An **Image Encryption Tool** using pixel manipulation involves reading image pixels, applying an encryption algorithm (like swapping or mathematical operations), and providing a way to decrypt the image back to its original form. Below is the code implementation in Python, with detailed explanations.

### ****Code Implementation****

from PIL import Image

import numpy as np

def encrypt\_image(image\_path, key):

"""

Encrypts an image using pixel manipulation.

:param image\_path: Path to the input image

:param key: Encryption key (integer)

:return: Encrypted image as a PIL Image object

"""

# Load the image

img = Image.open(image\_path)

img\_array = np.array(img) # Convert image to a NumPy array

# Perform encryption: Add the key to each pixel value

encrypted\_array = (img\_array + key) % 256 # Ensure values remain in the 0-255 range

# Convert the encrypted array back to an image

encrypted\_img = Image.fromarray(encrypted\_array.astype('uint8'))

return encrypted\_img

def decrypt\_image(encrypted\_image, key):

"""

Decrypts an encrypted image using pixel manipulation.

:param encrypted\_image: Encrypted image as a PIL Image object

:param key: Decryption key (integer)

:return: Decrypted image as a PIL Image object

"""

encrypted\_array = np.array(encrypted\_image) # Convert image to a NumPy array

# Perform decryption: Subtract the key from each pixel value

decrypted\_array = (encrypted\_array - key) % 256 # Ensure values remain in the 0-255 range

# Convert the decrypted array back to an image

decrypted\_img = Image.fromarray(decrypted\_array.astype('uint8'))

return decrypted\_img

# Main scriptif \_\_name\_\_ == "\_\_main\_\_":

import os

# Input: Image file and encryption key

image\_path = input("Enter the path to the image file: ")

if not os.path.exists(image\_path):

print("Error: File not found!")

exit()

key = int(input("Enter the encryption key (integer): "))

# Encrypt the image

print("Encrypting the image...")

encrypted\_img = encrypt\_image(image\_path, key)

encrypted\_img.save("encrypted\_image.png")

print("Encrypted image saved as 'encrypted\_image.png'.")

# Decrypt the image

print("Decrypting the image...")

decrypted\_img = decrypt\_image(encrypted\_img, key)

decrypted\_img.save("decrypted\_image.png")

print("Decrypted image saved as 'decrypted\_image.png'.")

### ****Explanation of the Code****

**Importing Libraries**:

PIL.Image: Used to handle image operations (load, manipulate, and save images).

numpy: Used for efficient array manipulation (converting images to arrays for pixel-level operations).

**Encryption Function**:

**Input**: Path to the image and a numerical encryption key.

**Operation**: Adds the key to each pixel value in the image. The modulo operation ensures the values stay within the 0-255 range (valid pixel range for an 8-bit image).

**Output**: An encrypted image.

**Decryption Function**:

**Input**: Encrypted image and the encryption key.

**Operation**: Subtracts the key from each pixel value to revert to the original pixel values. The modulo operation ensures no overflow occurs.

**Output**: A decrypted image.

**Main Script**:

Prompts the user for an image file path and encryption key.

Calls the encrypt\_image function and saves the encrypted image.

Calls the decrypt\_image function and saves the decrypted image.

### ****How It Works****

**Encryption**:

Each pixel value (R, G, B) is altered using a simple arithmetic operation (e.g., addition with the key).

For example:

* + 1. Original pixel value = (120, 200, 255)
    2. Key = 50
    3. Encrypted pixel = (170, 250, 49) (values wrapped using % 256).

**Decryption**:

Reverse the operation (subtraction with the key) to retrieve the original pixel values.

For example:

* + 1. Encrypted pixel = (170, 250, 49)
    2. Key = 50
    3. Decrypted pixel = (120, 200, 255).

### ****Features****

**Custom Key**:

Users can specify an integer key for encryption, adding a layer of customization and security.

**Image Format Compatibility**:

Supports multiple formats (e.g., .png, .jpg) due to the flexibility of the PIL library.

**Ease of Use**:

Simple command-line interface for user input and output.

**Reversible**:

The encryption is lossless, so the original image can be retrieved without any degradation in quality.

### ****Sample Output****

* Input Image: sample\_image.jpg
* Key: 50
* Encrypted Image: encrypted\_image.png
* Decrypted Image: decrypted\_image.png (matches the original).

### ****Advantages****

* **Lightweight**: Simple mathematical operations make this approach fast and lightweight.
* **Customizable**: The key can be modified for different encryption schemes.
* **Cross-Platform**: Works on any platform with Python and required libraries installed.

### ****Limitations****

* **Basic Security**: Simple addition/subtraction is not highly secure against advanced attacks. Stronger encryption algorithms (e.g., AES) are recommended for sensitive data.
* **Key Dependency**: Loss of the key makes decryption impossible.