**Practical No: 5**

**Practical Name: Study and implement a program for Vigenère Cipher.**

**Original Approach:**

**Introduction:** The **Vigenère Cipher** is a method of encrypting alphabetic text using a series of different Caesar ciphers based on the letters of a key. It was named after the French cryptographer **Blaise de Vigenère** and is a **polyalphabetic substitution cipher**. The cipher improves upon the Caesar cipher by using different shifts based on a repeating key.

**How the Vigenère Cipher Works:**

The Vigenère cipher uses a **key** to shift each letter of the plaintext by a varying number of positions. The position shift for each letter is determined by the corresponding letter in the key.

1. **Alphabet**: It uses the standard alphabet (A–Z) and treats letters cyclically. That is, after Z, it wraps around to A.
2. **Key**: The key is repeated to match the length of the plaintext if it's shorter.
3. **Encryption**: Each letter in the plaintext is shifted by the number of positions corresponding to the letter of the key.
   * A = 0, B = 1, C = 2, ..., Z = 25.

For example, if the key letter is "B" (which represents a shift of 1), the plaintext letter will be shifted by 1 position in the alphabet.

**Example of Vigenère Cipher:**

Let's encrypt the plaintext "ATTACK AT DAWN" using the key "BATTLE."

**1. Write the Plaintext and Repeat the Key**

* Plaintext: **ATTACK AT DAWN**
* Key: **BATTLEBATTLEBA**

The key "BATTLE" is repeated to match the length of the plaintext.

**2. Map the Letters to Their Numerical Positions**

We convert each letter into its corresponding numerical value:

* **Plaintext**:  
  A = 0, T = 19, T = 19, A = 0, C = 2, K = 10, A = 0, T = 19, D = 3, A = 0, W = 22, N = 13
* **Key**:  
  B = 1, A = 0, T = 19, T = 19, L = 11, E = 4, B = 1, A = 0, T = 19, T = 19, L = 11, E = 4

**3. Shift Each Letter**

For each letter in the plaintext, shift it by the number corresponding to the matching letter in the key.

* **A (0)** + **B (1)** = 1 → **B**
* **T (19)** + **A (0)** = 19 → **T**
* **T (19)** + **T (19)** = 38 → (38 % 26 = 12) → **M**
* **A (0)** + **T (19)** = 19 → **T**
* **C (2)** + **L (11)** = 13 → **N**
* **K (10)** + **E (4)** = 14 → **O**
* **\* + B(1) → \***
* **A (0)** + **A (0)** = 1 → **A**
* **T (19)** + **T (19)** = 38 → (38 % 26 = 12) → **T**
* **\* + T (19) → \***
* **D (3)** + **L (11)** = 14 → **O**
* **A (0)** + **E(4)** = 4 → **E**
* **W (22)** + **B (1)** = 23 → **X**
* **N (13)** + **A (0)** = 13 → **N**

**4. Encrypted Text**

The final encrypted message is:  
**BTMTNO\*AM\*OEXN**

Let’s decrypt the ciphertext "BTMTNO BT WTHR" using the key "BATTLE."

**1. Write the Ciphertext and Repeat the Key**

* **Ciphertext**: **BTMTNO\*AM\*OEXN**
* **Key**: **BATTLEBATTLEBA**

**2. Convert Ciphertext and Key into Numerical Values**

We convert each letter into its corresponding numerical value (A = 0, B = 1, ..., Z = 25):

* **Ciphertext**:  
  B = 1, T = 19, M = 12, T = 19, N = 13, O = 14, A = 0, M = 12, O = 14, E= 4, X = 23, N = 13
* **Key**:  
  B = 1, A = 0, T = 19, T = 19, L = 11, E = 4, B = 1, A = 0, T = 19, T = 19, L = 11, E = 4

**3. Shift the Ciphertext Backwards**

For each letter in the ciphertext, shift it **backward** by the corresponding key letter’s numerical value.

* **B (1)** − **B (1)** = 0 → **A**
* **T (19)** − **A (0)** = 19 → **T**
* **M (12)** − **T (19)** = -7 → (wrap around: -7 + 26 = 19) → **T**
* **T (19)** − **T (19)** = 0 → **A**
* **N (13)** − **L (11)** = 2 → **C**
* **O (14)** − **E (4)** = 10 → **K**
* **\*** − **B (1)** = 0 → **\***
* **A (19)** − **A (0)** = 19 → **A**
* **T (22)** − **T (19)** = 3 → **T**
* **\*** − **T (19)** = 0 → **\***
* **O (7)** − **L (11)** = -4 → (wrap around: -4 + 26 = 22) → **D**
* **E (17)** − **E (4)** = 13 → **A**
* **X (17) − B (1) = 13 → W**
* **N (17)** − **A (0)** = 13 → **N**

**4. Decrypted Text**

The final decrypted message is:  
**ATTACK AT DAWN**

**Source Code:**

alphabet\_lower = "abcdefghijklmnopqrstuvwxyz"

alphabet\_upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def preprocess\_text(plaintext, filler):

processed\_text = ''

For char in plaintext:

if char != ' ':

if char in alphabet\_lower:

index = alphabet\_lower.index(char)

processed\_text += alphabet\_upper[index]

else:

processed\_text += char

else:

processed\_text += filler

return processed\_text

def adjust\_key(plaintext, keys):

plain\_length = len(plaintext)

key\_length = len(keys)

while(key\_length != plain\_length):

if key\_length > plain\_length:

keys = keys[:plain\_length]

elif key\_length < plain\_length:

keys = (keys \* (plain\_length // key\_length + 1))

return keys

def encryption(message,keys, filler):

plaintext = preprocess\_text(message,filler)

keys = adjust\_key(plaintext,keys)

result = ''

for i in range(len(plaintext)):

char = plaintext[i]

key = keys[i]

if char in alphabet\_upper:

c = alphabet\_upper.index(char)

k = alphabet\_upper.index(key)

p = (c + k) % 26

result += alphabet\_upper[p]

else:

result += char

return result

def decryption(ciphertext, keys, filler):

keys = adjust\_key(ciphertext, keys)

result = ''

for i in range(len(ciphertext)):

char = ciphertext[i]

key = keys[i]

if char in alphabet\_upper:

c = alphabet\_upper.index(char)

k = alphabet\_upper.index(key)

p = (c - k + 26) % 26

result += alphabet\_upper[p]

elif char == filler:

result += ' '

else:

result += char

return result

message = input("Enter the plaintext: ")

keys = "BATTLE"

filler = '\*'

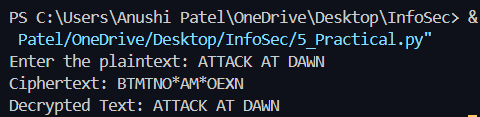
ciphertext = encryption(message, keys, filler)

print("Ciphertext:", ciphertext)

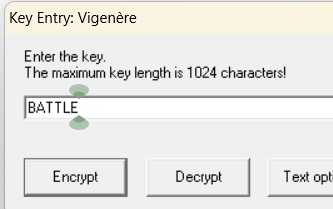
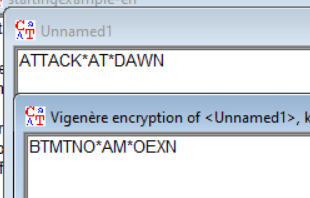
decrypted\_text = decryption(ciphertext, keys, filler)

print("Decrypted Text:", decrypted\_text)

**Output:**

****

**Cryptool:**

**Revised Approach:**

**Introduction**: The revised approach enhances the traditional Vigenère cipher by incorporating additional layers of encryption and a dynamic key extension mechanism. It employs double encryption: the first stage utilizes a split and reversed key to obscure the plaintext, while the second stage applies an autokey approach to further obfuscate the message. This autokey extension dynamically adjusts the key based on the plaintext, thereby increasing resistance to frequency analysis and other attacks that exploit repeating keys. The key splitting and reversal process—dividing the key into even and odd parts, with the odd part reversed—adds further complexity to the encryption, making the overall scheme more secure and intricate.

**Drawbacks of the Vigenère Cipher**

The Vigenère cipher, while more secure than a simple Caesar cipher, still has notable drawbacks:

1. **Repetitive Key Usage**: If the key is shorter than the plaintext, it repeats itself, which can create patterns in the ciphertext that can be exploited. This makes it vulnerable to frequency analysis attacks if the key is reused or too short.
2. **Pattern Recognition**: The repeating nature of the key can lead to recognizable patterns in the ciphertext, especially if the plaintext contains repetitive elements.
3. **Key Length Sensitivity**: The security of the Vigenère cipher is heavily dependent on the key length. A shorter key increases vulnerability, while a key of the same length as the plaintext (known as a one-time pad) provides perfect secrecy, but practical use of one-time pads is limited by key management issues.

**How the Modified Code Improves Security**

The provided code introduces several modifications to enhance the basic Vigenère cipher:

1. **Double Encryption**:
   * **Stage 1**: Uses a Vigenère cipher with a split key, applying different key segments to different halves of the plaintext.
   * **Stage 2**: Applies a second layer of encryption with an autokey method, where the key is dynamically extended with characters from the plaintext. This adds complexity and reduces the likelihood of pattern recognition.
2. **Autokey Extension**: In the second stage of encryption, the key is extended with characters from the plaintext, creating a more complex key that varies with each message, thus reducing the effectiveness of frequency analysis.
3. **Dynamic Key Adjustment for Decryption**: During decryption, the key is dynamically adjusted based on previously decrypted characters. This makes it harder for an attacker to reverse-engineer the decryption process.

**Example:**

Let's walk through the encryption process using the plaintext 'ATTACK AT DAWN' and the key 'BATTLE':

**Encryption**

1. **Preprocess Text**:
   * Input: 'ATTACK AT DAWN'
   * Processed: 'ATTACK\*AT\*DAWN' (replace spaces with '\*')
2. **Encryption Stage 1**:
   * Key is adjusted: BATTLEBATTLEBA
   * First half: 'ATTACK\*'
     + Encrypt using 'BTLBTLB'
     + Result: BMEBVV\*
   * Second half: 'AT\*DAWN'
     + Encrypt using 'AETAETA'
     + Result: AX\*DEPN
   * Combined Result from Stage 1: BMEBVV\*AX\*DEPN
3. **Encryption Stage 2**:
   * Generate autokey: BATTLEATTACKAT
   * Encrypt BMEBVV\*AX\*DEPN with this autokey
   * Final Ciphertext: CMXUGZ\*TQ\*FOPG

**Decryption**

1. **Decryption Stage 2**:
   * Generate autokey from the final ciphertext and base key
   * Decrypt using this autokey
2. **Decryption Stage 1**:
   * Decrypt the result from Stage 2 using the split key method
   * Result: 'ATTACK\*AT\*DAWN'
3. **Post-process Text**:
   * Replace '\*' with spaces
   * Final Decrypted Plaintext: 'ATTACK AT DAWN'

**Source Code:**

alphabet\_lower = "abcdefghijklmnopqrstuvwxyz"

alphabet\_upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def preprocess\_text(plaintext, filler):

processed\_text = ''

for char in plaintext:

if char != ' ':

if char in alphabet\_lower:

index = alphabet\_lower.index(char)

processed\_text += alphabet\_upper[index]

else:

processed\_text += char

else:

processed\_text += filler

return processed\_text

def adjust\_key\_for\_encryption(plaintext, base\_key):

auto\_key = base\_key.upper()

for char in plaintext:

if char.isalpha():

auto\_key += char

return auto\_key[:len(plaintext)]

def split\_key(key):

even\_key = ""

odd\_key = ""

for i in range(len(key)):

if i % 2 == 0:

even\_key += key[i]

else:

odd\_key += key[i]

return even\_key, odd\_key

def encryption\_1(message, base\_key, filler):

plaintext = preprocess\_text(message, filler)

key = base\_key.upper()

even\_key, odd\_key = split\_key(key)

mid = len(plaintext) // 2

first\_half = plaintext[:mid]

second\_half = plaintext[mid:]

result = ''

for i in range(len(first\_half)):

char = first\_half[i]

if char in alphabet\_upper:

key\_char = even\_key[i % len(even\_key)]

c = alphabet\_upper.index(char)

k = alphabet\_upper.index(key\_char)

p = (c + k) % 26

result += alphabet\_upper[p]

else:

result += char

for i in range(len(second\_half)):

char = second\_half[i]

if char in alphabet\_upper:

key\_char = odd\_key[i % len(odd\_key)]

c = alphabet\_upper.index(char)

k = alphabet\_upper.index(key\_char)

p = (c + k) % 26

result += alphabet\_upper[p]

else:

result += char

return result

def encryption\_stage2(ciphertext, base\_key, filler):

plaintext = preprocess\_text(ciphertext, filler)

auto\_key = adjust\_key\_for\_encryption(plaintext, base\_key)

result = ''

for i in range(len(plaintext)):

char = plaintext[i]

if char in alphabet\_upper:

key\_char = auto\_key[i]

c = alphabet\_upper.index(char)

k = alphabet\_upper.index(key\_char)

p = (c + k) % 26

result += alphabet\_upper[p]

else:

result += char

return result

def adjust\_key\_for\_decryption(ciphertext, base\_key, decrypted\_text):

auto\_key = base\_key.upper() + decrypted\_text

return auto\_key[:len(ciphertext)]

def decryption\_stage2(ciphertext, base\_key, filler):

auto\_key = adjust\_key\_for\_decryption(ciphertext, base\_key, "")

result = ''

decrypted\_text = ''

for i in range(len(ciphertext)):

char = ciphertext[i]

if char in alphabet\_upper:

key\_char = auto\_key[i]

c = alphabet\_upper.index(char)

k = alphabet\_upper.index(key\_char)

p = (c - k + 26) % 26

decrypted\_char = alphabet\_upper[p]

result += decrypted\_char

auto\_key += decrypted\_char

elif char == filler:

result += ' '

else:

result += char

return result

def decryption\_stage1(ciphertext, base\_key, filler):

key = base\_key.upper()

even\_key, odd\_key = split\_key(key)

mid = len(ciphertext) // 2

first\_half = ciphertext[:mid]

second\_half = ciphertext[mid:]

decrypted\_text = ''

for i in range(len(first\_half)):

char = first\_half[i]

if char in alphabet\_upper:

key\_char = even\_key[i % len(even\_key)]

c = alphabet\_upper.index(char)

k = alphabet\_upper.index(key\_char)

p = (c - k + 26) % 26

decrypted\_text += alphabet\_upper[p]

else:

decrypted\_text += char

for i in range(len(second\_half)):

char = second\_half[i]

if char in alphabet\_upper:

key\_char = odd\_key[i % len(odd\_key)]

c = alphabet\_upper.index(char)

k = alphabet\_upper.index(key\_char)

p = (c - k + 26) % 26

decrypted\_text += alphabet\_upper[p]

else:

decrypted\_text += char

return decrypted\_text

def complete\_encryption(message, base\_key, filler):

ciphertext\_stage1 = encryption\_1(message, base\_key, filler)

ciphertext\_stage2 = encryption\_stage2(ciphertext\_stage1, base\_key, filler)

return ciphertext\_stage2

def complete\_decryption(ciphertext, base\_key, filler):

decrypted\_text\_1 = decryption\_stage2(ciphertext, base\_key, filler)

decrypted\_text = decryption\_stage1(decrypted\_text\_1, base\_key, filler)

return decrypted\_text

message = input("Enter the plaintext: ")

base\_key = "BATTLE"

filler = '\*'

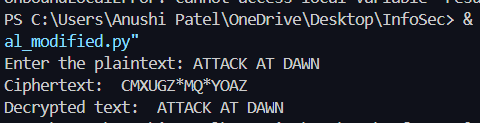
ciphertext = complete\_encryption(message, base\_key, filler)

print("Ciphertext: ", ciphertext)

decrypted\_text = complete\_decryption(ciphertext, base\_key, filler)

print("Decrypted text: ", decrypted\_text)

**Output:**

****

**Cryptanalysis on Vigenère Cipher:**

1. **Kasiski Examination:**

* **Description:** This method detects repeating patterns in the ciphertext that occur when the key is repeated. By identifying these repetitions, an attacker can estimate the length of the key. Once the key length is known, the ciphertext can be divided into segments, each corresponding to a Caesar cipher, which can be solved using frequency analysis.
* **Impact:** Kasiski examination significantly reduces the Vigenère cipher's security, especially when a short key is repeatedly used. It allows attackers to effectively break the cipher without knowing the key, making the cipher vulnerable to plaintext recovery.

1. **Known-Plaintext Attack:**

* **Description:** Involves having a portion of plaintext and its corresponding ciphertext to deduce the keyword.
* **Impact:** If the attacker knows part of the plaintext and its ciphertext, they can calculate shifts for those characters and infer parts of the keyword.

**Conclusion:**

The traditional Vigenère cipher improves upon the Caesar cipher by using a repeated key to shift each letter of the plaintext according to the corresponding letter in the key. While this method offers greater security than simple substitution ciphers, it remains vulnerable to frequency analysis, especially if the key is short and repetitive. Your modified approach addresses these weaknesses by introducing key splitting and double encryption. By dividing the key into segments for even and odd indices and dynamically adjusting it based on the plaintext, this method significantly enhances the encryption's complexity. This dynamic adjustment and increased key complexity add layers of obfuscation, making the cipher more resistant to cryptanalytic attacks. Thus, your approach provides a notable improvement over the classical Vigenère cipher, offering a more secure and robust method for encrypting messages.

**References:**

1. Grošek, Otokar, Eugen Antal, and Tomáš Fabšič. "Remarks on breaking the Vigenère autokey cipher." *Cryptologia* 43.6 (2019): 486-496.
2. Subandi, A., et al. "Vigenere cipher algorithm modification by adopting RC6 key expansion and double encryption process." *IOP Conference Series: Materials Science and Engineering*. Vol. 420. No. 1. IOP Publishing, 2018.