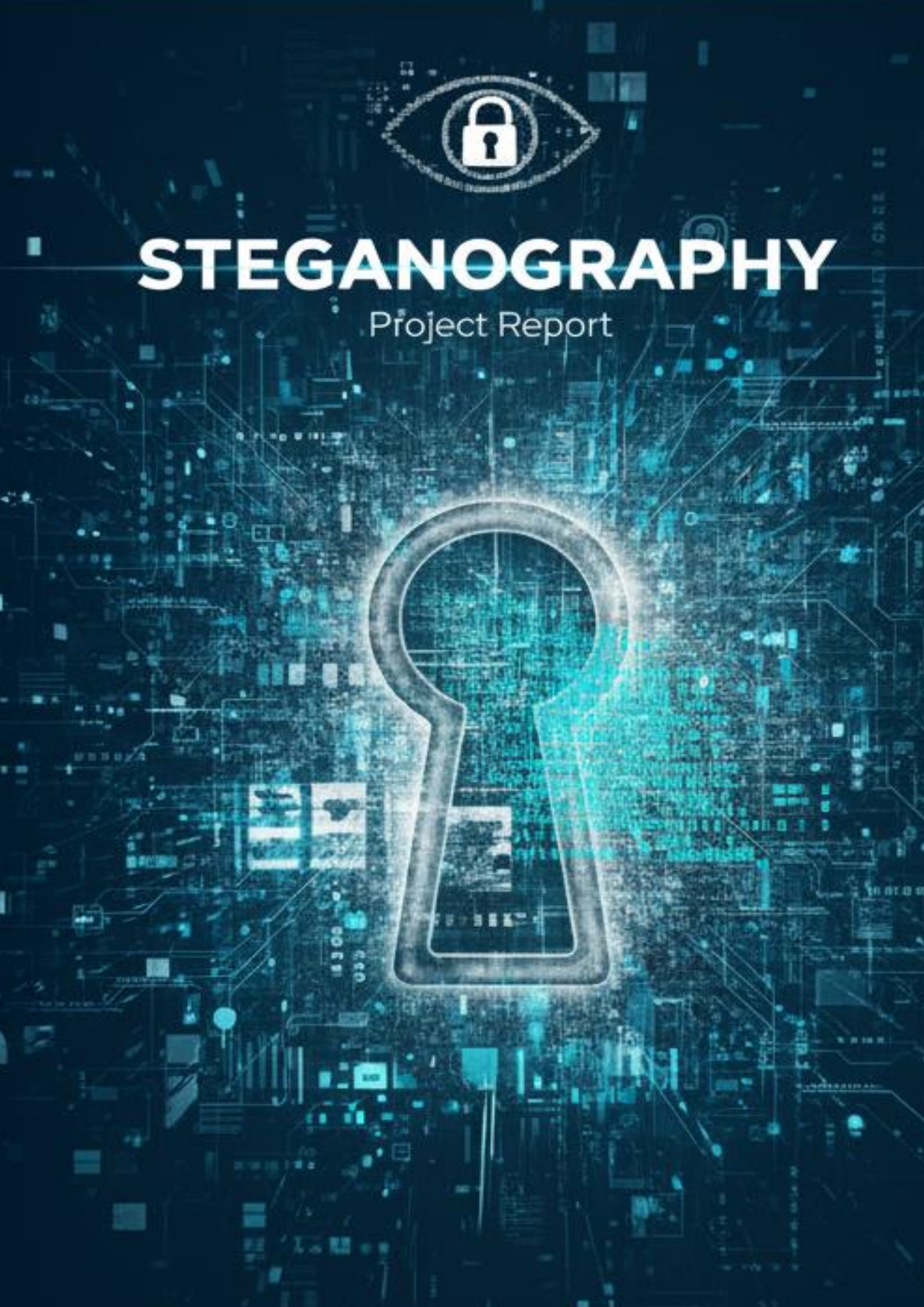

STEGANOGRAPHY
Project Report

TABLE OF CONTENTS

Abstract

Steganography is the art of concealing information within non-secret media to avoid detection. This project demonstrates a practical implementation of steganography in the C programming language, where secret data is securely hidden inside a 24-bit BMP image using the **Least Significant Bit (LSB) substitution method**.

The encoding process embeds the contents of a secret file into the pixel data of an image without noticeably altering its appearance, while the decoding process extracts and reconstructs the original hidden file. To ensure data integrity, a predefined **magic string** is encoded to verify the presence of hidden information.

This system successfully supports both **encoding and decoding operations**, with validation checks for file formats, error handling, and efficient use of image capacity. The output stego-image remains visually indistinguishable from the original, thus preserving confidentiality.

The project highlights the integration of **file handling, bit-level manipulation, and data security concepts**. It demonstrates how steganography can be applied for **secure communication, digital watermarking, and covert data storage**, making it a valuable contribution to the domains of information security and image processing.

Introduction

In today's digital era, secure communication has become a critical requirement due to the increasing risk of data breaches, cyberattacks, and unauthorized surveillance. While encryption ensures confidentiality by converting data into unreadable formats, it often attracts attention because the presence of encrypted data itself is visible. **Steganography** addresses this limitation by concealing secret information within ordinary media files, such as images, audio, or video, in a way that is visually or statistically undetectable.

This project focuses on implementing **image-based steganography** using the **Least Significant Bit (LSB) technique** in the C programming language. The LSB approach is chosen for its simplicity and effectiveness—it embeds secret data in the least significant bits of pixel values in a bitmap (BMP) image. Since changes in the LSBs do not produce perceptible variations in the image, the secret remains hidden from unintended viewers.

The tool developed supports both **encoding** (hiding secret data in an image) and **decoding** (retrieving hidden data from a stego image). A **magic string marker** is used to validate stego images before decoding, ensuring that data extraction is performed only when hidden content is present.

By combining **low-level file handling, bit manipulation, and error validation**, this project demonstrates how steganography can be practically applied for:

- Secure transmission of sensitive information
- Digital watermarking and copyright protection
- Covert communication in security-critical domains

This work serves as a foundation for further advancements, such as supporting compressed image formats (JPEG/PNG), integrating cryptography for double-layered security, and creating user-friendly graphical interfaces.

Objectives

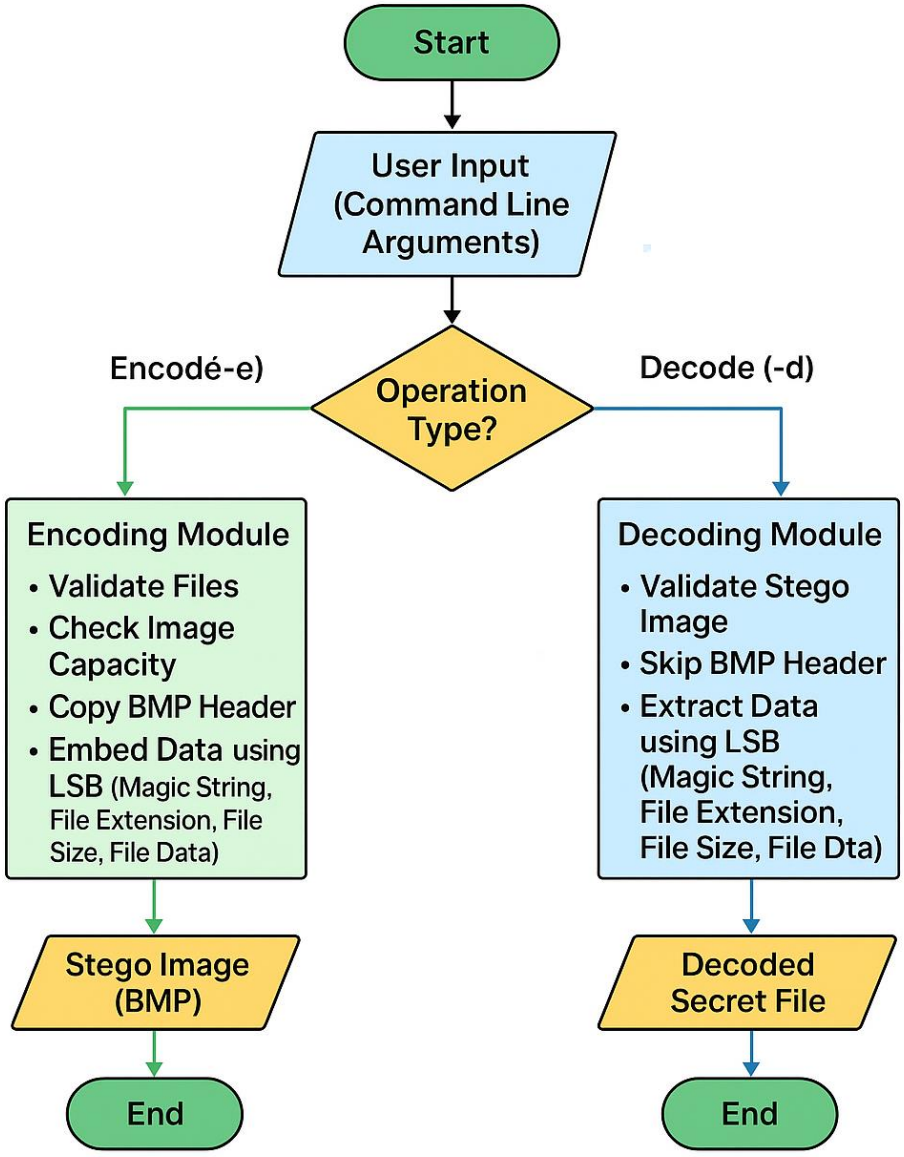
The primary objectives of this project are:

- To implement a steganography system in the C programming language using the Least Significant Bit (LSB) technique for hiding data within 24-bit BMP images.
- To develop an encoding module that securely embeds a secret text file into a BMP image without altering its visual quality.
- To design a decoding module capable of accurately extracting and reconstructing the hidden secret file from the stego image.

- To validate file formats and ensure data integrity through proper error handling, file extension checks, and the use of a predefined magic string marker.
- To demonstrate practical applications of steganography in secure communication, digital watermarking, and covert data storage.
- To provide a foundation for future enhancements, such as extending support to other image formats, integrating encryption for added security, and developing a user-friendly interface.

System Architecture

The system architecture of the steganography tool is designed to handle both **encoding** (data hiding) and **decoding** (data extraction) operations in a structured and modular way. The architecture is divided into the following major components:



The project consists of four main modules: Input, Encoding, Decoding, and Output.

The Input Module handles command-line arguments, validating file types and presence; for encoding, it requires a source BMP image and a secret file, while for decoding, it takes a stego BMP image.

The Encoding Module opens the necessary files and checks that the BMP image has sufficient capacity to store the secret data. It copies the BMP header and embeds the data using Least Significant Bit (LSB) technique in a specific sequence: a predefined magic string, secret file extension, file size, and the file contents, before copying the remaining image data unchanged to preserve appearance.

The Decoding Module opens the stego image, skips the header, and reads the embedded data, verifying the magic string, then extracting the secret file's extension, size, and contents to reconstruct the original file accurately.

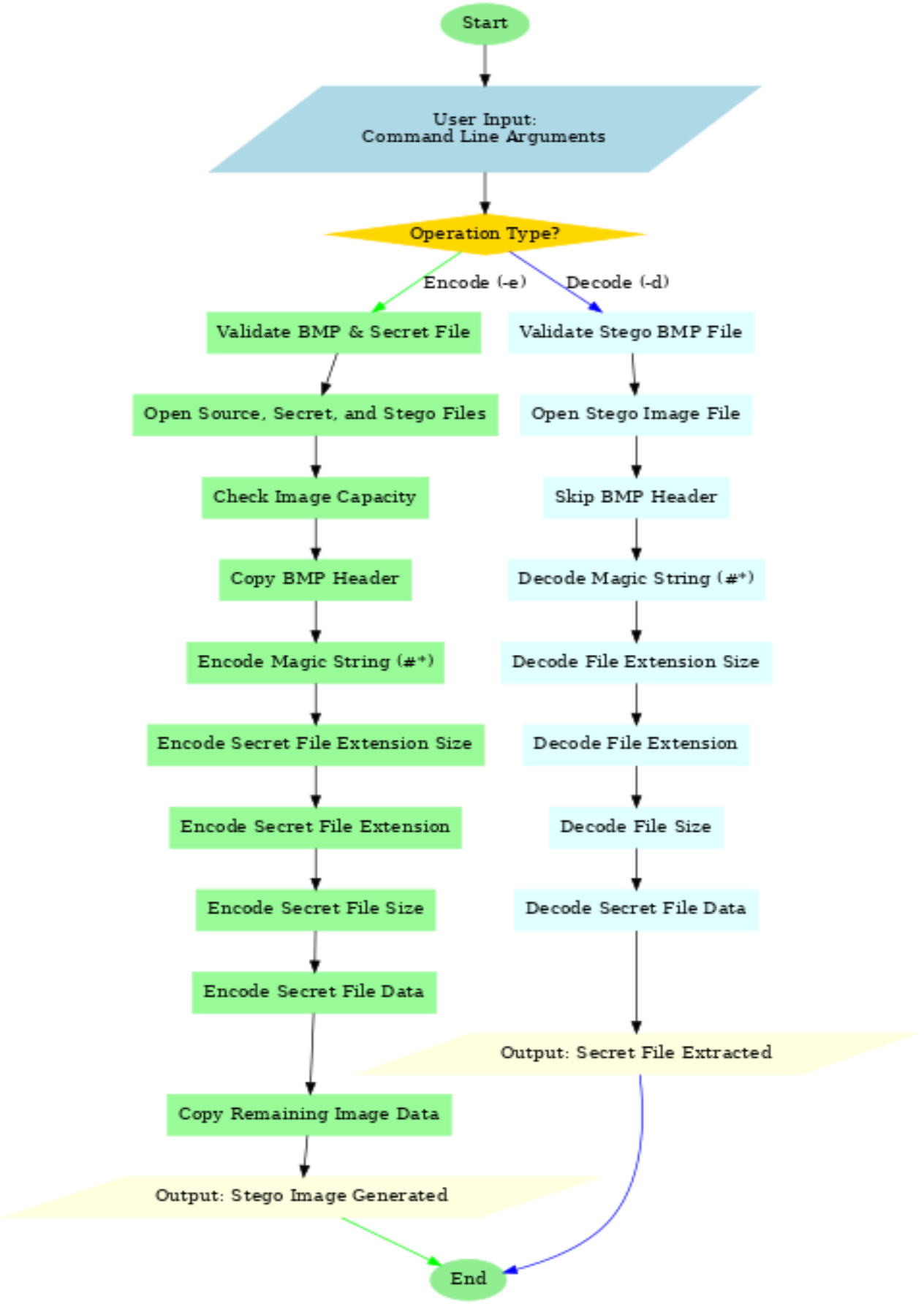
Finally, the Output Module generates either the stego BMP image in encoding or the original secret file in decoding, providing detailed console messages for successful execution or errors.

Requirements

Software Requirements:

- **Operating System:** Windows, Linux (Ubuntu), or macOS
- **Compiler:** GCC (GNU Compiler Collection) or any standard C compiler
- **Editor/IDE:** Code::Blocks, Visual Studio Code, Turbo C, Vim, or any text editor with terminal support
- **Terminal:** Command Prompt (Windows), Terminal (Linux/macOS), or an IDE-integrated console.

Workflow/Working



Visual workflow of the Steaganography Process

Encoding Workflow:

Goal: Hide a secret file inside a BMP image without altering the image visibly.

Step 1: Validate Input Arguments

- The program checks if the user has provided **exactly two files**:
    1. Source BMP image
    2. Secret file to embed
- Ensures the BMP image is of **valid format** (header and file signature).
- If validation fails → display an error and exit.

Step 2: Open Files

- Open **source BMP** in read mode.
- Open **secret file** in read mode.
- Create **stego image** (output) in write mode.
- Check that files opened successfully.

Step 3: Copy BMP Header

- BMP images have a **header of fixed size (usually 54 bytes)** containing metadata: width, height, color depth, etc.
- Copy the header **as-is** from source image to stego image to ensure the output is still a valid BMP.

Step 4: Encode Secret Data

The encoding is done using **Least Significant Bit (LSB) technique**, replacing the least significant bit of each byte in the image data.

Sequence of encoding

1. **Magic String (#*)**
    - Acts as a **marker** to identify the start of hidden data.
    - Encodes it character by character into LSBs of BMP pixel data.
2. **Secret File Extension Size**
    - Size of extension (e.g., .txt → 4 bytes)
    - Needed to reconstruct the file during decoding.
3. **Secret File Extension**
    - Stores the actual extension to reconstruct the original file type.
4. **Secret File Size**
    - Encodes the total number of bytes in the secret file.
    - Allows decoding program to know **how much data to read**.
5. **Secret File Data**
    - Reads the secret file **byte by byte** and embeds each bit into the LSBs of image data sequentially.

Step 5: Copy Remaining Image Data

- Any leftover BMP data after embedding the secret is copied **as-is** to the stego image.
- This ensures the image **looks visually identical** to the original.

Step 6: Close Files

- Close source BMP, secret file, and stego image.
- Encoding complete → output file: Encoded_Image.bmp

Summary of Workflow

Encoding:

Input → BMP Header → Magic String → Extension Size → Extension → File Size → File Data → Remaining Image → Output BMP

Decoding Workflow:

Goal: Extract the hidden secret file from a BMP image.

Step 1: Open Stego Image

- Open Encoded_Image.bmp in read mode.
- Check for successful file opening.

Step 2: Skip BMP Header

- Skip the first **54 bytes** (or actual header size) to start reading hidden data.

Step 3: Decode Magic String

- Read the first few bits to reconstruct the magic string.
- Verify it matches #*.
- If not → report **no hidden data found** and exit.

Step 4: Extract Secret File Metadata

1. **File Extension Size** → number of bytes for file extension
2. **File Extension** → .txt, .pdf, etc.
3. **Secret File Size** → total bytes of the secret data

Step 5: Extract Secret File Data

- Read **secret file size** bytes sequentially from BMP LSBs.
- Reconstruct the original file byte by byte.

Step 6: Write Output File

- Create a new file with the extracted extension, e.g., Decoded_File.txt.
- Write the reconstructed data into this file.

Step 7: Close Files

- Close stego image and output file.
- Decoding complete → the secret file is successfully recovered.

Summary of Workflow

Decoding:

Input BMP → Skip Header → Verify Magic String → Read Metadata → Extract Secret Data → Write Output File

Testing/Verification

The steganography tool was thoroughly tested to ensure that secret files can be accurately hidden and retrieved from BMP images without data loss or corruption.

1. Test Setup

- Source Image: beautiful.bmp
- Secret File: secret.txt containing My password is SECRET ;)
- Output Files:
  - Encoded_Image.bmp (after encoding)
  - Decoded_File.txt (after decoding)
- Environment: C program compiled using GCC on Linux/Windows, executed via command line.

2. Testing Steps

1. Encoding Test:
   - Input the source BMP image and secret file.
   - Execute the program in encode mode.
   - Check that the stego image is created successfully.
   - Verify that the BMP image remains visually unchanged.
2. Decoding Test:

- Input the stego image.
- Execute the program in decode mode.
- Confirm that the decoded file is generated with the correct name and extension.

3. Verification

- The decoded file content was compared with the original secret file:

  Original Secret: My password is SECRET ;)

  Decoded Secret: My password is SECRET ;)

- The contents match exactly, confirming that the encoding and decoding processes preserve the secret data.
- Console messages indicate successful encoding and decoding.

4. Additional Checks

- Tested with different file types (.txt, .c, .sh).
- Ensured the program handles large files within BMP capacity limits.
- Verified error handling for missing files, invalid formats, and insufficient image capacity.
-

Test Case	Operation	Output	Result
beautiful.bmp + secret.txt	Encoding	Encoded_Image.bmp	Success, image visually unchanged
Encoded_Image.bmp	Decoding	Decoded_File.txt	Success, content matches original
beautiful.bmp + script.c	Encoding	Encoded_Script.bmp	Success, file embedded correctly
Encoded_Script.bmp	Decoding	Decoded_Script.c	Success, original code recovered
beautiful.bmp + large file	Encoding	N/A	Failed if file exceeds BMP capacity

Sample Outputs

Encoding:

Fig1:Input Image

Fig2:Encoded Image

Decoding:

Fig1:Input Image

Output:

My password is SECRET ;)

Features

- Encode Secret Files – Hide any text file inside a BMP image using the LSB (Least Significant Bit) method.
- Decode Hidden Files – Extract the original secret file from a stego image.
- Magic String Verification – Ensures only valid stego images are decoded using #*.
- Supports Multiple File Types – Validates file extensions such as .txt, .c, .sh.
- Custom Output Names – Allows specifying output file names for both encoded and decoded files.
- Error Handling – Displays clear success, failure, and info messages in the console.
- Data Integrity – Ensures decoded content matches exactly with the original secret file.
- Simple & Efficient – Minimal dependencies, pure C implementation with fast encoding/decoding.
- Portable – Works on Windows, Linux, or macOS with any standard C compiler.

Advantages

- Secure Data Hiding: The tool successfully hides secret files within BMP images using LSB steganography, preventing casual detection.
- File Type Flexibility: Supports any file type (text, code, scripts) as long as the BMP has sufficient capacity.
- Visual Integrity: The stego image remains visually identical to the original BMP, ensuring minimal noticeable changes.
- Reliable Recovery: Magic string verification and metadata (file size, extension) allow accurate reconstruction of the hidden file.
- User-Friendly: Provides clear console messages for success and error notifications.

Limitations

- File Size Constraint: The secret file size is limited by the capacity of the BMP image. Large files may not fit.
- Image Format Restriction: Currently works only with BMP images; other formats like PNG or JPEG are not supported.
- Basic Security: LSB technique is susceptible to advanced steganalysis; no encryption is applied.

Conclusion

This project successfully implements steganography using the Least Significant Bit (LSB) technique in BMP images, enabling secure embedding and retrieval of secret text files without data loss. The tool validates input files, preserves image integrity, and ensures accurate decoding, demonstrating a robust encoding and decoding pipeline.

Through this project, key concepts in file handling, bit-level data manipulation, and low-level programming in C were applied effectively. The work also highlights practical applications in secure communication, digital watermarking, and covert data storage, while providing a strong foundation for exploring advanced steganography methods, encryption, and multimedia security.

Future Scope

This project can be extended and enhanced in several ways to increase its functionality and security:

- Support for additional image formats such as PNG or JPEG to broaden compatibility.
- Data encryption before embedding to provide an extra layer of security.
- GUI-based application for a more user-friendly experience.
- Batch encoding and decoding of multiple files for efficiency.
- Integration with cloud storage for secure remote data hiding and retrieval.
- Advanced steganography techniques like audio/video steganography for multimedia applications.

These improvements can make the tool more versatile, secure, and practical for real-world applications in cybersecurity, digital forensics, and confidential communication.

Source Code

👉 https://github.com/reddyganeshbathala/Steganography-LSB-BMP.git