A

MAJOR PROJECT REPORT

ON

INTERATING CRYPTOGRAPHY AND STEGANOGRAPHY FOR ROBUST DATA PROTECTION IN IMAGES

Submitted in partial fulfillment of the requirements
For the award of Degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted By

GOODA RUSHENDAR REDDY	2453-21-733-154
K. SATHWIKA	2453-21-733-318
SURY SUPRIYA	2453-21-733-319

Under the guidance

Of

Dr.R.Srilakshmi

Assistant Professor

Department of Computer Science and Engineering

AFFILIATED TO OSMANIA UNIVERSITY HYDERABAD

A

MAJOR PROJECT REPORT

ON

INTERATING CRYPTOGRAPHY AND STEGANOGRAPHY FOR ROBUST DATA PROTECTION IN IMAGES

Submitted in partial fulfillment of the requirements For the award of Degree of
BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted By

GOODA RUSHENDAR REDDY	**2453-21-733-154**
K. SATHWIKA	**2453-21-733-318**
SURY SUPRIYA	**2453-21-733-319**

Under the guidance

Of

Dr.R.Srilakshmi

Assistant Professor

Department of Computer Science and Engineering

NEIL GOGTE INSTITUTE OF TECHNOLOGY
Kachavanisingaram Village, Hyderabad, Telangana 500058.

June 2025

NEIL GOGTE INSTITUTE OF TECHNOLOGY
A Unit of Keshav Memorial Technical Education (KMTES)
Approved by AICTE, New Delhi & Affiliated to Osmania University, Hyderabad

CERTIFICATE

This is to certify that the project work entitled "**Integrating cryptograpy and steganography for robust data protection in image**" is a **Bonafide** work carried out **by Gooda Rushendar Reddy(245321733154), K. Sathwika (245321733318), Sury Supriya (245321733319)** of IV-year VIII semester **Bachelor of Engineering** in **Computer Science and Engineering** by Osmania University, Hyderabad during the academic year 2021-2025 is a record of Bonafide work carried out by them. The results embodied in this report have not been submitted to any other University or Institution for the award of any degree.

Internal Guide | **Head of Department**

Dr.R.Srilakshmi | Dr. P. VAISHALI

External

DECLARATION

We hereby declare that the Major Project Report entitled**, "Integrating cryptograpy and steganography for robust data protection in image"** submitted for the B.E degree is entirely our work and all ideas and references have been duly acknowledged. It does not contain any work for the award of any other degree.

Date:

GOODA RUSHENDAR REDDY	**2453-21-733-154**
K. SATHWIKA	**2453-21-733-318**
SURY SUPRIYA	**2453-21-733-319**

ACKNOWLEDGEMENT

We are happy to express our deep sense of gratitude to the principal of the college **Prof R. SHYAM SUNDAR**, Neil Gogte Institute of Technology, for having provided us with adequate facilities to pursue our project.

We would like to thank **Dr. P. VAISHALI, Head of the Department**, Computer Science and Engineering, Neil Gogte Institute of Technology, for having provided the freedom to use all the facilities available in the department, especially the laboratories and the library.

We would also like to thank my internal guide **Dr.R.Srilakshmi, Assistant Professor** for his Technical guidance & constant encouragement.

We sincerely thank our seniors and all the teaching and non-teaching staff of the Department of Computer Science & Engineering for their timely suggestions, healthy criticism and motivation during the course of this work.

Finally, we express our immense gratitude with pleasure to the other individuals who have either directly or indirectly contributed to our need at the right time for the development and success of this work.

ABSTRACT

In the rapidly evolving realm of digital communication, the need for robust data protection has grown exponentially. This project introduces an innovative approach known as "crystography" seamlessly blending the strengths of cryptography and steganography to enhance information security. Addressing the heightened demand for fortified encryption techniques in the exchange of confidential and private data, this approach utilizes steganography to covertly embed encrypted information within images – a prevalent medium of communication. The core strategy involves segmenting images and employing established cryptographic methods to encrypt the message content. Subsequently, a unique 3-3-2 Least Significant Bit (LSB) insertion technique, leveraging human visual perception limitations, conceals the encrypted segments within the image. The project offers a comprehensive examination, covering introduction, historical context, procedural intricacies, experimental validation, comparative analysis with existing methodologies, and conclusive insights. By effectively integrating cryptography and steganography, crystography emerges as a pioneering solution in safeguarding sensitive information amidst the dynamic landscape of digital data exchange.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER - 1

INTRODUCTION

1.1 Problem Statement

The problem at hand is the increasing demand for stronger data protection in digital communication. Existing cryptographic methods may not be sufficient to meet this demand, leading to the need for innovative solutions. "Crystography" is introduced as a blend of cryptography and steganography to enhance information security by covertly embedding encrypted data within images, a common medium of communication. This problem statement emphasizes the necessity of such innovative approaches in safeguarding sensitive information in the digital data exchange landscape.

1.2 Motivation

In an era dominated by digital communication and data exchange, the security and confidentiality of information have become paramount. Traditional data protection techniques such as cryptography offer strong encryption mechanisms to protect data from unauthorized access. However, encrypted data can still attract the attention of malicious actors due to its apparent structure, making it a visible target. On the other hand, steganography—hiding information within seemingly innocuous files such as images—offers the advantage of concealing the very existence of data, thus reducing the likelihood of detection. Despite their individual strengths, each method has limitations when used in isolation. Cryptography secures content but not its presence, while steganography conceals data but lacks strong encryption. This project is motivated by the need to create a more secure, multi-layered approach to data protection by combining the strengths of both cryptography and steganography.

1.3 Scope

The proposed method presents a robust foundation for secure data transmission by integrating cryptography and steganography, termed "crystography." The scope of this project can be significantly expanded by incorporating stronger cryptographic algorithms alongside the existing steganographic technique to enhance security further. By extending beyond single or dual-level encryption, a multilevel encryption strategy can be implemented, which would make it considerably more resistant to attacks and data breaches. This opens opportunities for broader applications in areas requiring high levels of data confidentiality such as military communications, digital forensics, and secure financial systems. The adaptability of this system allows for future integration with advanced cryptographic schemes, making it a scalable solution for evolving cybersecurity demands.

1.4 Outline

The project presents a novel technique called crystography, which combines cryptography and steganography to enhance the security of digital data transmission. The report begins with an overview of steganography, its significance in concealing information, and its limitations when used independently. To address these limitations, the proposed system encrypts secret messages using a symmetric key cryptographic algorithm and then hides them within images using a custom 3-3-2 Least Significant Bit (LSB) insertion method. The report details the system architecture, private key generation, encryption and decryption processes, and implementation specifics using Python with modules like PIL, Hashlib, and Tkinter. A comprehensive literature survey supports the methodology by evaluating existing techniques and identifying gaps. The results demonstrate the successful embedding and retrieval of both text and files from stego images via a user-friendly GUI. The conclusion emphasizes the robustness and imperceptibility of the proposed method and outlines its potential applications in secure communication, digital watermarking, DRM, and IoT.

CHAPTER - 2

LITERATURE SURVEY

2.1 Existing Work

In existing systems for information security, standard cryptographic algorithms such as AES (Advanced Encryption Standard), RSA (Rivest–Shamir–Adleman), and Diffie-Hellman are widely utilized to ensure the confidentiality, integrity, and authentication of digital data. These encryption algorithms form the backbone of secure communication, effectively safeguarding sensitive information from unauthorized access during transmission. Complementing these cryptographic methods, steganography—particularly the Least Significant Bit (LSB) technique—is employed to embed secret data within multimedia files such as images or audio. This technique conceals information in such a way that the cover medium remains visually or audibly unchanged, thereby avoiding suspicion and adding a layer of covert communication to the security framework.

2.2 Limitations of Existing work

- **Vulnerability to Attacks:** Many existing LSB-based steganographic methods are prone to detection through statistical or image processing attacks.
- **Low Robustness:** Techniques often fail under common operations like compression, resizing, or noise addition, leading to data loss or corruption.
- **Static Embedding Methods:** Use of fixed or predictable patterns (e.g., basic LSB) reduces security and increases the chance of steganalysis.
- **Limited Integration:** In several cases, cryptography and steganography are used independently, rather than in a tightly integrated, mutually reinforcing manner.
- **Complexity in Key Management:** Some methods that include encryption rely on complex key generation or distribution schemes, making them harder to implement securely.
- **Capacity vs. Quality Trade-off:** Many techniques struggle to hide large amounts of data without degrading the image quality, affecting PSNR and visual imperceptibility.
- **Limited Real-world Testing:** Many methods are evaluated only on theoretical or ideal conditions without testing robustness in diverse real-world scenarios.
- **Poor Adaptability:** Existing systems often lack flexibility to adapt to different types of images, file formats, or varying message sizes.

System	Focus	Methodology	Limitations
A new approach for data hiding in graylevel images (2008)	Data embedding in grayscale images	Block-based LSB embedding	Limited security scope
Sequential Colour Cycle Algorithm (2010)	Improved LSB image steganography	Sequential color cycle for multi-LSB embedding	Limited payload, integration difficulty
LSB-based image steganography using secret key (2012)	Secure steganography image	Secret key-based LSB position shifting	Sensitive to processing, less robust
Pixel Indicator with Randomization (2012)	Watermarking and authentication	RGB pixel indicator with randomized key	Limited real-world testing
Improved LSB Technique for RGB Images (2013)	Steganography withspixel random embedding	Random pixel-based LSB with edge detection	Lacks extraction clarity, may reduce image quality

Table 2.2.1: Comparative Analysis

CHAPTER -3

SOFTWARE REQUIREMENTS SPECIFICATION

3.1 Overall Description

The project aims to enhance the confidentiality and security of digital communication by blending two major data protection techniques: cryptography and steganography, into a unified approach termed Crystography. The system first encrypts the user's message using a custom symmetric key encryption algorithm, which employs dynamically generated private keys. Once encrypted, the ciphertext is embedded within a cover image using a novel 3-3-2 Least Significant Bit (LSB) insertion method, leveraging human visual perception limits to ensure that the hidden data remains imperceptible.

The system is developed using Python, with a graphical user interface (GUI) built in Tkinter, offering functionalities for encryption, decryption, stego image generation, message embedding, and extraction. Users can either enter text directly or upload a .txt file to be securely hidden in a .png image. During decryption, the application retrieves and verifies the embedded encrypted data using the stego key and private key before reconstructing the original message. The software also includes error handling to ensure usability and robustness.

The target users of this application include individuals or organizations needing secure imagebased data transmission, especially in contexts such as secure messaging, digital watermarking, rights management, and covert communication. This software runs on Windows 8 or higher and Mac OS X v10.7 or above, requiring minimal hardware (2GHz processor, 4GB RAM, 256GB HDD). The development environment includes Python IDLE and dependencies such as PIL, hashlib, os, pathlib, and random.

In summary, this software provides a practical, efficient, and secure means of embedding and retrieving encrypted data in images, addressing modern challenges in data privacy and secure communication.

3.2 Operating Environment

Software Requirements

Operating System	:	Windows 8 or above, MAC OS X v10.7 or higher
Front End	:	Tkinter
Language	:	Python
Modules	:	Hashlib PIL,Pathlib, random ,os
Development Tools	:	Python IDLE

Hardware Requirements

Processor	:	Recommended 2Ghz or more
RAM	:	Recommended 4 GB or more
Hard Disk	:	Recommended 256 GB or more

3.3 Functional Requirements

The Functional Requirements of the system must encrypt a secret message using symmetric key cryptography and embed it into an image using a 3-3-2 LSB steganography technique. It should allow users to upload text or image files, generate and save stego images, and later retrieve and decrypt the hidden message securely.

Cryptography Functional Requirements

Private Key Generation

- Generate a random position within the message.
- Extract ASCII value of the character at that position.
- Convert to binary and choose a 4-bit key where value ≥ 8.
- Retry the above steps if no suitable key is found.

Symmetric Encryption Algorithm

- For each character in the message:

- Convert to ASCII.
- Divide ASCII by key to get quotient and remainder. Represent quotient and remainder in 4-bit binary.
- Concatenate and reverse 8-bit binary string. Output is the ciphertext.

Decryption Algorithm

- Reverse the 8-bit encrypted bits.
- Extract 4 MSB and 4 LSB.
- Multiply MSB by key, add LSB.
- Convert result to ASCII to recover original character.

Steganography Functional Requirements

Image Processing and LSB Embedding

- Accept only .png images as cover images.
- Implement 3-3-2 LSB embedding:

    Embed 3 bits in Red channel.
    Embed 3 bits in Green channel.
    Embed 2 bits in Blue channel.

- Use cyclic bit positions within LSBs for increased security.

Encoding Steps

- Upload cover image and secret message (.txt or text input).
- Embed encryption metadata (key, stego key, message length) in the last row of the image.

- Compute bit positions and embed message bits in RGB channels accordingly.
- Generate and preview the stego image.

Decoding Steps

- Load stego image.
- Extract embedded keys and message length.
- Identify initial pixel using stego key.
- Use cyclic bit positions to extract message bits.
- Reconstruct ciphertext and decrypt using the embedded private key.

GUI Functional Requirements (Tkinter-based)

Encryption Tab

- Upload cover image.
- Enter message manually or upload a .txt file.
- Generate and display stego image.
- Save stego image locally.

Decryption Tab

- Upload stego image.
- Extract and display hidden message.
- Save extracted message as a .txt file.

Input Validations

- Show popup if:

- Cover image is not uploaded.
- Message input is missing.
- No stego image is selected for decryption.
- No hidden message is found in image.

Testing and Validation Requirements

Test Case Functionality

- Upload text/image inputs.
- Validate correct encryption, embedding, extraction, and decryption.
- Display PSNR, MSE, or error messages when applicable.

System Requirement-Specific Functionalities

File Handling

- Accept .png for images and .txt for text files.
- Read and write file content using GUI file dialogues.

Error Handling

- Handle cases where the image lacks capacity for embedding.
- Gracefully manage missing or corrupted files.

3.4 Non-Functional Requirements

The non-functional requirements the system should satisfy are:

1. **Performance**
   - Must generate and extract stego images within 3 seconds for standard size inputs (up to 1 MB).
2. **Usability**
   - Simple, intuitive GUI built with Tkinter.

3. Reliability

- Application should not crash under normal use.
- Graceful handling of unexpected inputs and errors.

4. Portability

- Must run on Windows 8 or higher and Mac OS X v10.7 or higher**.**

5. Maintainability

- Modular code structure separating UI, encryption, decryption, and steganography logic.

6. Security

- Embeds encrypted data to ensure confidentiality even if stego image is intercepted.
- Uses 8-bit symmetric key for encryption.

CHAPTER - 4

DESIGN

4.1 System Architecture of Integrating cryptography and steganography for robust data protection in image:

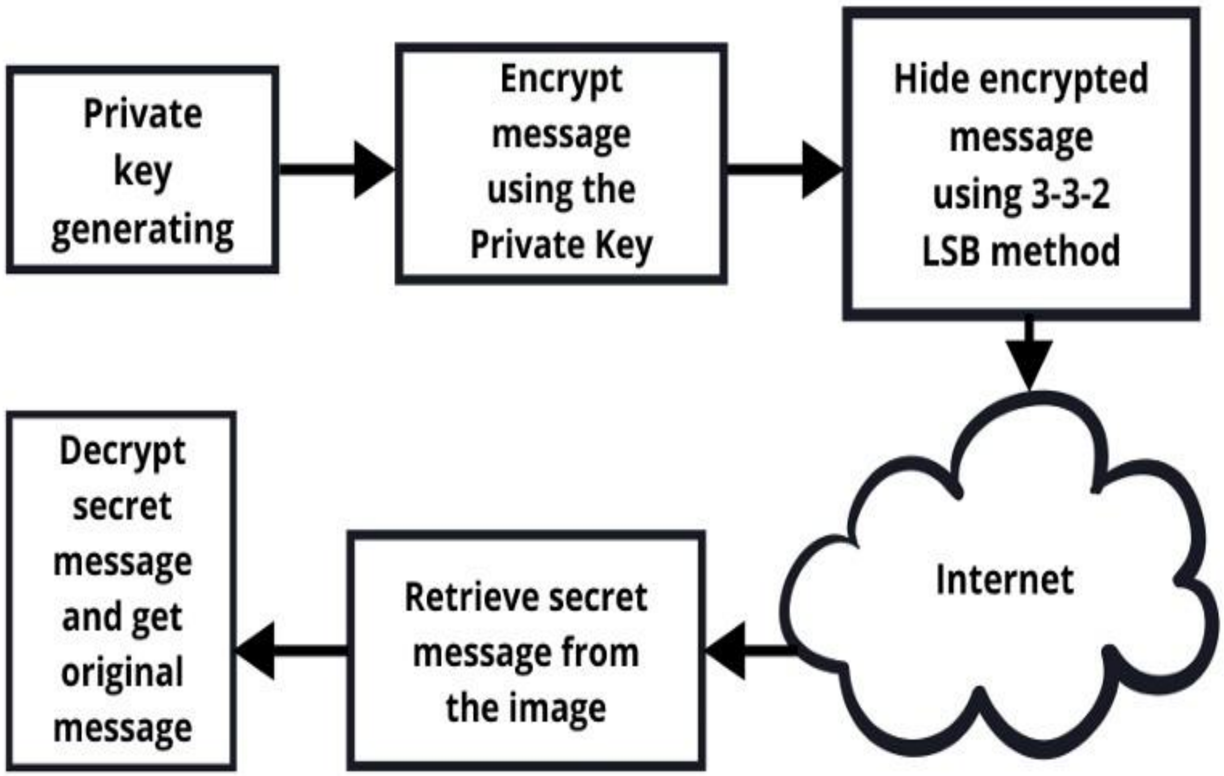

Fig 4.1: System Architecture

In this System Architecture Cryptography and Steganography are merged together. As shown in Figure 3.1, the system first encrypts the secret message by using a Symmetric Key Cryptography Algorithm by using the private key K generated by a key generating algorithm. Then in the second step, the secure encrypted secret message (ciphertext) is hidden as a payload inside the cover image by using a novel technique using the 3-3-2 LSB insertion method. For retrieval of the hidden message, the same reverse technique will be applied.

4.2 Class Diagram of Integrating cryptography and steganography for robust data protection in image:

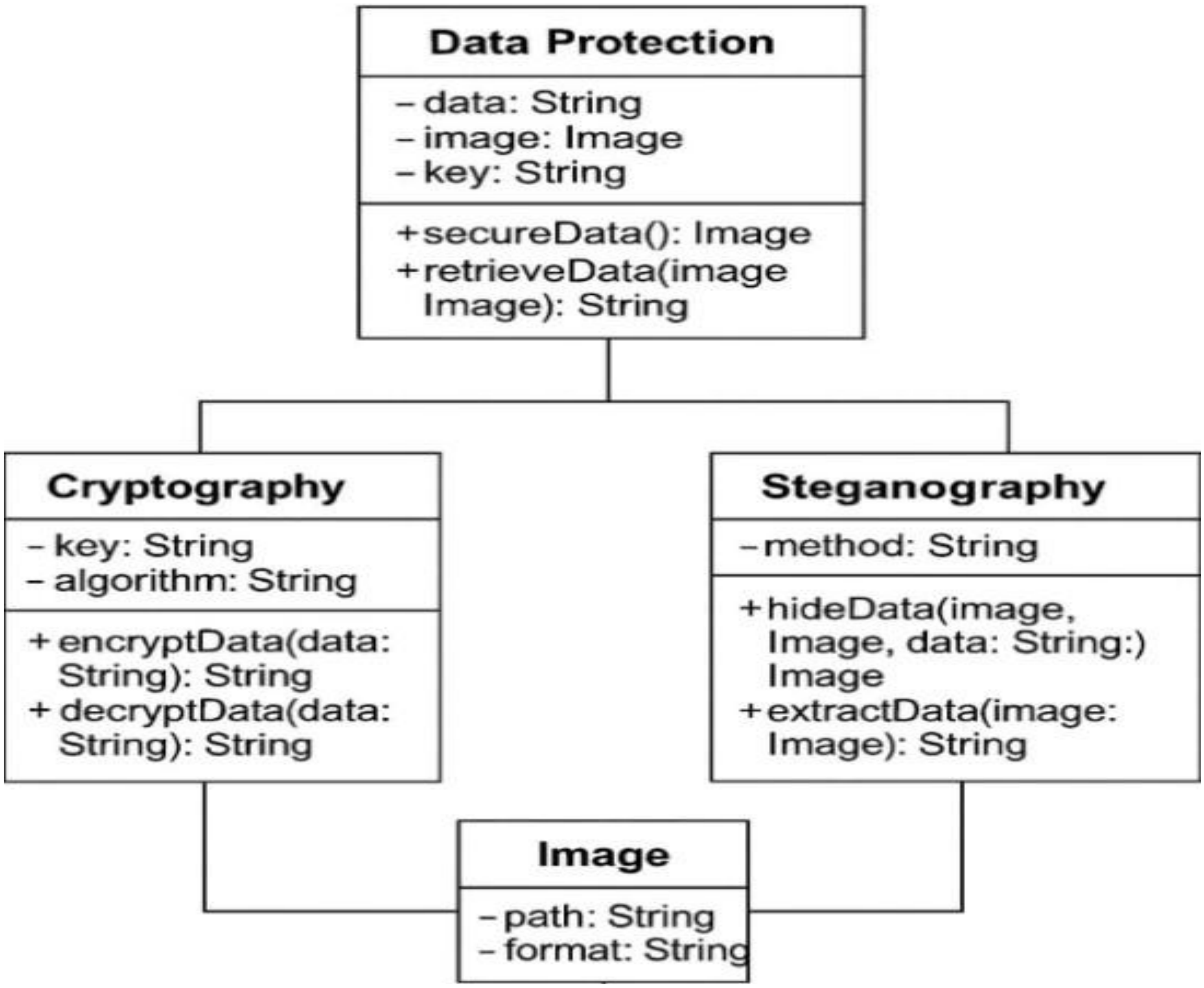

Fig 4.2: Class Diagram

The class diagram illustrates a data protection system that integrates **cryptography** and **steganography** techniques. The central class DataProtection has attributes data, image, and key, and defines two operations: secureData() and retrieveData(image). This class is associated with three other classes: Cryptography, Steganography, and Image.

4.3 Use Case Diagram of Integrating cryptography and steganography for robust data protection in image:

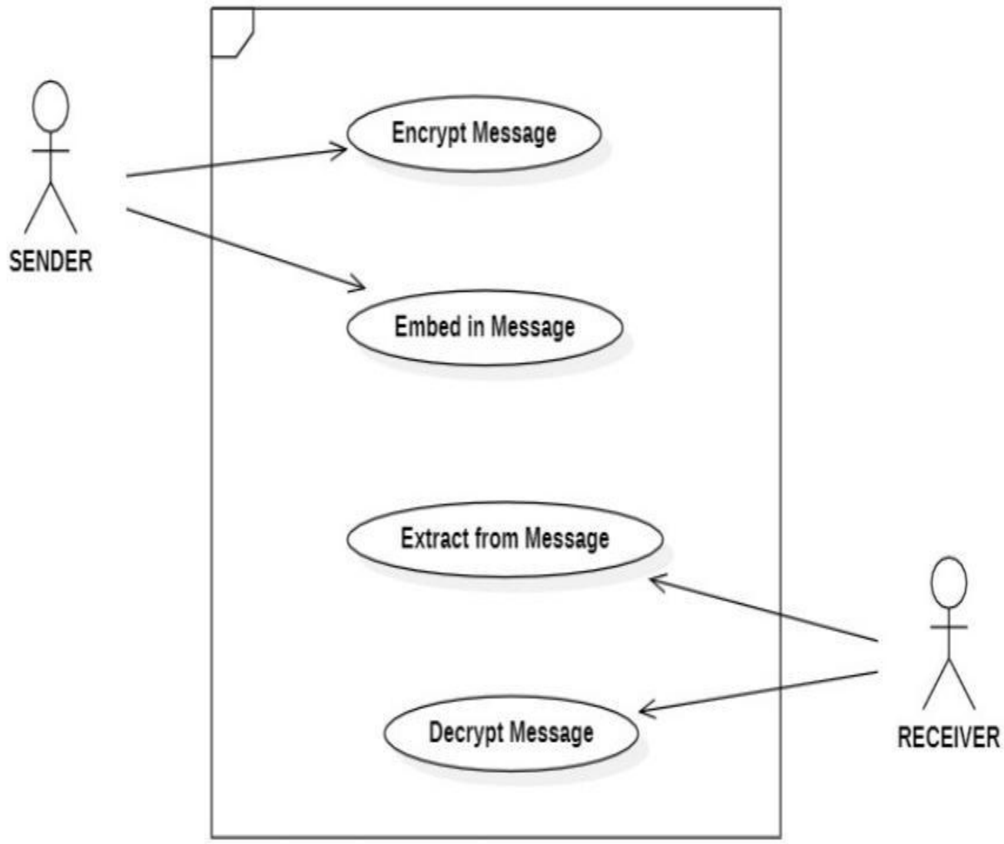

Fig 4.3: Use Case Diagram

This Use Case Diagram includes an actor ("User") and four use cases related to the crystography system: "Encrypt Message," "Decrypt Message," "Embed in Image," and "Extract from Image." The arrows indicate the flow of interaction between the user and the system, as well as the relationships between the use cases.

4.4 Sequence Diagram of Integrating cryptography and steganography for robust data protection in image:

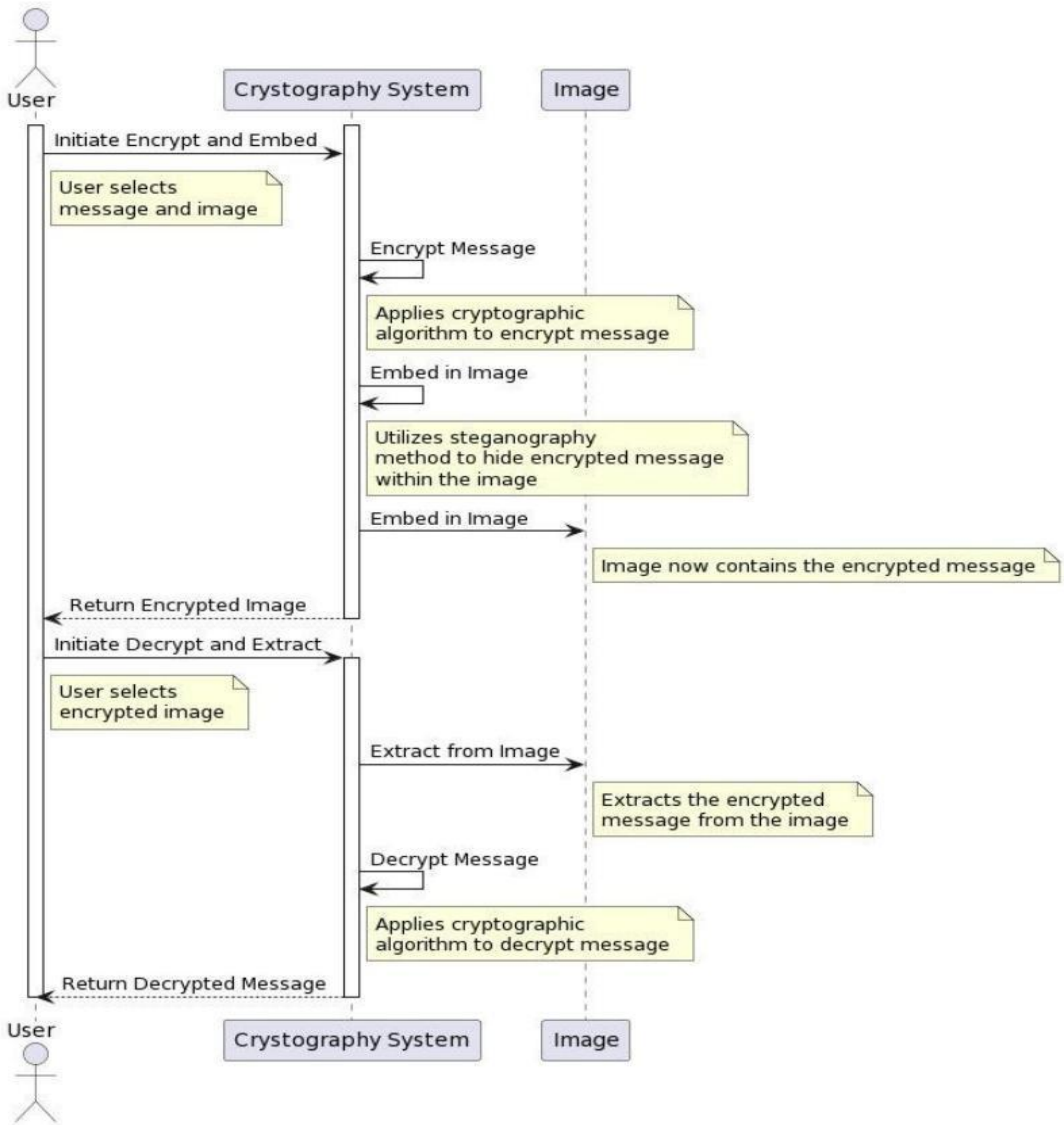

Fig4.4: Sequence Diagram

This sequence diagram in illustrates the flow of interactions between the User, Crystography System (CS), and Image. The steps include initiating encryption and embedding, followed by decrypting and extracting the message.

4.5 Flowchart of Integrating cryptography and steganography for robust data protection in image:

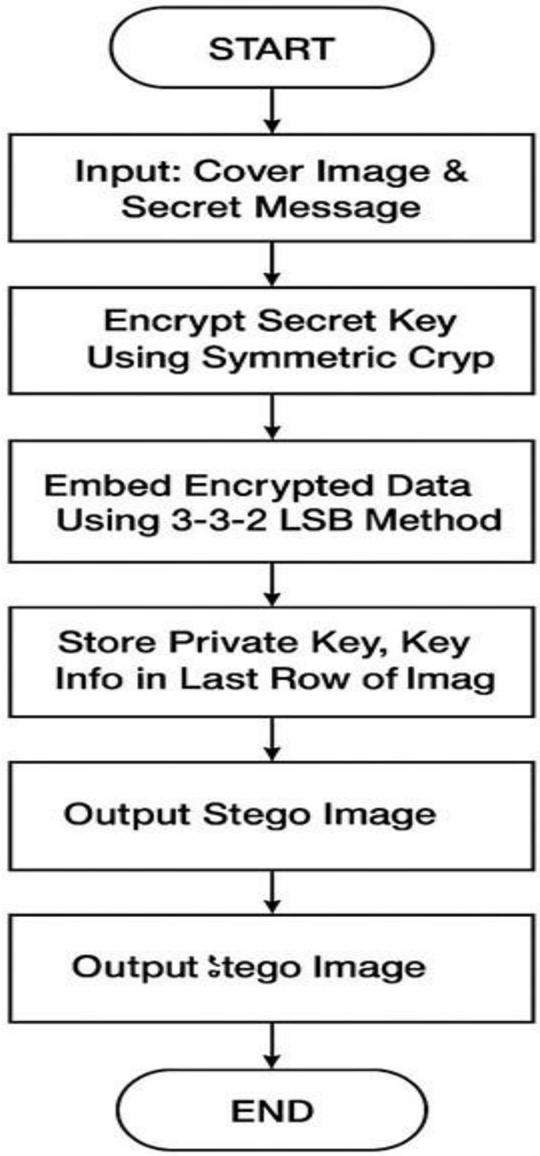

Fig 4.5: Flow Chart

The flow of the data protection system begins with the user providing the original data, a cover image, and an encryption key. The DataProtection class first uses the Cryptography module to encrypt the data, producing ciphertext. This encrypted data is then passed to the Steganography module, which embeds it into the image using a hiding method such as 3-3-2 LSB. The resulting stego image, containing the hidden encrypted message, is saved or transmitted securely.

CHAPTER - 5

IMPLEMENTATION

5.1 Sample Code

Application.py

```
from pathlib import Path from
steganography.steganography import Steganography
import tkinter as tk import
tkinter.ttk as ttk
from PIL import ImageTk, Image from
tkinter.messagebox import  from pathlib import Path
from steganography.steganography import
Steganography
# Application class has all functionalities of
GUI  class
    Application(Steganography):
    # Application intializer for all tabs
        def _init_(self):
        # Corrected constructor
            name  try:  self._buffer =
            None
        self.cover_filepath=none self.
            stego_filepath = None

            self._app = tk.Tk()
self.____app.geometry('600x360')
self._app.title("Major Project")
self._app.resizable(False, False)  #
Assign PhotoImage to an attribute
to prevent garbage collection
self.icon =
```

```
ImageTk.PhotoImage(Image.open('
images/title.ico'))

self._app.iconphoto(False, self.icon)


                self._tabs = ttk.Notebook(self._app)
                 # Corrected  self._app reference
                self._createEncryptionTab()
                self._createDecryptionTab()
                # self._createAboutTab()

 self._tabs.pack(expand=1, fill="both")
self._app.mainloop()
except Exception as e:
 messagebox.showerror("IncognitoBit", f"Oops! Some error has occurred:
             {str(e)}")

             self._app.destroy()

     # Create encryption tab
            Def
createEncryptionTab(self):
         self._encryption_tab = tk.Frame(self._tabs)
         self._tabs.add(self._encryption_tab,
         text="Encryption")  self._generateCanvas()
         self.___uploadCoverImageButton()      self.
         saveStegoImageButton()                     self.
         messageInput()
```

```
160, 300)  self._stegoGenerateButton()
self._exitButton(self._encryption_tab, 400,
300)


# Generate frames in encryption tab
def _generateCanvas(self):
self._bg_frame = tk.Frame(self._encryption_tab, bg="black", height=180,
width=598) self._bg_frame.pack() self.__bg_frame.pack_propagate(0) self.
bg_frame.place(x=1, y=2) self._cover_frame = tk.Frame(self._encryption_tab,
bg="white", height = 178, width=300) self.__cover_frame.pack() self.
cover_frame.pack_propagate(0) self.____cover_frame.place(x=2, y=3)
self._stego_frame = tk.Frame(self._encryption_tab, bg="white", height = 178,
width=292) self.__stego_frame.pack() self.__stego_frame.pack_propagate(0)
self._stego_frame.place(x=303, y=3)


# Action on clicking upload button in
encryption tab def
onClickuploadCoverImageButton(self):
self._buffer = str(filedialog.askopenfile(filetypes=[("PNG Images", "*.png")]))
if (self._buffer == "None" and self.cover_filepath == "None") or

self._buffer != "None": self._cover_filepath = self._buffer

else:  return self._cover_filepath =
self.cover_filepath[self._cover_filepath.find("'") + 1:]
self._cover_filepath =
self.cover_filepath[:self._cover_filepath.find("'")]
```

```
self._cover_img = Image.open(self._cover_filepath)

self._cover_img = self._cover_img.resize((298, 176))

self._cover_img =

ImageTk.PhotoImage(self._cover_img)

self._cover_panel   =   tk.Label(self.encryption_tab,   image=self._cover_img,
height=172,width=296)  self._cover_panel.place(x=2, y=3.5)

# Create upload cover image button in

encryption tab def

uploadCoverImageButton(self):

self._upload_img_btn   =   tk.Button(self.encryption_tab,   text="Upload
Image",  command=self._onClickuploadCoverImageButton)  self.
upload_img_btn.place(x=110, y=194)


# Action on clicking save stego image

button

def _onClickSaveStegoImageButton(self):

self._stego_filepath   =
str(filedialog.asksaveasfile(initialfile = 'stego.png',
defaultextension=".png",filetypes=[("PNG Images","*.png")]))  if
self._stego_filepath == "None":
    return
self._stego_filepath =

self.stego_filepath[self._stego_filepath.find("'") + 1:]

self._stego_filepath =

self.stego_filepath[:self._stego_filepath.find("'")]

self._stego_object._saveStegoImage(self._stego_filepath)

messagebox.showinfo("IncognitoBit", "Stego Image saved

successfully") self._onClickResetButton()
```

Create stego image button in

encryption tab def

saveStegoImageButton(self):

self._save_stego_img_btn = tk.Button(self.encryption_tab, text="Save Stego Image",
command=self._onClickSaveStegoImageBut

ton) self.

save_stego_img_btn.place(x=385, y=194) self.

save_stego_img_btn["state"] = DISABLED

Create message box for taking input from user in encryption tab def

messageInput(self): self._message_bg = tk.Frame(self._encryption_tab,

bg="black", height=54, width=486) self._message_bg.pack() self.

message_bg.pack_propagate(0) self __ padding_left = 109 self._padding_top =

235 self._message_bg.place(x=self._padding_left, y=self._padding_top)

self._message_label = tk.Label(self._encryption_tab, text="Enter the

Message") self._message_label.place(x=2.5, y=self._padding_top)

self._message = tk.Text(self._encryption_tab,
undo=True, height=2.5, width=60, wrap=WORD)
self._message.place(x=self._padding_left+1, y=self._padding_top+1)

Action on clicking text file upload

button def

onClickUploadTextFileButton(self):

self._text_filepath =

str(filedialog.askopenfile(filetypes=[("

Text Files", "*.txt")])) if self. text_filepath

== "None":

```
return self._text_filepath =

self.text_filepath[self._text_filepath.find("'") + 1:]

self._text_filepath =

self.text_filepath[:self._text_filepath.find("'")]

with open(self._text_filepath, 'r') as f:
   self._message.insert(1.0, f.read())

# Create text file upload button in encryption tab
def _uploadTextFileButton(self):
self._upload_txt_btn = tk.Button(self.encryption_tab, text="Upload Text",
command=self._onClickUploadTextFileButton) self._upload_txt_btn.place(x=16, y=260)

# Action on clicking reset
button

defonClickResetButton(self):

   self._app.destroy() self._init_()

# Create reset button in encryption and decryption tab def
resetButton(self, parent_widget, padding_left, padding_top):
self._reset_btn = tk.Button(parent_widget, text="Reset",
command=self._onClickResetButton)

   self._reset_btn.place(x=padding_left, y=padding_top)

# Stick stego image on stego panel label for

preview def _stickStegoImage(self):

   self._stego_img = ImageTk.PhotoImage(self._stego_object._stego_image.resize((298,
   176))) self._stego_panel = tk.Label(self.encryption_tab,
image=self._stego_img, height=172, width=287) self.
stego_panel.place(x=303, y=3.5)
```

```
# Action on clicking stego generate

button

defnClickStegoGenerateButton(self):

    if self._cover_filepath is None or self._cover_filepath == "None":
        messagebox.showerror("IncognitoBit", "Must upload a cover image")
        return
    if len(self._message.get(1.0, "end-1c")) == 0:
        messagebox.showerror("IncognitoBit", "Must enter some message or upload text file
        to be
hidden")
        return self._message.configure(state="disabled") self._upload_img_btn["state"]

    = DISABLED self._stego_generate_btn["state"] = DISABLED

        self._upload_txt_btn["state"] = DISABLED self._stego_object =

    Steganography(self.cover_filepath, self._message.get(1.0, "end-1c"))  self._status

    = self._stego_object._generateStegoImage() if self._status != "Stego Image

    generated successfully": messagebox.showerror("IncognitoBit",

    self._status)  self.

        onClickResetButton()

    else:  self._stickStegoImage()
        messagebox.showinfo("IncognitoBit", self.

        status)  self.

        save_stego_img_btn["state"] =

        NORMAL


# Create stego image generate button in

encryption tab def _stegoGenerateButton(self):

    self._stego_generate_btn = tk.Button(self.encryption_tab, text="Generate Stego
    Image",
command=self._onClickStegoGenerateButton)         self.

stego_generate_btn.place(x=235, y=300)  # Create exit
```

```
button in encryption and decryption tab def
exitButton(self, parent_widget, padding_left,
padding_top):

self._exit_btn = tk.Button(parent_widget, text="Exit", command=self._app.destroy) self.
exit_btn.place(x=padding_left, y=padding_top)
# Create decryption tab

defcreateDecryptionTab(

self):

self._decryption_tab = tk.Frame(self._tabs)

self._tabs.add(self._decryption_tab,

text="Decryption") self.

createDecryptionCanvas() self.

uploadStegoImageButton() self.

retrieveMessageButton() self.

createSaveTextButton() self.

createLoadingBar()

self._resetButton(self._decryption_tab, 172.5, 272.5)
self._exitButton(self._decryption_tab, 270, 272.5)


# Generate frames in decryption tab
def _createDecryptionCanvas(self):
self._stego_image_bg_frame =

tk.Frame(self._decryption_tab, background="black",

height=262, width=598) self.  stego_image_bg_frame.pack()

self. stego_image_bg_frame.pack_propagate(

0) self.

stego_image_bg_frame.place(x=1, y=1)

self._decrypted_text = scrolledtext.ScrolledText(self._decryption_tab, width=30,
height=16, wrap=CHAR)
```

```
self.
decrypted_text.configure(state=DISABLED)
self._decrypted_text.place(x=334, y=2)
self._stego_image_fg_frame  =
tk.Frame(self._decryption_tab, background="white",
height=260, width=331) self.  stego_image_fg_frame.pack()
self. stego_image_fg_frame.pack_propagate(
0)  self._stego_image_fg_frame.place(x=2,
y=2)


# Action on clicking retrieve message
button def
onClickRetrieveMessageButton(self):
if self._stego_filepath is None or self._stego_filepath == "None":
    messagebox.showerror("IncognitoBit", "Upload an image for decryption")  return
self._retrieve_btn.configure(state=DISABLED)
self.
upload_stego_img_btn.configure(state=DISABLE
D) self._stego_object =
Steganography(self._stego_filepath)  self._original_msg
=
self.stego_object._retrieveMessage(self.decryption_tab, self._loading_bar) self.
decrypted_text.configure(state=NORMAL)
self._decrypted_text.insert(INSERT,
self._original_msg) self.
decrypted_text.configure(state=DISABLED)
```

```
if len(self._decrypted_text.get(1.0, "end-1c")) == 0:
    messagebox.showinfo("IncognitoBit", "No hidden message
    found") self._onClickResetButton()

else:
    messagebox.showinfo("IncognitoBit", "Message retrieved successfully")
    self._save_text_btn.configure(state=NORMAL)


# Retrieve message button in decryption tab
def _retrieveMessageButton(self):
self._retrieve_btn = tk.Button(self.decryption_tab, text="Retrieve Message",
command=self._onClickRetrieveMessageButton) self._retrieve_btn.place(x=355, y=272.5)



# Action on clciking stego image upload

button def

onClickuploadStegoImageButton(self):

    self._stego_filepath = str(filedialog.askopenfile(filetypes=[("PNG Images", "*.png")])) if
    self._stego_filepath == "None":
        return self._stego_filepath =

    self.stego_filepath[self._stego_filepath.find("'") + 1:]

    self._stego_filepath =

    self.stego_filepath[:self._stego_filepath.find("'")] self._stego_img

    = Image.open(self._stego_filepath) self._stego_img =

    self._stego_img.resize((327, 255)) self._stego_img =

    ImageTk.PhotoImage(self._stego_img)
self._stego_panel = tk.Label(self.decryption_tab, image=self._stego_img,
height=255, width=327) self._stego_panel.place(x=2, y=3)
```

```
# Create upload stego image button in

decryption tab def

uploadStegoImageButton(self):

   self._upload_stego_img_btn      =  tk.Button(self.decryption_tab,
             text="Upload            Image",
command=self._onClickuploadStegoImageButton)
    self._upload_stego_img_btn.place(x=35, y=272.5)



# Action on clicking save text

button def

onClickSaveTextButton(self):

   self._msg_filepath           =          str(filedialog.asksaveasfile(initialfile  =

       'message.txt', defaultextension=".txt",filetypes=[("Text Files","*.txt")])) if self.

msg_filepath == "None":

       return self._msg_filepath =

    self.msg_filepath[self._msg_filepath.find("'") + 1:]

    self._msg_filepath =

    self.msg_filepath[:self._msg_filepath.find("'")] with open(self.

    msg_filepath, 'w+') as f:

       f.write(self._decrypted_text.get(1.0, "end-1c"))
   f.close()
   messagebox.showinfo("IncognitoBit", "Message saved successfully")  self.
   onClickResetButton()

# Create save text button in

decryption tab def

createSaveTextButton(self):

self._save_text_btn            =

       tk.Button(self.decryption_tab,
```

```
text="Save   Message",

command=self._onClickSaveTextButt

    on) self.

    save_text_btn.place(x=490,

    y=272.5)

    self._save_text_btn.configure(state=DISABLED)


# Create loading bar in decryption tab
def _createLoadingBar(self):
     self._loading_bg_frame  = tk.Frame(self._decryption_tab,      background="red",
height=30,  width=600) self.

loading_bg_frame.pack() self.

loading_bg_frame.pack_propagate(

    0) self.

    loading_bg_frame.place(x=0,

    y=308)

    self._loading_bar = ttk.Progressbar(self._decryption_tab, orient="horizontal",
    length=592,
value=0) self.

    loading_bar.place(x=1.5,
    y=310) self __

    loading_bar['value'] = 0


# Application GUI Initializer
def main():
    Application()
```

Encryption.py

```
# Import required packages and

modules import random

class Encryption:
    # Encryption

   initializer def _init

   (self, msg): self.

   plain_text = msg #

   Generate private

   key def

   _getPrivateKey(sel

   f):

   # Iterate until private key is not

   generated while True:

      # Take a random integer between 1 and message

      length self._r = random.randint(1, len(self.

      plain_text))
     # Take the first 4 binary bits of that random number self.

      private_key = int(format(ord(self._plain_text[self._r]), "08b")[:4], 2)

      # If it is greater than 111 then take it as

      private key if self._private_key >= 8:

         break
     # Take the last 4 binary bits of that random number self._private_key

      = int(format(ord(self._plain_text[self._r]), "08b")[4:], 2)

     # If it is greater than 111 then take it as

private key if self._private_key >= 8: break
```

```
return self._private_key # Encrypt the plain text

def _encryptMessage(self, token):

  # Add token before the message to be

   embedded self._plain_text = token +

   self._plain_text self._encrypted_msg =

   []

   # Iterate for each character of plain text to be

   embedded for char in self._plain_text:

     # Convert each character to a specified code            temp  =
(format(ord(char)  //     self._private_key,  "04b")
 +  format(ord(char)   %   self._private_key,
    "04b"))[::-1]   # Append that code with
    encrypted message self.

     encrypted_msg.append(temp)

   return self._encrypted_msg
```

Decryption.py

```
# Decryption class has all functionalities for decryption of the cipher used in this project
class Decryption:

    def __init__(self):
        pass

    # Match received token with the original token
    def _matchToken(self, received_token, original_token, private_key):
        self.__private_key = private_key
        self.__token = []

        # Iterate for each 8-bit tuple of received token
        for binstring in received_token:
            # Reverse each tuple
            binstring = binstring[::-1]
            # Split the 8 bit tuple from middle
          # Then convert it into the corresponding character using private key
            self.__token.append(chr(int(binstring[:4], 2) * self.__private_key + int(binstring[4:], 2)))

        # Finally convert characters to get the token in string representation
        # Then compare it to original token
        if "".join(self.__token) == original_token:
            return True
        return False

    # Decrypt message from the encrypted text
    def _decryptMessage(self, msg):
        self.__encrypted_text = msg
```

```
        self.__plain_text = []

        # Iterate for each 8-bit tuple of received token
        for binstring in self.__encrypted_text:
            # Reverse each tuple
            binstring = binstring[::-1]
            # Split the 8-bit tuple from middle
            # Then conver it into the corresponding character using private key
            self.__plain_text.append(chr(int(binstring[:4], 2) * self.__private_key + int(binstring[4:],
2)))

        # Finally join characters to get the string representation of plain text
        return "".join(self.__plain_text)
```

Steganography.py

```
# Import required packages and modules
import os
import math
import random
import hashlib
import numpy as np
from PIL import Image
from pathlib import Path
from Encryption import Encryption
from Decryption import Decryption

# Custom exception for non-capable images
class ImageNotCapableError(Exception, Decryption):

    # Exception initializer
```

```
        def __init__(self, msg):
        self.__msg = msg

    # Exception message generator
    def __str__(self) -> str:
        return self.__msg


# Steganography class has all functionalities to embed a message inside an image
# Using the algorithm discussed in the README.md
class Steganography(Encryption):

    # initialization of encryption object state
    def __init__(self, path = None, msg = None):
        if path is not None and msg is not None:
            try:
                # Open cover image from the given path
                self.__cover_image = Image.open(path, 'r')
                # Extract height and width of image
                self.__cover_image_width, self.__cover_image_height = self.__cover_image.size
                # Extract the image array from the Image object
                self.__cover_image_arr = np.array(list(self.__cover_image.getdata()))
                # Get the mode of the cover image
                self.__cover_image_mode = self.__cover_image.mode
                # Get the no. of channels present in the cover image
                self.__cover_image_channel = len(self.__cover_image.mode)
                # Get the total no of pixels present in the cover image
                self.__cover_image_net_pixels = self.__cover_image_arr.size //
self.__cover_image_channel
```

```
        # Get stego key
        self.__getStegoKey()

        # Convert message into list of characters
        self.__plain_text = list(msg)

        # Create an object of Encryption class
        self.__encryption_object = Encryption(self.__plain_text)
        # Generate private key for encryption
        self.__private_key = self.__encryption_object._getPrivateKey()

        # Get authorization token
        self.__generateAuthorizationToken()

        # Store the no. of pixels required to hide the message
        self.__required_bits = (len(self.__plain_text) + 16) * 8

        # Get cipher from plain text
        self.__encrypted_msg = self.__encryption_object._encryptMessage(list(self.__token))
    except:
        pass
if path is not None and msg is None:
    try:
        self.__stego_image = Image.open(path, 'r')
        self.__stego_image_width, self.__stego_image_height = self.__stego_image.size
        self.__stego_image_arr = np.array(list(self.__stego_image.getdata()))
    except:
        pass
```

```
# Generate stego key from the image width
    def __getStegoKey(self):
        # Take a random number between 0 to cover image width as stego key
        self.__stego_key = random.randint(0, self.__cover_image_width)


    # Generate authorization token
    def __generateAuthorizationToken(self):
        # Do bitwise XOR between stego key and private key
        # Then take the string representation of the XOR outcome
        # Then take the first 16 characters of that MD5 hash as authorization token
        self.__token = hashlib.sha256(str(self.__private_key ^ self.__stego_key).encode('utf-
8')).hexdigest()[:16]


    # Use the last row of image to embed the necessary information
    # To decrypt the message at reciever's end
    # +--------------------+-----------+---------+
    # | message length | stego key | private key |
    # +--------------------+-----------+---------+
    # All data should be represented in the binary format
    def __getEncryptionPacket(self):
        self.__encryption_info = list(map(int, list(format(self.__private_key, '04b') +
format(self.__stego_key, f"0{self.__bits_required_for_stego_key}b") + format(self.__required_bits,
f"0{self.__bits_required_for_message_length}b"))))


    # hide encryption packet in the last row of cover image array
    def __hideEncryptionPacket(self):
        self.__bits_required_for_stego_key = math.ceil(math.log(self.__cover_image_width, 2))
        self.__bits_required_for_private_key = 4
```

```
self.__bits_required_for_message_length = self.__cover_image_width -
(self.__bits_required_for_stego_key + self.__bits_required_for_private_key)
        self.__getEncryptionPacket()
        index = 0
        i = -1
        inner_loop = False
        while True:
            second_for_loop = False
            for j in range(3):
                temp = list(map(int, list(format(self.__cover_image_arr[i][j], "08b"))))
                if j == 0:
                    for k in range(3):
                        if index == self.__cover_image_width:
                            inner_loop = True
                            break
                        temp[-(k + 1)] = self.__encryption_info[index]
                        index += 1
                elif j == 1:
                    if index == self.__cover_image_width:
                        inner_loop = True
                        break
                    temp[-4] = self.__encryption_info[index]
                    index += 1
                    for k in range(2):
                        if index == self.__cover_image_width:
                            inner_loop = True
                            break
                        temp[-(k + 1)] = self.__encryption_info[index]
```

```
                index += 1
            else:
                for k in range(2):
                    if index == self.__cover_image_width:
                        inner_loop = True
                        break
                    temp[-(k + 3)] = self.__encryption_info[index]
                    index += 1
            temp = list(map(str, temp))
            self.__cover_image_arr[i][j] = int("".join(temp), 2)
            if inner_loop:
                second_for_loop = True
                break
        if second_for_loop:
            break
        i -= 1

# hide plain text message within the cover image
def __hideEncryptedMessage(self):
    for i in range(self.__stego_key, self.__stego_key + (self.__required_bits // 8)):
        index = 0
        for j in range(3):
            temp = list(map(int, list(format(self.__cover_image_arr[i][j], '08b'))))
            if j == 0:
                for k in range(3):
                    temp[-(k + 1)] = int(self.__encrypted_msg[i - self.__stego_key][index: index + 1])
                    index += 1
            elif j == 1:
```

```
                temp[-4] = int(self.__encrypted_msg[i - self.__stego_key][index: index + 1])
                index += 1
                for k in range(2):
                    temp[-(k + 1)] = int(self.__encrypted_msg[i - self.__stego_key][index: index + 1])
                    index += 1
            else:
                for k in range(2):
                    temp[-(k + 3)] = int(self.__encrypted_msg[i - self.__stego_key][index: index + 1])
                    index += 1
            temp = list(map(str, temp))
            self.__cover_image_arr[i][j] = int("".join(temp), 2)


    def _generateStegoImage(self):
        try:
            # raise ImageNotCapableError("Image is not capable for hiding the message")
            self.__status = None
            if self.__cover_image_net_pixels - (self.__cover_image_width + self.__stego_key) <
self.__required_bits // 8:
                raise ImageNotCapableError("Image is not capable for hiding the message")
            self.__hideEncryptionPacket()
            self.__hideEncryptedMessage()
            self.__cover_image_arr  = self.__cover_image_arr.reshape(self.__cover_image_height,
self.__cover_image_width, self.__cover_image_channel)
            self._stego_image = Image.fromarray(self.__cover_image_arr.astype('uint8'),
self.__cover_image_mode)
            self.__status = "Stego Image generated successfully"
        except ImageNotCapableError as error:
            self.__status = error.__str__()
        except Exception:
```

```
self.__status = "Some error occured"
    finally:
        return self.__status

def _saveStegoImage(self, filepath):
    self._stego_image.save(Path(filepath))

def __getToken(self):
    self.__received_token = []
    for i in range(self.__stego_key, self.__stego_key + 16):
        binstring = ''
        for j in range(3):
            temp = list(format(self.__stego_image_arr[i][j], '08b'))
            if j == 0:
                for k in range(3):
                    binstring += temp[-(k + 1)]
            elif j == 1:
                binstring += temp[-4]
                for k in range(2):
                    binstring += temp[-(k + 1)]
            else:
                for k in range(2):
                    binstring += temp[-(k + 3)]
        self.__received_token.append(binstring)

def __retrieveEncryptedText(self):
    self.__encrypted_msg = []
    for i in range(self.__stego_key + 16, self.__stego_key + (self.__msg_length_in_bits // 8)):
```

```
            binstring = ''for j in range(3):
                temp = list(format(self.__stego_image_arr[i][j], '08b'))
                if j == 0:
                    for k in range(3):
                        binstring += temp[-(k + 1)]
                elif j == 1:
                    binstring += temp[-4]
                    for k in range(2):
                        binstring += temp[-(k + 1)]
                else:
                    for k in range(2):
                        binstring += temp[-(k + 3)]
            self.__encrypted_msg.append(binstring)
            progress = (i - self.__stego_key + 16 + 1) // ((self.__msg_length_in_bits // 8) / 100)
            self.__makeProgress(progress)

    def __getEncryptionInfo(self):
        self.__encryption_info = []
        index = 0
        i = -1
        inner_loop = False
        while True:
            second_for_loop = False
            for j in range(3):
                temp = list(format(self.__stego_image_arr[i][j], "08b"))
                if j == 0:
                    for k in range(3):
                        if index == self.__stego_image_width:
```

```
        inner_loop = True
            break
          self.__encryption_info.append(temp[-(k + 1)])
          index += 1
    elif j == 1:
        if index == self.__stego_image_width:
          inner_loop = True
          break
        self.__encryption_info.append(temp[-4])
        index += 1
        for k in range(2):
          if index == self.__stego_image_width:
            inner_loop = True
            break
          self.__encryption_info.append(temp[-(k + 1)])
          index += 1
    else:
        for k in range(2):
          if index == self.__stego_image_width:
            inner_loop = True
            break
          self.__encryption_info.append(temp[-(k + 3)])
          index += 1
    if inner_loop:
        second_for_loop = True
        break
  if second_for_loop:
    break;  i -= 1
```

```
self.__private_key = int("".join(self.__encryption_info[:4]), 2)
        temp = math.ceil(math.log(self.__stego_image_width, 2))
        self.__stego_key = int("".join(self.__encryption_info[4: temp + 4]), 2)
        self.__msg_length_in_bits = int("".join(self.__encryption_info[temp + 4:]), 2)

    def __makeProgress(self, value):
        self.__loading_bar['value'] = value
        self.__root.update_idletasks()

    def _retrieveMessage(self, masterwidget, loadingbar):
        try:
            self.__root = masterwidget
            self.__loading_bar = loadingbar
            self.__makeProgress(0)
            self.__getEncryptionInfo()
            if self.__private_key < 8 or self.__stego_key > self.__stego_image_width:
                return ''
            self.__generateAuthorizationToken()
            self.__getToken(
            self.__decryption_object = Decryption()
            if not self.__decryption_object._matchToken(self.__received_token, self.__token,
    self.__private_key):
                return ''
            self.__retrieveEncryptedText()
            self.__decryptedText = self.__decryption_object._decryptMessage(self.__encrypted_msg)
            self.__makeProgress(100)
            return self.__decryptedText
        except:
```

return "“

init_.py

Driver program to start the application

Import main

from _init_.py

from Application import main

Call main function to start the application main()

CHAPTER - 6

TESTING

The testing phase of the project *"Integrating Cryptography and Steganography for Robust Data Protection in Images"* was conducted through a series of functional test cases designed to validate the encryption, embedding, decryption, and error-handling mechanisms of the application. The first test case involved uploading a .png image and manually entering a message, which was then encrypted using a symmetric key cryptographic algorithm and embedded into the image using the 3-3-2 Least Significant Bit (LSB) insertion technique. The resulting stego image was successfully generated, saved, and later used to retrieve and decrypt the hidden message accurately. In the second test case, a .txt file was uploaded instead of typing a message. The text was encrypted, embedded into the image, and the message was later retrieved and saved successfully, confirming the functionality for file-based input. Several negative test scenarios were also considered. These included attempts to proceed without uploading a cover image, without providing a message or file, and attempting to decrypt an image without hidden content. Each of these cases triggered appropriate popup alerts such as "Must upload cover image" or "No hidden message found," demonstrating the robustness of the error-handling system. Overall, the testing phase validated that the application performs all intended functions correctly, maintains data confidentiality, and provides user-friendly feedback in case of errors.

6.1 Test Cases

Test Cases	Objective	Input	Expected Result	Status
Upload Valid Image and Text	Embed encrypted text into an image	PNG image + Text	Stego image generated successfully and displayed	Pass/Fail
Upload Text File with Image	Encrypt and embed a .txt file	PNG image + .txt file	Stego image created with embedded file	Pass/Fail
Upload Without Image (Encryption Tab)	Validate mandatory image upload	No image	Error popup: "Must upload cover image"	Pass/Fail
Upload Without Message	Validate message input is required	Image but no text or file	Error popup: "Must upload some message or upload a text file"	Pass/Fail
Upload Invalid Image for Decryption	Handle image without hidden data	PNG without stego content	Alert: "No hidden message found"	Pass/Fail

Table 6.1: Test Cases

CHAPTER–7

RESULTS

7.1 Uploading the image and text

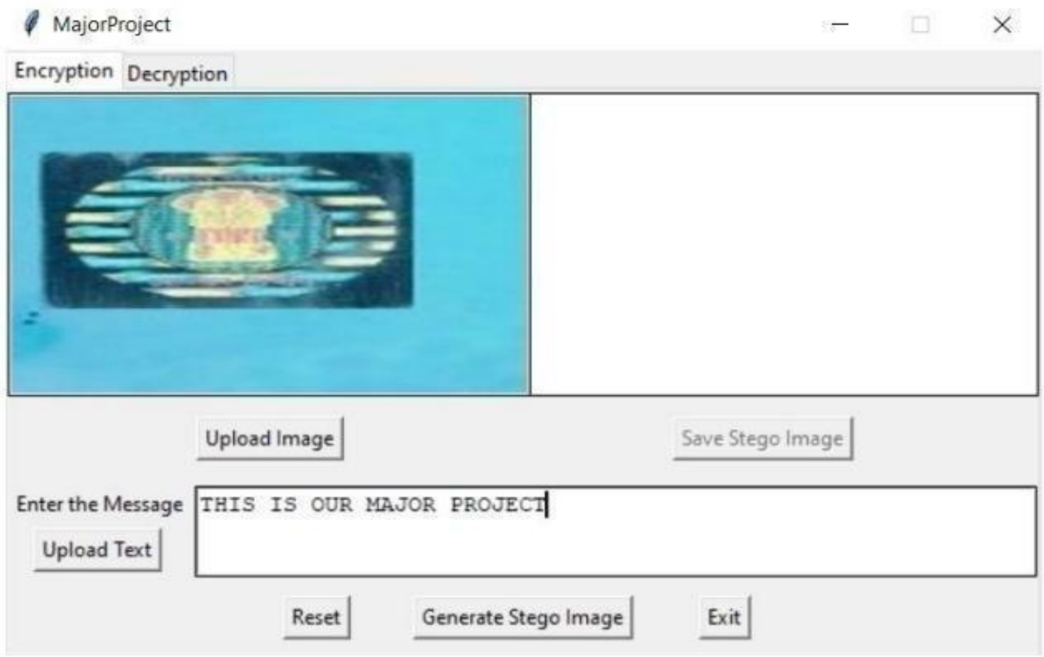

Fig 7.1: Uploading the image and text

1. In the user interface, both the image and the message to be embedded should be uploaded as input, as shown in Figure 7.1.
2. Images are uploaded Using “Upload Image” button. Only .png images can be uploaded.

7.2 Encrypting and storing message within the image

Fig 7.2: Encrypting and storing message within the image

3. On clicking the "Generate Stego Image" button, the uploaded text is encrypted and then embedded inside the image. The stego image, which hides the encrypted message, is displayed on the other side, as illustrated in Figure 7.2.

4. In Figure 5.3, the stego image hiding the encrypted message is generated successfully.

7.3 Generating and saving Stego image

Fig 7.3 Generating and saving Stego image

5. Stego Image can be saved using the "Save Stego Image" button. Figure 5.4 illustrates saving the previously generated stego image.

7.4 Extracting the hidden message from the stego image

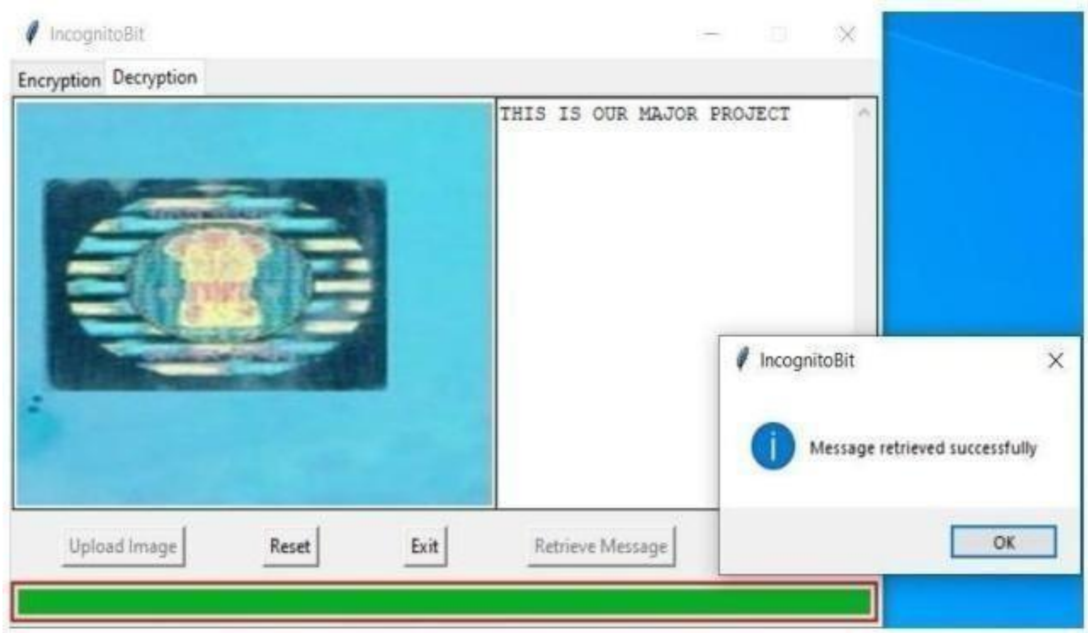

Fig 7.4: Extracting the hidden message from the stego image

6. To extract the previously embedded message from the stego image, Go to the Decryption tab and upload the saved stego image in the decryption tab. The 'Retrieve message' button decrypts the hidden message within the image. The decrypted message is then displayed on the other side, accompanied by a TopLevel widget with the text "Message retrieved successfully" as illustrated in Figure 7.4.

7.5 Extracting the hidden .txt file from the stego image & saving image

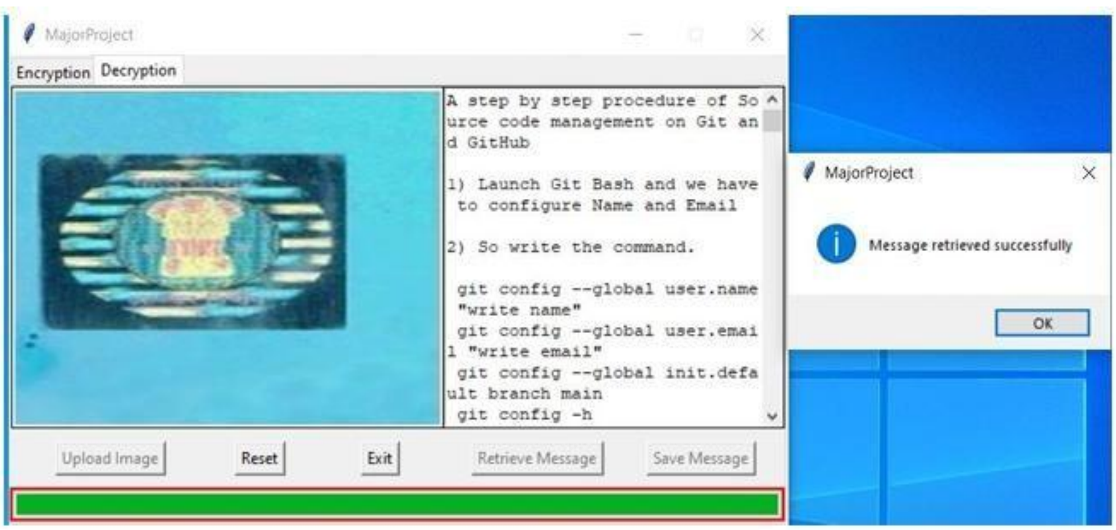

Fig 7.5: Extracting the hidden .txt file from the stego image

7. To extract the previously embedded message from the stego image, Go to the Decryption tab and upload the saved stego image in the decryption tab. The 'Retrieve message' button decrypts the hidden message within the image. The decrypted message is then displayed on the other side, accompanied by a TopLevel widget with the text “Message retrieved successfully” as illustrated in Figure 7.5.

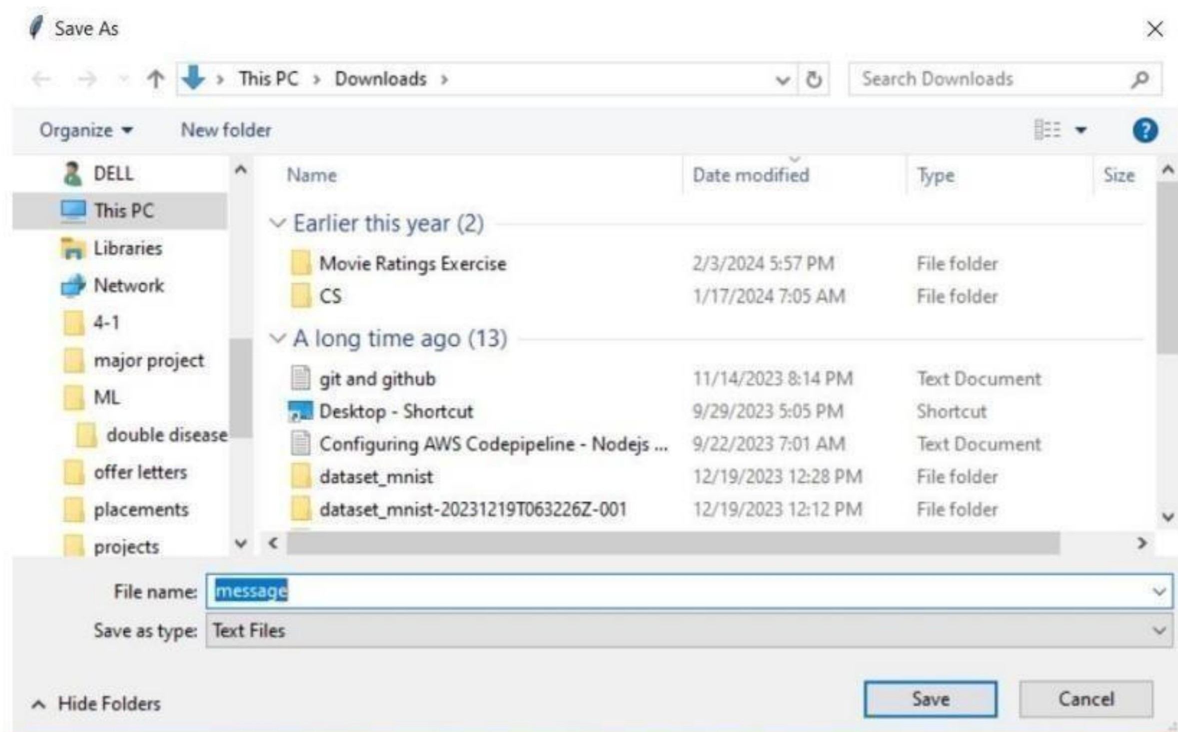

Fig 7.5: Saving the message

8. The retrieved message can be saved using “Save Message” button,
as shown in Figure 7.6.

CHAPTER - 8

CONCLUSION & FUTURE SCOPE

CONCLUSION

The proposed technique is a highly secure technique for embedding messages into images. Also, the symmetric key cryptographic algorithm used in this approach is very strong as it uses 8 bits key and a complex enciphering algorithm. It is almost computationally infeasible to retrieve the original message with a plain text attack. This technique also results in less distortion in an image after embedding. It has high PSNR (Peak Signal to Noise Ratio), less MSE (Minimum Squared Error), and it is imperceptible. This technique is also better than conventional LSB steganography. In this way, the system was strengthened using the LSB approach to provide a means of secure communication.

The strength of Steganography lies in the sheer amount of information that changes hands every day. It is very simple using digital technology to conceal any given digital information within other information, so virtually anything could contain a hidden meaning. There is no practical way to check it all. However, none of the steganography methods we examined could resist a concerted attack if someone knew that there was a message in a given document. For the greatest level of secrecy, a combination of both steganography and cryptography is necessary.

FUTURE SCOPE

- A strong cryptosystem can be built from the proposed method.
- A stronger cryptographic technique can be applied with the proposed steganographic technique in order to increase the security.
- Instead of single or double level; multilevel encryption can be applied with this technique to make the proposed method more secure.

CHAPTER – 9

REFERENCE & BIBLIOGRAPHY

[1] S. M. Masud Karim, M. S. Rahman and M. I. Hossain, "A new approach for LSB based image steganography using secret key," 14th International Conference on Computer and Information Technology (ICCIT 2011), 10.1109/ICCITechn.2011.6164800. Dhaka, Bangladesh, 2011, pp. 286-291, doi:

[2] Al-Taani, Ahmad & Al - Issa, Mohammed. (2008). A new approach for data hiding in gray-level images. 48-53.

[3] A. Singh and H. Singh, "An improved LSB based image steganography technique for RGB images," 2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, India, 2015, pp. 1-4, doi:
10.1109/ICECCT.2015.7226122.

[4] Farhan, Hamid & Alwan, Zena. (2018). Improved method using a two Exclusive-OR to binary image in RGB color image steganography.

[5] Zena Ahmed and M. Hamid Mohammed Farhan, "Secure Watermark Image Steganography by Pixel Indicator Based on Randomization", JMAUC, vol. 4, no. 2, pp. 101-110, Dec. 2012.

[6] S. Goyal, M. Ramaiya and D. Dubey, "Improved Detection of 1-2-4 LSB Steganography and RSA Cryptography in Color and Grayscale Images," 2015 International Conference on Computational Intelligence and Communication Networks (CICN), Jabalpur, India, 2015, pp. 1120-1124, doi: 10.1109/CICN.2015.220.

[7] Por, Yee & Beh Mei Yin, Delina & Ang, T.F. & Ong, Simying. (2013). An enhanced mechanism for image steganography using sequential color cycle Algorithm. International Arab Journal of Information Technology. 10.

[8] G. R., Manjula & Danti, Ajit. (2015). A Novel Hash Based Least Significant Bit (2-3-3) Image Steganography In Spatial Domain. International Journal of Security, Privacy and Trust Management. 4. 10.5121/ijsptm.2015.4102.