bagian 1-2024".
- [3] "(22) 🔒 Frequency Analysis in Cryptanalysis: Cracking Secrets Through Patterns | LinkedIn." Accessed: Jun. 15, 2025. [Online]. Available: <https://www.linkedin.com/pulse/frequency-analysis-cryptanalysis-cracking-secrets-through-aqeel-anwar-gw4lf>

STATEMENT

Hereby, I declare that the paper I have written is my own work, not an adaptation or translation of someone else's paper, and not a product of plagiarism.

Bandung, 20 June 2025



Stevanus Agustaf Wongso
13524020