Steganography and Cryptography in Digital Images

P1 Project – Steganography

AALBORG UNIVERSITY
STUDENT REPORT

Authors:

Kean Olsen
Mostafa Hassan
Nick Arge
Nicklas Søskov
William Pedersen

Supervisor:

Henning Thomsen

Semester:

1st Semester

Study Programme:

B.Eng, Cybersecurity

Date: December 21, 2025

Abstract

This project explores the integration of steganography and cryptography to develop a convenient application for entities who prefer to securely hide and retrieve data within digital images. The primary objective is to enhance confidentiality within text-based communication by combining two security layers: concealing any trace of communication through steganography and protecting its content using AES encryption. The theoretical framework covers a steganographic technique least significant bit (LSB), image file formats, and security considerations, emphasizing the use of lossless formats such as portable network graphics (PNG) for optimal data integrity. The implementation leverages Python and libraries like Pillow and PyCryptodome to embed encrypted messages using the LSB method, complemented by Euclid's algorithm for a structured method using Euclid's algorithm and LSB data distribution. This layered approach ensures robustness against detection and unauthorized access, addressing challenges in modern cybersecurity where steganography is increasingly exploited for covert communication. The project concludes with an analysis of limitations, future prospects, and the balance between security, data capacity, and performance.

Contents

1 Introduction

With the rapid growth of digital communication, guarding sensitive information has become a serious challenge. Cryptographic methods have proven to be effective in securing the content of communication over digital media, but they do not conceal the existence of communication itself. This visibility of communication may attract unwanted attention from bad actors, which increases the risk of targeted attacks.

Steganography offers a solution by hiding data within ordinary media, such as images, masking the presence of communication. In addition to ensuring confidentiality, encryption should be applied to the media, which is used to conceal communication, as data may be extracted if no encryption is applied. This project explores the integration of steganography and cryptography to develop a user-friendly application capable of hiding and retrieving encrypted messages within digital images. By utilizing the Least Significant Bit (LSB) technique, AES encryption, and a lossless image format (PNG), the project aims to balance security, usability, and performance. Euclid's algorithm is implemented to distribute hidden data across the image, which reduces detectable patterns while maintaining the means to reliably extract the data.

This report outlines the problem context, theoretical foundations, implementation, and an evaluation of the program, concluding with perspectives on future improvements and application.

2 Problem Statement

How can a user-friendly application, enable users to hide and retrieve data in images using steganography and cryptography, prioritizing confidentiality.

3 Methods

In this report, the aim was to investigate our problem and review relevant open-source codes using LSB-steganography for data embedding. Many of our sources consist of technical articles, documentation, and academic texts from reliable outlets. Our sources are relevant documentation from the last 15 years ranging between 2012 to current year.

We did not rely on many different methods in this project, as our project mainly consisted of research, code review, analysis, and discussion based on our findings.

The development followed an Object-Oriented Programming (OOP) process, which python primarily use. We first created a simple prototype "StegoV1". This prototype led our path to "StegoV2" which improved on data decoding, and now included AES cryptography.

4 Problem Analysis

With the ever-increasing communication over open and potentially insecure channels, there is a growing need to protect sensitive information. While cryptography ensures confidentiality, it does not conceal the fact that communication is taking place. Bad actors may become attracted to communication between certain parties, so they can intercept and analyze encrypted data. This limitation highlights the need for complementary techniques that not only protect data confidentiality but also reduce the likelihood of detection.

Steganography may be this complementary technique that can conceal information within ordinary digital media, such as images, which conceal the presence of communication. Combining cryptography and steganography provides a layered security approach by hiding and protecting a message and its content simultaneously.[1]

Improper implementation, such as the use of lossy image formats, will corrupt hidden data during encryption, and using predictable embedding patterns will increase detectability. These challenges show the necessity of combining steganographic and cryptographic methods in a structured and secure manner.

Another aspect of the problem concerns usability. Many existing steganographic solutions and tools are highly technical or lack intuitive interfaces, which limits their accessibility. This has created a demand for a tool that allows users to hide and retrieve data securely without the need for advanced technical knowledge.

This project addresses the problem statement:

"How can a user-friendly application, enable users to hide and retrieve data in images using steganography and cryptography, prioritizing confidentiality."

By combining AES encryption with Least Significant Bit (LSB) image steganography, with the lossless image format Portable Network Graphics (PNG) to preserve data integrity. Additionally, an embedding approach, based on Euclid's algorithm, is implemented to distribute hidden data across the image, which reduces detectable patterns while maintaining the means to reliably extract the data.[2]

In general, this project aims to achieve a balance between detectability and usability in the design of a practical application for hidden and secure data communication.

5 Theoretical Framework

5.1 Steganography

Steganography (often shortened to stego) is a technique that enables a person to hide text based messages, files or other forms of data within ordinary media such as images, videos or audio files. The key idea behind steganography, is not to make the message unreadable like cryptography does, but to conceal the existence of the message. This is achieved by embedding the data you want to hide in such a way that human senses can't see it in the chosen media by either sight or hearing. [3]

Unlike other methods of keeping information secure like cryptography, which scrambles the content of the data into something unreadable, only readable if you have the key used to decrypt it, stego focuses on keeping the communication itself hidden. A way to look at it is the image as a Trojan horse, and the people inside are the hidden data. When combining, they form the stego object which just looks like any ordinary file. [3]

In some instances, stego can also use a secret key to hinder unwanted eyes from reading or hearing the data, however, that is optional. [3]

Instead of measuring effectiveness by how well an algorithm can encrypt data, the effectiveness is measured on how well the data is embedded and how discernible it remains. Although, if alteration of the image is discovered, the data might be vulnerable, and read by others than the recipient, especially if no encryption has been applied alongside it.

5.1.1 Text Steganography

Text steganography involves hidden information within the textbook, documents, reports, or alternative textual data. Data are hidden with the help of each letter of any given word. It is a very difficult method to execute as variations or changes in any given secret data can change subtly. [4]

5.1.2 Image Steganography

Image steganography referse to the metode of hiding information often text-based within a digital image, such as a PNG file. This is commonly achieved by making changes to the pixel values withing the image to encoded the data, such as the least significant bit (LSB) method being one of the most common techniques. While the approach enables the embedding of information without making it visual apparent. This limits the amount of data hidden without noticeable distortion in the image. [4]

5.1.3 Audio Steganography

Audio steganography is achieved by caching dispatches or secret information within audiotape lines. [4]

5.1.4 Network/Protocol Steganography

Network or protocol steganography is a type of steganography which is defined by caching dispatches or secret information within network protocols or dispatches. It hides secret information in the usual flow of internet or network exertions. This type of steganography is commonly used to bypass traditional antivirus and security scanners, which rarely inspects image content at the bit level. [4]

5.2 Image File Formats

Choosing image file formats is essential to meet the goals of file size, number of bits in a pixel, quality, layout, and ease of sharing. Some formats take up significant space and do not compress well, while others compress more easily.

Another important aspect to think about is how easily the image file format is edited. Some images can only be edited with certain editing software, while others can be edited more easily. Additional considerations include animation capabilities, color limits, data retention, and transparency features. [5]

Each color pixel in an image represented using color formats of 8, 16, 24, or 32 bits.[6] In 16 bit color each pixel is represented using 16 bits or 2 bytes. There are 5 bits for red, 6 bits for green, and 5 bits for blue. The total number of distinct colors in 16 bit is 65,536.

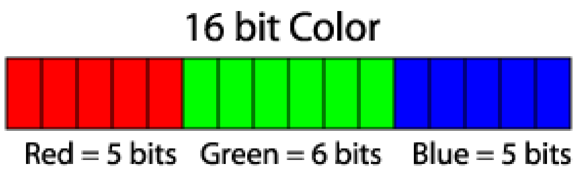


Figure 1: 16 bit color
[7]

For 24 bit color each pixel is represented by using 3 bytes. The 3 bytes represent red, green, and blue, which is referred to the RGB colors. Each byte has 8 bits and can represent 256 different shades, which is equivalent to 16,777,216 distinct colors.

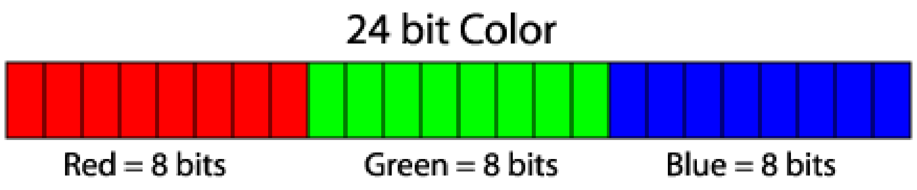


Figure 2: 24 bit color
[7]

The 32 bit color is somewhat different as it is the same as 24 bit but has an extra byte, which is used for transparency. This extra byte is often referred to as the alpha component.

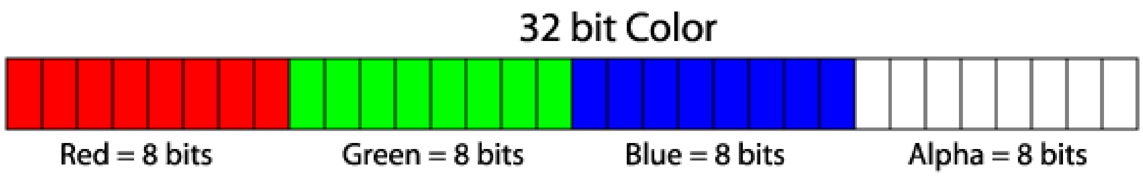


Figure 3: 32 bit color
[7]

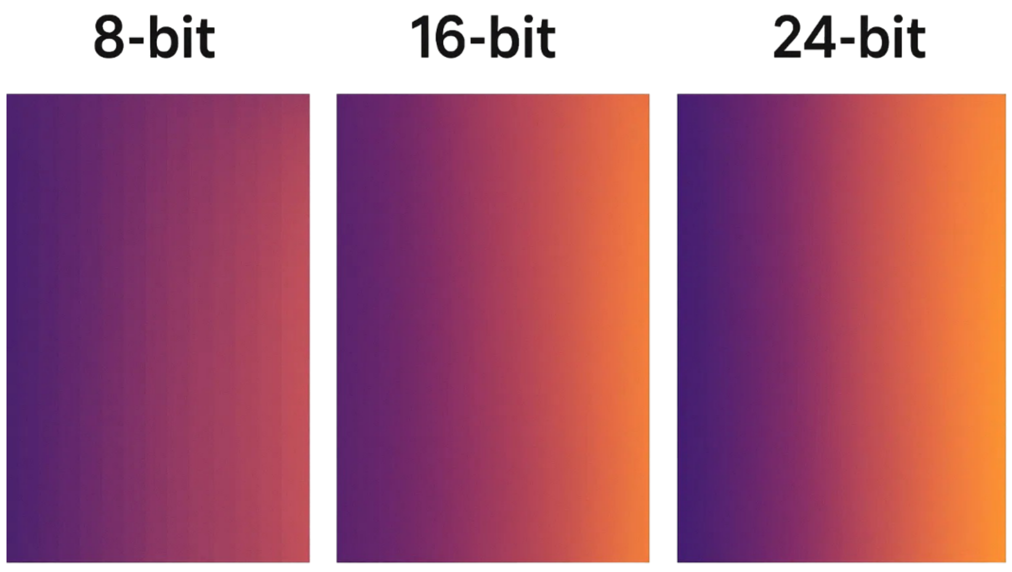


Figure 4: Color depth comparison between 8-bit, 16-bit and 24-bit
[8]

5.2.1 JPEG

JPEG stands for Joint Photographic Experts Group. JPEG is the go-to file format for digital images. As a natural upgrade to Graphics Interchange Format (GIF), which only had 8 bits per pixel resulting in 256 distinct colors, JPEG was made to have 24 bits per pixel, but lost the animation which GIF previously had.[9]

JPEG was made as a way to bring photo-realistic pictures to the average PC. The solution JPEG brought about was lossy compression, which removed visual data that the human eye could not see and averaged out color variation. The real value of JPEG was the ability to store metadata. Data such as where and when a picture was taken and also the camera settings when the picture was taken.[9]

JPEGs has various advantages including having a very small file size, using lossy compression which removes elements humans cannot see, and the post-processing being much easier as the white balancing and saturation in JPEGs are set with the click of the shutter. Although the compression method saves a lot of space and attempts to discard elements that cannot be seen, heavily compressed images still suffer a loss in quality, which makes JPEG unsuitable for steganography. [9]

5.2.2 PNG

A newer file format is Portable Networks Graphics (PNG) which is popular because it is 32 bits, therefore able to handle graphics with transparent or semi-transparent backgrounds. PNG was launched in 1995 and is the evolution of the GIF format. The GIF has several drawbacks, such as a required patent license and a limited range of 256 colors. Therefore, PNG does not have a required patent license and has an enormously expanded color range of 16 million colors. PNG images will not lose any of their data when compressed. This is an advantage over JPEG files, where some information and image quality disappear in the lossy compression process.[10]

As a single-image format, PNGs do not support animation like GIFs do. By retaining all the original data when a PNG is compressed, a PNG file will generally be larger in size than JPEGs. [10]

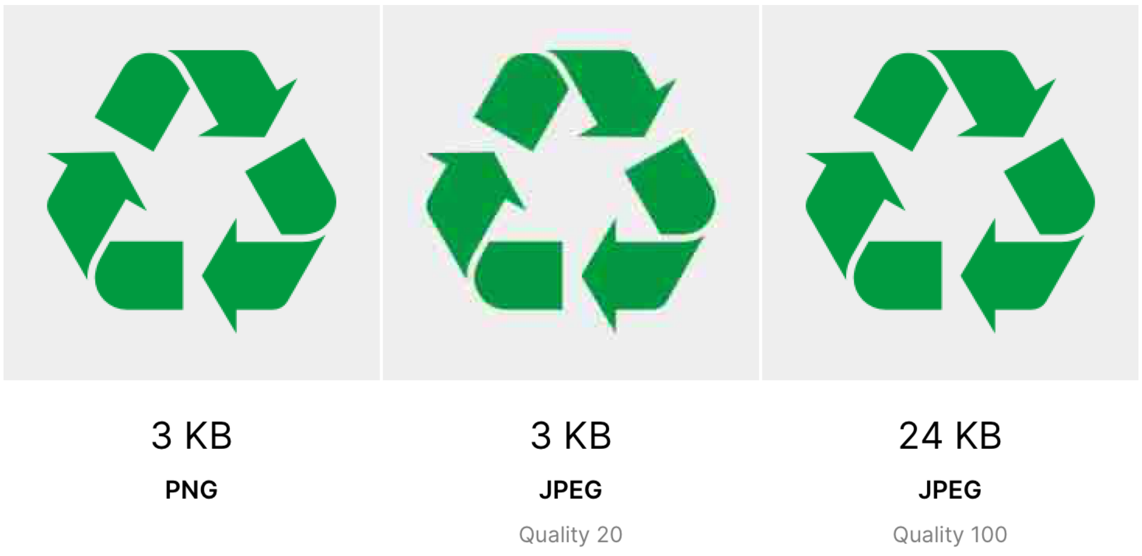


Figure 5: Comparing lossy and lossless compression
[11]

5.2.3 BMP

The BMP format is an uncompressed image file developed by Microsoft to display high-quality images on Windows and store printable photos. This makes BMP ideal for storing and displaying high-quality digital images. The lack of compression generally creates a larger file size than, for example, JPEGs and PNGs, which is great for hiding information within the picture.. BMP files are not a fixed bit size, but they support various color depths with 24 bit being the most common.[12]

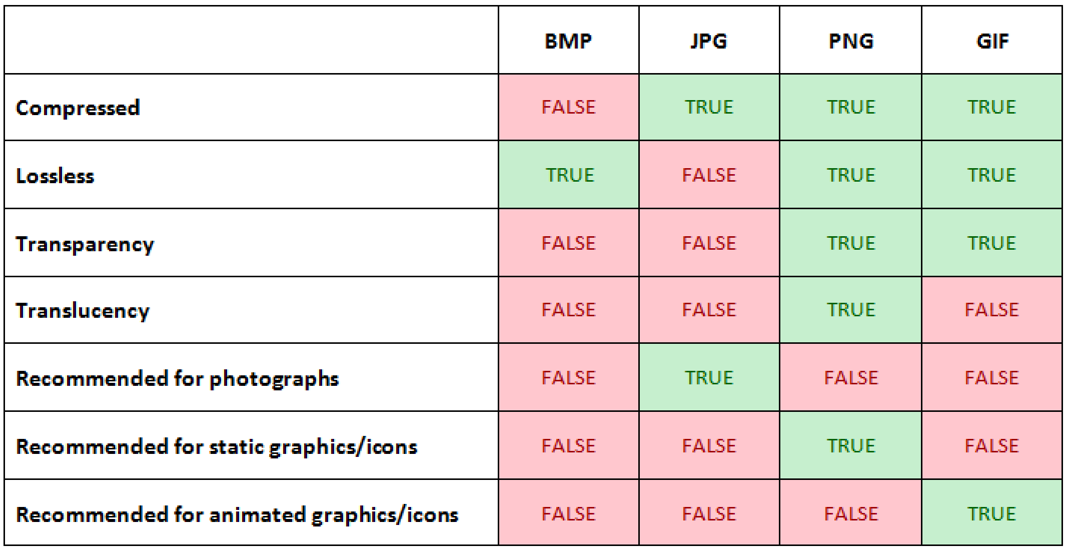
Microsoft originally developed the BMP format for its Windows operating system to maintain the resolution of digital images on different screens and devices. Nowadays, BMP files are also used on other devices such as the Mac and Android. Microsoft developed the BMP file format as a solution to make their devices independent of graphics adapter hardware. [12]

BMP is considered a device-independent bitmap (DIB). A DIB contains a color table that describes how the pixel values correspond to the RGB color values. A DIB also contains the following color and dimension information:

- The color format of the device on which the rectangular image was created.
- The resolution of the device on which the rectangular image was created.
- The palette for the device on which the image was created.

- An array of bits that maps RGB values to pixels in the rectangular image.
- Data compression identifier that indicates the data compression scheme used to reduce the size of the array of bits. [13]

Although BMP has some advantages, it is still considered undesirable by web developers and software developers due to the lack of compression.



	BMP	JPG	PNG	GIF
Compressed	FALSE	TRUE	TRUE	TRUE
Lossless	TRUE	FALSE	TRUE	TRUE
Transparency	FALSE	FALSE	TRUE	TRUE
Translucency	FALSE	FALSE	TRUE	FALSE
Recommended for photographs	FALSE	TRUE	FALSE	FALSE
Recommended for static graphics/icons	FALSE	FALSE	TRUE	FALSE
Recommended for animated graphics/icons	FALSE	FALSE	FALSE	TRUE

Figure 6: Image file comparisons
[14]

5.2.4 Lossless vs. Lossy Compression

Image compression methods can be broadly designateds into lossless and lossy techniques. Lossless compression retains all original data which means when an image is compressed and later decompressed, it is identical to the original. This is very crucial for steganography beacuse hidden information embedded in pixel bits remains complete. Formats like PNG are lossless, making them excellent for storing hidden data without risking corruption during compression.

Lossy compression used in formats such as JPEG removes data deemed visually insignificant to reduce file size. This does improve storage efficiency although it can distort or completely destroy hidden information and data embedded in the least significant bits (LSB). Minor optimizations or encoding to JPEG can render the concealed message unreadable.

For steganographic applications, lossless file formats are preferred because they preserve pixel integrity, ensuring that the embedded data remains undetectable yet recoverable. This makes PNG the optimal choice for projects requiring both security and reliability when it comes to steganography. [15]

5.3 Least Significant Bit

LSB is a common method for hiding information within digital media by exploiting the least significant bit (LSB) of a pixels sample values. These bits got very little effect on the overall visual appearance of the file, so the changes that are being applied are almost impossible to notice. The process in it self involves converting the message into a sequence of bits and then by embedding them into the host file by finding the LSB in the host file. This allows a large amount of data to be hidden in the file while the changes would be minuscule and difficult to detect, unless the person would zoom in on the picture and analyze the picture very thoroughly.

The LSB technique is common as it is efficient, simple and offers a high capacity for embedding data. Nevertheless, it does have some weaknesses. If the file is changed or processed into a different format, such as JPEG with certain optimizations or compression settings, the hidden data can be affected or lost. But when talking about PNG it is a lossless format and generally preserves embedded bits better than for example JPEG, operations like resizing, filtering, or re-saving with different tools will still be able to alter the least significant bits. It is also possible to detect patterns introduced by LSB changes through analysis, without encryption the hidden message can be easily recovered. For these reasons, modern approaches often combine LSB steganography with encryption or randomization to make it more secure. [16]

5.4 GCD and Euclid's Algorithm in the Encoding Process

Euclid's algorithm is an effective method for finding the greatest common divisor (GCD) of two integers, which means the largest number that divides both without leaving a remainder. The pseudocode below illustrates how Euclid's algorithm operates within a broader encoding process.

```
START
    INPUT x, y
    WHILE y is not 0 DO
        temp = y
        y = x MODULO y
        x = temp
    END WHILE
    OUTPUT x  // x is the GCD
END
```

Figure 7: Pseudocode of GCD

5.4.1 Process Overview

The algorithm starts by taking two values, x and y, extracted from the input (which in this case are the image pixel values). The decision point then checks whether y = 0:

- If yes, the algorithm returns x as the GCD
- If no, the algorithm replaces x with y and y with the remainder of x ÷ y and then repeats the check

This loop continues until y becomes zero, ensuring that the last non-zero value of x is the GCD. This particular principle works because any common divisor of x and y also divides their remainder, preserving the set of common divisors through the interactions.[17]

5.4.2 Integration in Encoding

Within the flowchart, Euclid's algorithm is applied after retrieving numeric values from the image. Upon computation of the GCD, it assists the encoding process by ensuring a substantially lesser detectable pattern. [17]

5.5 Security Regarding Steganography

5.5.1 Encryption

Modern stenographic systems combine stenography and cryptography to create a complex layered security approach. The operation typically involves encrypting a message by using algorithms like AES (symmetric encryption) or RSA (asymmetric encryption) before embedding it into the image. The encryption ensures that even if an attacker detects and extracts the hidden data, the content reaming unreadable without the correct decryption key.

The combining of these two methods ensures confidentiality because the encryption protects the actual message from unauthorized access and steganography hides the presence of the message, reducing the likelihood of detection, while keeping the message safe.

In practical applications, this dual approach is widely used in secure communications, digital watermarking, and data protection for sensitive environments such as government, healthcare and finance. Integrating steganography and encryption, organizations can achieve robust security. [18]

5.5.2 Decryption

Decryption is the processes of converting encrypted data back to its original readable form after it has been extracted from a steganographic object. The combination of cryptography and steganography means that the hidden message is first encrypted before embedding and at the receiver's end, the reverse steps occur: extract the ciphertext, then decrypt is using the correct key and algorithm.

This step is critical because steganography alone only hides the existence of the message, without decryption the extracted data remains unintelligible. Common algorithms include AES (symmetric encryption) and RSA (asymmetric encryption). The security of decryption depends on proper key management because if the key is lost or compromised, the message cannot be recovered. This layered approach makes sure that the confidentiality is kept with the secure communication. [18]

5.5.3 AES and RSA Encryption

AES (Advanced Encryption Standard) is a globally adopted symmetric encryption algorithm, utilizing the same encryption and decryption key. RSA on the other hand is a asymmetric encryption algorithm which utilizes both parties, having a public and private key. Using AES, both parties need to have attained the same key in a confidential manner, before sharing secret messages. Using RSA, you encrypt your secret message with your recipients public key. Afterwards, the recipient uses his private key to decode the secret message. The figure below illustrates how these different encryption methods work and what the different steps are whether you are using symmetric (AES) or asymmetric (RSA) encryption.

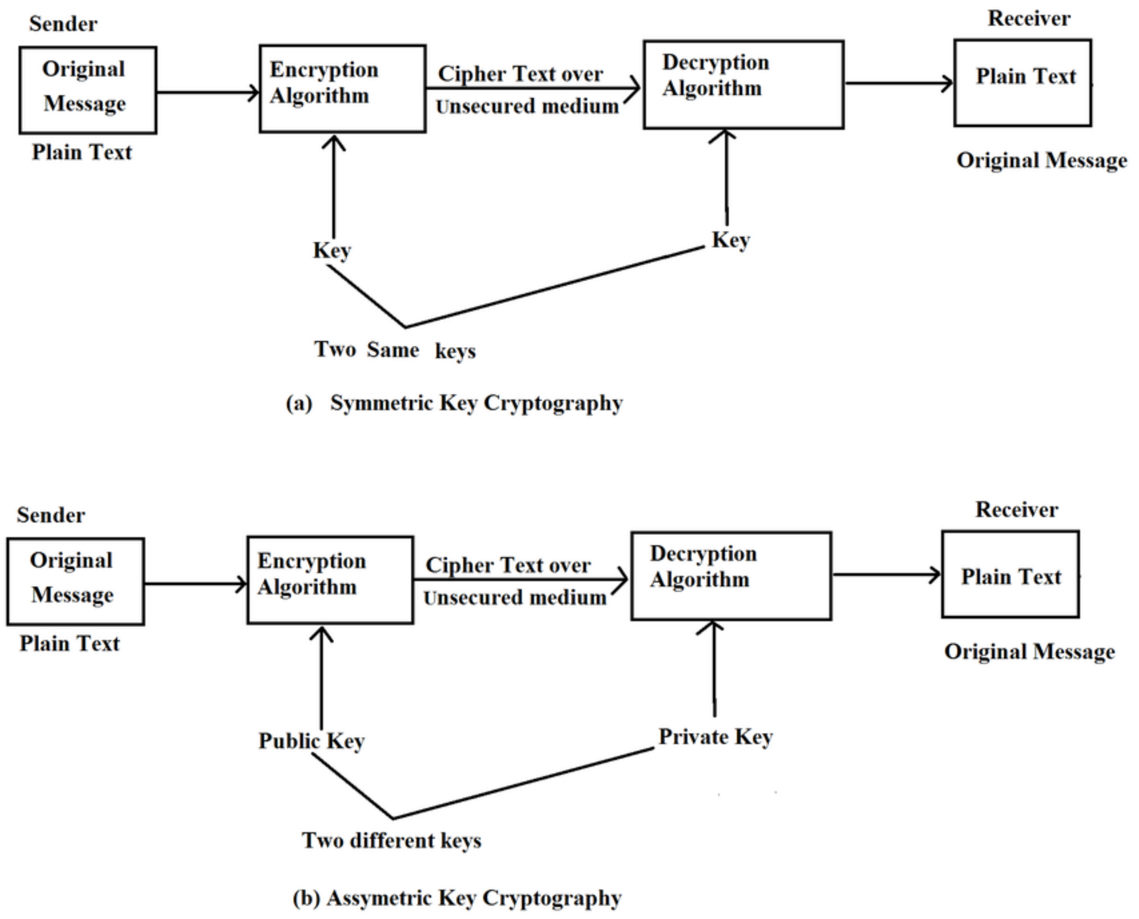


Figure 8: AES and RSA Encryption
[19]

5.5.4 Steganography in Cybersecurity

Steganography is increasingly being exploited in cybersecurity as a method to conceal malicious payloads within seemingly harmless images. This approach allows attackers to bypass traditional security measures relatively easily due to outdated traditional antivirus and security detection. Traditional antivirus and security detection focuses on executable files and known signatures, rarely inspecting image files for hidden binary patterns, which is the method we know as LSB embedding. Because of the vast amount of image data exchanged every day, performing through analysis requires significant resources, making detection even more challenging.

The concealed data in the stenographic content can include malware, ransomware or command-and-control instructions which enables attackers to establish backdoors or extract sensitive information without detection which is a big problem especially if the attacker gets access to a high profile employee or CEO. [20]

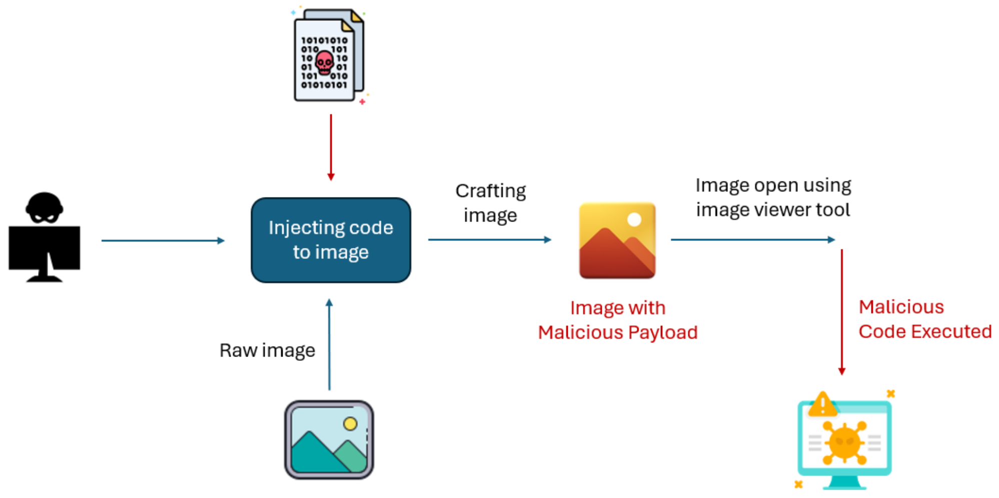


Figure 9: Attack Patterns Regarding Steganography
[20]

5.5.5 Why is Steganography So Hard To Detect?

Detection in steganography with regard to images is challenging because of the modifications introduced are almost invisible. Techniques like LSB embedding which has been mentioned before, alter only the smallest bits of pixel value which has almost no impact on the image appearance. This means that even high resolution images with hidden data looks identical to their original form. Lossless formats like PNG preserve these subtle changes, making the hidden data resilient against visual distortions or imperfections.

The reason steganography is so hard to detect is because traditional antivirus and security scanners rarely inspect image content at the bit level, allowing malicious payloads to bypass the detection protocols. Attackers therefore often combine steganography with encryption ensuring that even if the hidden data is extracted, it remains unreadable without the correct key. These different factors collectively makes steganography based attacks stealthy and difficult to identify using conventional antivirus and security detection. [20]

5.5.6 How To Defend Yourself Against Steganography

When defending against steganography based threats it requires a multi layerd approach. Organizations can deploy steganalysis tools to identify anomalies in image structures or statistical patterns, implement strict content filtering policies to limit image uploads and downloads in sensitive enviroments, and monitor network traffic for unusual behavior that may indicate undercover communication or anomalies. User education plays a critical role aswell reducing risk by promoting awareness of the dangers associated with downloading images from untrusted sources.

With the growing use of steganography in cybercrime underscores the need for integraing advanced detection methods and security practices. Understading these attack vectors will help organizations mitigate the risks posed by this evolving threat. [20]

6 Analysis

6.1 Python

In this project, we use python as our programming language. Python is a high-level language with lots of versatility, and many modules/libraries, such as Pillow, which we use to load an image, and manipulate it to our liking.

Python was created and released by Guido van Rossum in 1991. It's designed to to be easy to read and write, making it the ideal choice for beginners learning to code. Opposed to C or Java, Python focuses on readability, and fast development, rather than a deep control over hardware, such as memory blocks. [21]

When working with Python, there is no need for compiling the program before execution, making it easier to debug, and apply fixes. That in turn makes running Python code quite easy. We use an interpreter, which translates the code to "machine language" line by line, as the program is running live.

Opposed to C, which translates (compiles) the entire program before execution. By operating in such a way, the entire program needs to be recompiled whenever a minor fix, or other changes has been made. This also makes it hard to know which functions work, and which needs fixing. The `error messages` that C compilers give, is not always fulfilling enough, to know what causes the error.

Python runs the program live, only terminating whenever it runs into an error, making it easy to know when/where the program causes errors. Python is intuitive and readable for human comprehension, which makes it easier to hand-off to each other and continue the work. [22]

6.1.1 Required Libarys

To be able to make a functioning program, we have decided to use a few library's which are necessary to run the program the 2 is pillow and pycryptodome. Pillow is a library that allows us to manipulate images and perform some image processing tasks, its fast for our use case which is loading the image and then modify it with the hidden message and save the new image. The next library is pycryptodome that allows us to encrypt the message before embedding it to the image. In our case we are only importing the AES function to encrypt our message with CBC mode. There is one more module that we are using but it is not required since it is preinstalled with python, but OS is just as important, we use it to read write and create a file where we keep the key in.

6.2 PNG, Chosen File Format

Originally during the product planning phase, we wanted to use BMP files. We wanted to use BMP image files, as they are highly versatile, which is an uncompressed image file format making files very large in size, which makes it easier to hide data within the color bits of the images. In addition, being able to be stored, displayed across different devices and screens without losing quality, and are easy to edit and compress with numerous different programs.

Due to PNG image files being a more widely used image file format and also being lossless in image data, we chose to work further with PNG image files in our project. Although the PNG image file format does not have as large a size as BMP, it is still a lossless compressed format, which makes the file format bigger than file formats like JPEG and other compressed file formats. PNG is a 32 bit file format that opens up for more possibilities in regards to hiding data in the least significant bit.

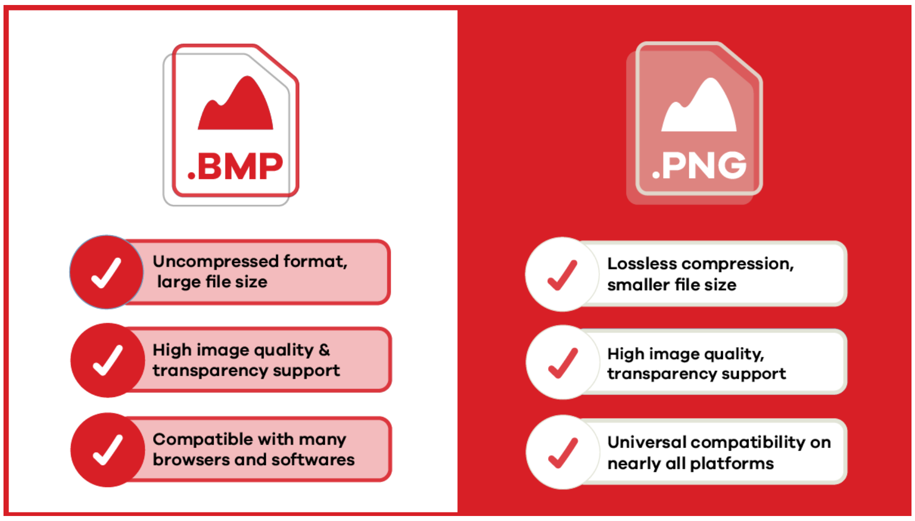

.BMP
Uncompressed format, large file size
High image quality & transparency support
Compatible with many browsers and softwares
.PNG
Lossless compression, smaller file size
High image quality, transparency support
Universal compatibility on nearly all platforms

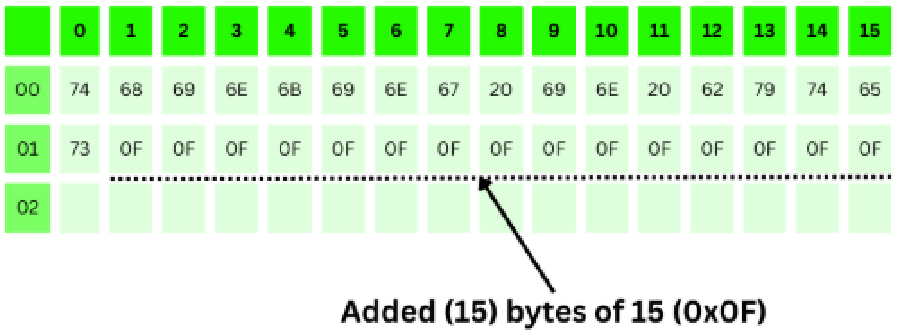
Figure 10: BMP VS PNG
[23]

6.3 Layered Security

The goal is to only let the sender and recipient read/understand the data. To do so, we cannot simply rely on a single security measure. We must assume that an adversary might detect the steganography, and may be able to extract the hidden bits. Therefore the use of AES is straightforward, as its implementation is also relatively simple with python. Having this layered security, assures the goal of confidentiality.

6.4 AES

The symmetric encryption algorithm AES is widely adopted due to its efficiency, and strong security properties. Our project focusses on steganograpyhy as a means to hide data by obscurity. But as Shannon's Maxxim states "The enemy knows the system" We must assume that if our secret embedded data were to ever be compromised, the contents of the data must still remain confidential. AES is the encryption standard chosen by NIST (AES is even an abbreviation of Advanced Encryption Standard) with good reason. AES is higly effecient, which allows it to run on farely primitive hardware by today's standards. The mathematical complexity, and large 128/256 bit-size key, also helps making it highly effective against brute-force attacks.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	74	68	69	6E	6B	69	6E	67	20	69	6E	20	62	79	74	65
01	73	OF	OF	OF	OF	OF	OF	OF	OF	OF	OF	OF	OF	OF	OF	OF
02																

Added (15) bytes of 15 (0x0F)

Figure 11: Block 00 and 01, 01 with padding [24]

As AES is a block-cipher every block needs to be 16-byte/128bit. With that in mind we do some padding to ensure that the length of the last block is 16-byte. We use Pkcs#7, which essentially just repeats "*0x0B*" until the 16-byte mark is reacehed for the last block.

6.5 LSBs role in the program

In the program, we have decided to hide the data in the least significant bit of the color red. We chose to just manipulate one color because it simplifies the implementation of the coding. The PNG image file format has 8 bits for a given color stream in each byte, which means that the last bit has a very minor impact on the value of the byte. The change is so minor that the difference cannot be seen with a human eye.

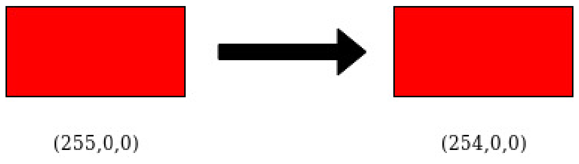


Figure 12: Significant bit difference
[25]

It is possible to change 3 bits per pixel without the change being noticeable, as it would be possible to change a bit in the colors green and blue aswell. This would triple the capacity of the space to hide data. For data capacity to be stored in a digital picture, the following mathematical formula can be used, where $Capacity$ is the number of pixels in the picture, W is the width of the picture and H is the height;

$$Capacity = W * H$$

This would mean if a picture has the size 1920x1080 and only 1 color is being edited then that would be 2,073,600 pixels, and also bits as we only use one bit in each pixel. This equals approximately the possibility of 253 KB of hidden data. The bits to KB value comes from the following formula where KB is the amount of data in kilobytes, divided by 8 converts bits to bytes and divided by 1024 converts bytes to kilobytes;

$$KB = bits/(8 * 1024)$$

Using all 3 streams of colors increases the capacity of hidden data, which can be seen using the same formula as before;

$$Capacity = 3 * W * H$$

For a picture with the size of 1920x1080 with 3 colors being edited, it would equal 6,220,800 bits, which is approximately 759 KB of hidden data.

It is important to note that any additional compression, conversions to different file formats, image filters, and resizing would destroy embedded bits in a digital picture.

6.6 Embedding The Information

Euclid's algorithm plays a crucial role when determining the embedding pattern for steganography. The algorithm calculates the GCD of the image's width and height, which we then use to distribute the hidden bits across the image in a structured yet obscure way. This approach ensures that the data is not concentrated in a single region which reduces the likelihood of detection while maintaining consistency for decoding.

The GCD based pattern works as follows: for each pixel at coordinates (i, j), the algorithm checks if (i + 1) * (j + 1) is divisible by the GCD. If this is true that specific pixel is selected for embedding a bit of the secret message. This method introduces a deterministic but irregular distribution, making the hidden data harder to detect through simple statistical analysis.

When using Euclid's algorithm it provides two main benefits:

- **Efficiency:** The algorithm in it self is simple and fast, even for the larger images, ensuring minimal overhead during encoding and decoding.
- **Unpredictability:** Spreading the data based on a mathematical pattern rather than sequentially, the embedding process in it self becomes less predictable which adds an extra layer of obscurity.

The integration of GCD in the steganographic process regarding our product exhibits how classic algorithms can enhance modern security capability by combining simplicity with effectiveness.

7 Implementation

Our program is split into multiple files: `Stegov2` for Steganograhy and `Crypto_utils` for Cryptography. This structure makes the files become more readable while also making the maintenance simple since keeping the different functions and the structure itself that inturn makes it easier to figure out where the bugs appeared from. As an example if something is wrong with the encryption of the message. The bug is most likely within the `Crypto_utils` or the function that calls it within the `Stegov2`. in the beginning we had a different approach, where we used heavily inspiration instead of interpreting in our own way, that is the reason behind version 2 stego. As mentioned we have different files that do different things. In the next part will get into each part separately and explain the functionality of the code.

7.1 Flowchart

A flowchart is a visual representation of a process using a variety of different shapes, it helps making a complex process easier to understand.

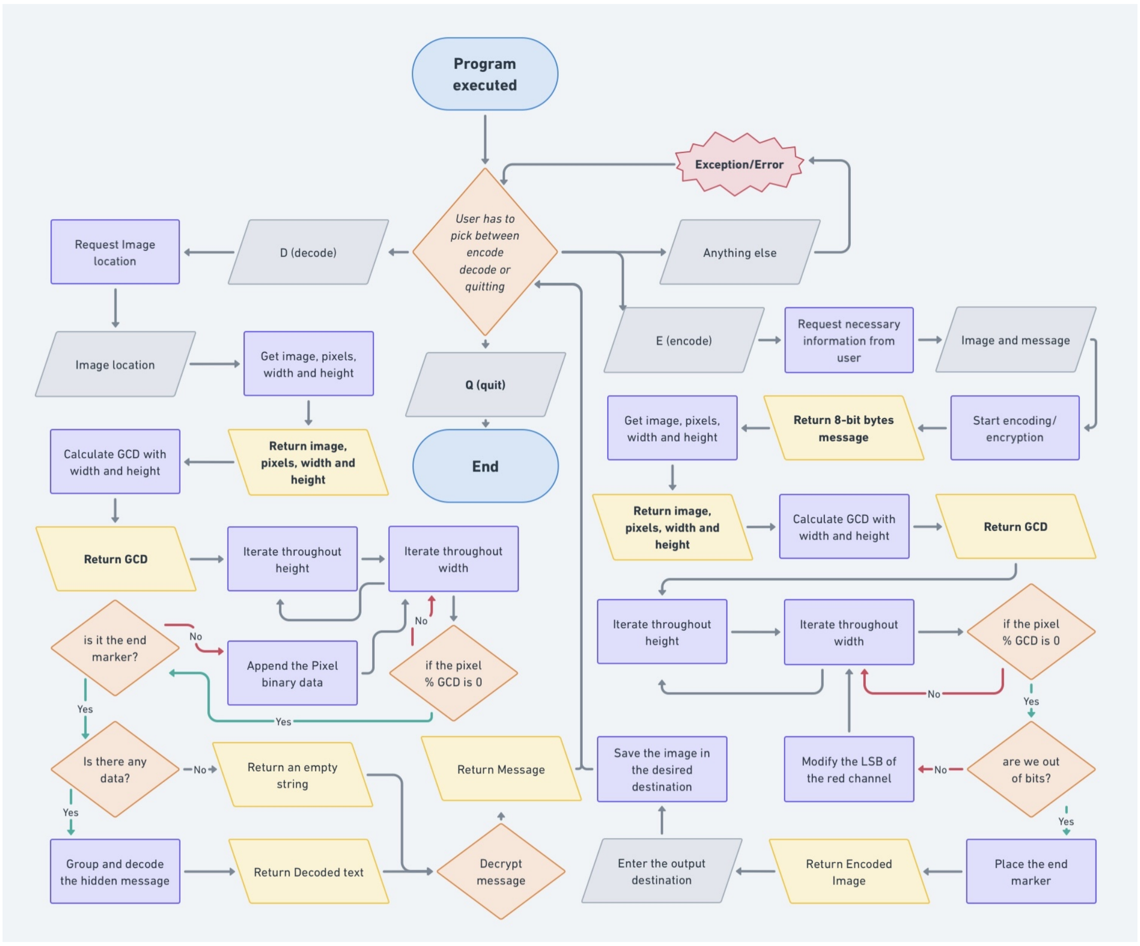

Program executed
Exception/Error
User has to pick between encode decode or quitting
D (decode)
Request Image location
Anything else
Image location
Get image, pixels, width and height
Q (quit)
E (encode)
Request necessary information from user
Image and message
Calculate GCD with width and height
Return image, pixels, width and height
End
Get image, pixels, width and height
Return 8-bit bytes message
Start encoding/ encryption
Return GCD
Iterate throughout height
Iterate throughout width
Return image, pixels, width and height
Calculate GCD with width and height
Return GCD
No
is it the end marker?
No
Append the Pixel binary data
if the pixel % GCD is 0
Iterate throughout height
Iterate throughout width
if the pixel % GCD is 0
Yes
Yes
No
Yes
Is there any data?
No
Return an empty string
Return Message
Save the image in the desired destination
Modify the LSB of the red channel
No
are we out of bits?
Yes
Yes
Group and decode the hidden message
Return Decoded text
Decrypt message
Enter the output destination
Return Encoded Image
Place the end marker

Figure 13: Flowchart of the implementation
[7]

7.2 Steganography

The Stegov2 file as the name implies is where we embed and extract the message that the user has inputted. In the implementation we will use both LSB and GCD. Although, the message the user inputs needs to not only be encrypted it needs to be converted to bytes that we then can embed in the image. So we start with the text-to-binary that takes the message (msg) and sends it to a function within the Crypto-utils file. After the message has been encrypted we need to convert it from hexadecimal to bytes since the output from the Crypto-utils is hexadecimal. Then we can return the message by taking the bytes and making them into 8-bit bytes.

```python
def text_to_binary(msg):
    ciphertext_hex = encrypt_data(msg)
    ciphertext = bytes.fromhex(ciphertext_hex)
    return ''.join(format(byte,'08b') for byte in ciphertext)
```

Figure 14: Text_To_Binary

For this part of the code we took the principles of Euclid's algorithm for the function **gcd** which is short for "greatest common divisor", so the function works by taking 2 numbers **x** and **y**, it then enters a loop and will continue as long as y is not zero. inside the loop it replaces **x** with **y** and **y** with the remainder of **y** using python's parallel (tuple) assignment. when **y** becomes zero the loop stops and the current value **x** is returned as the GCD.

```python
def gcd(x,y):
    while y != 0:
        x, y=y, y % x
    return x
```

Figure 15: GCD

But how does python's parallel (tuple) assignment work? At first python looks at the right hand side before making any assignments which is.

```python
    y, y % x
```

Figure 16: GCD

It evaluates both numbers using the old values of x and y, lets say **x = 12** and **y = 8**, so the first value is 8 and it will get assigned to x and the second value **8 % 12 = 8**, so 8 will be assigned to y.

```
(8,8)

x = 8
y = 8
```

Figure 17: Example on GCD

The **get_image** function loads an image from a file and returns some useful information we need to process the image. it uses the **Image.open** function to load the image from a file path and then converts the image to RGB color mode so that every pixel is represented by 3 color values using **.convert("RGB")**. to access the pixels we use **.load()** to return a pixel access object not an array from the library pillow, this allows us to read or modify individual pixels using **pixels[x, y]** we then get the image dimensions that we use for the gcd function with **.size** it returns a tuple with the **width** and **height**. now we return the image object, the pixel access object and the image dimensions.

```python
def get_image(location):
    img = Image.open(location).convert("RGB")
    pixels = img.load()
    w, h = img.size
    return img, pixels, w, h
```

Figure 18: Get_Image

For the **encode_image** function we embed the secret message into the image using **LSB**. first we convert the message to binary using the **text_to_binary** function. each char is then represented as 8 bits. we then prepare the image for encoding by locating and retrieving the necessary information with **get_image** function. The embedding pattern is calculated using the **GCD** of the image width and height so that its more spread out.

```python
    binary = text_to_binary(msg)    # convert message to binary
    bit_index = 0 # index to keep track

    img, pixels, w, h = get_image(image_location) # get image and its pixels
    pattern = gcd(w, h) # calculate the pattern using gcd
```

Figure 19: Encode_Image

to embed the binary message, the function will iterate over all the pixels in the image. For a pixel at coordinates **(i, j)**, the pixel is selected for embedding if **(i+1) * (j+1)** is divisible by the **GCD**. Each selected pixel's red channel is modified to store one bit of the message by changing its **LSB** while the green and blue channels remains unchanged. Once all bits of the message are embedded the functions marks the end of the message by setting the red channel of the next pixel in the GCD pattern to **0**, this allows us to find the marker later when we decode the message. after embedding the full message and adding the end marker, the image is then returned.

```python
    for i in range(h): # iterate through height
        for j in range(w): # iterate through width

            # logic
            if ((i + 1) * (j + 1)) % pattern == 0: # position matches the pattern

                # check if we're out of bits
                if bit_index >= len(binary):
                    # end marker → red = 0
                    r, g, b = pixels[j, i]
                    pixels[j, i] = (0, g, b)
                    return img

                # read next bit
                bit = int(binary[bit_index])
                bit_index += 1

                # modify LSB so the color red
                r, g, b = pixels[j, i]
                r = (r & ~1) | bit
                pixels[j, i] = (r, g, b)
    return img  # return the image
```

Figure 20: Encode_Image

To decode the embedded binary message from the image, the image is loaded with **get_image** providing the information needed to decode the message. the GCD pattern is then calculated so we insure that we are reading the pixel in the same sequence as in the encoding process. the function iterates over each pixel in the image. if a pixel at **(i, j)** matches the embedding pattern, if **(i+1) * (j+1)** is divisible by the **GCD**, the **LSB** of the red channel is extracted. the process continues until the message marker is found, which was indicated by a pixel where the red value is 0. to reconstruct the message all the bits are grouped into 8-bit sequences representing individual bytes. these bytes are then converted into a hexadecimal string which is the original message, if no data is found before finding a marker, an empty string is returned. the function then outputs the hidden message in hexadecimal format to further decrypt.

```python
def decode_image(image_location):
    img, pixels, w, h = get_image(image_location)
    binary_data = []
    pattern = gcd(w, h) # calculate the pattern using gcd

    for i in range(h): # iterate through height
        for j in range(w): # iterate through width
            if ((i + 1) * (j + 1)) % pattern == 0: # position matches the pattern
                r, g, b = pixels[j, i]
                if r == 0:

                    if not binary_data:
                        return ''

                    list_of_bytes = [''.join(str(b) for b in binary_data[i:i+8])
                    for i in range(0, len(binary_data), 8)]
                    cipher_bytes = bytes(int(byte, 2)
                    for byte in list_of_bytes if len(byte) == 8)
                    return cipher_bytes.hex()
                binary_data.append(r & 1)
```

Figure 21: Decode_Image

7.3 Cryptography

As mentioned in 6.4 under analysis that data encrypted with AES needs to be a certain length. So, to make sure that the data is the 16 bytes we make a check. We do that by first converting the message from a string to UTF-8 (bytes). Then we use the modulo operator % which gives us the remainder. That means when we take the length of the message to modulo 16 it will return a number that we then subtract with 16 to know how much to add before it becomes 16 bytes even if the message is 16 bytes. Then we combine the message with the padding amount and return it.

```python
def padding(msg):
    if isinstance(msg, str):
        msg = msg.encode('utf-8')
    padding_length = 16 - len(msg) % 16
    padding_bytes = bytes([padding_length] * padding_length)
    return msg + padding_bytes
```

Figure 22: Padding

How do we handle the key used for encrypting or decrypting, for a more secure encryption the **load_key** function manages the creation and retrieval of a symmetric encryption key and an initialization vector (iv) that will add randomness so the same message don't look the same. It checks for an existing key file named **.key**. This file is where we store both the encryption ket and the iv. if the **.key** file does not exist or if the size is not 48 bytes, the function generated a random 32-byte ket and a 16-byte iv, these values are then saved to the **.key** file as a single 48-byte sequence. if the **.key** file exists and has the correct size 48-bytes, the function reads the file and splits the data, the first 32 bytes are the encryption key and the remaining 16 bytes are the iv, and then returns it to be used.

```python
def load_key():
    if not os.path.exists(".key"):
        key = os.urandom(32)
        iv = os.urandom(16)
        with open(".key", "wb") as key_file:
            key_file.write(key + iv)
    elif os.path.getsize(".key") != 48:
        key = os.urandom(32)
        iv = os.urandom(16)
        with open(".key", "wb") as key_file:
            key_file.write(key + iv)
    elif os.path.getsize(".key") == 48:
        with open(".key", "rb") as key_file:
            data = key_file.read()
            key = data[:32]
            iv = data[32:]
    return key, iv
```

Figure 23: Load_Key

the encrypt the users message we use the **encrypt_data** function which provides a symmetric encryption using AES algorithm in CBC. first we load the key, the 32-byte symmetric key and the 16-byte iv. we then pad the message. The AES cipher object is created using the loaded key, CBC mode and the loaded iv. The ciphertext is then converted to a hexadecimal string **hex()**, if the encryption fails the function prints and error message and returns nothing.

```python
def encrypt_data(msg):
    key, iv = load_key()
    try:
        padded_msg = padding(msg)
        cipher = CryptoAES.new(key, CryptoAES.MODE_CBC, iv)
        ciphertext = cipher.encrypt(padded_msg).hex()
        return ciphertext
    except Exception as e:
        print("Encryption error:", e)
        return None, None, None
```

Figure 24: Encrypt_Data

To decrypt the ciphertext we use the **decrypt_data** function which is just reversing the process performed by **encrypt_data**, recovering the original message. it uses AES in CBC mode with the pre-shared key and iv. We first load the key, the 32-byte symmetric key and the 16-byte iv. the imputed ciphertext which should be in hexadecimal is then getting decrypted using the AES cipher object with the loaded key, CBC mode and the loaded iv. then we remove the padding and decode from bytes to a UTF-8 string recovering the original message. if it fails for any reason, the function prints an error and returns nothing.

```python
def decrypt_data(ciphertext_hex):
    key, iv = load_key()
    try:
        ciphertext = bytes.fromhex(ciphertext_hex)
        cipher = CryptoAES.new(key, CryptoAES.MODE_CBC, iv)
        decrypted = cipher.decrypt(ciphertext)
        print("Decrypted (raw):", decrypted)
        return unpadding(decrypted).decode("utf-8")
    except Exception as e:
        print("Decryption error:", e)
        return None
```

Figure 25: Decrypt_Data

As can be seen in the return function the function unpadding is getting called with the parameter decrypted message.

The unpadding takes the message and looks at the last bytes and by looking from the end of the string it can register how much padding has been added. If the last part is **0x04** then there had been added 4 bytes of padding. Then it verifies that the padding length is within the 16 byte block cipher that AES has the output. If its over or under that then it could be incomplete data and the system breaks. Then we return the message and removing the padding in bytes that then gets changed from bytes to UTF-8 in decrypt.

```python
def unpadding(msg):
    padding_length = msg[-1]
    if padding_length < 1 or padding_length > 16:
        raise ValueError("Invalid padding encountered")
    return msg[:-padding_length]
```

Figure 26: Unpadding

7.4 Text-based User Interface

We use a different file to handle the user interface for the steganography. it allows the user to encode the messages within an image. it displays a banner and a brief description of its purpose, it has an interactive menu where the user can pick to either encode, decode or quit the program. the encoding workflow requires the user to provide a path to the image and the message to hide, then the message is encoded and encrypted which is saved to the specified output. For the decoding work flow the user provides the path to the image and then decrypts it. And if the user chooses to quit the program it will exit the loop. Any invalid options will tell the user to choose a valid option.

```python
def main():
    print_banner()
    print("Steganography using GCD Pattern")
    while True:
        choice = input("Choose (e)ncode, (d)ecode or (q)uit: ").lower()
        if choice == 'e':
            image_location = input("Enter image file path: ")
            msg = input("Enter message to hide: ")
            encoded_img = encode_image(image_location, msg)
            output_path = input("Enter output image file path: ")
            encoded_img.save(output_path)
            print(f"Message encoded and saved to {output_path}")
            continue
        elif choice == 'd':
            image_location = input("Enter image file path: ")
            hidden_msg = decode_image(image_location)
            decrypted_msg = decrypt_data(hidden_msg)
            print("Hidden message:", decrypted_msg)
            continue
        elif choice == 'q':
            print("Quitting the program.")
            break
        else:
            print("Invalid choice. Please choose 'e', 'd', or 'q'.")
            continue
```

Figure 27: Main

8 Discussion

8.1 The Use of Steganography

The products primary strength lies in hiding data in an image without visibly noticing it, making it particularly valuable in situations where discretion is critical. Application regarding our product includes protecting privacy and securing sensitive data without making it possible for adversaries to become aware of.

If we write bits in all three color channels (r, g and b), we get more space, but the images is changed further and it becomes easier to detect with simple statistical tests. If we only use a single LSB per pixel and keep the payload small, the image changes less and the risk of detection drops because the size in particular does not look suspicious. To avoid obvious patterns, we distribute bits across the image via a GCD pattern and when possible switch channels in a key-driven way so changes are harder to spot.[16]

8.2 Security

While hiding data in images reduces the likelihood of detection, it does not guarantee confidentiality if the content is extracted. To address this, our product uses AES encryption before embedding, ensuring that even if an adversary discovers the steganographic payload and gains knowledge to the data, the message remains unreadable without the correct key. Poor key management on the other hand can completely undermine security, even with the use of algorithms like AES. For example, if encryption keys are stored in plain text or reused across multiple sessions, attackers can easily compromise confidentiality. Keys must be generated using secure random sources, stored in protected environments, and rotated periodically to reduce the risk of exposure. Without keeping these practices in mind, steganography combined with AES encryption only offers an illusion of security, as the hidden data becomes vulnerable once the key is leaked or mismanaged.

Malicious actors often use techniques like LSB embedding to bypass traditional antivirus and detection systems, which rarely inspect image files at the bit level. This makes detection very difficult and resourceful for organizations and the modifications introduced with LSB embedding it is almost invisible to see for the human eye, unless you do a deep analysis of the image.[26]

Organizations often deploy steganalysis tools, enforce strict content filtering, and monitor network traffic for anomalies. The big problem here is that organizations often do not have the resources nor the time, or are not even aware. When they lack steganalysis tools or are not even aware adversaries could do data exfiltration without detection. If a breach occurs and the organization identifies it, finding the root cause becomes harder because steganographic payloads are stealthy and the amount of files and organizations receive is quite large. This can lead to a prolonged investigations, higher recovery costs and reputational damage, especially if the organization complies with GDPR.

8.3 Limitations

As we are utilizing the LSB method, changing the present RGB (red) value, the size of the hidden data stored is highly affected by the number of pixels having an R value <1 present in the picture. For example, if we had a pure black/blue/green image, we would not be able to store any hidden data, as there are no pixels having any R value. This means the "hidden storage capacity" linearly grows as the file size grows

Theoretically, we would be able to store 253.125 kb data in a 6075kb

$$Imagesize : 6075KB = 6075 * 1024 = 6.220.800byte$$

Each pixel uses 3 bytes (RGB), so numer of pixels =:

$$6.220.800/3 = 2.073.600$$

We're only altering the red channel storing 1 bit per pixel, that is 2.073.600 bits of hidden data, converting to bytes, we have:

$$2.073.600/8 = 259200bytes = 253,125kb$$

This means we have a theoretical ratio of "hidden storage capacity" of about 1:24 per file size But this is purely the case, if we have an image, where all the pixels contain an R value of above 1. If that is not the case, the "hidden storage capacity" could be greatly decreased. If we were to utilize all 3 RGB channels, the theoretical capacity would be thrice as big, with a ratio of 1:8.

Another important limitation appear from the use of our GCD-based embedding pattern. A pixel is only used for hiding a bit when (i + 1) * (j + 1) is divisible by our GCD. This pattern increase obscurity and reduces the chances of detection, but it also significantly decreases the effective data capacity of the image. Since only one out of every GCD pixels is selected for embedding, the theoretical maximum capacity W * H bits is reduced to W * H / GCD and for many common resolutions, this reduction is significant. A 1920x1080 image has a GCD of 120 and if we place that into our equation it looks as follows 1920 * 1080 / 120 = 17,280 bits[27]. This means although our GCD-pattern improves security through irregular distribution, it imposes a big constraint on payload size. As a result of this, the method becomes inadequate for embedding larger messages and is highly dependent on the chosen image capacity. Therefore images with a large shared divisor between width and height suffer the most from this limitation.

Perhaps the biggest Limitation when deploying steganographic images, lies in the distribution. If we send the file via typical messaging platforms, such as Messenger or Instagram, the file would be further compressed, or even converted to a new file format, ruining the embedded data. This results in, when delivering the PNG with the embedded data, we have to transfer it to the recipient via a platform, that does not alter the file whatsoever. This could be via E-mail or Microsoft Teams, or any other platform that handles the transfer as a file-transfer rather than an image-transfer.

8.4 Future of Stegonography (AI)

When speculating on the future of steganography, the biggest mover is of course, AI. As seemingly everything else in this world, AI will also impact steganography. As presented earlier, with steganography comes steganalysis tools to detect the existence of hidden data. When analysing an image file, steganalysis detection consists of 3 types. Blind, Comparitive and Targeted.
Blind detection, is trying to uncover hidden data by analysing the picture for anomalies, such as unnatural pixel values.
Comparative is comparing the original image, to the received image.
Targeted is focused on detecting specific known stegonagraphic algrorithms or patterns.
AI brings GAN (Generative Adversarial Networks) utilizing Diffusion models, which aims to decieve all 3 detection types. GAN works by a "Generator" generate a realistic/natural image from scratch injecting hidden data during the denoising, which is then parsed to a "Discriminator" which job is to evaluate the realism of the image. This results in a seemingly "empty" image, which only decoders with the right key can extract data from. [28]

The AI model learns to generate images with natural pixel distribution, spreading the data across the entire image. Since the image is generated, not altered, theres also no obvious artifacts or inconsistencies, further hindering Blind detection when analyzing for anomalies. Comparative detection is useless, as the image was created with the sole-purpose of exchanging hidden data, meaning there is no "clean" image to compare it to.

AI may create its own stegonagraphic algorithms and patterns, being one step ahead in the Targeted "cat and mouse" chase

9 Perspectives

A significant limitation of the current system is that users have to share their key, in order to decode a message sent by another person. If this key is transmitted through an insecure or unencrypted communication channel, it can be intercepted, compromising everything. To address this, an implementation using Diffie-Hellman key exchange was discussed as a potential improvement. this method would allow two users to securely make a shared encryption key over an insecure channel without ever having the send the actual key compromising it. due to time constrains, this feature was not implemented, despite the fact it would significantly increase the security and communication.

Another feature that ended up not being implemented due to lack of time, is alternating between RGB color channels when embedding data. in our current implementation, we only use the red channel to store the hidden bits, whilst it is simple and effective, distributing the embedded data across red, green and blue channels could increase both the capacity and security of the process, in theory making it less prone to detection.

Additionally implementing color alternation would be more difficult despite the fact its more secure, however it would require a precise method to determine which color to use at each embedding position. this logic would need to be flawlessly mirrored in the decoding process to avoid data corruption. Furthermore, managing channel switching and bit positions would increase the risk of synchronization errors or bugs. Due to time constraints and the added complexity, this feature was considered but not implemented.

Another feature we discussed was implementing file validation using SHA-256 hashing to verify that the program has not been tampered with. By generating a SHA-256 hash of the original program files, users could compare their local version to a trusted hash, to know wether it has been modified or tampered with.

A further quality of life improvement that could have been added if time allowed to, is calculation a maximum text size per image, as well as implementing a graphical user interface (GUI). in the current implementation, the program does not tell the user if the image they have selected or is the necessary size to store the whole encrypted message, which can lead to incomplete or failed encoding. A capacity check based on the image height and width would allow the program to tell the user their approximate maximum message size.

Furthermore, the program is operated entirely through a command line, which is less intuitive for non technical users. Developing a GUI would make the program more accessible by allowing the user to select images, enter messages and save the output through visual controls.

10 Conclusion

This project set out to explore how steganography and cryptography can be combined into a user-friendly application capable of securely embedding and retrieving hidden messages within digital images. With the integration of AES encryption, LSB embedding and a GCD-based distribution pattern with euclid's algorithm, the product demonstrates a layered security approach that obscures the existence of communication and protects its content.

Practical implementation and theoretical research presents that while steganography alone provides concealment, it remains vulnerable if the data embedded is extracted. Likewise with cryptography, alone it protects the message, but cannot hide different types of communication, that has taken place within a messaging platform. By merging these two methods the product offers both secrecy and confidentiality, strengthening the overall security. The use of PNG as the chosen image format, has proved effective because of its lossless nature persevering the embedded bits without degradation. The use of Euclid's algorithm was to distribute hidden data which reduced predictable patterns and minimized detectability.

Despite the strengths of our product, it does have its limitations. The GCD-based embedding pattern significantly reduces capacity in images with large GCDs and the usage of only the red color channel further constrains storage that can be hidden inside the image. Other practical challenges such as message size limitations, key-sharing requirements, and risk of file compression depending on the communication platform, highlights areas where future improvements are necessary.

The product itself, works as a strong foundation for future development, and shows the potential of steganography as a relevant security technique in the modern cybersecurity landscape, which is increasingly challenged by data interception and digital surveillance.

Overall, the project demonstrates that combining cryptography and steganography can create a robust solution for covert data communication. With further enhancements such as secure key exchange, GUI, multi-channel embedding and capacity estimation, the product could be even more secure and user-friendly.

11 Appendix

https://github.com/Mostafa4800/Hidden-Within

References

[1] Cham Springer. Cryptstego. https://link.springer.com/chapter/10.1007/978-3-031-56728-5_44, Accessed: 2025-12-19.

[2] Molta Danlami. Hybridization of cryptography and steganography to achieve secret communication. https://ijaem.net/issue_dcp/Hybridization%20of%20Cryptography%20and%20Steganography%20to%20Achieve%20Secret%20Communication.pdf, Accessed: 2025-12-19.

[3] Nagham Hamid, Abid Yahya, R Badlishah Ahmad, and Osamah M Al-Qershi. Image steganography techniques: an overview. *International Journal of Computer Science and Security (IJCSS)*, 6(3):168–187, 2012.

[4] GeeksforGeeks. What is steganography? https://www.geeksforgeeks.org/computer-networks/what-is-steganography/, Accessed: 2025-11-14.

[5] Adobe. A guide to image file formats and image file types. https://www.adobe.com/acrobat/hub/guide-to-image-file-formats, Accessed: 2025-11-14.

[6] Willamette University. Image file formats. https://people.willamette.edu/~gorr/classes/GeneralGraphics/imageFormats/, Accessed: 2025-12-06.

[7] William Ette. Image file formats. https://people.willamette.edu/~gorr/classes/GeneralGraphics/imageFormats/, Accessed: 2025-12-01.

[8] Lyna. What is color depth and why does it matter for lcd displays? https://huaxianjing.com/color-depth-processing-for-lcd-panels-with-led-backlighting/, Accessed: 2025-12-19.

[9] Adobe. Jpeg files. https://www.adobe.com/creativecloud/file-types/image/raster/jpeg-file.html, Accessed: 2025-11-11.

[10] Adobe. Png files. https://www.adobe.com/creativecloud/file-types/image/raster/png-file.html, Accessed: 2025-11-11.

[11] Johannes Siipola. What's the best lossless image format? comparing png, webp, avif, and jpeg xl. https://siipo.la/blog/whats-the-best-lossless-image-format-comparing-png-webp-avif-and-jpeg-xl, Accessed: 2025-12-19.

[12] Adobe. Bmp files. https://www.adobe.com/dk/creativecloud/file-types/image/raster/bmp-file.html, Accessed: 2025-11-11.

[13] Microsoft. Dib files. https://learn.microsoft.com/en-us/windows/win32/gdi/device-independent-bitmaps, Accessed: 2025-11-11.

[14] Shumon Saha. Jpeg vs png vs bmp vs gif vs svg. https://superuser.com/questions/53600/jpeg-vs-png-vs-bmp-vs-gif-vs-svg, Accessed: 2025-11-14.

[15] Arpit Bhayani. Everything that you need to know about image steganography. https://www.codementor.io/@arpitbhayani/internals-of-image-steganography-12qsxcxjsh.

[16] Daniel Lerch. Lsb steganography in images and audio. https://daniellerch.me/stego/intro/lsb-en/, Accessed: 2025-11-18.

[17] William Dunham. Euclidean algorithm. https://www.britannica.com/science/Euclidean-algorithm, Accessed: 2025-11-14.

[18] David Tidmarsh. What is steganography in cybersecurity? https://www.eccouncil.org/cybersecurity-exchange/ethical-hacking/what-is-steganography-guide-meaning-types-tools/, Accessed: 2025-11-16.

[19] Researchgate. Block diagram of symmetric and asymmetric keycryptography. https://www.researchgate.net/figure/Block-Diagram-of-Symmetric-and-Asymmetric-key-cryptography_fig1_337689228, Accessed: 2025-12-9.

[20] Lorenzo Langeli. Malware hidden inside images: How steganography works and how to protect yourself. https://8bitsecurity.com/posts/malware-hidden-inside-images-how-steganography-works-and-how-to-protect-yourse
Accessed: 2025-12-9.

[21] GeeksforGeeks. C vs c++ vs java vs python vs javascript. https://www.geeksforgeeks.org/java/c-vs-java-vs-python/, Accessed: 2025-11-10.

[22] GeeksforGeeks. Difference between compiler and interpreter. https://www.geeksforgeeks.org/compiler-design/difference-between-compiler-and-interpreter/, Accessed: 2025-11-16.

[23] Leon Olagh. Bmp vs. png. https://proshotmediagroup.com/blog/bmp-vs-jpeg/, Accessed: 2025-12-05.

[24] Chad Mando. Aes-cbc padding explained. https://thinkinginbytes.com/posts/aes-cbc-padding-explained/, Accessed: 2025-12-11.

[25] BoiteAKlou. Steganography tutorial: Least significant bit (lsb). https://www.boiteaklou.fr/Steganography-Least-Significant-Bit.html, Accessed: 2025-12-09.

[26] Paranoid Cybersecurity. Case study: Caminho malware loader conceals .net payloads inside images via lsb steganography. https://www.paranoidcybersecurity.com/case-study-slug/threat/caminho-malware-loader-conceals-net-payloads-inside-images-via-lsb-steganograph
Accessed: 2025-12-17.

[27] Mathportal. Find gcd of 1920 1080. https://www.mathportal.org/calculators/popular-problems/gcd.php?rb1=primefactmethod&val1=1920%201080, Accessed: 2025-12-17.

[28] Mehdi Mirza Bing Xu David Warde-Farley Sherjil Ozair Aaron Courville Yoshua Bengio Ian J. Goodfellow, Jean Pouget-Abadie. Generative adversarial nets. https://proceedings.neurips.cc/paper_files/paper/2014/file/f033ed80deb0234979a61f95710dbe25-Paper.pdf, Accessed: 2025-12-17.