# Project Report - Group 24

## Problem Statement

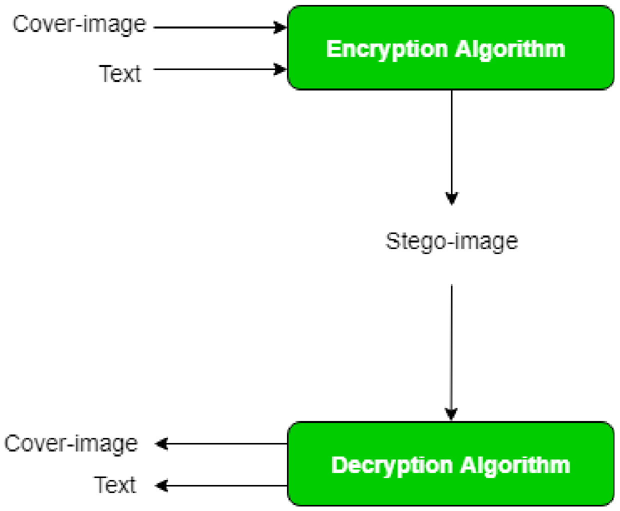
The problem we aim to solve is to provide a secure and efficient way to hide and retrieve secret messages within PNG files, while underlining the specific advantages of Rust for steganography and comparing these to the capabilities of other programming languages.

We propose the development of a comparative implementation of a command-line program that allows users to hide secret messages within PNG files, with a strong emphasis on Rust's benefits for steganography. This project aims to provide a secure and user-friendly way to encode, decode, remove, and manage hidden messages within PNG images.

The project includes two distinct implementations: one in Rust, serving as the primary target, and the other in C/C++ for the purpose of comparison. PNG files are chosen as the medium for secret message storage due to their widespread use and the flexibility they offer for embedding information. PoPL aspects are explained in more detail into the subhead : PoPL Aspects of the project.

## Software Architecture

Steganography is the practice of concealing information within another seemingly innocuous piece of data in such a way that it's difficult to detect. This can involve hiding messages, images, or other data within digital media, such as images, audio files, or text, without altering the apparent content of the carrier file.



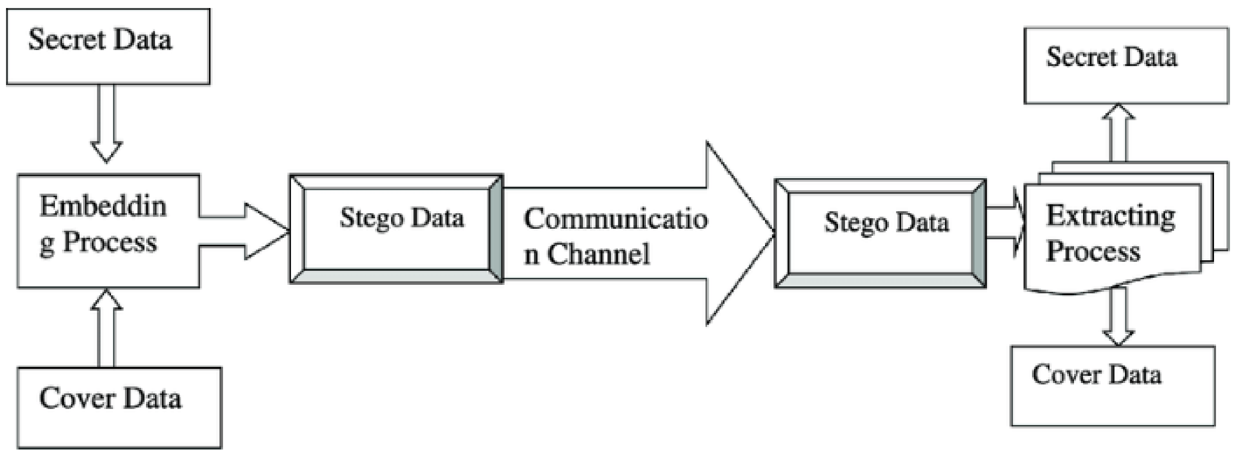
The goal of steganography is to hide the existence of the concealed information so that it goes unnoticed. There are multiple types of steganography such as image, audio, video, text, etc. Here we focus on Image based steganography.

1. Steganography works by concealing information in a way that avoids suspicion. One of the most prevalent techniques is called 'least significant bit' (LSB) steganography. This involves embedding

the secret information in the least significant bits of a media file.

2. In an image file, each pixel is made up of three bytes of data corresponding to the colors red, green, and blue. Some image formats allocate an additional fourth byte to transparency, or ‘alpha’.

3. LSB steganography alters the last bit of each of those bytes to hide one bit of data. So, to hide one megabyte of data using this method, you would need an eight-megabyte image file.
4. Modifying the last bit of the pixel value doesn’t result in a visually perceptible change to the picture, which means that anyone viewing the original and the steganographically-modified images won’t be able to tell the difference.



Block diagram of steganography.

Both the implementations were tested locally and do not require a database.
The following are the details of both the implementations.

The CPP implementation makes use of two external libraries (refer Code External section) to help process the input images and convert it into an array upon which the algorithms are applied.

1. Main.cpp : This function contains the main function and
2. Image.h : Contains the definition of struct image and the rest of the functions that are implemented in image.cpp. It also contains the definition of the enum ImageType.
3. Image.cpp :

   **Image Class:**

   - **Attributes:**
     - w: Width of the image.

    - `h`: Height of the image.
    - `channels`: Number of color channels.
    - `size`: Total size of the image data.
    - `data`: Pointer to the image data.
- **Methods:**
    - `Image(const char* filename)`: Constructor for reading an image from a file.
    - `Image(int w, int h, int channels)`: Constructor for creating an empty image.
    - `Image(const Image& img)`: Copy constructor.
    - `~Image()`: Destructor for freeing allocated memory.
    - `bool read(const char* filename)`: Reads an image from a file.
    - `bool write(const char* filename)`: Writes the image to a file in various formats.
    - `ImageType getFileType(const char* filename)`: Determines the file type based on the extension.
    - `Image& grayscale_lum()`: Converts the image to grayscale using luminance method.
    - `Image& grayscale_avg()`: Converts the image to grayscale using average method.
    - `Image& colorMask(float r, float g, float b)`: Applies a color mask to the image.
    - `Image& colorSwapRG()`: Swaps red and green color channels.
    - `Image& remRed()`: Removes red color by swapping red and green channels.
    - `Image& encodeMessage(const char* message)`: Encodes a message in the image using steganography.
    - `Image& decodeMessage(char* buffer, size_t* messageLength)`: Decodes a message from the image using steganography.

**STB Image Library:**

- `#include "stb_image.h"` and `#include "stb_image_write.h"`: External libraries used for reading and writing images.

**Usage:**

- The `main` function or other parts of the code would likely create instances of the `Image` class and use its methods for various image processing tasks.

**Dependencies:**

- The code relies on the STB image library for image I/O operations.

**Steganography:**

- The `encodeMessage` and `decodeMessage` methods provide functionality for hiding and retrieving messages within the image.

**Error Handling:**

- The code includes some basic error handling, such as printing error messages when file reading fails or unsupported image types are encountered.

This Rust code defines a simple image processing library and demonstrates steganography by hiding a message within an image.

**1. Libraries and Imports:**

a. The code uses the `std::ptr` module for dealing with raw pointers.

b. It relies on the `image` crate for image processing.

**2. Image Trait:**

c. Defines a trait named `Image` with various methods for image manipulation.

d. Methods include reading and writing images, converting to grayscale, applying color masks, and encoding/decoding messages.

**3. ImageType Enum:**

e. Enumerates different image file types.

**4. Constants:**

f. STEG_HEADER_SIZE: A constant representing the size of the header used for steganography, assumed to be 256 bits.

**5. RustImage Struct:**

g. Represents an image with RGB pixels.

h. Implements the `Image` trait for image manipulation.

**6. Image Trait Implementation for RustImage:**

i. Implements methods from the `Image` trait for the `RustImage` struct.

j. Reading and writing images, converting to grayscale, applying color masks, and steganography methods (encoding and decoding messages) are defined here.

7. **Main Function:**

k. Creates a `RustImage` from a file ("image.png").

l. Encodes a hidden message in the image using steganography.

m. Creates another `RustImage` from the steganographically modified image.

n. Decodes the hidden message and prints it.

8. **Steganography (encode_message and decode_message):**

o. The `run` method hides a message in the image by modifying the least significant bit of each pixel's red channel to represent the message bits.
It also decodes the message hidden in the file present in the results directory.

9. **Main Function (Continued):**

q. The main function prints the decoded message.

Libraries used in the code :

1. image crate: - The `image` crate is an external dependency used for image processing. It provides functionality for reading, writing, and manipulating images in various formats.

The `image` crate is used for tasks such as opening images, converting them to RGB format, and saving images in different file formats. This crate simplifies image-related operations in Rust by providing a high-level API for common tasks.

In the code, you can see the following import statements related to these libraries:

use image::{Rgb, RgbImage, DynamicImage, GenericImageView};

- `use std::ptr;` imports the `ptr` module from the standard library.

- `use image::{Rgb, RgbImage, DynamicImage, GenericImageView};` imports specific types and traits from the `image` crate.

These imports allow the code to use the functionalities provided by the `ptr` module and the `image` crate throughout the program.

# POPL Aspects of the Project

Yes, the principles of memory safety and lifetimes are used in this Rust code. Let's explore where and how these principles are applied:

**1. Memory Safety:**

- Rust's ownership system ensures memory safety by enforcing strict rules about ownership, borrowing, and lifetimes. The absence of explicit memory management, like manual memory allocation and deallocation, prevents many common programming errors related to memory safety.

- In the code, there is no explicit use of raw pointers or manual memory allocation. The `image` crate is used for image processing, and it abstracts away the low-level details of memory management. The ownership system helps manage the memory of the `RustImage` struct and other variables.

- The `Vec<u8>` used for the `decoded_buffer` is managed by Rust's ownership system, and the buffer's memory is automatically freed when it goes out of scope.

**2. Lifetime:**

- Lifetimes in Rust ensure that references remain valid for the duration they are used. In this code, lifetimes are implicitly managed by Rust's ownership system.

- The `&str` references in the `new_from_file` and `read` methods have lifetimes tied to the function scope, and Rust's borrow checker ensures that these references do not outlive the data they point to.

- The references to pixels in the `encode_message` and `decode_message` methods are short-lived and are appropriately managed within the method scope. The borrow checker ensures that these references do not lead to dangling pointers.

- The lifetime of the references used in the methods is inferred by the borrow checker and is not explicitly annotated in the code. Rust's ownership system statically enforces these lifetimes at compile-time.

The principles of memory safety and lifetimes are implicitly applied through Rust's ownership system and borrow checker. These features help prevent common memory-related errors and ensure safe concurrent access to data.

**3.Traits –**

In the provided Rust code, traits are used to define a set of methods that types implementing those traits must provide. The primary trait used in the code is the `Image` trait. Let's explore how traits are used in this context:

1. Trait Definition:

- The `Image` trait is defined with several method signatures, outlining the behavior expected from types that implement this trait.

```
struct RustImage {
    image: RgbImage,
}
```

2. Struct Implementation:

- The `RustImage` struct is defined, representing an image with RGB pixels.

```
impl Image for RustImage
```

3. Trait Implementation for RustImage:

- The `impl Image for RustImage` block indicates that the `RustImage` type is implementing the `Image` trait.

4. Method Implementations:

- The methods defined in the `Image` trait are implemented for the `RustImage` type.

```
fn new_from_file(filename: &str)

fn new_rgb(w: u32, h: u32)
```

Each of the methods declared in the `Image` trait is implemented for the `RustImage` type. For example, `new_from_file`, `new_rgb`, `read`, `write`, `grayscale_lum`, `encode_message`, etc., are all implemented for the `RustImage` type.

By implementing the methods of the `Image` trait for the `RustImage` type, you make it possible to create polymorphic code that can work with any type that implements the `Image` trait. This allows for more modular and reusable code, and it enables the use of trait objects, allowing different types to be treated uniformly when they share a common trait

.

**4. Error Handling:**

Rust encourages explicit error handling through Result types, making it clear when a function can return an error. This helps in writing robust and predictable code.

In the Rust code, error handling is used in several places, primarily through the `Result` type and its `Ok` and `Err` variants. Instances where error handling is applied:

1.Reading an Image:

```
fn read(&mut self, filename: &str) -> bool {

    match image::open(filename) {

        Ok(image) => {

            let rgb_image = image.to_rgb8();

            self.image = RgbImage::from_vec(rgb_image.width(), rgb_image.height(), rgb_image.into_raw()).unwrap();

            true

        }

        Err(_) => false,

    }

}
```

- The `image::open(filename)` function attempts to open and decode an image from the specified file.

- The `Ok(image)` arm of the match block handles the case where the image is successfully opened.

- If there's an error (`Err(_)`), the function returns `false`.

2. Writing an Image:

```
fn write(&self, filename: &str) -> bool {
        match Self::get_file_type(filename) {

            ImageType::PNG => self.image.save(filename).is_ok(),

            _ => {

                println!("Unsupported image type.");

                false

            }

        }

    }
```

- The `self.image.save(filename)` function attempts to save the image to the specified file.

- The `ImageType::PNG => self.image.save(filename).is_ok()` line checks if the image is saved successfully as a PNG file.

- If the file type is not supported, it prints a message and returns `false`.

3. Checking File Type:

```
    fn get_file_type(filename: &str) -> ImageType {

        let ext = filename.split('.').last();

        match ext {

            Some("png") => ImageType::PNG,

            Some("jpg") => ImageType::JPG,

            Some("bmp") => ImageType::BMP,

            Some("tga") => ImageType::TGA,

            _ => ImageType::UNKNOWN,

        }

    }
```

- The `get_file_type` function attempts to determine the file type based on its extension.

- It returns an `ImageType`, and if the extension is not recognized, it defaults to `ImageType::UNKNOWN`.

4. Encoding Message:

```
fn encode_message(&mut self, message: &str) -> &mut Self {
    // ...
    if len + 256 > self.image.dimensions().0 * self.image.dimensions().1 * 3 {
        println!(
            "[ERROR] This message is too large ({} bits / {} bits)",
            len + 256,
            self.image.dimensions().0 * self.image.dimensions().1 * 3
        );
        return self;
    }
    // ...
}
```

- The function checks whether the message size exceeds the capacity of the image before encoding.

- If the message is too large, an error message is printed.

Overall, error handling in this code is mainly focused on image-related operations, and it involves checking the success of functions that may fail. The specific error values are not explicitly handled in some cases, and the code often opts for a simple boolean indication of success or failure. Depending on your use case, you might want to handle errors more gracefully by returning `Result` and providing more detailed error information.

The C++ code relies on return values and error codes, which may not be as explicit or standardized as Rust's Result type, potentially leading to overlooked error cases.

**5. String Handling:**

Rust's String type ensures memory safety and prevents common string-related bugs, such as buffer overflows or null-terminated string issues. C++ relies on null-terminated strings and manual memory management, which can lead to vulnerabilities if not handled carefully.

String handling is being used in the following functions:

1. `fn encode_message(&mut self, message: &str) -> &mut Self;`

2. `fn decode_message(&self, buffer: &mut [u8], message_length: &mut usize) -> &Self;`

3. `fn run();` (inside `run`, the payload path is specified as a string: `let payload_path = "./message.txt";`)

4. `fn main();` (a message string is commented out in the main function: `// let message = "Hello, this is a hidden message!";`)

**6. Ownership and Borrowing:**

The concepts of ownership and borrowing in Rust are fundamental to the language's memory safety guarantees. They help prevent common programming errors such as use-after-free, data races, and null pointer.

1.Ownership in `new_from_file` Function:

```
fn new_from_file(filename: &str) -> Self {

    // ...

    let mut rust_image = RustImage {

        image: RgbImage::new(1, 1),

    };

    // ...

    rust_image

}
```

In this function, ownership of the `RustImage` instance is transferred to the calling code. The function takes a reference to a string (`&str`) representing the filename. Ownership of the filename is not transferred; instead, the function borrows it. This ensures that the ownership and responsibility for cleaning up the memory associated with the `RustImage` instance are clear.

2. Borrowing in `read` Function:

```
fn read(&mut self, filename: &str) -> bool {

    match image::open(filename) {

        Ok(image) => {

            // ...

                self.image = RgbImage::from_vec(rgb_image.width(), rgb_image.height(),
rgb_image.into_raw()).unwrap();

            true

        }

        Err(_) => false,

    }

}
```

The `&mut self` parameter in the `read` function indicates that the function borrows mutable ownership of the `RustImage` instance. This allows the function to modify the `image` field of the `RustImage`. Borrowing mutably ensures exclusive access to the data, preventing simultaneous modifications from other parts of the code.

3. Borrowing in Pixel Manipulation Functions:

```
fn grayscale_lum(&mut self) -> &mut Self {

    for pixel in self.image.pixels_mut() {

        // ...

    }

    self
}
```

Functions like `grayscale_lum`, `grayscale_avg`, and others take a mutable reference (`&mut self`). This means they borrow mutable ownership of the `RustImage` instance and can modify its content. The borrow checker ensures that at any given time, there is either one mutable reference or multiple immutable references to a piece of data, preventing data races.

4. Borrowing in `encode_message` Function:

```
fn encode_message(&mut self, message: &str) -> &mut Self {

    // ...

    self

}
```

The `encode_message` function takes a mutable reference to `self` and borrows a string slice (`&str`) representing the message. This borrowing ensures that the function can read the message without taking ownership of it. The function can modify the `RustImage` to encode the message without consuming or altering the original message.

Ownership and borrowing in Rust ensure clear ownership semantics and prevent common memory-related errors. They contribute to the code's safety by enforcing strict rules about how data can be accessed and modified, and they eliminate the need for explicit memory management or garbage collection.

**7. Code Readability:**

Rust's syntax and strong conventions contribute to code readability. Explicit ownership and borrowing annotations help understand the flow of data and prevent unexpected side effects.

## Results

Raw image

Encoded image

Both the Rust and C++ implementations you provided are similar in functionality, as both perform steganography operations on images.

To differentiate between two images in Rust, you can compare the pixel values of corresponding pixels in both images. One simple approach is to calculate the Mean Squared Error (MSE) between the two images. The MSE provides a measure of the average squared difference between corresponding pixel intensities.
We can also use the Structural Similarity Index method to achieve this.

**Advantages of the Rust Implementation:**
Memory Safety:Rust has a strong ownership system that ensures memory safety without sacrificing performance. The Rust implementation uses the ownership system effectively, preventing common memory-related errors. The C++ implementation, on the other hand, uses manual memory management with new and delete, which can lead to memory-related bugs if not handled carefully.

- Error Handling: Rust enforces explicit error handling through its Result type, making it clear when a function can fail. The C++ implementation relies on return values and may not provide clear indications of errors.

- Concurrency and Safety: Rust's ownership system and borrowing rules make it easier to write concurrent and parallel code without data races. This is a language feature that can be advantageous for certain applications.

- Pattern Matching: Rust allows for powerful pattern matching, which can lead to more concise and expressive code. In the Rust implementation, pattern matching is used for handling different image types.

- Trait System: Rust's trait system enables code reuse through trait implementations. In the Rust implementation, the Image trait is defined, providing a clear interface for different image operations.

Rust's emphasis on memory safety and its ownership system can contribute to better space complexity and reduced memory-related errors compared to manual memory management in C++. However, both languages have the potential for high runtime performance, and the actual performance may vary based on implementation details and compiler optimizations.

# Potential for future work and applications

The LSB method of steganography used has a major disadvantage as it is vulnerable to steganalysis and therefore cannot be considered secure. To improve upon the least significant bit method, modifications are needed. One possible solution is to combine least significant bits of pixel in a sequence with midpoint circle approach to choose which pixels are used to hide messages.

1. Enhanced Security Measures:
   - Advancements in technology may lead to increased use of steganography for embedding authentication information, ensuring data integrity, and adding an extra layer of security.

2. Digital Rights Management (DRM):
   - Steganography could play a role in strengthening digital rights management systems by embedding invisible watermarks or information to track and protect intellectual property.

3. Privacy and Anti-Surveillance:
   - In response to increased surveillance, steganography may see use as a means of private communication by hiding messages within everyday digital content.

4. Machine Learning and Steganalysis:
   - Advancements in machine learning and artificial intelligence will likely impact steganalysis, contributing to the ongoing cat-and-mouse game between steganographers and detectors.

5. Biometric Data Security:
   - Steganography could be applied to secure biometric data, such as fingerprints or facial recognition information, enhancing privacy and preventing unauthorized access.

6. Internet of Things (IoT) Security:
   -Steganography may find applications in securing communication between IoT devices, involving the hiding of authentication credentials or ensuring data integrity.

7. Communication in Restricted Environments:
   - In regions with strict censorship and surveillance, steganography might continue to be used as a tool for covert communication, allowing individuals to share information without detection.

## Code Original

Rust_Implementation/rust_impl/src/main.rs
CPP_Implementation/src/main.cpp
CPP_Implementation/src/image.cpp
CPP_Implementation/src/image.h

## Code External

CPP_Implementation/src/stb_image.cpp
CPP_Implementation/src/stb_image_write.h
(The above two open-source libraries were used to convert the .png images to the array data[])