BSc Honours CS and IT

ITRI 615

Encryption & Decryption Application

Hereman van Heerden – 34402926

Phuti Malota - 41370015

Monica Serwala - 32711514

NORTH-WEST UNIVERSITY VAAL TRIANLGE CAMPUS

Contents

1 Introduction

The Vernam Cipher, also known as the One-Time Pad (OTP), is a symmetric key cipher where each bit or character of the plaintext is XORed with a corresponding bit or character from a randomly generated key of equal length. This encryption method is known for being theoretically unbreakable if the key is truly random, used only once, and kept completely secret. The Vernam Cipher offers a high level of security when implemented correctly.

In addition to the Vernam Cipher, this project includes two classical encryption algorithms: the Vigenère Cipher and the Transposition Cipher. They are substitution-based cryptosystems that are traditionally used for teaching and theoretical cryptography studies.

This project also includes the creation of a “Own Algorithm”, which is a custom hybrid-based encryption and decryption algorithm. This algorithm is based on the improvement of fundamental cryptography principles, with the focus on logic.

To extend beyond the classical approaches, the implementation of a Product Cipher was also included aswell as documented.

2 Own Algorithm (Herman van Heerden)

2.1 Overview

The algorithm created is a custom hybrid-based encryption and decryption algorithm. The algorithm combines the following fundamental cryptographic principles:

- Caesar shifting
- XOR operations
- Salted keys

2.2 Encryption Steps

I. A file is selected through the GUI.
II. The option is provided for the key to be randomly generated or to be entered manually by the user.

III. The key would be converted to a byte array (key_bytes) and mixed with a 16-byte, with the inclusion of being randomly salted.

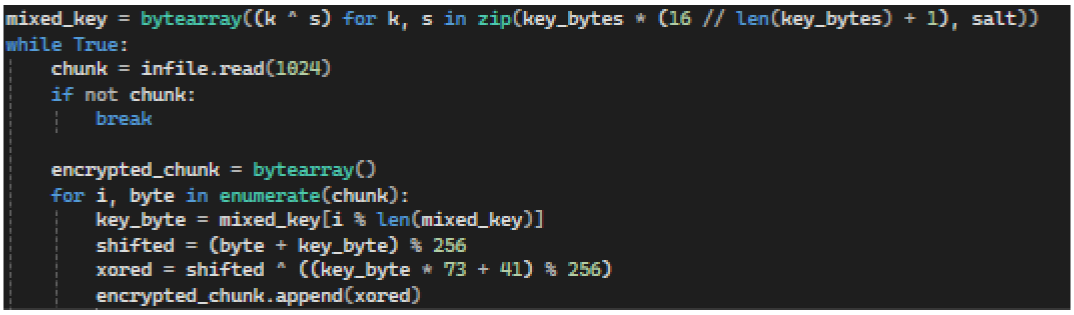
IV. The following information is prepended to the file for decryption validation:

- Key length
- Key
- Salt

V. Each individual byte within the file is:

- Shifted using the corresponding mixed_key byte created.
- XORed with the application of a key-dependent transformation: (key_byte * 73 + 41) % 256.

VI. The output is then saved as filename.ext.enc



```python
mixed_key = bytearray((k ^ s) for k, s in zip(key_bytes * (16 // len(key_bytes) + 1), salt))
while True:
    chunk = infile.read(1024)
    if not chunk:
        break

    encrypted_chunk = bytearray()
    for i, byte in enumerate(chunk):
        key_byte = mixed_key[i % len(mixed_key)]
        shifted = (byte + key_byte) % 256
        xored = shifted ^ ((key_byte * 73 + 41) % 256)
        encrypted_chunk.append(xored)
```

2.3 Decryption Steps

I. The application reads the key length, original bytes, and salt from the encryption file.

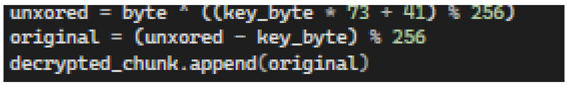
II. Then validates the key retrieved against the stored one.

III. The mixed_key would then undergo reconstruction with the use of the stored key and salt.

- Each byte would be:
- (Un-XORed) with the use of a key-dependent formula.

IV. Would then be unshifted using the mixed key.

V. Writes the original file, then the encrypted version would be deleted.



```python
unxored = byte ^ ((key_byte * 73 + 41) % 256)
original = (unxored - key_byte) % 256
decrypted_chunk.append(original)
```

2.4 Conclusion

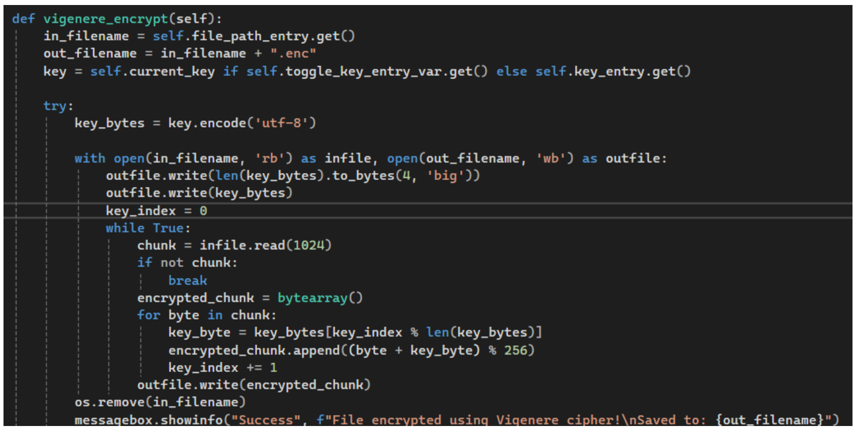
This is a hybrid algorithm, and makes use of fundamental cybersecurity principles. The salting process ensures that even in the case of where identical files are encrypted with the same key, they would still yield different outputs. The use of XOR transformation applies a nonlinear approach, which only amplifies the complexity. Chunked binary processing is the foundation on which all logic is based, thus making it compatible with any type of file.

3 Vigenere Cipher (Phuti Malota)

3.1 Overview

The Vigenère Cipher is a polyalphabetic substitution cipher that uses a keyword as an alphabet shifter for plaintext letters.

3.2 Encryption process (snippets code and explanation)



```python
def vigenere_encrypt(self):
    in_filename = self.file_path_entry.get()
    out_filename = in_filename + ".enc"
    key = self.current_key if self.toggle_key_entry_var.get() else self.key_entry.get()

    try:
        key_bytes = key.encode('utf-8')

        with open(in_filename, 'rb') as infile, open(out_filename, 'wb') as outfile:
            outfile.write(len(key_bytes).to_bytes(4, 'big'))
            outfile.write(key_bytes)
            key_index = 0
            while True:
                chunk = infile.read(1024)
                if not chunk:
                    break
                encrypted_chunk = bytearray()
                for byte in chunk:
                    key_byte = key_bytes[key_index % len(key_bytes)]
                    encrypted_chunk.append((byte + key_byte) % 256)
                    key_index += 1
                outfile.write(encrypted_chunk)
        os.remove(in_filename)
        messagebox.showinfo("Success", f"File encrypted using Vigenere cipher!\nSaved to: {out_filename}")
```

This script encrypts a file using the Vigenère cipher (byte version) and output it to a new .enc file.

I. Gets the path of the user's choice from the GUI input box.
II. Corrects the name of the output (encrypted) file by appending .enc to the name of the input file.
III. Choosing the encryption key:
    - If the user has generated a key, it uses self.current_key.
    - If the user enters a key, it uses that from the input box.
IV. Converting the key to a byte string (so it can be used with encryption).
V. Reading and Writing Files:
    - Reads the input file in binary mode.
    - Writes to the output file in binary mode.
VI. Appends the key to the encrypted file, so it can be accessed during decryption:

- Writes the key length as 4 bytes (big-endian).
- Appends the key itself in bytes.

VII. Reads file in 1024 byte chunks in case of large files.

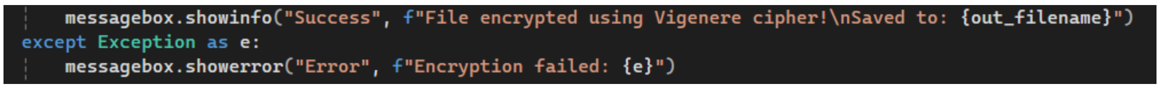
VIII. For each byte of the chunk:

- Appends corresponding key byte to it through modular arithmetic (wraps around 256).
- key_index % len(key_bytes) causes the key to repeat.

IX. Writes encrypted data chunk into output file.

X. Destroys original input file after successful encryption.

XI. Shows pop-up message of successful encryption and path of the file.

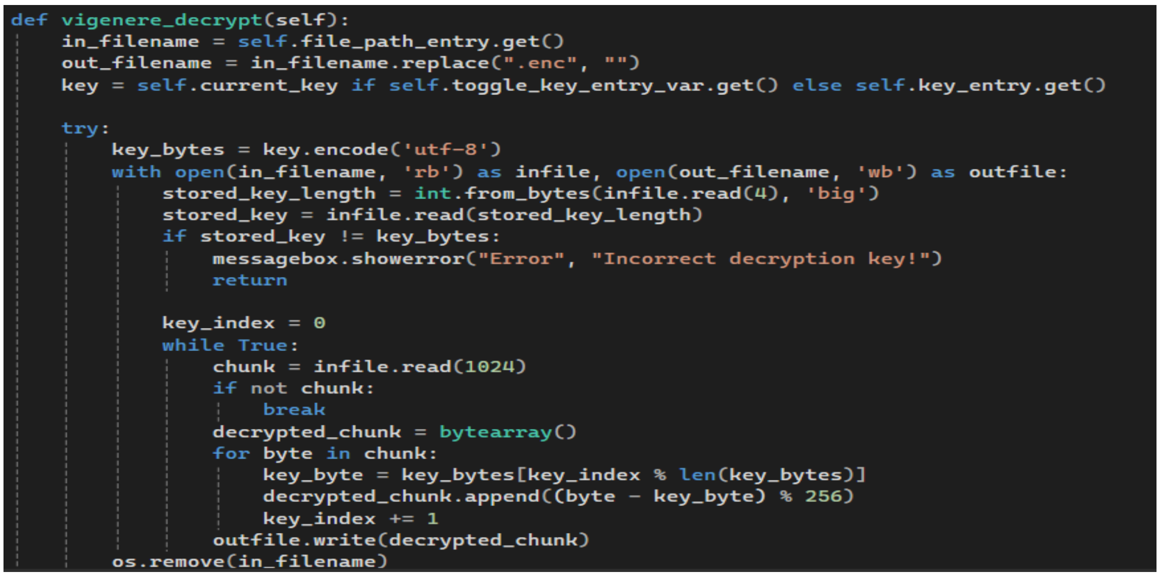


```
        messagebox.showinfo("Success", f"File encrypted using Vigenere cipher!\nSaved to: {out_filename}")
    except Exception as e:
        messagebox.showerror("Error", f"Encryption failed: {e}")
```

XII. Catches any encryption failure.

XIII. Pops up an error box with the message so the user is aware instead of the application crashing.

3.3 Decryption process (snippets code and explanation)



```
def vigenere_decrypt(self):
    in_filename = self.file_path_entry.get()
    out_filename = in_filename.replace(".enc", "")
    key = self.current_key if self.toggle_key_entry_var.get() else self.key_entry.get()

    try:
        key_bytes = key.encode('utf-8')
        with open(in_filename, 'rb') as infile, open(out_filename, 'wb') as outfile:
            stored_key_length = int.from_bytes(infile.read(4), 'big')
            stored_key = infile.read(stored_key_length)
            if stored_key != key_bytes:
                messagebox.showerror("Error", "Incorrect decryption key!")
                return

            key_index = 0
            while True:
                chunk = infile.read(1024)
                if not chunk:
                    break
                decrypted_chunk = bytearray()
                for byte in chunk:
                    key_byte = key_bytes[key_index % len(key_bytes)]
                    decrypted_chunk.append((byte - key_byte) % 256)
                    key_index += 1
                outfile.write(decrypted_chunk)
        os.remove(in_filename)
```

Decrypts a file that was encrypted using the Vigenère cipher.

I. Retrieve file path (in_filename) and path to output file (out_filename).

II. Retrieve decryption key from GUI.

III. Read encrypted file opening using:

- Original key length.
- Original key itself.

IV. Ensure that the provided key is the same as the original:

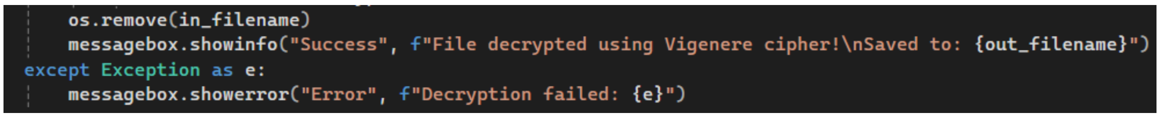
- Otherwise, show an error and stop.

V. Read the file in chunks other than the first one, decrypting each byte using the Vigenère formula:

   - (byte - key_byte) % 256.

VI. Write out decrypted data to the output file.

VII. Delete the original encrypted file.



```python
        os.remove(in_filename)
        messagebox.showinfo("Success", f"File decrypted using Vigenere cipher!\nSaved to: {out_filename}")
    except Exception as e:
        messagebox.showerror("Error", f"Decryption failed: {e}")
```

VIII. Show success message when the file is decrypted and saved.

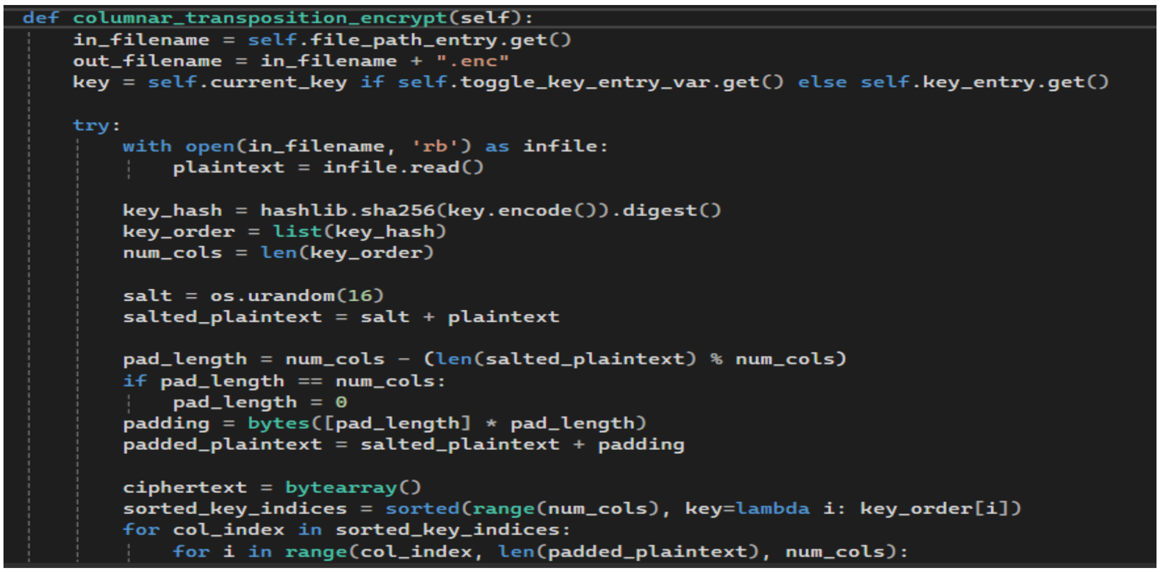
IX. Catches exceptions and show an error message if decryption fails.

4 Columnar Transposition Cipher (Phuti Malota)

4.1 Overview

A Transposition Cipher encrypts the plaintext by rearranging its characters according to a provided key or pattern.

4.2 Encryption process (Snippets code and explanation)



```python
def columnar_transposition_encrypt(self):
    in_filename = self.file_path_entry.get()
    out_filename = in_filename + ".enc"
    key = self.current_key if self.toggle_key_entry_var.get() else self.key_entry.get()

    try:
        with open(in_filename, 'rb') as infile:
            plaintext = infile.read()

        key_hash = hashlib.sha256(key.encode()).digest()
        key_order = list(key_hash)
        num_cols = len(key_order)

        salt = os.urandom(16)
        salted_plaintext = salt + plaintext

        pad_length = num_cols - (len(salted_plaintext) % num_cols)
        if pad_length == num_cols:
            pad_length = 0
        padding = bytes([pad_length] * pad_length)
        padded_plaintext = salted_plaintext + padding

        ciphertext = bytearray()
        sorted_key_indices = sorted(range(num_cols), key=lambda i: key_order[i])
        for col_index in sorted_key_indices:
            for i in range(col_index, len(padded_plaintext), num_cols):
```

Encrypts a file using the Columnar Transposition Cipher:

I. Fetch input/output file and key

   - READS file path from the GUI input.
   - Appends.enc to generate the output file name.

- Appends.enc to generate the output file name.

II. Read file contents

- Opens the file in binary mode and reads its content.

III. Generate key order

- Hashes the key with SHA-256.
- Use the hash bytes to decide the column order.

IV. Add salt

- Appends 16 random bytes (salt) to the beginning of the plaintext for additional randomness.

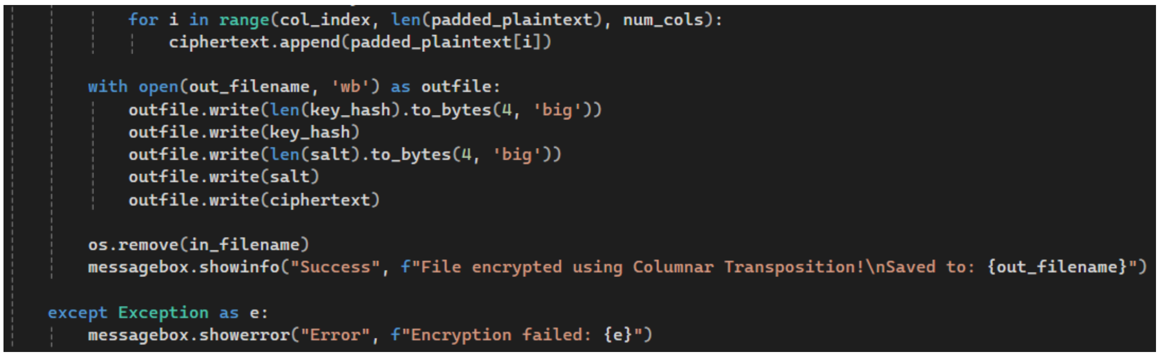
V. Pad plaintext

- Pads the data so that it perfectly fits into a rectangular block (depending on column number).

VI. Encrypt by columns

- Transposes the padded plaintext column-by-column according to the sorted order of the hash of the key.

VII. The end loop (starting with for col_index in sorted_key_indices:) continue encrypting the content by writing out the text column-wise based on the hashed key order.



```
            for i in range(col_index, len(padded_plaintext), num_cols):
                ciphertext.append(padded_plaintext[i])

        with open(out_filename, 'wb') as outfile:
            outfile.write(len(key_hash).to_bytes(4, 'big'))
            outfile.write(key_hash)
            outfile.write(len(salt).to_bytes(4, 'big'))
            outfile.write(salt)
            outfile.write(ciphertext)

        os.remove(in_filename)
        messagebox.showinfo("Success", f"File encrypted using Columnar Transposition!\nSaved to: {out_filename}")

    except Exception as e:
        messagebox.showerror("Error", f"Encryption failed: {e}")
```

VIII. For each byte of the chunk:

- Appends corresponding key byte to it through modular arithmetic (wraps around 256).
- key_index % len(key_bytes) causes the key to repeat.

IX. Writes encrypted data chunk into output file.

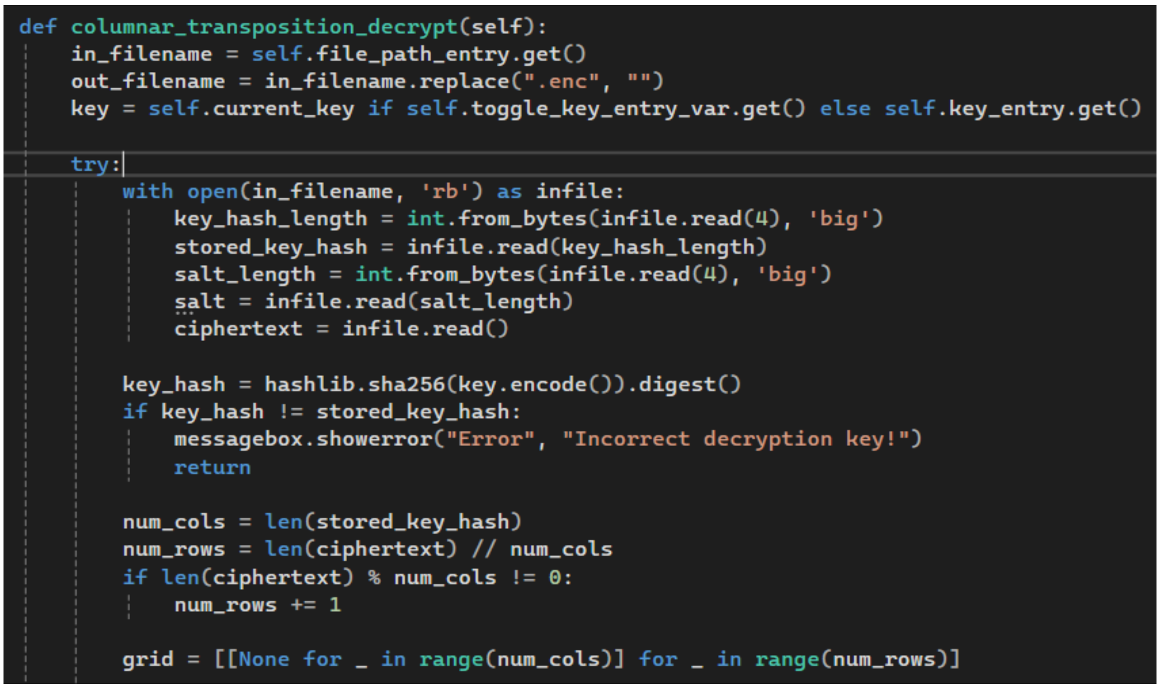
X. Destroys original input file after successful encryption.

XI. Shows pop-up message of successful encryption and path of the file.

XII. Catches any encryption failure.

XIII. Pops up an error box with the message so the user is aware instead of the application crashing.

4.3 Decryption process (Snippets code and explanation)

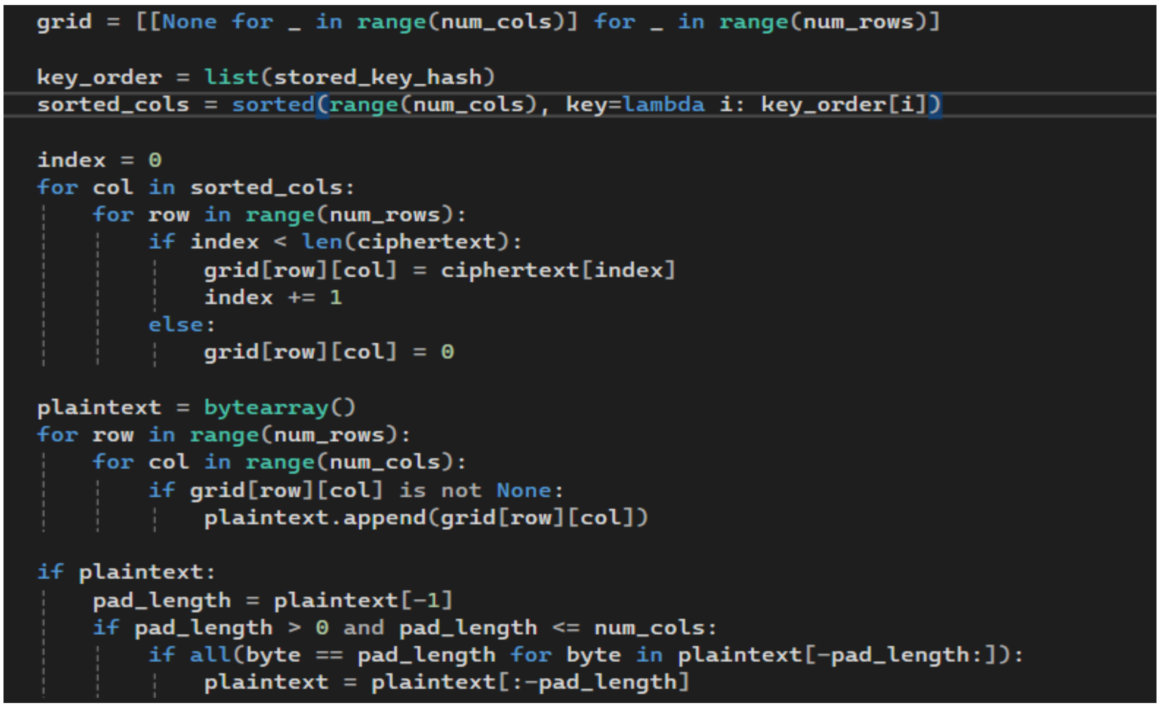


```python
def columnar_transposition_decrypt(self):
    in_filename = self.file_path_entry.get()
    out_filename = in_filename.replace(".enc", "")
    key = self.current_key if self.toggle_key_entry_var.get() else self.key_entry.get()

    try:
        with open(in_filename, 'rb') as infile:
            key_hash_length = int.from_bytes(infile.read(4), 'big')
            stored_key_hash = infile.read(key_hash_length)
            salt_length = int.from_bytes(infile.read(4), 'big')
            salt = infile.read(salt_length)
            ciphertext = infile.read()

        key_hash = hashlib.sha256(key.encode()).digest()
        if key_hash != stored_key_hash:
            messagebox.showerror("Error", "Incorrect decryption key!")
            return

        num_cols = len(stored_key_hash)
        num_rows = len(ciphertext) // num_cols
        if len(ciphertext) % num_cols != 0:
            num_rows += 1

        grid = [[None for _ in range(num_cols)] for _ in range(num_rows)]
```

I. Reads encrypted file details:
   - Gets the file path and decryption key.
   - Reads the key hash, salt, and ciphertext from the encrypted file.

II. Checks the key
   - Hashes the given key.
   - Checks against the saved key hash to see if the correct key is employed.
   - Incorrect key is found, reports an error, and stops.

III. Prepare grid reconstruction:
   - Determines number of columns according to key hash size.
   - Calculates required rows according to ciphertext length.
   - Construct an empty grid (grid) with enough rows and columns to hold decrypted content.



```
grid = [[None for _ in range(num_cols)] for _ in range(num_rows)]

key_order = list(stored_key_hash)
sorted_cols = sorted(range(num_cols), key=lambda i: key_order[i])

index = 0
for col in sorted_cols:
    for row in range(num_rows):
        if index < len(ciphertext):
            grid[row][col] = ciphertext[index]
            index += 1
        else:
            grid[row][col] = 0

plaintext = bytearray()
for row in range(num_rows):
    for col in range(num_cols):
        if grid[row][col] is not None:
            plaintext.append(grid[row][col])

if plaintext:
    pad_length = plaintext[-1]
    if pad_length > 0 and pad_length <= num_cols:
        if all(byte == pad_length for byte in plaintext[-pad_length:]):
            plaintext = plaintext[:-pad_length]
```

IV. Determine column sequence using the key hash stored

- Maintains stored key hash in list form.
- Orders the column indices based on their hashed key values.
- This gives the correct order to fill the columns since they were encrypted.

V. Fill grid column by column

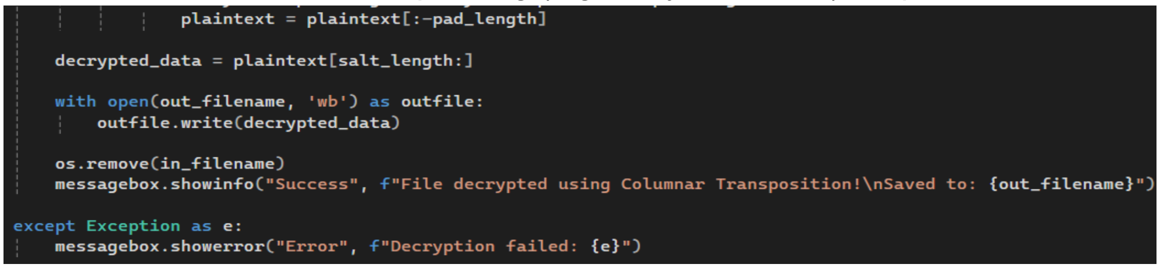
- Traverses the ordered sequence of columns.
- Fills each cell in the grid from top to bottom.
- Ciphertext is shorter than the grid, remaining cells are filled with 0s.

VI. Read grid row by row to reconstruct plaintext

- Now that the grid is completed column-wise, read it row by row (left to right, top to bottom) to get the original order of data.

VII. Remove padding if present

- Checks the last byte of plaintext to see if it indicates length of padding.
- If discovered to contain valid padding (e.g., 4 bytes of \x04), strips it.



```
                plaintext = plaintext[:-pad_length]

    decrypted_data = plaintext[salt_length:]

    with open(out_filename, 'wb') as outfile:
        outfile.write(decrypted_data)

    os.remove(in_filename)
    messagebox.showinfo("Success", f"File decrypted using Columnar Transposition!\nSaved to: {out_filename}")

except Exception as e:
    messagebox.showerror("Error", f"Decryption failed: {e}")
```

VIII. Remove the Salt

- When encrypting, a random salt was added to the start of the plaintext.

- This line deletes the salt (based on the salt_length) to bring back the original file content.

IX. Write decrypted FileCreates a new file based on the original name (removing .enc).

- Writes pure, decrypted data (salt- and padding-free).

X. Deletes the encrypted .enc file in case of successful decryption.

XI. Display Success or Error Message

5 Vernam Cipher (Monica Serwala)

5.1 Overview

The Vernam Cipher, also known as the One-Time Pad (OTP), is a symmetric key cipher where each bit or character of the plaintext is XORed with a corresponding bit or character from a randomly generated key of equal length. This encryption method is known for being theoretically unbreakable if the key is truly random, used only once, and kept completely secret. The Vernam Cipher offers a high level of security when implemented correctly.

5.2 Key Features

- Perfect Secrecy: If the key is truly random and as long as the plaintext, the Vernam Cipher is considered perfectly secure.
- Symmetric Key Encryption: The same key is used for both encryption and decryption.
- Bitwise XOR Operation: Each character in the plaintext is XORed with a corresponding character in the key, creating the ciphertext.

5.3 Key Generation for Vernam Cipher

The Vernam Cipher relies on a key that is:

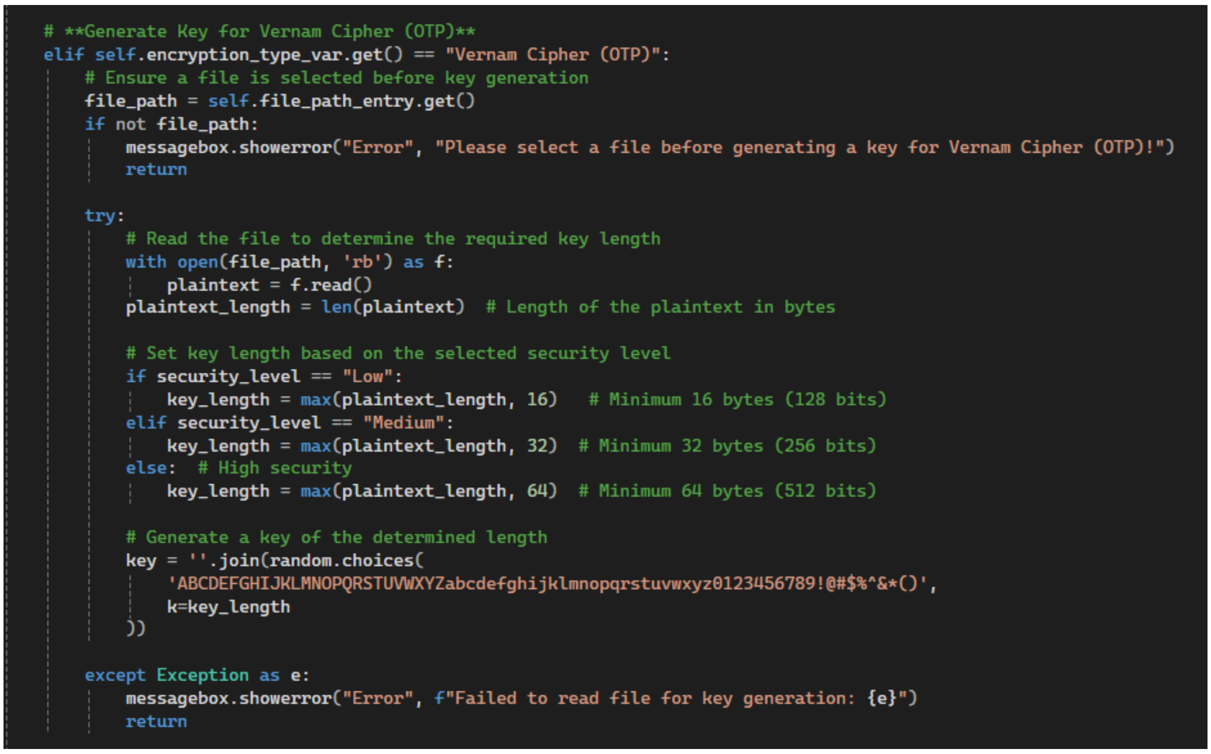
- At least as long as the plaintext to be encrypted.
- Completely random to ensure unbreakable security.
- Unique for each message to avoid reuse vulnerabilities.

5.4 Key Generation Logic in Cryptography App

For the Vernam Cipher, the key is generated using a secure random selection of characters from a wide range of possible values, including uppercase and lowercase letters, digits, and special characters. The length of the key is determined by the size of the plaintext file, ensuring a perfect one-to-one mapping during the XOR operation. The minimum lengths for different security levels are:

- **Low Security**: At least 16 characters (128 bits)
- **Medium Security**: At least 32 characters (256 bits)
- **High Security**: At least 64 characters (512 bits)

5.5 Key Generation Code



```python
# **Generate Key for Vernam Cipher (OTP)**
elif self.encryption_type_var.get() == "Vernam Cipher (OTP)":
    # Ensure a file is selected before key generation
    file_path = self.file_path_entry.get()
    if not file_path:
        messagebox.showerror("Error", "Please select a file before generating a key for Vernam Cipher (OTP)!")
        return

    try:
        # Read the file to determine the required key length
        with open(file_path, 'rb') as f:
            plaintext = f.read()
        plaintext_length = len(plaintext)  # Length of the plaintext in bytes

        # Set key length based on the selected security level
        if security_level == "Low":
            key_length = max(plaintext_length, 16)   # Minimum 16 bytes (128 bits)
        elif security_level == "Medium":
            key_length = max(plaintext_length, 32)  # Minimum 32 bytes (256 bits)
        else:  # High security
            key_length = max(plaintext_length, 64)  # Minimum 64 bytes (512 bits)

        # Generate a key of the determined length
        key = ''.join(random.choices(
            'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()',
            k=key_length
        ))

    except Exception as e:
        messagebox.showerror("Error", f"Failed to read file for key generation: {e}")
        return
```

Implementation in Cryptography App

5.6 Methods

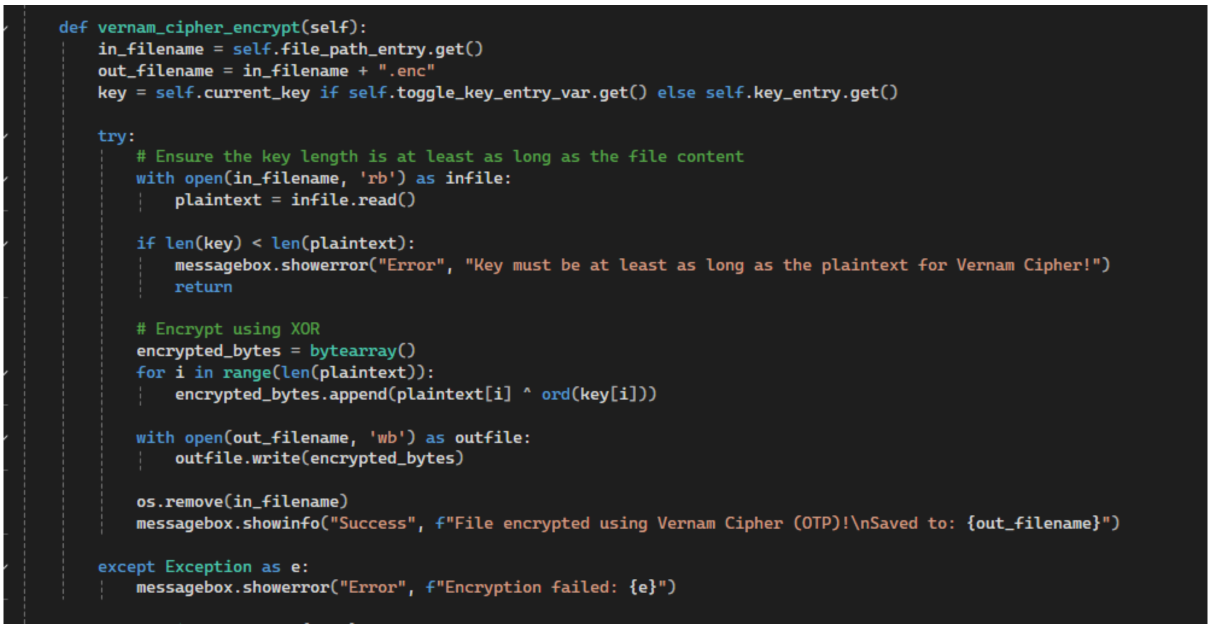
5.6.1 vernam_cipher_encrypt

This method encrypts a file using the Vernam Cipher. It reads the plaintext from a file, encrypts it using a randomly generated or manually provided key, and writes the encrypted data to a new file.

Key Steps:

I. File Reading: Reads the plaintext from the selected input file.
II. Key Length Verification: Ensures that the key is at least as long as the plaintext.
III. Encryption Process:
    a. XORs each byte of the plaintext with the corresponding character of the key.
    b. Creates a bytearray to store the encrypted data.
IV. File Writing: Writes the encrypted data to a new file with the ".enc" extension.
V. Cleanup: Deletes the original plaintext file to enhance security.

Method Code:



```
def vernam_cipher_encrypt(self):
    in_filename = self.file_path_entry.get()
    out_filename = in_filename + ".enc"
    key = self.current_key if self.toggle_key_entry_var.get() else self.key_entry.get()

    try:
        # Ensure the key length is at least as long as the file content
        with open(in_filename, 'rb') as infile:
            plaintext = infile.read()

        if len(key) < len(plaintext):
            messagebox.showerror("Error", "Key must be at least as long as the plaintext for Vernam Cipher!")
            return

        # Encrypt using XOR
        encrypted_bytes = bytearray()
        for i in range(len(plaintext)):
            encrypted_bytes.append(plaintext[i] ^ ord(key[i]))

        with open(out_filename, 'wb') as outfile:
            outfile.write(encrypted_bytes)

        os.remove(in_filename)
        messagebox.showinfo("Success", f"File encrypted using Vernam Cipher (OTP)!\nSaved to: {out_filename}")

    except Exception as e:
        messagebox.showerror("Error", f"Encryption failed: {e}")
```

5.6.2 vernam_cipher_decrypt

This method reverses the encryption process, taking the encrypted file and the same key used for encryption to recover the original plaintext.

Key Steps:

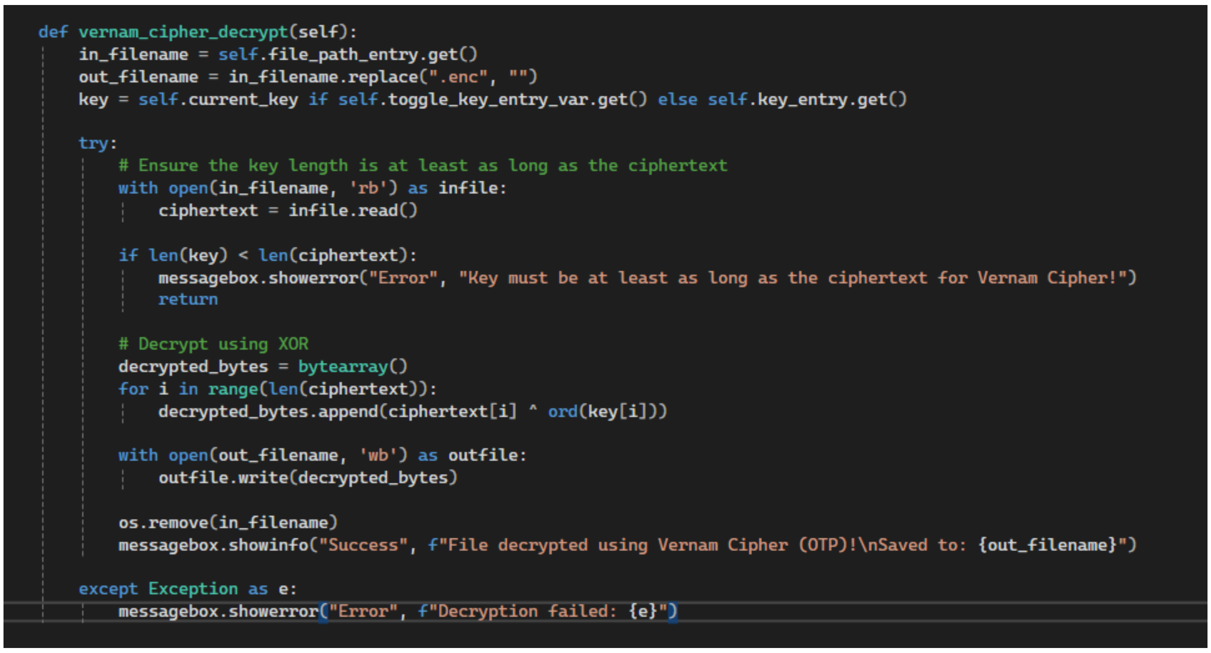
I. File Reading: Reads the ciphertext from the encrypted file.
II. Key Length Verification: Ensures the key is at least as long as the ciphertext.
III. Decryption Process:
    a. XORs each byte of the ciphertext with the corresponding character of the key.

b. Recreates the original plaintext as a bytearray.

IV. File Writing: Writes the decrypted data to a file with the original filename.

V. Cleanup: Deletes the encrypted file after successful decryption.

Method Code:



```
def vernam_cipher_decrypt(self):
    in_filename = self.file_path_entry.get()
    out_filename = in_filename.replace(".enc", "")
    key = self.current_key if self.toggle_key_entry_var.get() else self.key_entry.get()

    try:
        # Ensure the key length is at least as long as the ciphertext
        with open(in_filename, 'rb') as infile:
            ciphertext = infile.read()

        if len(key) < len(ciphertext):
            messagebox.showerror("Error", "Key must be at least as long as the ciphertext for Vernam Cipher!")
            return

        # Decrypt using XOR
        decrypted_bytes = bytearray()
        for i in range(len(ciphertext)):
            decrypted_bytes.append(ciphertext[i] ^ ord(key[i]))

        with open(out_filename, 'wb') as outfile:
            outfile.write(decrypted_bytes)

        os.remove(in_filename)
        messagebox.showinfo("Success", f"File decrypted using Vernam Cipher (OTP)!\nSaved to: {out_filename}")

    except Exception as e:
        messagebox.showerror("Error", f"Decryption failed: {e}")
```

Usage

I. **Encryption**:

- Select a plaintext file using the "Browse" button.
- Generate a key or manually enter one.
- Click the "Encrypt" button to encrypt the file.

II. **Decryption**:

- Select the encrypted ".enc" file.
- Provide the same key used for encryption.
- Click the "Decrypt" button to recover the original plaintext.

Security Considerations

- Key Length: Ensure the key is at least as long as the plaintext to maintain security.
- Key Management: The key must remain secret and never reused for another message.

- File Deletion: The original plaintext file is deleted after encryption to prevent unauthorized access.

Conclusion

The Vernam Cipher implementation in this project provides a strong form of encryption, but its security depends heavily on proper key management and length. Use this method only when the key can be securely generated, distributed, and destroyed after use.

6 Product Cipher (Herman van Heerden)

6.1 Overview

This algorithm is the combination of two different encryption techniques in a sequence, with the purpose of creating a stronger composite algorithm. The two encryption techniques used are:

- Vigenere Cipher (Substitution)
- Columnar Transposition Cipher (Permutation)

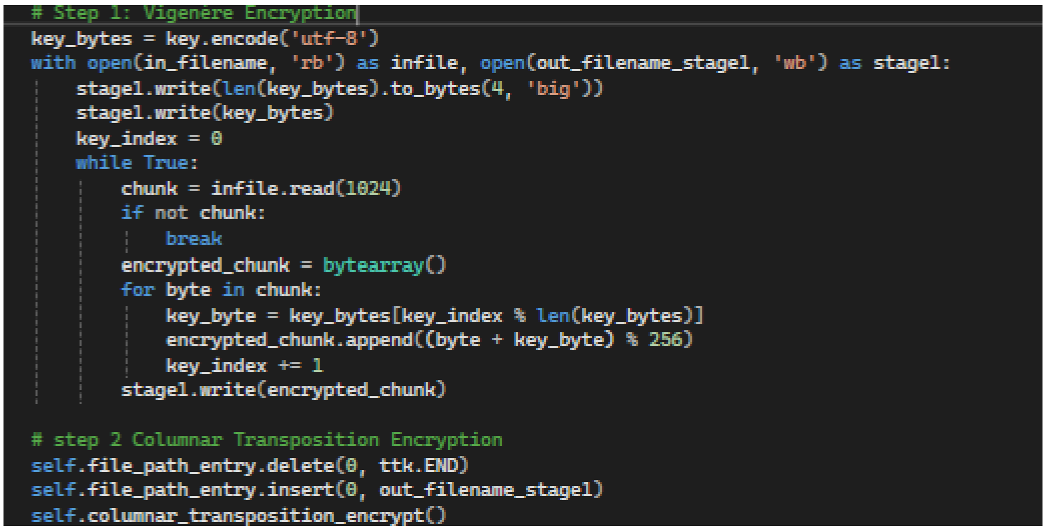
6.2 Encryption Steps

I. The first encryption technique is applied, which is the Vigenere Cipher:

- Thus each byte is shifted, with the use of a repeated key (byte + key_byte) % 256.
- The key length and key are then written to the file.

II. The file is then received by the Columnar Transposition Cipher where:

- The column order is determined by the key of a SHA-256 hash.
- Salt is then added before transposition
- And finally the file is then saved as .enc



```
# Step 1: Vigenere Encryption
key_bytes = key.encode('utf-8')
with open(in_filename, 'rb') as infile, open(out_filename_stage1, 'wb') as stage1:
    stage1.write(len(key_bytes).to_bytes(4, 'big'))
    stage1.write(key_bytes)
    key_index = 0
    while True:
        chunk = infile.read(1024)
        if not chunk:
            break
        encrypted_chunk = bytearray()
        for byte in chunk:
            key_byte = key_bytes[key_index % len(key_bytes)]
            encrypted_chunk.append((byte + key_byte) % 256)
            key_index += 1
        stage1.write(encrypted_chunk)

# step 2 Columnar Transposition Encryption
self.file_path_entry.delete(0, ttk.END)
self.file_path_entry.insert(0, out_filename_stage1)
self.columnar_transposition_encrypt()
```

6.3 Decryption Steps

I. Columnar decryption is applied first:

- The salt is read aswell as the key hash, in order to construct the original matrix.
- Salt is then removed aswell as padding.

II. Output is decrypted, including the Vigenere algorithm:

- Key is validated against metadata.
- And finally it performs (byte – key_byte) % 256

6.4 Conclusion

The use of both substitution and permutation will allow for stronger cryptographic diffusion, as well as possible patterns within the plaintext being scrambled.