**Introduction**

In an age where digital communication is ever-present, ensuring the privacy and confidentiality of data has become more critical than ever. This project explores the concept of **secure steganography**, which involves hiding an encrypted message within an image in a way that the presence of the message is undetectable to an observer. Steganography on its own can conceal the existence of a message, but when combined with **encryption**, it offers a second layer of security: even if someone discovers the hidden data, they still cannot read it without the correct key.

This project implements a secure steganography system that hides encrypted messages inside BMP image files. The goal is to combine simple encryption with pixel-level data hiding, making the message both invisible and unreadable without the correct decryption key. The system utilizes the Least Significant Bit (LSB) method to hide bits of the message in 24-bit bitmap files, and XOR encryption for obscuring the actual message content. The application is fully written in C++. Additionally, it provides a practical demonstration of how two classic cybersecurity techniques - cryptography and steganography can be integrated into a single application for enhanced message protection.

**Program Interface**

The SecureSteganography program is operated via a terminal-based command-line interface and does not require any graphical environment. It is designed to be simple and self-explanatory for any user familiar with basic console input/output.

To compile the program, open a terminal in the project directory and run the following command:

g++ -o stegatool main.cpp securestega.cpp encryptor.cpp hideandseek.cpp steganography.cpp

This produces an executable named stegatool.

After launching, the user is guided step-by-step through the following input prompts:

* Entering the path of the input BMP file
* Creating an encryption key
* Choosing between encoding or decoding modes

The program terminates naturally once the chosen operation (encode or decode) is completed. It displays a confirmation or the result, and then exits without requiring any special command.

**Program Execution**

The program is operated via a command-line interface. It guides the user through encoding (hiding) or decoding (retrieving) a message inside a BMP image. The process is interactive and follows a simple step-by-step dialogue.

**Launching the Program**

After compiling the program as described in the *Program Interface* section, the user launches it by entering the following command in a terminal:

./stegatool

This starts the application and displays a prompt asking the user to provide input parameters.

**Loading the Image and Choosing the Operation Mode**

Once the program is launched, the user is first prompted to enter the path to the BMP image file they wish to work with:

Enter image file name (e.g., image.bmp):

After the user enters the image file name, the program loads the image and prints feedback messages such as:

Calculated buffer size: 11943936

Image loaded: output.bmp (11943936 bytes)

These confirm that the image file has been successfully read into memory and is ready for manipulation.

Then, the user is prompted to enter the encryption key:

Enter encryption key:

This key is required whether the user chooses to encode or decode. It is used to encrypt or decrypt the hidden message using XOR.

After these inputs, the user is asked to choose whether they want to encode or decode:

Do you want to encode or decode? (e/d):

* Typing e (for encode) starts the message-hiding process.
* Typing d (for decode) starts the message-retrieving process.
* Any other input results in an error message and ends the program.

Invalid option. Exiting.

**Encoding Mode: Hiding a Secret Message**

If the user chooses encoding mode by entering e, they are guided through the following steps:

**Step 1: Enter Image File Name**  
The user is prompted to input the name of the image file to be used:

Enter image file name (e.g., image.bmp):

The image must be a valid 24-bit uncompressed BMP file. If the file does not exist or is in the wrong format, the program will terminate with an error message.

**Step 2: Enter Encryption Key**

Enter encryption key:

The user provides a string that will be used to encrypt the message. The same key must later be used for successful decryption.

**Step 3: Enter Message to Hide**

Enter message to hide:

The user types the full message they want to conceal within the image. The message may include any characters supported by the ASCII encoding.

**Step 4: Completion Message**  
Once the message is encrypted and embedded within the image:

Message successfully hidden and image saved as output.bmp

The output image, now containing the hidden encrypted message, is saved under the name output.bmp in the working directory.

**Decoding Mode: Revealing a Hidden Message**

If the user chooses decoding mode by entering d, the following steps occur:

**Step 1: Enter Image File Name**

Enter image file name (e.g., output.bmp):

The program expects an image file that was previously generated by the encoding function (i.e., a .bmp file containing hidden data).

**Step 2: Enter Encryption Key**

Enter encryption key:

The user inputs the same key that was used to encode the message. If an incorrect key is entered, the output will be unreadable (random or meaningless characters).

**Step 3: Message Output**  
If successful, the decoded message is displayed on the terminal:

Hidden message: [decrypted message]

If the message was corrupted or the key was incorrect, the output may appear distorted.

**Program Termination**

Once the encoding or decoding process is complete and the output is shown or saved, the program terminates automatically. No special commands are required to exit.

**Example Sessions**

**Encoding Session:**

Enter image file name: image.bmp

Enter encryption key: secret123

Enter message to hide: cholpon dura.

Message successfully hidden and image saved as output.bmp

**Decoding Session (Correct Key):**

Enter image file name: output.bmp

Enter encryption key: secret123

Hidden message: karina legenda.

**Decoding Session (Wrong Key):**

Enter image file name: output.bmp

Enter encryption key: wrongkey

Hidden message: ¢¿ÀAÈ×ð3ƒ±%›Ê^…

These examples demonstrate that the program is user-friendly and intentionally simple to follow, while still incorporating powerful encryption and data-hiding capabilities.

**Input and Output**

**Input**

* Image file (24-bit BMP) //important
* Encryption key (string)
* Message to hide (string)

**Output**

* Encoded image saved as output.bmp
* Decrypted message printed to console when decoding

Example:

Input:

File: image.bmp

Key: mypass

Message: Hello

Output:

File: output.bmp (with "Hello" hidden)

**Program Structure**

**Overview**

The program follows an object-oriented architecture and is composed of multiple source (.cpp) and header (.h) files. These are grouped into logical units based on responsibility: encryption, steganography, image handling, and application control.

**Main Modules**

* **main.cpp**
  + This file contains the main() function and handles all user interaction.
  + It prompts the user for the image file name, encryption key, and operation mode.
  + It creates an instance of SecureSteganography and calls the appropriate method (encode() or decode()).
* **securestega.cpp / securestega.h**
  + This module defines the SecureSteganography class.
  + It combines both encryption and steganography through multiple inheritance from Encryptor and TextSteganography.
  + It overrides encode() and decode() to add encryption and decryption steps around the steganographic operations.
* **encryptor.cpp / encryptor.h**
  + Contains the Encryptor class, which performs simple XOR-based encryption and decryption.
  + The key is stored as a member variable and used in a loop to encrypt/decrypt each character of the message.
  + The class also defines an overloaded << operator for potential debugging or display purposes.
* **hideandseek.cpp / hideandseek.h**
  + Contains the TextSteganography class, which inherits from Steganography.
  + Implements encode() and decode() using the LSB (Least Significant Bit) technique on BMP pixel data.
  + Converts text to binary using stringToBinary(), embeds it in the pixel data, and reconstructs it using binaryToString().
* **steganography.cpp / steganography.h**
  + Contains the abstract base class Steganography, which provides fundamental operations for image processing.
  + Defines image loading (loadImage()) and saving (saveImage()) logic.
  + Declares pure virtual functions encode() and decode() that must be implemented by child classes.
  + Includes helper functions for converting strings to binary and vice versa.

**Class Descriptions**

* **Encryptor**
  + Base class that manages encryption key and text transformation logic.
  + Public Methods:
    - encrypt(std::string) → returns XOR-encrypted string
    - decrypt(std::string) → returns XOR-decrypted string
  + Protected Member:
    - std::string key
* **Steganography (Abstract Class)**
  + Provides the framework for steganographic functionality.
  + Public Methods:
    - loadImage() — loads the BMP file into memory
    - saveImage() — writes the modified BMP buffer to disk
  + Pure Virtual Methods:
    - encode(std::string) — to be implemented by derived class
    - decode() — to be implemented by derived class
  + Protected Members:
    - std::string filePath
    - unsigned char\* imageBuffer
    - size\_t bufferSize
    - BitmapFileHeader, BitmapInfoHeader
* **TextSteganography (inherits from Steganography)**
  + Implements the actual encoding/decoding logic using LSB.
  + Uses imageBuffer directly to modify pixel data.
  + Uses ~ as a delimiter to mark the end of the hidden message.
* **SecureSteganography (inherits from Encryptor + TextSteganography)**
  + Adds security to steganography by combining both base class functionalities.
  + Overrides:
    - encode() — encrypts message first, then calls TextSteganography::encode()
    - decode() — calls TextSteganography::decode(), then decrypts the result

**Data Structures and Types**

* std::string — used for file names, messages, encryption keys
* unsigned char\* imageBuffer — dynamically allocated buffer for raw BMP data
* BitmapFileHeader, BitmapInfoHeader — packed structs used to parse BMP headers
* std::bitset<8> — used in binary conversion functions

**Program Flow Summary**

1. User provides image file name and encryption key.
2. Image is loaded into imageBuffer using loadImage().
3. Depending on user choice:
   * **Encode:**
     + Message → encrypt() → binary → encode() into LSB of BMP
     + Save modified image via saveImage()
   * **Decode:**
     + Extract binary → decode() → decrypt message using decrypt()
4. Output message is printed or saved.

**Testing and Verification**

Testing was performed using multiple sample BMP files with different dimensions. The following tests were run:

* Encoding and decoding with valid inputs
* Using wrong decryption key (produces garbage output as expected)
* Using malformed image (program throws exception)
* Very long messages (exceeding image capacity) were prevented

Error handling was verified using try/catch blocks and exceptions.

**Improvements and Extensions**

* Add frontend
* Support more image formats (e.g. PNG, JPG)
* Add different encryption methods instead of XOR for stronger security
* Allow user to choose output filename
* Visualize how much space is available for hiding

**Difficulties Encountered**

* Understanding BMP file structure and padding rules
* Ensuring XOR encryption didn't break special characters
* Bit-level manipulation and buffer indexing was tricky
* Validating whether the message can fit in the image safely
* Windows bmp file format not supported
* Paint bmp file format not supported
* Encoded messages were shorter by 1 symbol than intended
* Unable to save to output.bmp
* Wrong filepaths resulted in output not generating in files
* Having to ensure the exact bit calculations for output file to be successfully read
* Overestimating capabilities, therefore not using the external library like planned for manipulating other image formats

**Conclusion**

This project successfully demonstrates a basic, functional steganography tool combining encryption and image processing. It shows the importance of data hiding, bitwise operations, and object-oriented design in C++. The result is a working tool that can be easily extended into a more complex steganography system in future.

**Deviations from the Original Plans**

In the initial project proposal, it was planned to use the **OpenCV library** for image processing and pixel manipulation. However, during implementation, this approach was abandoned in favor of manually parsing **24-bit BMP files** using raw byte manipulation and custom header structures. This decision was made to:

* Simplify dependencies
* Gain low-level experience with file structures
* Ensure portability and full control over memory and file operations

Additionally, the proposal mentioned the possibility of using **Visual Studio Code on Windows 11**, but the development process was eventually conducted in a more cross-platform-friendly environment using standard g++ compilation and command-line testing.

Finally, encryption was listed as a potential **additional feature**, but it was successfully implemented and integrated into the final solution using XOR logic. This extended the project beyond the minimum scope initially proposed.

These deviations reflect changes made for educational value, efficiency, and to meet all core project requirements without relying on external libraries.

**References**

* <https://metanit.com/cpp/tutorial/>
* <https://www.learncpp.com/>
* <https://cplusplus.com/reference/bitset/bitset/>