Overview

This project implements a steganography library for hiding messages in BMP image files. Steganography is the practice of concealing messages within other non-secret data or a physical object. In this implementation, messages are hidden in the least significant bits (LSB) of pixel data in BMP images, making the changes imperceptible to the human eye.

Features

Message Hiding: Encode any binary data within BMP images
Message Extraction: Decode hidden messages from encoded images

Encryption Support: Optional password-based XOR encryption

Distribution Methods:

- Sequential bit embedding
- Pseudorandom bit distribution for increased security

Bit Position Selection: Configurable bit position (0-7) for embedding

Progress Monitoring: Callback function for displaying encoding/decoding progress

Error Handling: Comprehensive validation and error reporting

Components

The project consists of three files:

- `stego.h` : Header file defining the public API
- `stego.c` : Implementation of the steganography algorithms
- `main.c` : Command-line interface for using the library

To run the program the user needs to have

- a bmp image, meeting all requirements
- text file, containing secret message

Technical Details

To start working with the program, check if all required tools are available on the machine

Requirements

- C compiler (GCC recommended)

- Make
- BMP image files (24-bit, uncompressed)

Building from Source

1. Clone or download the source code
2. Open a terminal and navigate to the source directory
3. Run the make command:

```
make
```

This will compile the program and make exe file named `stegoapp`.

Encoding Process

1. Validates input BMP file format (24-bit, uncompressed)
2. Calculates available capacity based on image size
3. Prepares metadata with signature "STEG" and message size
4. Optionally encrypts message using XOR with provided password
5. Embeds metadata and message in pixel data using LSB technique
6. Writes the modified image to the output file

Decoding Process

1. Validates input BMP file format
2. Extracts metadata to verify signature and message size
3. Extracts hidden message bits from pixel data
4. Decrypts message using the provided password
5. Returns the extracted message

Storage Format

The embedded data consists of:

- **Signature (4 bytes):** "STEG" marker to identify valid steganographic images
- **Message Size (4 bytes):** Size of the hidden message
- **Seed (4 bytes):** Random seed (used for random distribution mode)
- **Message Data:** The actual message content

API Reference

Data Structures

`stego_distribution_t`

```
ctypedef enum {
    STEGO_SEQUENTIAL = 0,  // Use sequential bits in the image
    STEGO_RANDOM = 1       // Use pseudorandom bit positions
} stego_distribution_t;
```

`stego_config_t`

```
ctypedef struct {
    uint8_t bits_per_byte;        // Number of bits to use per byte of image
    uint8_t bit_position;         // Position of bit to use (0-7, with 0 being
LSB)
    stego_distribution_t distribution_mode; // Distribution mode
    char password[32];            // Encryption password (optional)
    status_callback_t status_callback; // Status callback function (optional)
} stego_config_t;
```

Functions

1.

```
void stego_init_config(stego_config_t *config)
```

- Initializes configuration with default values:
- 1 bit per byte
- LSB (bit position 0)
- Sequential distribution
- No encryption
- No status callbacks

2.

```
int stego_encode(const char *input_image, const char *output_image, const uint8_t
*message, size_t message_size, stego_config_t *config)
```

Encodes a message into an image. Returns 1 on success, 0 on failure.

3.

```
uint8_t *stego_decode(const char *input_image, size_t *message_size,
stego_config_t *config)
```

Decodes a message from an image. Returns the message buffer on success, NULL on failure.

4.

```
const char *stego_get_error()
```

Returns the last error message if an operation failed.

Command-Line Usage Basic Syntax

`/stego <operation> [options]`

Operations

encode: Encode a message into an image
decode: Decode a message from an image

Options

- -i : Input image file (BMP format)
- -o : Output file (image for encode, message for decode)
- -m : Message file to encode (encode mode only)
- -p : Password for encryption/decryption
- -b : Bit position to use (0-7, default: 0 = LSB)
- -r: Use random bit distribution

Examples Encoding a message:

Bash

```
./stego encode -i input.bmp -o output.bmp -m message.txt -p secret
```

Decoding a message:

```
./stego decode -i stego_image.bmp -o extracted_message.txt -p secret
```

Limitations

- Only works with 24-bit uncompressed BMP images

- Maximum message size is limited by image dimensions

The size of the message you can hide depends on the size of the image. Each pixel in a 24-bit BMP image has 3 bytes (R, G, B), and the tool uses one bit from each byte by default. The maximum message size can be roughly estimated as:

```
Max Message Size ≈ (Image Size in Bytes - 54) / 8
```

- Changes to image compression or format will corrupt the hidden message
- No error correction - pixel data modifications can lead to decoding failures
- XOR encryption provides only basic security - use strong passwords

Security Considerations

- The presence of steganography can be detected through statistical analysis
- Always use the random distribution mode (-r) for sensitive data
- Consider using a strong password for the XOR encryption
- For maximum security, consider adding additional encryption to your message before encoding
  File Size Limitations

Troubleshooting

Not a valid BMP file

- Make sure your image is a proper BMP file with 24-bit color depth
- Try converting your image using an image editor

Message too large for the image

- Use a larger image or reduce the message size
- Consider compressing your message file before encoding

Invalid steganography signature

- The input image doesn't contain a hidden message
- The password might be incorrect
- The bit position specified might be wrong

Notes

I had problems with random distribution, several trials and even assistance of AI didn't help me find the problem why during decoding there were issues with signature. The program is however fully functional when the option `-r` is excluded from encoding/decoding.