Secure Transmission via Text-Image Steganography Integrated with RSA and Modified RSA Cryptography: A Comparative Analysis

SHAHEDA AKTER

M. Sc. in

Computer Science and Telecommunication Engineering

Department of Computer Science and Telecommunication Engineering
Noakhali Science and Technology University

Noakhali-3814
Bangladesh

Noakhali Science and Technology University

Department of Computer Science and Telecommunication Engineering

Project Title:
Secure Transmission via Text-Image Steganography Integrated with RSA and Modified RSA Cryptography: A Comparative Analysis
Course Code: CSTE-6101

Submitted by
Name: Shaheda Akter
Roll: BKH2001MSc119F
Session: 2019-20

Department of Computer Science and Telecommunication Engineering,
Noakhali Science and Technology University

Under The Supervision of
Dr. Md. Humayun Kabir
Professor
Department of Computer Science and Telecommunication Engineering,
Noakhali Science and Technology University

Date of Submission: 23.11.2025

The research work entitled **“Secure Transmission via Text-Image Steganography Integrated with RSA and MSA Cryptography: A Comparative Analysis”**, submitted by Shahedaa Akter, Roll No: BKH2001MSc119F has been accepted Master of Science in Computer Science and Telecommunication Engineering in Noakhali Science and Technology University.

Board of Examiners

Affirmation

This is hereby declared that the thesis entitled **"Secure Transmission via Text-Image Steganography Integrated with RSA and MSA Cryptography: A Comparative Analysis"** is an original research work carried out by **Shaheda Akter** in the partial fulfillment of the requirements for the degree of "Master of Science in Computer Science and Telecommunication Engineering as **M.Sc. Engg**." under the department of Computer Science and Telecommunication Engineering, Noakhali.

Supervisor
Dr. Mohammed Humayun Kabir
Professor,
Department of Computer Engineering and Telecommunication Engineering
Noakhali Science and Technology University

……………………………..
Signature of the Candidate

Acknowledgement

The authors would like to thank Dr. Mohammed Humayun Kabir, Professor, Department of Computer Science and Telecommunication Engineering (CSTE), Noakhali Science and Technology University who took time out of his hectic schedule to guide me and provide me with all the necessary materials and sufficient knowledge that was the major requirement.

Abstract

Ensuring the security of data transmission is crucial in resource-sharing over data communication networks. This paper proposes a solution for securely transmitting sensitive information by integrating encryption and steganography. The process starts with the sender inputting a plaintext message through a Java-based graphical user interface (GUI). The message is then converted into ASCII integer values and encrypted using the RSA algorithm, ensuring confidentiality and integrity. To further conceal the encrypted data, the system employs steganography, embedding the encoded message within the RGB color channels of an image. The intensity levels of the red, green, and blue colors are adjusted to hide the encrypted data, producing an image that appears normal to the human eye. For added security, the image is embedded in a Microsoft Word document, containing unrelated content to further obscure the presence of hidden data. This multi-layered approach ensures that, even if intercepted, the hidden message remains undetectable without knowledge of the encryption and steganography methods used. The recipient can extract the image, decrypt the data using RSA, and recover the original message. By combining RSA encryption with text-image-based steganography, this method provides secure, covert, and reliable transmission of sensitive information.

Key words: ***RSA Cryptography, Steganography, Text Cover, RGB Encoding.***

Table of Content

Affirmation i

Acknowledgement ii

Abstract iii

List of Figures vi

List of Tables vii

1. **CHAPTER ONE** 1

INTRODUCTION 1

1.1 Background 1

1.2 Research Objectives 1

1.3 Thesis Organization 2

2. **CHAPTER TWO** 3

RELATED WORK 3

2.1 Literature Review 3

2.1.1 Feno Heriniaina Rabevohitra's approach 4

2.2 Motivation and Contribution 6

3. **CHAPTER THREE** 7

CRYPTOGRAPHY 7

3.1 What is cryptography? 7

3.2 Techniques of cryptography 8

3.2.1 Symmetric Key Cryptography 8

3.2.2 Assymetric Key Cryptography 8

3.3 RSA Algorithm 9

3.3.1 RSA Encryption and Decryption 10

3.3.2 Why RSA is Needed? 11

3.4 Objectives of cryptography 11

4. **CHAPTER FOUR** 12

STEGANOGRAPHY 12

4.1 What is steganography? 12

4.2 Techniques of steganography 12

4.2.1 Text Steganography 12
4.2.2 Image Steganography 13
4.2.3 Audio steganography 14
4.2.4 Video Steganography 14
4.3 Importance of steganography 15
4.4 Cryptography Vs Steganography 15
5. **CHAPTER FIVE** 17
RESEARCH METHODOLOGY 17
5.1 Materials 17
5.1.1 Java Development Environment 17
5.1.2 Frameworks and Libraries 17
5.1.3 ASCII Values 17
5.1.4 Embedding medium: MS Word 17
5.2 Our Proposed System 18
5.3 Implemantation of Using RSA 20
5.3.1 Essential Java Imports for Our Research 20
5.3.2 Code for Encryption 21
5.3.3 Embedding the Image in a Word Document 25
5.3.4 Decryption Process 25
5.4 Implementation Using MRSA 27
5.4.1 Encryption Process 28
5.4.2 Decryption Process 31
FileWriter for Output 31
6. **CHAPTER SIX** 34
RESULT ANALYSIS 34
6.1 The GUI Output 34
6.1.1 Output for RSA 34
6.1.2 Output for MRSA 36
6.2 Analyzing Time Complexity 38
6.2.1 RSA Encryption 38
6.2.2 RSA Decryption 39

6.2.3 MRSA Encryption 40
6.2.4 MRSA Decryption 41
6.3 Analyzing Space Complexity 42
6.3.1 RSA Encryption 42
6.3.2 RSA decryption 44
6.3.3 MRSA Encryption 45
6.3.4 MRSA Decryption 46
6.5 CPU Utilization 47
7. **CHAPTER SEVEN** 48
CONCLUSION 48
REFERENCES 49
APPENDIX 50

List of Figures

Figure 3.1: Cryptographic Process 7
Figure 3.2: Private Key Cryptography 8
Figure 3.3: Public Key Cryptography 9
Figure 3.4: Block Diagram of Key Generation 10
Figure 4.1: Text Steganography Techniques 12
Figure 4.2: Image Steganography 13
Figure 4.3: Audio Steganography 14
Figure 4.4: Vedio Steganography 14
Figure 4.5: Steganography Types 15
Figure 5.1: Overall Process of Encryption and Decryption 18
Figure 6.1: Ciphertext Interface 34
Figure 6.2: Plaintext Interface 34
Figure 6.3: Encrypted Image(PNG Type) 34
Figure 6.4: Final Sending File 34
Figure 6.5: Dialog Box for Openning Image File 35
Figure 6.6: Final Decrypted Output 35
Figure 6.7: User Interface for Input 36
Figure 6.8: Encrypted Image for MRSA 36
Figure 6.9: Resulted Encrypted Sending Document 37
Figure 6.10: Choosing Image for Decrytion 37
Figure 6.11: Output after Decryption 38
Figure 6.12: Execution Time Comparison for two Schemes 47

List of Tables

Table 2.1: Rabevohitra's Algorithm 4
Table 5.1: Newly Proposed Algorithm 19
Table 5.2: RSA Encryption 19

CHAPTER ONE
INTRODUCTION

1.1 Background

Steganography involves hiding information within different carriers, ensuring that private data remains inaccessible to unauthorized users [1].

The core concept behind any digital steganography echnique is to secretly embed private or confidential data within a cover medium. The secret data can take various forms, such as images, text, binary files, or videos. Similarly, the cover medium can also include videos, images, or text. The data embedded within the cover media are referred to as "hidden data," and the result of this embedding process is known as "stego media." Text steganography, however, is considered particularly challenging due to the limited redundancy in textual files compared to other digital media types, such as audio, images, or video files [2].

Cryptography, on the other hand, is a technique used to disguise a message—whether it be in text, images, or other formats—so that only the sender and receiver can interpret the actual content. This process ensures that no third party or adversary can access the information. The act of converting plain data into an unreadable or encrypted form is called encryption, while the reverse process, turning encrypted data back into its original form, is called decryption [3].

Steganography and cryptography are often interrelated and share the common goal of ensuring the confidentiality of sensitive data, which is a key aspect of computer security [1].

RSA, introduced in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman, is a prominent public key cryptosystem essential for secure data transmission. It is widely applied in various cryptographic contexts, including banking, email security, and e-commerce. The algorithm converts information into an unreadable format through encryption, ensuring that only authorized parties can access the original data. This paper utilizes the RSA cryptosystem, which continues to be one of the most prevalent algorithms in public key cryptography, enabling message encryption without the necessity of exchanging a secret key beforehand.

The combination of both steganography and cryptography methods enhances the overall security of the information.

1.2 Research Objectives

This research focuses on ensuring the secure transmission of data. The primary objectives of this study are as follows:

- Develop a method to protect confidential information during transmission over data communication networks using encryption and steganography.
- Encrypt the plaintext message using the RSA algorithm to ensure the confidentiality and integrity of the transmitted data.
- Employ steganography to hide the encrypted data within an image, ensuring that the presence of hidden information remains undetectable to unauthorized parties.
- Further protect the concealed data by embedding the image within an unrelated Microsoft Word document, adding a layer of complexity to prevent detection.
- Combine RSA encryption with text-image steganography to create a robust and covert solution for secure data transmission.
- Ensure that the recipient can extract and decrypt the hidden message using the appropriate tools and methods to recover the original plaintext.
- Guarantee that, even if intercepted, the encrypted and hidden message remains secure and undetectable without prior knowledge of the encryption and steganography techniques.

1.3 Thesis Organization

Chapter 1 introduces the background and objectives of the system.

Chapter 2 focuses on the literature review, examining past research that addresses the discussed system and offering insight into existing solutions.

Chapter 3 covers cryptography, focusing on the encryption and decryption algorithm, while Chapter 4 discusses the importance of steganography and its techniques.

Chapter 5 presents the proposed system, integrating steganography with RSA and modified RSA, with detailed implementation steps.

Chapter 6 shows the graphical user interface (GUI) outputs and compares the two processes.

Finally, Chapter 7 summarizes the key findings and conclusions of the research.

CHAPTER TWO
RELATED WORK

2.1 Literature Review

Text has long been utilized as a medium for steganography [4], [5], [6], [7], and its prominence has surged in recent years, garnering increased scholarly attention due to its pervasive application in the digital domain[8], [9], [10], [11].

The research describes a steganographic approach that conceals information using invisible characters and tailored RGB color intensities, resulting in safe, undetectable data embedding inside text while increasing capacity and resilience to discovery [4].

The research describes a revolutionary text steganography approach for Microsoft Word that uses invisible characters like spaces and carriage returns with hidden font colors to discreetly encode data. This technology overcomes the inherent limits of redundancy in text-based steganography, transforming language into a safe and efficient medium for concealed data transfer [5].

The paper offers a steganographic method for embedding data in Microsoft Word documents using change tracking. By inserting deliberate grammatical errors that are later "corrected," the secret message is concealed inside the changes, giving the document the illusion of being part of a collaborative editing process. The change tracking technique ensures that both the original content and the hidden message are recoverable, rendering the approach untraceable and resistant to manipulation [6].

The paper introduces a linguistic steganography technique that leverages word indexing compression (WIC) to efficiently compress and embed secret messages within carefully selected candidate texts. By utilizing synonym substitutions and optimizing text selection, the method achieves high imperceptibility and enhanced resistance to steganalysis. Experimental results demonstrate superior embedding efficiency and performance compared to traditional compression algorithms [7].

The article introduces a steganographic technique that embeds secret messages within Ci-poetry using a Markov chain model. By leveraging a state transfer matrix based on a tone pattern, the method enhances the naturalness of the generated poetry while maintaining a competitive embedding rate [8].

The research offers a high-embedding-rate text steganography system that uses LSTM-based RNNs to produce excellent Chinese quatrains while overcoming duplication through template limitations and mutual information-based [9].

The paper presents RITS, a real-time text steganography model utilizing RNNs and reinforcement learning to embed secret messages within natural, coherent dialogues. Trained on 5808 dialogues, it ensures efficient, high-quality information [10].

The study introduces *RNN-Stega*, an advanced linguistic steganography technique leveraging recurrent neural networks to autonomously generate sophisticated text covers for concealing confidential information. By integrating both fixed-length and variable-length encoding strategies grounded in conditional probability distributions, the method significantly enhances the efficacy of information embedding. Empirical evaluations reveal that *RNN-Stega* surpasses prior methodologies in terms of stealth, embedding efficiency, and data-hiding capacity, establishing a new paradigm in the domain of text-based steganography [11].

Goshew used the crypto layer that using RSA crypto algorithm and is a public cryptographic system that depends on two keys: one for encryption and the other for decryption. The reason for choosing this cryptography algorithm among other crypto schemes is that it is still considered safe and secure using required mathematics to prove its effectiveness[12].

2.1.1 Feno Heriniaina Rabevohitra's approach

In examining recent literature on text-based steganography methods, we evaluated Feno Heriniaina Rabevohitra's and Yantao Li's approach, which employs the font color of invisible characters for information concealment. Table 2.1 shows the idea to use the binary representation of the secret message and hide it in the form of colors to embed in a Word file so that it is invisible to the naked eye when carefully added [4].

Table 2.1: Rabevohitra's Algorithm

Steps	**Descripton**
Step 1:	Selecting the secret message: "hello world"
Step 2:	Convert the secret message into its bitstream value: 01101000011001010110110001101100011011110010000001110111 10111101110010011011000110010 0
Step 3:	Ensure the message bitstream length modulo three is equal to zero. If this is not the case, prepend zero(s) to the bitstream till its length modulo three equals zero. In our case, the update is needed and the updated bitstream is: 000110100001100101011011000110110001101111001000000111011 1011011110111001001101100011001 00
Step 4:	As the updated bit stream modulo three equals zero, we can create a sequence of 3-bit, which we will call triplets. The triplets created from the update bitstream is: 000 110 100 001 100 101 011 011 000 110 110

	001 101 111 001 000 000 111 011 101 101 111 011 100 100 110 110 001 100 100 A three-bit triplet can be represented by a decimal value in other words, so we can convert the updated bit stream into the decimal values of the triplets. Our triplets' decimal values: 0 6 4 1 4 5 3 3 6 ...
Step 5:	Prepend one zero to each of the decimal value generated in step 4 that originally had one single digit: 00 06 04 01 04 05 03 03 00 06
Step 6:	From the decimal values created in step 5, build triplets: 000604 010405 030300 06...
Step 7:	Use the triplets to generate the corresponding hexadecimal color for painting the invisible characters used for hiding the secret message.

In this algorithm, the hexadecimal color range is from 000000 to 070707, where each RGB value can have a value from 00 to 07, making a possible color palette of 83 that are all dark and are similar to black to the naked eye. The decoding can be achieved by completing the process in reverse order.

After embedding the secret message, with an optimal condition, the content of the generated stego document (word file + Lorem Ipsum passage + modified secret message) would look like this:

"Lorem *ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.*"

In Rabevohitra's algorithm,

- Invisible characters are used to conceal information with the expectation that they will not be detected. However, tools or actions such as text selection can expose them.
- Steganographic security depends on how effectively data is hidden. Without strong encryption or obfuscation, visible patterns can reveal the hidden message.
- The paper focuses on optimizing color intensities for data embedding, a process that can be computationally intensive and may negatively impact performance.

2.2 Motivation and Contribution

As the demand for secure communication escalates, steganography is increasingly used to conceal sensitive information within ubiquitous text documents.Rabevohitra’s approach employs invisible characters and optimized font colors for data concealment, yet it remains vulnerable to detection through textual analysis, particularly when combined with traditional steganographic techniques. Building on Rabevohitra’s work with font color optimization, we propose a refined steganographic approach that amalgamates encryption, image encoding, and document obfuscation, thereby fortifying resilience against detection and mitigating the inherent limitations of traditional methodologies.The system seamlessly integrates text-based steganography, encryption, and image embedding to facilitate secure communication and data transmission, particularly within Word documents. The key contributions of this work are outlined as follows:

Building on Feno Heriniaina Rabevohitra's foundational work in text-based steganography using invisible characters and font colors, our method refines this technique by embedding encrypted data within RGB color intensities. This enhancement not only increases the capacity for data concealment but also significantly bolsters the resilience of the hidden message against detection.We employ RSA encryption on ASCII values before embedding them, thereby fortifying data security. This ensures that decryption is required to access the embedded information, offering a more robust defense compared to traditional steganographic approaches.The encrypted data is then embedded using RGB encoding, encapsulated as an image, and seamlessly inserted into a Word document. This multi-layered process enhances the concealment of the hidden message while strengthening its overall security.

In summary, this work combines RSA encryption, RGB-based text steganography, and image embedding within Word documents, providing a sophisticated and highly secure solution for safeguarding sensitive data.

CHAPTER THREE
CRYPTOGRAPHY

3.1 What is cryptography?

Cryptography is the study of advanced mathematical techniques designed to protect key aspects of information security, such as confidentiality, data integrity, and authentication. While essential, it is just one of many approaches within the broader field of information security. Various rigorous criteria assess the effectiveness of cryptographic methods. A cryptosystem refers to a set of integrated cryptographic primitives that work together to provide security services, with encryption, primarily for confidentiality, being the most common application.

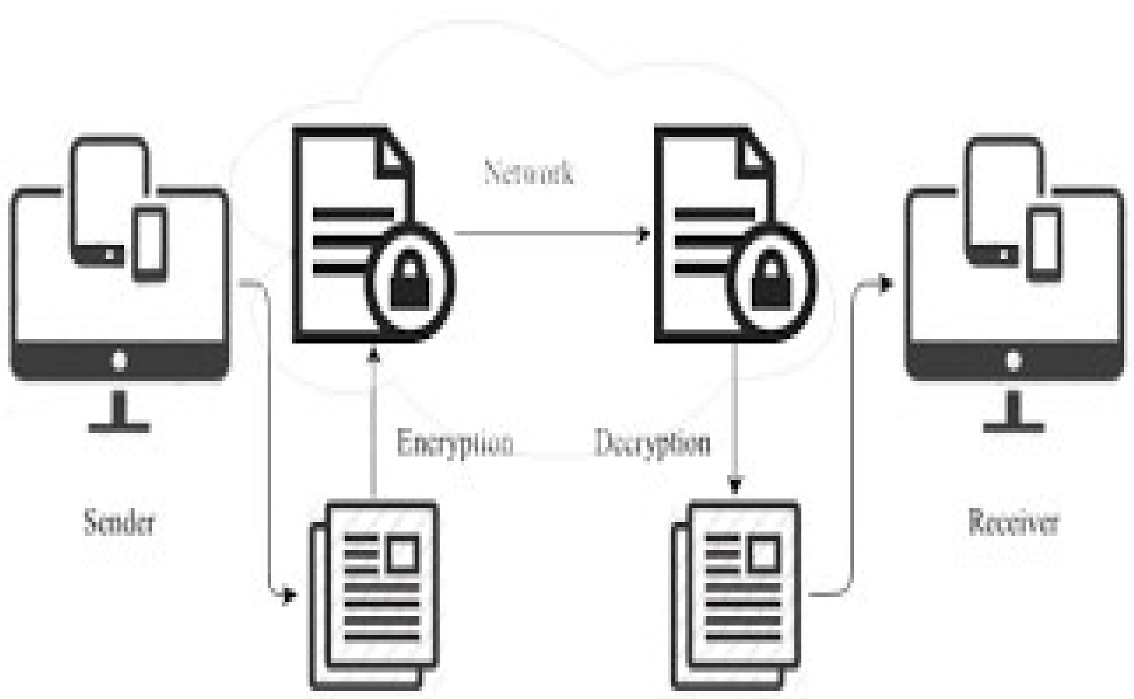


Figure 3.1: Cryptographic Process

In computer science, cryptography refers to the application of mathematical principles and algorithmic processes to ensure the security of information and communication. It involves a set of systematic, rule-based procedures designed to transform messages into formats that are difficult, if not impossible, to decipher without the proper decryption keys. These deterministic methods are employed for a range of purposes, including the generation of cryptographic keys, digital signatures, and verification mechanisms, all aimed at safeguarding data privacy. Cryptography plays a vital role in securing online activities, such as internet browsing, as well as protecting sensitive communications, including credit card transactions and email exchanges.

3.2 Techniques of cryptography

Cryptography is nearly related to the disciplines of cryptology and cryptanalysis. It includes methods such as microdots, merging words with images, and other ways to hide information in storage or transit. However, in today's science-centric world, cryptography is most often attached with scrambling plaintext (ordinary text, sometimes referred to as cleartext) into ciphertext (a process called encryption), then back again (known as decryption). People who practice this field are known as cryptographers.

3.2.1 Symmetric Key Cryptography

In symmetric key cryptography, also known as private-key cryptography, a secret key is either kept by one party or shared by the message's originator and recipient. When utilizing this cryptographic method for secure communication, both the transmitter and the receiver must have an identical copy of the secret key to encrypt and decode the data delivered. The most famous symmetric key cryptography system is Data Encryption System (DES).

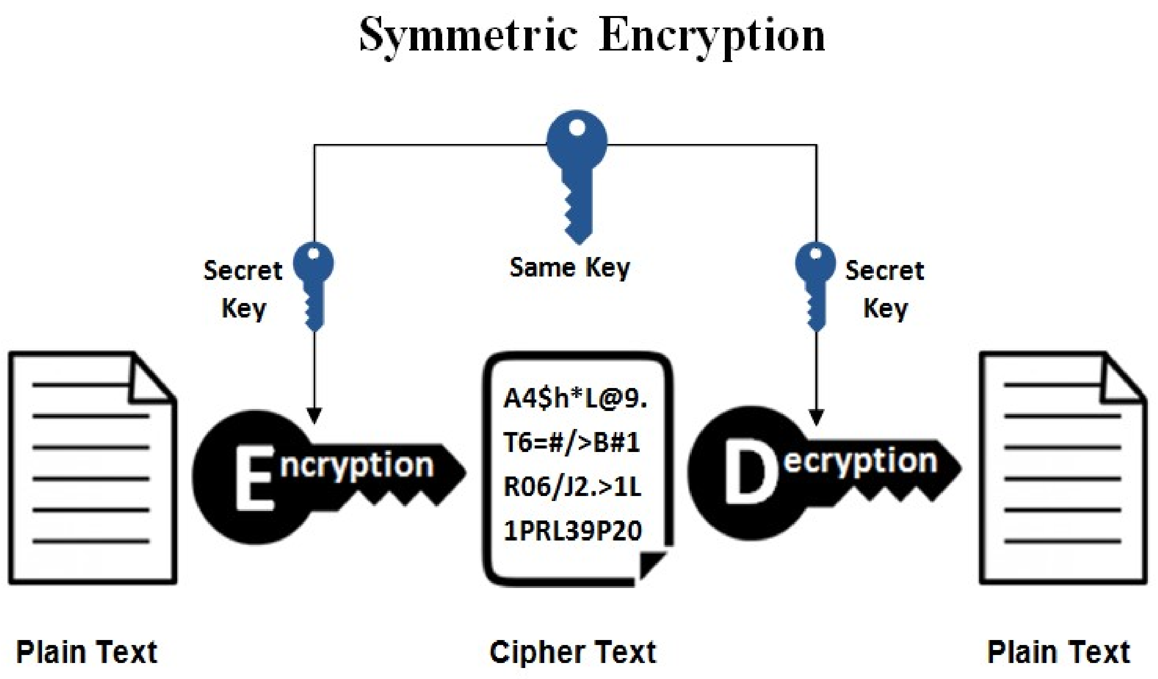


Figure 3.2: Private Key Cryptography

3.2.2 Assymetric Key Cryptography

In the two-key, or public-key cryptosystem, one key is used for encryption, and a mathematically linked key is used for decryption. The sender encrypts the message first with the recipient's public key and then again with their own private key, which remains confidential. Upon receipt, the recipient decrypts the message using their private key, followed by the sender's public key.

This dual-layer encryption not only preserves the confidentiality of the communication but also facilitates mutual authentication, ensuring both the identity and secrecy of the parties involved.

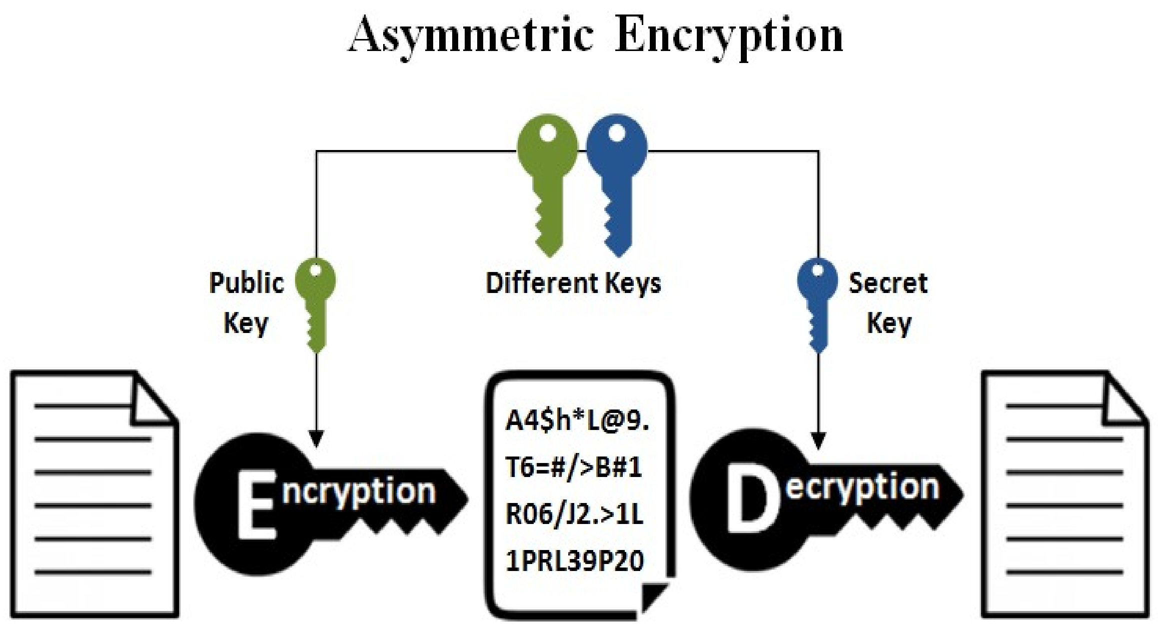


Figure 3.3: Public Key Cryptography

3.3 RSA Algorithm

RSA has emerged as a cornerstone in the realm of electronic communications, making a significant milestone as the first public-key cryptosystem and, notably, the only one that has withstood over three decades of cryptographic attacks. This enduring resilience has solidified RSA as the algorithm of choice for a wide range of applications, including authenticating phone calls, securing online credit card transactions, safeguarding email communications, and facilitating various other critical internet security functions. For their groundbreaking work, Ron Rivest, Adi Shamir, and Leonard Adleman were honored with the prestigious Alan Turing Award by the Association for Computing Machinery in 2002. Given its widespread use and the continued importance of its security, RSA remains a central subject of cryptographic research, both in theoretical studies and practical implementations.

RSA uses modular exponentiation, where the system selects integers e, d, and N such that for any A smaller than N, the equation (A^e mod N)d mod N=A holds. This allows encryption with e and decryption with d, or the reverse, a process called signing and verification. The public key consists of (e, N), which anyone can share, while the private key, (d, N), must remain confidential. In RSA, e is the public exponent, d is the private exponent, and N is the modulus. RSA is asymmetric: anyone can encrypt with the public key, but only the private key holder can decrypt. The private key owner can also encrypt messages, ensuring authenticity with digital signatures[17].RSA faces several threats: the forward search attack, where predictable message spaces allow attackers to brute-force decryption; the common modulus attack, which exploits a shared modulus with different (e, d) pairs; low encryption exponents, allowing decryption when small exponents and messages are used; and the multiplicative property, which enables chosen-ciphertext attacks, as attackers can leverage the relationship between the product of ciphertexts and the encryption of plaintext products.

The block diagram for key generation is presented in Figure 3.4, while the block diagram for the RSA key generation algorithm is shown in Figure 3.5.

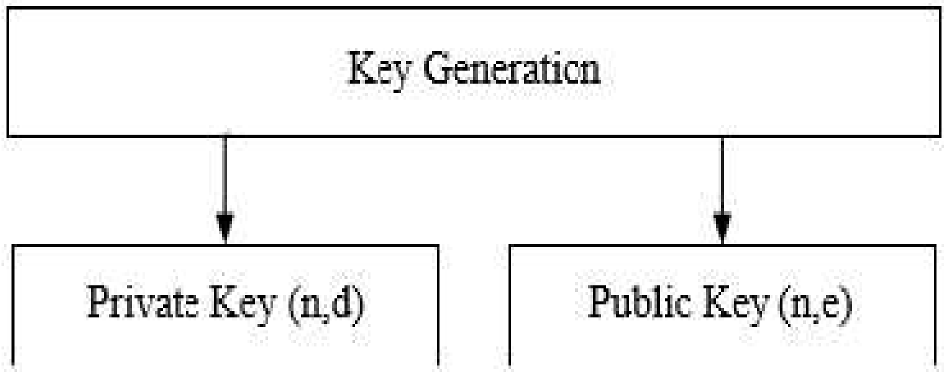


Figure 3.4: Block Diagram of Key Generation

3.3.1 RSA Encryption and Decryption

The RSA algorithm can be used for both key exchange and digital signatures. Although employed with numbers using hundreds of digits, the mathematics behind RSA is relatively straightforward. To create an RSA public and private key pair, the following steps can be used[10].

I. Choose two prime numbers, p and q. From these numbers you can calculate the modulus, $n=pq$.
II. Select a third number, f, that is relatively prime to (i.e., it does not divide evenly into) the product (p-1)(q-1); the number f is the public exponent.
III. Calculate d, the private exponent, from the equation: f^-1 mod (p-1)(q-1).
IV. The public key is the number pair (n,e). Although these values are publicly known, it is computationally infeasible to determine d from n and f if p and q are large enough.
V. To encrypt a message, M, with the public key, create the ciphertext, C, using the equation: $C = M^f \bmod n$.
VI. The receiver then decrypts the ciphertext with the private key using the equation $M = C^d$ mod n.

3.3.2 Why RSA is Needed?

RSA executes its security from the difficulty of factoring large integers that are the product of two large prime numbers. Multiplying those two numbers is normally easy, but determining the exact prime numbers from the total -- or factoring -- is considered infeasible due to the time it would take using even today's supercomputers.

3.4 Objectives of cryptography

Modern cryptography shows itself with the following four objectives:

- **Confidentiality**: The information or message cannot be understood by anyone for whom it was unintended.
- **Integrity:** The information or message cannot be altered in storage or transit between sender and intended receiver without the alteration being detected.
- **Non-repudiation**: The sender of the confidential information cannot deny at a later stage his or her intentions in the creation or transmission of the information or message.
- **Authentication**:The sender and receiver can confirm each other's identity and the source/destination of the confidential information.

CHAPTER FOUR
STEGANOGRAPHY

4.1 What is steganography?

Steganography conceals sensitive secret information in a regular, non-secret text or document to prevent discovery. The recipient extracts the confidential data at its destination. We can use steganography in conjunction with encryption as an additional measure to conceal or safeguard data. The word "steganography" derives from the Greek words **steganos** (meaning hidden or covered) and the Greek root **graph** (meaning to write).

Steganography can conceal a wide range of digital content, such as text, images, videos, or audio. One can hide data within nearly any type of digital file. Prior to arranging the information into a cover text file or data stream that appears innocuous, the user encrypts the secret text. Steganography serves to preserve the confidentiality of a message. Although many legitimate users apply steganography for secure communication, cybercriminals also exploit it to mask the transmission of harmful software.

People have used forms of steganography for centuries, employing various techniques to hide a secret message in an otherwise harmless container. For example, people have used invisible ink to conceal secret messages in otherwise undetectable text, hidden documents, or files recorded on microdots, some as small as 1 millimeter in diameter, embedded in correspondence that appears legitimate, and even shared messages within multiplayer gaming environments.

4.2 Techniques of steganography

4.2.1 Text Steganography

Text steganography embeds hidden information within a text file by modifying formatting or text features, such as font size or spacing.It aims to ensure reliable extraction of data while keeping

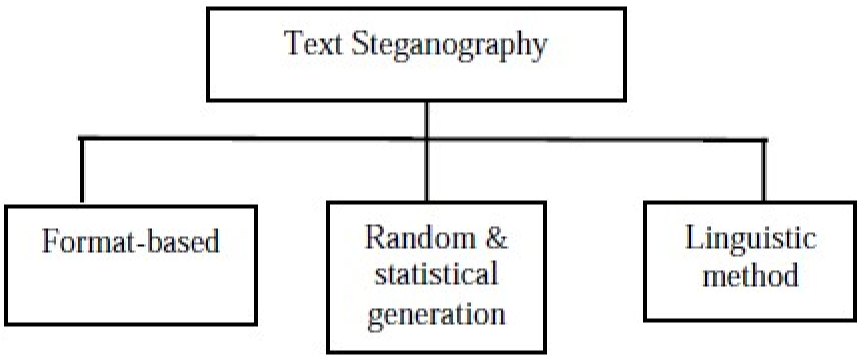


Figure 4.1: Text Steganography Techniques

visible changes minimal. Formats like PostScript2, TeX, or @off describe the document's structure and appearance. This technique inserts data discreetly, making it invisible to the reader yet retrievable, even with distortions. There are Three types of text steganography.

Format-Based Approaches: These techniques conceal data by manipulating the visual presentation of text. They alter existing content by incorporating extra spaces, deliberately introducing spelling errors, or adjusting font sizes. Although these strategies can mislead human observers, Bennett highlighted that automated systems can easily identify them.

Random and Statistical Fabrication: This approach generates cover text based on statistical characteristics. It encodes information within arbitrary sequences of characters that appear random to an external observer. Additionally, it capitalizes on the statistical patterns of word lengths and letter frequencies to produce "words" that mimic natural language without carrying semantic meaning. Word sequences can embed data, using terms or predefined mappings between vocabulary and bit patterns to conceal the secret message.

Linguistic Techniques: These methods exploit the inherent structure of language to embed concealed information. They modify syntactic elements of the text, encoding hidden data within the grammatical framework. By adjusting linguistic constructs, these strategies effectively camouflage steganographic content while preserving the document's fluent readability.

4.2.2 Image Steganography

Image steganography remains a favored method due to the limitations of the Human Visual System (HVS). The eye fails to perceive slight modifications in color or minute degradations in image quality resulting from steganographic techniques. Consequently, individuals increasingly adopt image-based approaches to conceal data.

Figure 4.2: Image Steganography

4.2.3 Audio steganography

Audio steganography presents a challenge due to the human ability to perceive even subtle variations in sound quality. In this technique, one embeds confidential messages within a digitized audio signal, causing minor modifications to the binary sequence of the corresponding audio file. Several advanced methods, such as LSB coding, phase coding, spread spectrum, and echo hiding, facilitate this process.

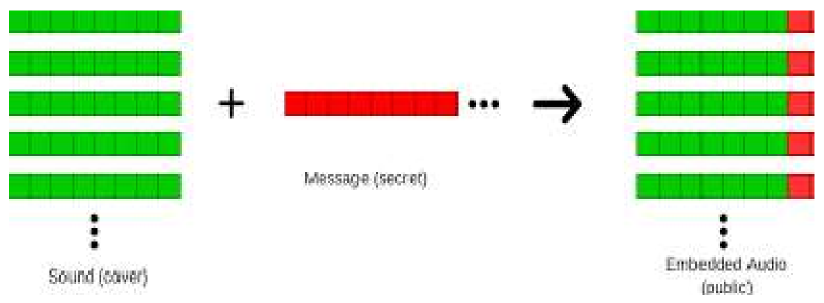


Figure 4.3: Audio Steganography

4.2.4 Video Steganography

Video Steganography is a technique to conceal any kind of files into a cover video file. The use of the video-based steganography can be more secure than other multimedia files, because of its size and complexity.

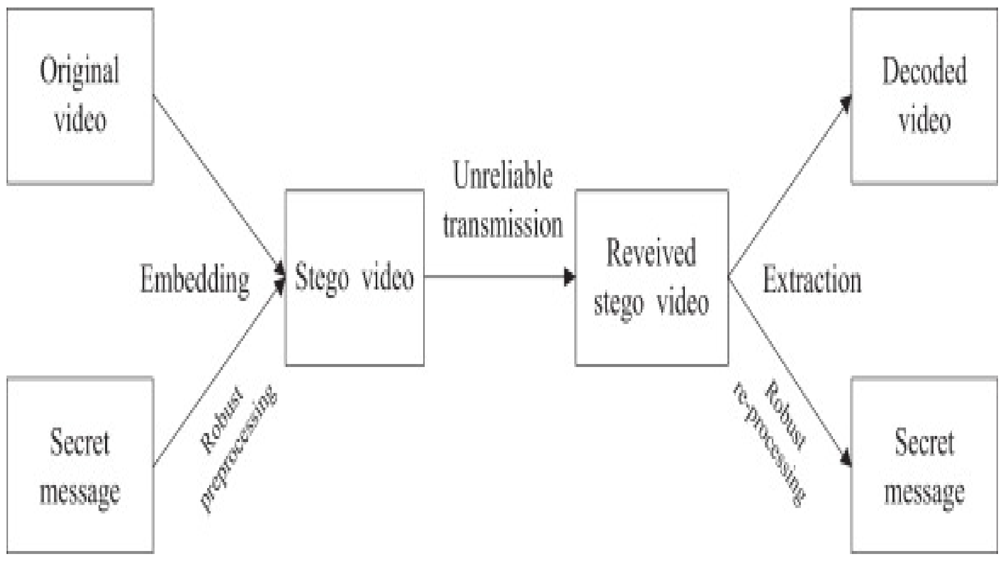


Figure 4.4: Vedio Steganography

In the context of key exchange, three primary types involve the concealment of secret keys during the exchange process. These types focus on how cryptographic keys conceal and exchange. The three types are:

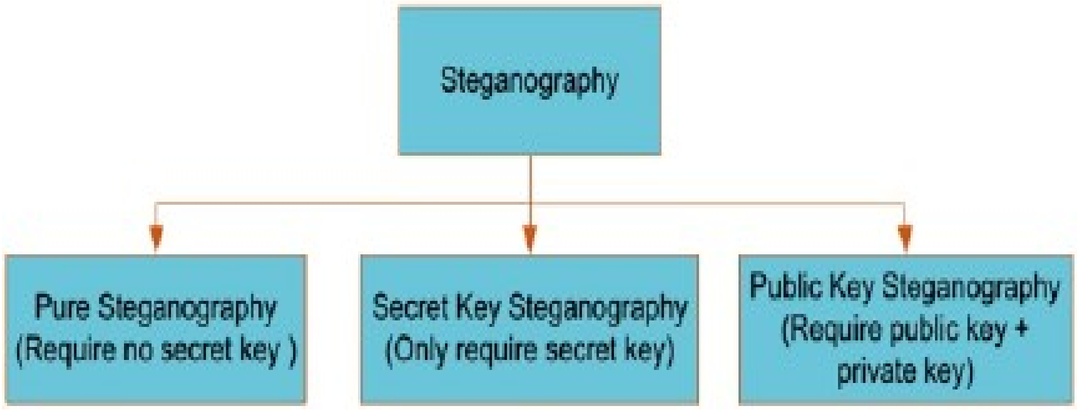

Steganography
Pure Steganography (Require no secret key)
Secret Key Steganography (Only require secret key)
Public Key Steganography (Require public key + private key)

Figure 4.5: Steganography Types

Pure steganography enables message transmission without exchanging a secret key or prior information. Its security depends entirely on the ability to maintain confidentiality and secrecy.

Secret key steganography is a technique that encrypts confidential data using a secret key or password, guaranteeing safe communication by limiting access to only those who have permission.

In the third steganographic technique, the system embeds concealed information within a carrier using a public key and enables extraction with a private key, ensuring secure, restricted access to the data.

4.3 Importance of steganography

As the volume and variety of data to store and transmit across different mediums continue to grow, defining security measures at various levels of medium access and addressing authentication and authorization challenges becomes increasingly difficult. Many steganographic and data-embedding techniques manipulate the original data to hide sensitive information or enforce access control over the medium. These mediums, such as images, video, and audio, often alter certain parts or the entire space with meaningful data. This work emphasizes the role of steganographic techniques in information processing algorithms to ensure data security. It focuses on data security concerns, particularly in text and image formats, and identifies the essential characteristics that steganographic algorithms must have.

4.4 Cryptography Vs Steganography

Cryptography uses mathematical algorithms to convert data into an unreadable format, ensuring that only those with the correct decryption key can access the original information. Its primary purpose is to protect the content of the message by making it incomprehensible to unauthorized users, thus ensuring privacy and security.

Steganography, on the other hand, hides the message itself within other non-secret data, such as images, videos, or text. It doesn’t alter the content but conceals the existence of the message, making it invisible to anyone unaware of the hidden information. Unlike cryptography, which secures data through encryption, steganography focuses on ensuring the message goes unnoticed by blending it seamlessly into another medium.

CHAPTER FIVE
RESEARCH METHODOLOGY

5.1 Materials

In this study, an advanced, multi-faceted approach is meticulously devised to ensure the secure transmission of sensitive information across data communication networks. The materials integral to this methodology consist of several indispensable components, each contributing to the overall security and effectiveness of the system:

5.1.1 Java Development Environment

- **Operating System:** The system runs on Windows 10 with an HP Core i5 processor, ensuring optimal compatibility with Java and efficient execution of cryptographic and image tasks.
- **IDE/Code Editor:** Development is conducted in Visual Studio Code (VSCode), a streamlined editor with features like syntax highlighting, IntelliSense, and debugging tools.
- **Java JDK:** The JDK, including the Java Runtime Environment (JRE), is used to compile and run Java applications, supporting GUIs, cryptography, image processing, and I/O operations.

5.1.2 Frameworks and Libraries

- The GUI was constructed using Swing, which provides elements like windows, buttons, and text fields for user input and the display of encrypted results.
- Alongside Swing, Java AWT manages layout and handles events, providing elements like text fields and buttons.
- Handle file I/O operations for reading and writing image files using Java IO, and ImageIO supports PNG and other formats.
- Enable image manipulation by storing and altering pixel data and embedding the encrypted message as RGB values using the BufferedImage and Color classes.

5.1.3 ASCII Values

The plaintext's characters are transformed to their corresponding ASCII values, which are then encrypted using the RSA technique. The encrypted values are then stored in the picture as RGB pixels.

5.1.4 Embedding medium: MS Word

Microsoft Word hides encrypted messages by embedding a steganographically concealed image within the document, which appears as ordinary content, ensuring the hidden information remains undetected.

5.2 Our Proposed System

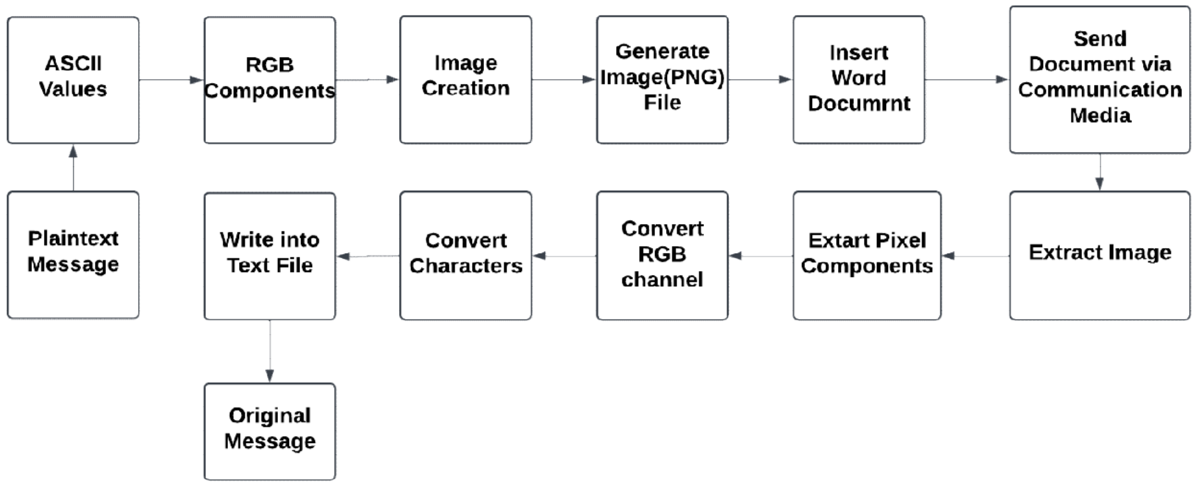


Figure 5.1: Overall Process of Encryption and Decryption

The proposed encryption algorithm begins by converting the plaintext into its corresponding ASCII values, where each character transforms into a unique numerical representation. To prepare the data for further processing, we adjust the length of the plaintext to ensure it is divisible by 3, as the next step necessitates grouping the data into triplets. If the length fails to meet this requirement, we pad the plaintext with zeros or a suitable placeholder until it satisfies the condition. Once we modify the length, we apply the RSA encryption algorithm to each value in the plaintext, utilizing a public-private key pair to encrypt the data. The RSA algorithm ensures that the original message remains secure, transforming the plaintext into ciphertext. We then partition the resulting ciphertext into triplets, with each triplet containing three encrypted values. Next, we map these triplets to RGB color values, assigning each numerical value from the triplet to the red, green, and blue channels of a pixel. This mapping converts the encrypted data into a visual form, with the RGB values determining the color of each pixel in the resulting image. By embedding the encrypted message within the image, we obscure the original plaintext, making it invisible to anyone without the correct decryption key. Only those with the appropriate private key can decrypt the image and retrieve the original message. In Table 5.1, we provide a comprehensive overview of the entire process, from plaintext to encrypted image, while Table 5.2 delves into the application of the RSA encryption technique. This approach effectively secures data, embedding it within an image and offering both confidentiality and visual camouflage, ensuring that the hidden information remains protected from unauthorized access.

Table 5.1: Newly Proposed Algorithm

Steps	Description
Step 1	Using the "plaintext" Jtextfield, enter the secret message "hello world."
Step 2	First, translate the secret message into ASCII decimal values: 104 101 108 108 111 32 119 111 114 108 100
Step 3	Verify that the length of the transformed decimal message modulo three equals zero. If not, add zero or zeros to the final decimal numbers until their length modulo three equals zero: 104 101 108 108 111 32 119 111 114 108 100 0
Step 4	Use RSA encryption. I'll elucidate RSA's subsequent work. The new decimal values are as follows: 15 118 146 146 144 87 102 144 31 146 155 0
Step 5	Create triplets for RGB color: 15 118 146 146 144 87 102 144 31 146 155 0

Table 5.2: RSA Encryption

Steps	Description
Step 1	Take prime number p=17,q=11
Step 2	calculate n=p*q=187
Step 3	calculate phi=(p-1)*(q-1)=160
Step 4	calculate f= 13 and d= f^-1 mod phi=37
Step 5	Take Plaintext: M_1=104 M_2=101 M_3=108 M_4=108 M_5=111

	M_6=32 M_7=119 M_8=111 M_9=114 M_{10}=108 M_{11}=100 M_{12}=0
Step 6	Using the equation for encryption C=M^f mod n where M= messages, C= cyphertext. For M_1, C_1=104^13 mod 187= 15 ; for M_2, C_2= 101^13 mod 187= 118 and so on…So, the new values after encryption are given below: 15 118 146 146 144 87 102 144 31 146 155 0

An image is generated using RGB encoding to obfuscate the encrypted 'hello world.' To retrieve the message, the RGB values are extracted, converted to integers, and subjected to RSA decryption.The reverse process is employed to decrypt the ciphertext and restore the original plaintext.

5.3 Implamantation of Using RSA

The program uses Object-Oriented Programming (OOP) principles in Java. Sun Microsystems created Java in 1995, and Oracle now owns it. Java is a high-level programming language known for its "write once, run anywhere" capability. With a rich set of libraries and frameworks, Java is highly versatile, making it suitable for a variety of applications, from web servers to complex graphical user interfaces (GUIs). It is particularly known for its stability, security features, and strong support for multithreading.

The program includes two main classes: **javaConstructor.java** and **ColorPicky.java**. It follows a top-down design, starting with a basic structure and gradually adding more features.

This Java program builds a graphical user interface (GUI) that combines RSA encryption with a form of steganography. Here's a detailed look at how it works.

5.3.1 Essential Java Imports for Our Research

We use Java libraries or packages, which offer pre-written classes and methods, to enhance our applications. By including import statements at the beginning of our files, we can access these external libraries. Below is a breakdown of each import statement:

- We use javax.swing.* to import all classes from the Java Swing package, a versatile library for building graphical user interfaces. Swing offers a broad range of interface components, such as buttons, labels, text fields, and panels, which enable us to create highly interactive and visually engaging applications. By incorporating this package, we can design and customize the user interface with ease, improving the overall user experience.
- The statement java.awt.* imports all classes from the Abstract Window Toolkit (AWT) package, which provides essential tools for building graphical user interfaces in Java. AWT offers various core components, such as buttons, frames, text areas, and labels,

enabling us to create dynamic and interactive user interfaces. It also provides functionality for handling user input, including mouse events and keyboard interactions. Including this package in our project allows us to effectively manage both visual elements and user interactions.

- The java.awt.event.* import statement grants us access to all classes in this package, which provides essential tools for managing events in Java. This package encompasses classes that represent various types of events, such as mouse clicks, keyboard inputs, window actions, and more. It also includes listeners like ActionListener, MouseListener, and KeyListener, which enable us to capture user interactions with graphical user interface (GUI) elements and define the appropriate responses. By incorporating this package, we can efficiently handle and respond to user actions within our project.
- The java.awt.image.* import statement gives us access to all classes in this package, which offers a comprehensive set of tools for managing images in Java. This package includes classes that allow us to load, edit, and render images in a variety of formats. It supports tasks such as image manipulation, creating image buffers, and processing pixel data. For example, classes like BufferedImage enable us to work with images directly in memory. By importing this package, we can seamlessly integrate image processing capabilities into our project.
- The javax.imageio.* import statement gives us access to classes that allow us to read, write, and manipulate image files in formats like JPEG, PNG, GIF, and BMP. It provides various tools for loading, saving, and converting images between different formats. Key classes, such as ImageIO, enable basic image input and output operations, while ImageReader and ImageWriter offer advanced features for processing images. By using these classes, we can efficiently handle image-related tasks in our project.
- The java.io.* import statement grants us access to a wide range of classes that facilitate input and output (I/O) operations in Java. This package provides the necessary tools to read from and write to files, streams, and other data channels. It includes classes like File, which lets us manipulate files and directories, and BufferedReader and BufferedWriter, which improve the performance of reading and writing text data. By utilizing this package, we can efficiently manage file handling and data transfer tasks within our project.

5.3.2 Code for Encryption

the encryption process is done in javaconstructor.java class. The working procedure of this class is given below step by step:

Class Declaration: The `JavaConstructor` class manages encryption and steganography while providing a graphical user interface (GUI) for user interaction. The interface allows users to input plaintext, initiate encryption, and view the resulting steganographic image. The class also evaluates the encryption time and storage requirements of the program, offering details on the size of the encrypted data and the generated image.

Jframe Set Up: We create a frame object by using the JFrame class with the title "Merging Steganography with RSA." Inside, we have two JTextField objects for input and output, as well as a "Encrypt" JButton that initiates encryption. Finally, we scaled the frame to 400x400 pixels, positioned it in the upper-left corner, set the exit close option, and utilized a blue backdrop.

JPanel for Input Area: A Jpanel (open) arranges the input elements, such as the JTextField for plaintext and the JButton for encryption, into a cohesive layout. It sets the background to a custom light blue shade and contains the components before adding them to the main frame.

New Jframe for Cipher Text: Upon the user's click of the "Encrypt" button, the application creates a new JFrame to showcase the encoded message. The window features colored buttons that symbolize RGB values, visually aligning with the ciphered text. The application adjusts the frame's visibility, dimensions, and placement to correspond with those of the primary window.

New JPanel for Encrypted Colors: After encryption, the system converts the data into RGB values that visually represent the encrypted text. It organizes these values as interactive buttons in a newly created JPanel, with each button's background reflecting one of the RGB components. The program constructs a new JPanel (newJPanel) to hold these buttons, arranging them in a grid layout. Each button receives a color corresponding to its respective RGB value, and the system sets the text color to white for better visibility. Additionally, the program customizes each button's appearance by removing borders and adjusting the dimensions to 20x20 pixels.

RSA Encryption function:

The power(int x, int y, int p) function performs modular exponentiation, a fundamental operation in RSA encryption. It efficiently computes x^y mod p, where x is the base, y is the exponent, and p is the modulus. This function plays a pivotal role in both the encryption and decryption processes.

- The result 'res' begins at 1, and the reduction of x modulo p keeps it within a manageable range. If x mod p = 0, the function returns 0.
- Using its binary form, a loop iteratively processes the exponent y, gradually decreasing it. When y is odd, the process multiplies x by the result, applying the modulo p operation. By shifting y to the right (bitwise), the process reduces the number of phases with every iteration.
- At each loop, x squares and reduces modulo p, limiting the number of calculations needed for handling large exponents.
- After the loop concludes, the function returns the value of res.

This method significantly improves the efficiency of modular exponentiation, making it feasible to handle large exponents effectively, which is essential for the secure operation.

Constructor Definition

```
public javaConstructor() throws IOException {
   initialize();
}
```

The constructor calls the initialize() method, which sets up the GUI and encryption logic.

Initialize Method

```
public void initialize() throws IOException { ... }
```

This method handles all the GUI components and encryption-related setup. It also manages user input, encrypts the text, and displays the result in both text and image form.

RSA Encryption Logic

```
int p=17, q=11;
int n = p * q;
int phi = (p - 1) * (q - 1);
int f = 13;
double d = 1.0;
d = (1 + 3 * phi) / f;
```

- The process uses two primes for RSA key generation.
- Modulus n is a crucial component of both the public and private keys.
- Phi is smaller than n, and it is also coprime with n; they have no common divisors other than 1.
- Here, f is the public key exponent, which should be coprime with phi(n), and d is the private key component.

Encryption Button Action Listener

```
encrypt.addActionListener(new ActionListener() { ... });
```

The system listens for a click on the "Encrypt" button, and upon activation, it obtains the plaintext that the user enters, encrypts it using RSA, and displays ciphertext both as text and as an image.

Text Processing and Conversion to ASCII

Padding the Text for Steganography

```
int length;
```

```
if (l % 3 == 0) { ... } else { ... }
```

The system modifies the input text length to make it divisible by three, enabling seamless representation in RGB format. If the size modulo 3 equals 1, it appends two characters; if it equals 2, it adds one character.

Conversion and Encryption:

Text to ASCII conversion: The system converts each character of the input text into its corresponding ASCII value. It replaces the last character of the string with a space (ASCII 32) to ensure uniformity and possibly serve as padding or alignment for subsequent processes.

RSA Encryption: The power() method encrypts each character through RSA, utilizing modular exponentiation with the public key.

Steganographic RGB Color Assignment

RGB Conversion: The system groups the ASCII values into triplets, adjusting for the padded length of the text. Each triplet maps to the RGB color model, representing red, green, and blue components.

Image Generation: The system assigns the RGB values to the r[], g[], and bcol[] arrays, creating a color palette that encodes the message within the image.

Creating and Saving the Image:

```
BufferedImage img = new BufferedImage(rlength, rlength, BufferedImage.TYPE_INT_RGB);
```

Image Creation: The system generates an image using BufferedImage with the RGB color model. The image size depends on the number of hues derived from the input text.

```
File myFile = new File(multiply + "MyFile.png");
ImageIO.write(img, "PNG", myFile);
```

File Creation: The process saves the image to disk as a PNG file, with the filename reflecting the computed product of all RGB components.

Main Method

```
public static void main(String[] args) throws IOException {
javaConstructor a = new javaConstructor();
}
```

The main method creates an instance of the javaConstructor class, initializing the GUI and the RSA encryption process.

5.3.3 Embedding the Image in a Word Document

The image with the hidden message is embedded in a Word document alongside harmless content, further obscuring its presence. Dummy or irrelevant text is included to mask the secret, making it more difficult for attackers to detect. The image is saved in a lossless format, such as PNG, ensuring the integrity of the hidden message is preserved throughout the embedding process.

5.3.4 Decryption Process

The sender sends the document to the receiver through any communication medium, such as email. The receiver first extracts the image file containing the secret message from the document. Next, the receiver initiates the decryption process using the ColorPicky class. The ColorPicky.java class follows a structured method to decode the encrypted data, as outlined in the subsequent steps:

Import packages

- We utilize the java.awt.Color class from Java's AWT to define colors in the RGB model. It enables us to create hues by specifying the red, green, blue, and alpha components. We can extract the distinct color components using functions like getRed(), getGreen(), and getBlue().
- We use the BufferedImage class to hold images in memory, as it extends the Image class. It enables direct manipulation of pixels with methods like getRGB() and setRGB(). When loading or generating images, we often depend on BufferedImage to manage and modify pixel data in Java.
- We use the java.io.File class to represent path names for files and directories within the file system. It determines the location of a resource and provides methods like getAbsolutePath() to retrieve its directory path.
- The java.io.FileWriter class facilitates writing character-based data to a file. It uses buffering to increase processing performance for huge datasets and permits output to a file.
- The java.io.IOException class addresses input/output failures, including issues with file and network operations. It governs I/O errors, and methods executing such tasks usually mandate that the calling code either handle or propagate the exception.
- An effective and adaptable method for reading and writing pictures is the javax.imageio.ImageIO class. It enables ImageIO to read pictures and write them in the given format.

Main Method (main()): The decryption program's entry point is the main method.

```
public static void main(String[] args) throws IOException {
```

File Selection and Path Handling

This part of the code triggers a file selection dialog, enabling the user to choose an image file. The JFileChooser component presents a visual interface for browsing documents. Once the user picks a resource, the program obtains its full path for subsequent handling.

File Writer Setup

```
    FileWriter writer = new FileWriter("pixel_values.text");
```

The FileWriter class employs character encoding to transform characters into byte data before writing them to a file. By initializing its object with a path (e.g., "pixel_values.text"), we designate the location for storing the data. The program either generates or opens the file within the current working directory. If the file is absent, the program creates it; if it already exists, the program overwrites it by default, though appending data remains an alternative. Once initialized, the FileWriter becomes ready to write text to the designated file.

Image Reading

```
    File file = new File(path);
    BufferedImage img = ImageIO.read(file);
```

The `File` object specifies the selected image file, and `ImageIO.read(file)` handles the picture and saves it as a `BufferedImage` instance, making its pixel information accessible for further examination.

Loop for Image Processing

This section uses img.getRGB(x, y) to iterate through the image's pixels and retrieve the RGB value for each pixel at coordinates (x, y). After that, the procedure creates a color object for each pixel using its RGB value and separates the red, green, and blue components for additional processing.

Decryption of RGB values

The program decrypts the RGB values of each pixel by applying the private key exponent d and modulus n through the power() method. The power method works the same as the encryption technique. This approach independently decrypts each color channel (red, green, and blue) to recover the original values for subsequent analysis.

Writing Decrypted Values to File

```
        char rstr = (char) r;
        char gstr = (char) g;
        char bstr = (char) b;
        writer.append(rstr);
        writer.append(gstr);
        writer.append(bstr);
        writer.flush();
```

The program transforms the decrypted RGB values into characters and records them in pixel_values.txt using the append() method. It adds the restored pixel values (rstr, gstr, and bstr) derived from RSA decryption. Subsequently, the program invokes the flush() method to write all buffered data to the file.

Closing the File Writer

```
    writer.close();
    System.out.println("RGB values are stored in the file.");
}
```

Upon completion of the processing, the program terminates the FileWriter to commit the data and free up resources. It then displays a message indicating the successful storage of the RGB values in the file.

Finally, when the receiver opens the file using a text editor like WordPad, they will be able to access and view the intended secret message stored within.

5.4 Implementation Using MRSA

MRSA (Modified RSA) uses four prime numbers, which sets it apart from the standard RSA algorithm, which typically uses two prime numbers. To implement MRSA, we employ three classes that manage the key generation, encryption, and decryption processes. These classes make use of the same packages as the previous RSA system, with the addition of the java.awt.geom and java.math packages.

- The java.awt.geom package imports the RoundRectangle2D class, enabling the construction of rounded-cornered shapes. Within the code, RoundRectangle2D.Double modifies the shape of the JFrame window, substituting the conventional rectangular outline with a rounded one. This alteration enhances the window's aesthetic appeal, providing a more contemporary and refined look.
- The java.math package imports the BigInteger class, which handles huge numbers that exceed the range of primitive types (int or long). BigInteger manages the large numbers for key generation, encryption, and decryption, and methods like modPow() and modInverse() perform the necessary mathematical operations in the MRSA algorithm.

5.4.1 Encryption Process

Main Class

```
public class Main extends JFrame {
```

The Main class is the entry point for the program. It extends JFrame, which means this class is a type of window or graphical interface.

```
public static BigInteger encrypt(BigInteger message, BigInteger m, BigInteger n) {

   return message.modPow(m, n);

}
```

This method encrypts a message using RSA by applying modular exponentiation. Its inputs are the modulus, the public exponent, and the message, and it uses 'message.modPow(m, n)' to return the encrypted message.

Constructor of Main Class

```
public Main() throws IOException {

   super("Modified RSA");

   encrypt();

}
```

The Main class's initializer executes upon instantiating an object. It assigns the JFrame's window title by calling super("Modified RSA").The constructor then sets up the window and starts the encryption mechanism by using the encrypt() function, which coordinates the cryptographic processes and the GUI elements.

GUI Setup

- Through the removal of decorations (such as the title bar and border), this code personalizes the window. It positions the window at coordinates (400, 150) on the screen and sets its size to 400x470 pixels. The code sets the window shape to a rounded rectangle with a 40-pixel radius and arranges components vertically using a BoxLayout.
- We create a text field for user input, a label displaying the title "RGB Steganography With Modified RSA," and a "Hide" button to trigger the encryption and steganography process. We also define two panels: one to hold UI components and one for the title and related elements.
- The code adjusts the background color of the "Hide" button to a greenish tint, makes sure the backdrop is completely opaque, and turns off the focus highlight effect that typically

happens when users click the element. It also sets the control's text color to white and removes the default border, giving the button a flat appearance.

- The code establishes the text field's layout. It inserts a 2px dark gray bottom border and positions the text field at coordinates (0, 27) with dimensions of 300x30 pixels. Furthermore, it makes the text field's backdrop translucent.
- The implementation sets the text color of the label to purple, applies a BorderLayout, makes the caption's background transparent, and positions it at coordinates (0, 27) with a size of 300x30 pixels.
- The next section of the code structures the graphical user interface by positioning components within two panels (title and main content). It incorporates appropriate spacing between elements to maintain an uncluttered layout and arranges the components vertically, ensuring a smooth, top-to-bottom progression. The code positions the title panel at the top, while it places the content panel, housing the interactive elements, beneath. This arrangement guarantees a coherent, aesthetically pleasing, and intuitive interface.

User Interaction

```
en.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
```

In Java Swing, an ActionListener actively monitors user interactions with interface components, such as buttons. When the user clicks the "Hide" (en) button, the ActionListener detects the event and triggers the actions defined for it. After that, the program calls the actionPerformed method, which executes the logic that triggers activities like starting the encryption process. It specifies how the application reacts to user input.

Coprime Check and Key Generation

In MRSA We use four prime numbers (p, q, r, s). In order to generate keys, we compute the modulus and Euler's totient function (phi). After that, we choose a public exponent (m).

```
while (phi.gcd(m).compareTo(BigInteger.ONE) > 0) {
   m = m.add(BigInteger.ONE);}
BigInteger d = m.modInverse(phi);
```

- The bit of code first calculates the GCD between phi and the public exponent (m) and then determines whether or not it equals 1. If the GCD is greater than one, the condition evaluates to true, indicating that phi and m are not coprime. Consequently, the program increments m until it becomes coprime with phi. When the GCD equals one, the condition evaluates to false, confirming that phi and m are coprime, at which point the loop terminates. This procedure ensures the selection of a legitimate public exponent.

- Next, we compute the modular inverse of m with respect to phi using the modInverse function, which determines the value of m that satisfies the modular equation.

Text encryption and RGB conversion:

In this section, the program accomplishes three main tasks: first, it transforms the plaintext message into ASCII values, then encrypts it using MRSA, and finally, it converts the encoded information into RGB values.

The program starts by receiving a string input message from the user via a text-entry field on a graphical interface. After that, it retrieves and processes the entered message using the getText() function, ensuring it formats correctly for future usage.

The next step modifies the length of the input string to make its total length a multiple of three. This adjustment ensures that the program can process the string in three-character chunks.

The program converts the plaintext string into ASCII codes, representing each character by its corresponding number. It then transforms these codes using the MRSA encryption function, which operates similarly to the power method in RSA. The program stores the encoded data in the encrypted[] array. The loop iterates through each character of the plaintext, processes it, and displays the resulting encrypted values on the console. When the loop reaches the end of the plaintext, it inserts a fixed encrypted number (128) to represent the space.

The do-while loop handles the ciphered message and assigns the coded values to the r[], g[], and b[] arrays, which stand for the RGB channels of the image. For every set of three values, it allocates z to the red channel, x to the green channel, and y to the blue channel. If there are not enough encrypted values for the green or blue channels, the program fills in the missing data with zero.

The loop processes all encrypted data and incorporates it into the RGB channels. The program increments the index (u) by three to handle the next set of values, while the flag variable monitors the current pixel, ensuring the message inserts correctly in the image in sequential order.

Image Creation and File Saving:

```
BufferedImage img = new BufferedImage(
    imglength,imglength, BufferedImage.TYPE_INT_RGB );
for (int a = 0; a < r.length; a++) {
   int rgb = (r[a] << 16) | (g[a] << 8) | b[a];
   img.setRGB(a, a, rgb);}
```

```
ImageIO.write(img, "PNG", new File(name + "MyFile.png"));
```

- The BufferedImage class stores an image (square in shape) in memory, allowing for direct pixel manipulation. It represents each pixel using 24-bit integers, with 8 bits allocated for each of the three color channels (red, green, and blue).
- The for loop assigns RGB values to each pixel by combining the red, green, and blue components into a single integer using bitwise shifts and OR operations. The program then applies this integer, which represents the pixel's color, to the image at coordinates (a, a).
- Finally the image is saved to a file in PNG format. The file is named dynamically by combining the string "name" with "MyFile.png" and specifies the save location.

5.4.2 Decryption Process

Decrypt Function

```
public static BigInteger decrypt(BigInteger c, BigInteger d, BigInteger n) {

  return c.modPow(d, n);

}
```

The method implements the MRSA decryption using the modPow(d, n) function, which performs modular exponentiation to reverse the encryption system and retrieve the original plaintext.

Main Method

A file chooser dialog opens to allow the user to select an image file for decryption.The application retrieves the location of the chosen picture file when the dialog box appears.

We follow the identical procedures for the coprime check, key creation, and decryption as we did for the encryption process, ensuring consistency in both stages.

FileWriter for Output

This portion creates a FileWriter object that allows the program to write the decrypted message to a file named "decrypted.text." The FileWriter writes the data to this file.

Reading the Image

```
BufferedImage img = ImageIO.read(file);
int widthimg = img.getWidth();
```

The program reads the selected image file into memory and retrieves its width to determine the number of pixels, which we use for iteration during processing.

Processing Image Pixels

This part retrieves the RGB value of the pixel at the specified coordinates in the image. Subsequently, it transforms the pixel value into a Color object using the getRed, getGreen, and getBlue methods and extracts these elements for additional processing.

Decrypting RGB Values

```
BigInteger bigred = BigInteger.valueOf(red);
BigInteger biggreen = BigInteger.valueOf(green);
BigInteger bigblue = BigInteger.valueOf(blue);
BigInteger rr = decrypt(bigred, d, n);
BigInteger gg = decrypt(biggreen, d, n);
BigInteger bb = decrypt(bigblue, d, n);
```

First, it converts the red, green, and blue values into BigInteger objects for MRSA decoding. Then, it applies the decryption procedure in the same manner as the encryption process.

Converting to Character

```
int rri = rr.intValue();
int ggi = gg.intValue();
int bbi = bb.intValue();
char rsaR = (char) rri;
char rsaG = (char) ggi;
char rsaB = (char) bbi;
```

In this section, the code converts the decrypted values into numeric form and then translates these numbers into characters. These characters represent the decoded color components of the pixel.

Writing Decrypted Data

```
write.append(rsaR);
write.append(rsaG);
write.append(rsaB);
write.flush();
```

The code appends the decrypted RGB values as characters to the output file and ensures it writes the data immediately, preserving the integrity of the content.

Final output: The code prints a message confirming that it has successfully stored the decrypted RGB values in the file, indicating completion of the decryption and saving process.

CHAPTER SIX
RESULT ANALYSIS

6.1 The GUI Output

6.1.1 Output for RSA

The application features a message input interface for the sender. When the user types the message "hello world" in the plaintext field and clicks the encrypt button, the application transforms it into RGB colors. The resulting ciphertext is then presented in a separate frame (Figures 6.1 and 6.2). A PNG image file is subsequently generated (Figure 6.3) and added to an MS Word document accompanied by text annotations (Figure 6.4).

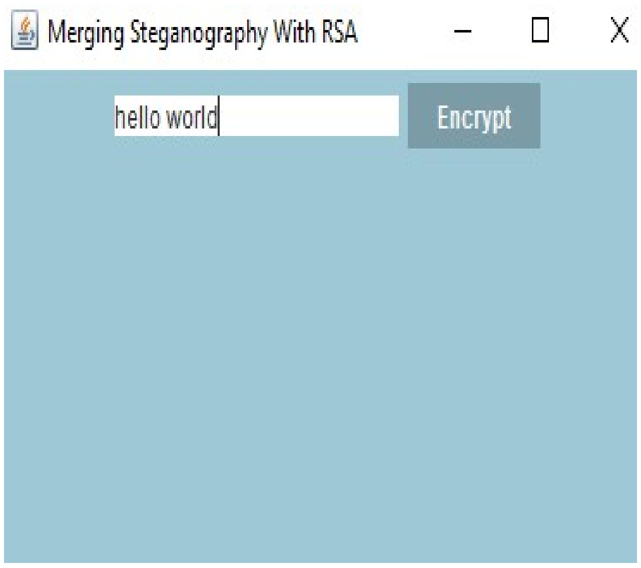


Figure 6.2: Plaintext Interface

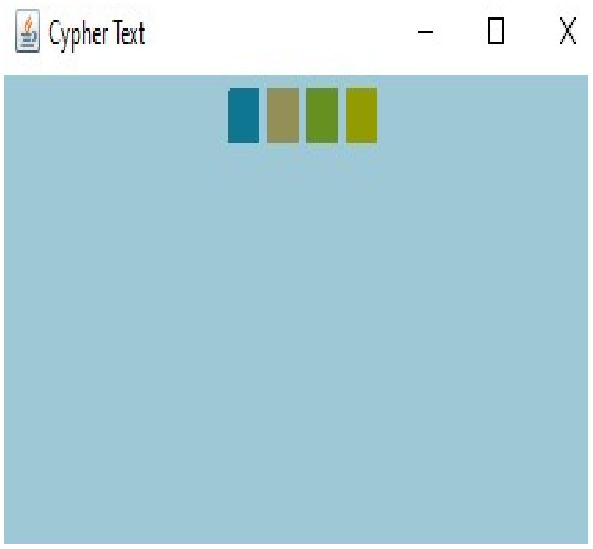


Figure 6.1: Ciphertext Interface

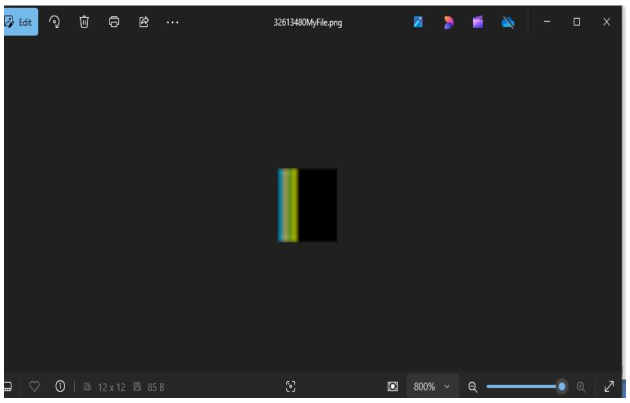
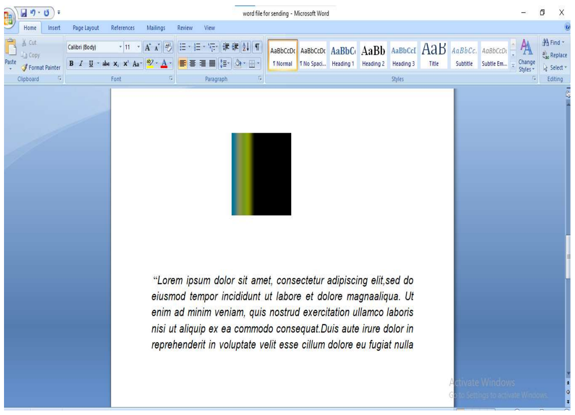


Figure 6.3: Encrypted Image(PNG Type)



"Lorem ipsum dolor sit amet, consectetur adipiscing elit,sed do eiusmod tempor incididunt ut labore et dolore magnaaliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla

Figure 6.4: Final Sending File

At the receiver's end, the file transmitted via any communication medium is received. The receiver subsequently extracts the image from the text cover. Upon executing the decryption process, a dialog box is prompted, allowing the user to select the image file shown in figure (6.5). Once the file is chosen, the image is decrypted and restored to its original plaintext message, which is shown in figure (6.6).

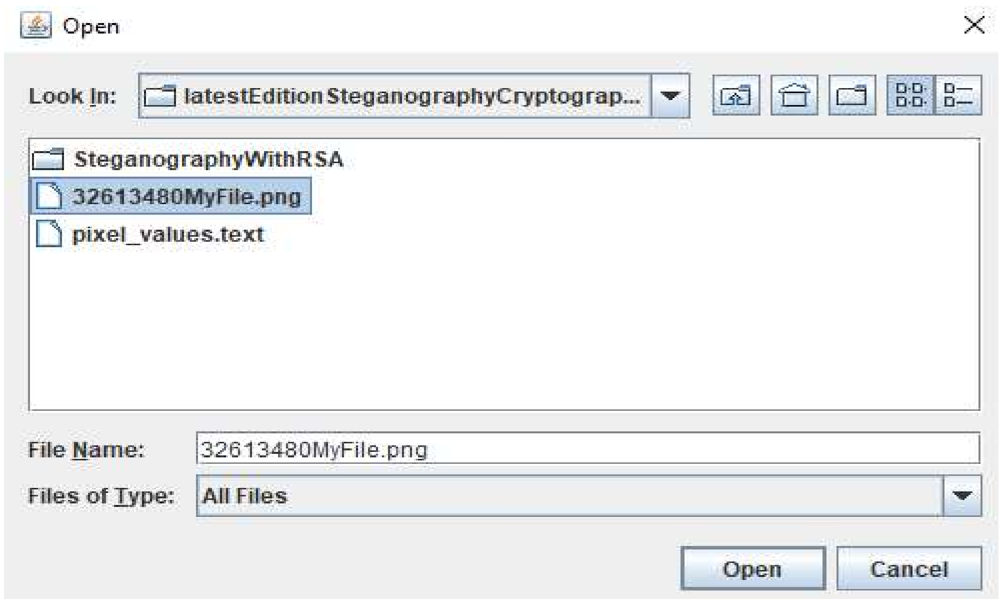
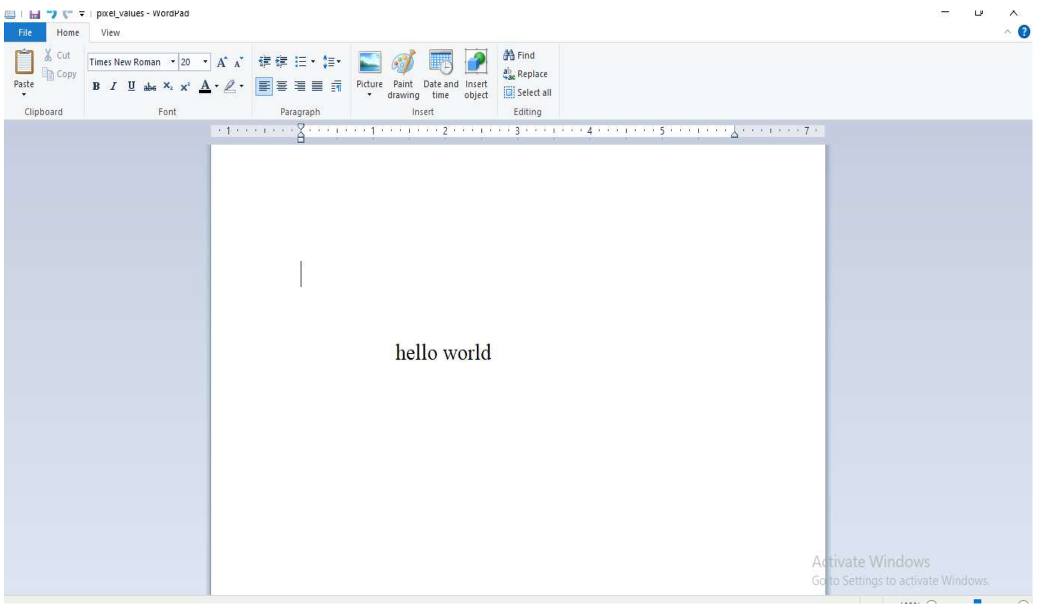


Figure 6.5: Dialog Box for Openning Image File

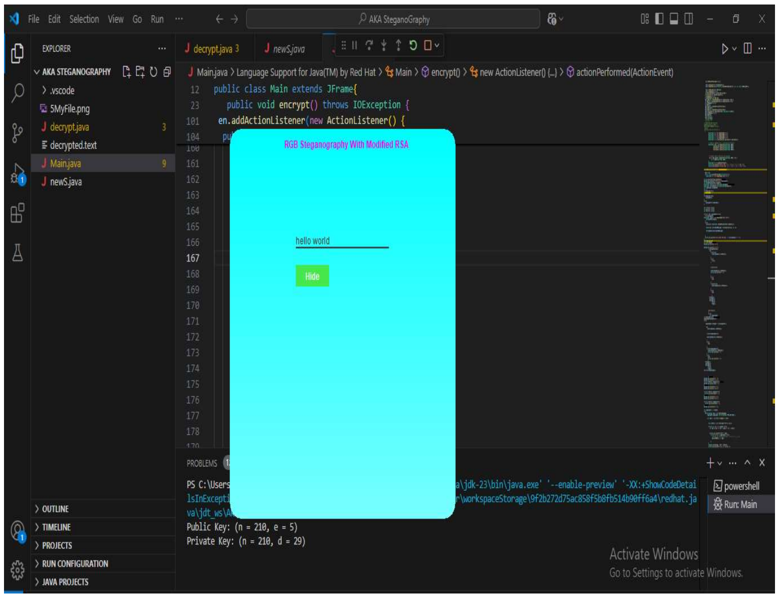


hello world

Figure 6.6: Final Decrypted Output

6.1.2 Output for MRSA

In the case of MRSA, we utilize an enhanced graphical user interface (GUI) to facilitate user input. However, we do not employ such an interface for displaying the ciphertext. The process follows the same steps as RSA for tasks such as image creation, transmission, reception, and decryption.



```java
public class Main extends JFrame{
  public void encrypt() throws IOException {
en.addActionListener(new ActionListener() {
 pu
```

```
PS C:\Users                                          a\jdk-23\bin\java.exe' '--enable-preview' '-XX:+ShowCodeDetai
lsInExcepti                                          r\workspaceStorage\9f2b272d75ac858f5b8fb514b90ff6a4\redhat.ja
va\jdt_ws\A
Public Key: (n = 210, e = 5)
Private Key: (n = 210, d = 29)
```

Figure 6.7: User Interface for Input

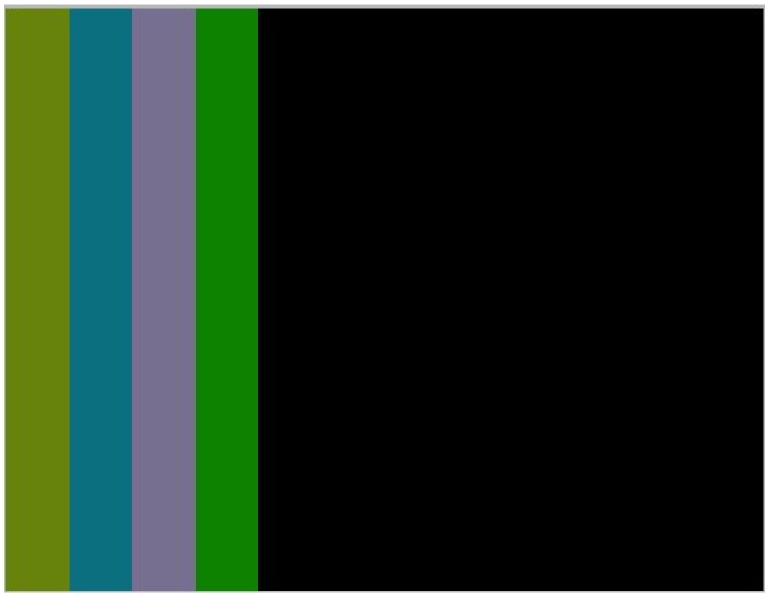
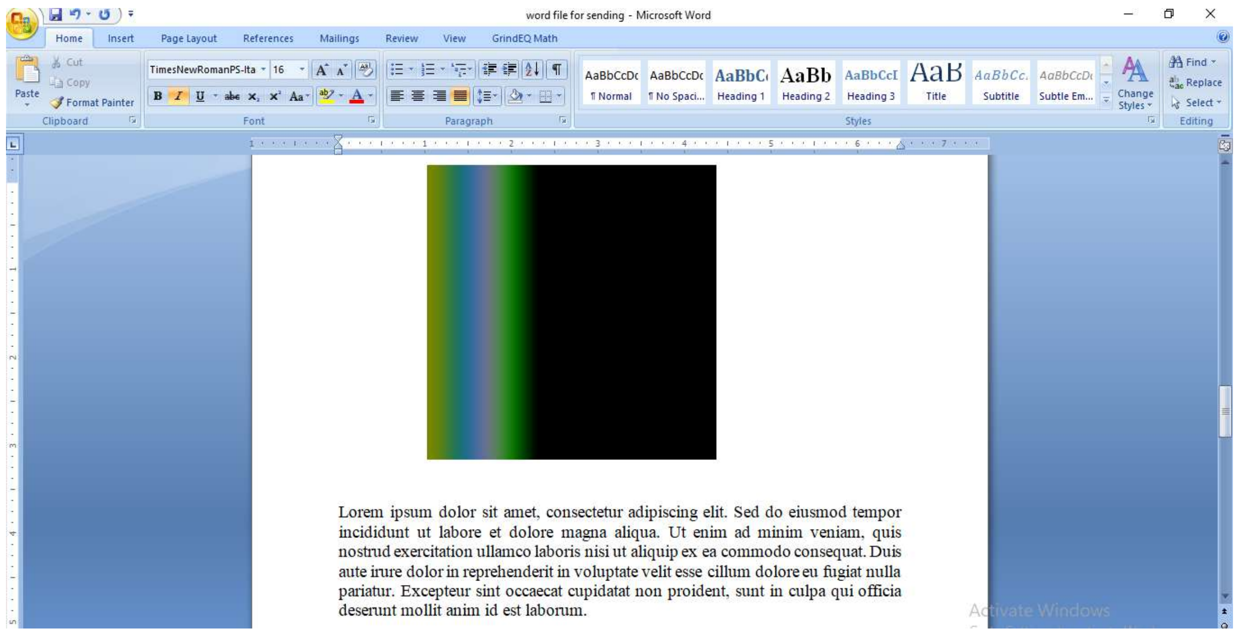


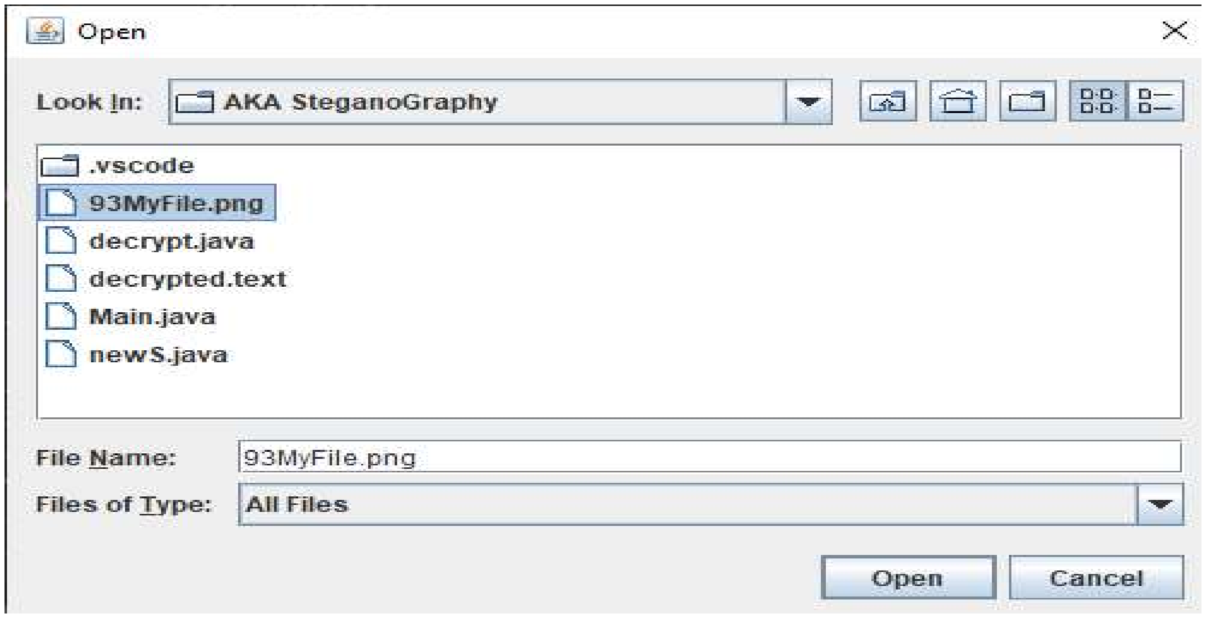
Figure 6.8: Encrypted Image for MRSA



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 6.9: Resulted Encrypted Sending Document

Figures 6.7, 6.8, 6.9, 6.10, and 6.11 display the corresponding outputs for these steps. These figures visually represent the process from start to finish, showcasing how each phase of MRSA functions, from the initial image creation to the final decrypted output.

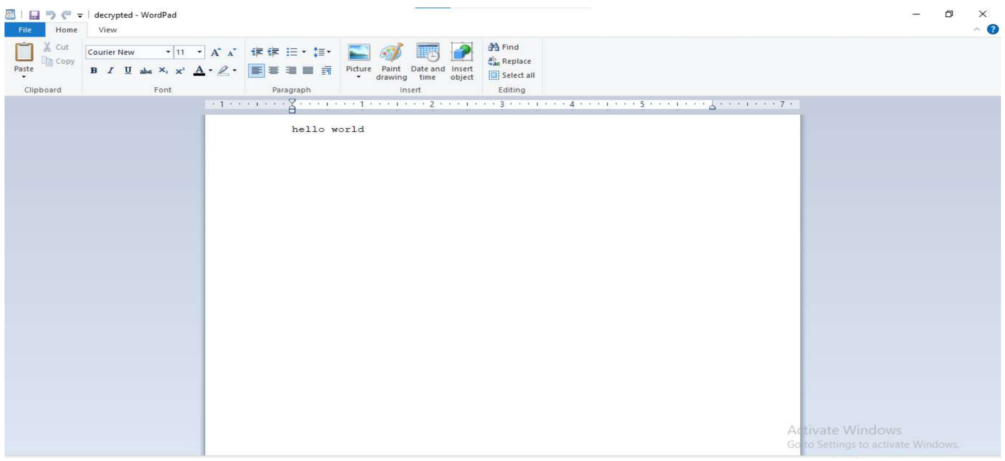


Look In: AKA SteganoGraphy

.vscode
93MyFile.png
decrypt.java
decrypted.text
Main.java
newS.java

File Name: 93MyFile.png
Files of Type: All Files

Figure 6.10: Choosing Image for Decrytion



```
hello world
```

Figure 6.11: Output after Decryption

6.2 Analyzing Time Complexity

6.2.1 RSA Encryption

This research combines RSA encryption with picture synthesis for steganography, dissecting it into logical chunks to examine each component's purpose and function.

- The declarations, initialization, and configuration of the UI components have no loops or complicated operations, hence the time complexity is constant, O(1).
- Th javaConstructor just invokes the initialize() function. The constructor's time complexity is constant for the best, worst, and average cases since it does not conduct any computations or loops.
- The simple arithmetic operations that generate RSA parameters has constant time complexity of the best, worst, and average cases.
- We loop through each character of the input string (length l) for text to ASCII conversion, so this is O(l).
- The RSA encryption algorithm involves performing exponentiation via squaring, which has a time complexity of O(log f), where f is the public exponent. For each character in the message, the encryption process applies this operation, resulting in a total time complexity of O(l log f), where l is the length of the input text. Since f is constant, this can be considered O(1) for practical purposes.
- The next block of code divides the array into three colors (RGB) and stores the RGB values in an image. In this portion, We iterate over the full string (l characters) and convert it to RGB values. This section includes looping through the plaintext, which has a length of l.

- The image creation process involves two nested loops, each iterating rlength times per cycle. The size of the picture is rlength^2, where rlength is proportional to the input string's length (l). Therefore, the time complexity for generating the image is O(l^2).
- When creating the image, we save it to a file. Writing a picture to a file increases proportionally to its size, O(rlength^2).

Therefore, the overall time complexity is: O(l^2).

Best case: This occurs when everything works smoothly, with minimum padding and a tiny picture size. The ideal scenario is dominated by the picture generation and file writing processes.So, the cost is O(l^2).

Worst-case: In this instance, all operations (encryption, RGB conversion, picture production, and file writing) contribute to a quadratic computational cost. Therefore, the overall efficiency is O(l^2).

Average case: The picture formation and file writing phases drive the computational cost, both growing quadratically with input length. The entire average proficiency is O(l^2).

6.2.2 RSA Decryption

- **Reading the Image:** The time complexity of reading the image is proportional to the number of pixels, so the temporal cost is O(n * m), where n is the width and m is the height of the picture.
- **Iterating through Image Pixels:** The outer loop iterates n times, corresponding to the width of the image. For each iteration of the outer loop, the inner loop executes n times, once for each row. Each operation within the inner loop requires a constant amount of time, so the overall time complexity is O(n^2).

```
for (int y = 0; y < img.getWidth(); y++) {
   for (int x = 0; x < img.getWidth(); x++) {
      //Operations for extracting pixel values
   }
}
```

- **power() method:** The computational complexity of decrypting a picture using RSA for each RGB component involves modular exponentiation with a cost of O(log d) per element. Since each image unit contains three channels (red, green, and blue), the per-pixel complexity is O(log d). For an n×n image, with n^2 total pixels, the overall time complexity is O(n^2* log d).
- **Writing to the File:** The procedure transforms each decrypted RGB value into a character and stores it in the file. Each append action for a color channel takes constant time. Writing to the file for n^2 image elements yields a total time complexity of O(n^2).

Combining these operations, we get the overall time complexity as: O(n^2 * log d).

Best Case: This case mirrors the general case, as both the image size nn and the RSA key dd stay constant regardless of the input data. Consequently, the time complexity is O(n^2 * log d).

Worst Case: The worst-case scenario aligns with the best case, as all operations (loading the image, processing the pixels, performing modular exponentiation, and saving to the file) require consistent time for any given input. Thus, the time complexity remains O(n^2 * log d).

Average Case: The average case follows the same pattern as the best and worst cases, as the image dimensions and RSA decryption complexity do not vary with different inputs. Therefore, the time complexity is O(n^2 log * d).

6.2.3 MRSA Encryption

The key generation Phase: It consists of three stages:

i. Calculating phi: This step involves multiplying four integers and takes constant time.
ii. Finding the GCD (m, phi): Using the Euclidean method, this procedure computes the greatest common divisor in time that grows logarithmically with the size of the product of the prime numbers.
iii. Calculating m.modInverse(φ): The current operation computes the modular inverse via the extended Euclidean method, which also runs in O(log n) time.

Therefore, the overall key generation time complexity is **O(log n)**.

The encryption process relies on modular exponentiation, utilizing the "exponentiation by squaring" technique. This operation's time complexity depends on the size of the exponent. The small size of the exponent ensures that the time complexity remains effectively constant, as long as it stays relatively small.

Message Encryption Loop: The encryption process loops through each character in the plaintext, converting it to a BigInteger based on its ASCII value (O(1)). It then encrypts every character, which has a time complexity of **O(log n)** due to the exponentiation by squaring method.

Consequently, the cumulative time complexity for encrypting the entire message is **O(plength * log n)**, where plength is the length of the plaintext and n is the modulus (typically the product of two primes in RSA).

Splitting into RGB values: This section processes the encrypted data and embeds it into RGB values by performing simple assignments and logical checks, such as verifying the remainder. Since the loop iterates once over the data, with each cycle involving constant-time operations, the time complexity is **O(plength)**.

Image Creation and Writing: The process begins by constructing a BufferedImage from the encrypted message data, and it involves two loops: the first converts RGB values into pixel hues, and the second maps these hues to the image element. The first loop operates with a time complexity of **O(r.length)**, while the image writing step has a complexity of **O(imglength^2)**. As a result, the total time complexity is **O(r.length + imglength^2)**.

Final Time Complexity:

The total time complexity of the encrypt() function is dominated by the encryption loop and the image creation loop. Therefore, the overall complexity is therefore:

O(plength * log n + imglength^2)

The best case happens when the plaintext is very small, such as a single character. In this case, the time complexity is dominated by the O(logn) complexity from RSA encryption. So, the best-case overall complexity is: O(logn).

The worst case occurs when the plaintext is very large, and thus both the RSA encryption and the image creation become expensive. The complexity is dominated by O(imglength^2) for the image creation. So, the worst-case overall complexity is: O(imglength^2).

The average case is when the plaintext is of moderate length. Since RSA encryption takes O(plength·logn) and image creation takes O(imglength^2), the overall time complexity will be dominated by the O(imglength^2) term in practice for most typical message lengths. So, the average-case overall complexity is: O(imglength^2).

6.2.4 MRSA Decryption

Image Reading: With W standing for the image's width and H for its height, the operation's temporal complexity is **O(W*H)**. This occurs as a result of the procedure's requirement to load every pixel into memory.

Iterating Over the Pixels: Using a for loop, the process extracts the RGB value for each pixel. Since the iteration focuses solely on the width of the image, the time complexity for this operation is **O(W)**.

Color Component Extraction: These operations have constant time complexity because retrieving each color component (red, green, and blue) involves a simple lookup. As a result, the time complexity for this process is **O(1)**, indicating that it takes the same amount of time regardless of the image size.

Decryption Operation: The decryption process involves a modular exponentiation operation where each calculation takes **O(log n)** time complexity. Here, the MRSA algorithm uses n as the modulus.

Writing Decrypted Data to a File: Writing three characters for each pixel takes O(1) time. Since we write data for each pixel, the total time complexity for this step is **O(W)**.

The total time complexity is dominated by the image reading and decryption steps. Therefore, the overall time complexity is: **O(W*H)+O(W*log n)**.

When n is small or height is constant, the **optimal situation** arises. In this scenario, the width of the image dominates the complexity, and the decryption term remains **O(W*log n)**.

The worst-case scenario arises when dealing with exceptionally large images, where both the width and height are substantial, and the modulus value is also significantly large. Both the picture reading and decryption processes in this case greatly increase the total time complexity, which is **O(W*H+W*log n)**.

Average Case Complexity is also Similar to the worst case, O(W*H+W*log n).

6.3 Analyzing Space Complexity

6.3.1 RSA Encryption

Space complexity is a measure of memory utilization relative to input size. We'll examine how it changes in the best, worst, and average circumstances and split down the spatial complexity of our study into variables, methods, classes, and code blocks.

Instance Variables (Field Variables)

- The worst, average, and best-case spatial complexity of the GUI components are often constant.
- The int[] arr array has O(l) space complexity, where l is the length of the input string, as it stores the ASCII values of the characters.
- The int[] r, g, and bcol arrays hold the RGB values for each letter in the input text. Every array has a capacity that is a multiple of 3, resulting in O(l) memory usage, where l is the adjusted size of the input string.
- The double[] cypher array records the encrypted values of each character in the input string. Since its size equals the input length, it has a spatial complexity of O(l).
- The integer variables s, b, j, k, and flag handle iteration and processing, each consuming constant space, O(1).
- The int multiply variable calculates a product and also requires constant space, O(1).

- The BufferedImage img contains the image, and its size depends on the input string's length. For a square image (rlength x rlength), the space complexity is O(r^2), where r is the length of the array holding the RGB values. Since r relates to the input length, the space complexity reaches O(l^2) in the worst case.
- The file myFile occupies constant space, O(1), as its size does not depend on the input length.

Methods

Now we calculate space complexity of individual methods and their execution:

- The power method consumes a fixed amount of memory because it simply keeps a small number of integers and doesn't build any big data structures. On the basis of x, y, and p values, it iteratively calculates.
- In the intialize method, the arrays made to store the input and RGB values (arr, r, g, bcol, and cipher), the image creation by BufferedImage, and saving the image into the file myFile are the main sources of space complexity. we previously discussed them.
- The main method simply creates an instance of the javaConstructor class, so it doesn't use significant memory by itself. It has constant space complexity, O(1).

The **best case** occurs when the input string is either empty or very short, such as when l = 1. In this case, the arrays arr, r, g, bcol, and cypher each consume O(1) space, resulting in minimal memory usage. However, the image buffer still requires space proportional to the input size. As a result, the overall space complexity remains O(l^2).

The **worst-case** scenario arises when the input string is very long. As the length l increases, the space required by the arrays (arr, r, g, bcol, cypher) scales linearly, or O(l). The BufferedImage creation during initialization also expands quadratically, requiring O(l^2) space to store the image data. Consequently, the overall worst-case space complexity is O(l^2), with the BufferedImage object contributing significantly to the increased space usage as the input text length grows.

The **average scenario** is the condition between best and worst. When l is moderate, the arrays use O(l) space, and the image buffer requires O(l^2) space, which primarily determines the space complexity. Therefore, the space complexity in the average case is likely to approach O(l^2), with the image generation playing a significant role in the increased space usage.

In summary, the most significant space usage in this program is driven by the image storage (BufferedImage), which leads to an overall worst-case space complexity of O(l^2), where l is the length of the input string.

6.3.2 RSA decryption

Power function: The space complexity of the exponentiation function is O(1), as it utilizes a fixed number of integer variables, which remain independent of the input size. The function performs fundamental arithmetic operations without allocating additional memory. Consequently, the memory consumption remains constant, ensuring optimal space efficiency.

File selection: The space complexity of JFileChooser and the File instance is O(1), as both consume a constant amount of memory without loading the file itself. The string representing the file path has O(l) complexity, where l denotes the length of the path, which is independent of the file size.

RSA Components: The RSA decryption process uses various integers like p, q, n, phi, f, and d (which don't depend on input size), so their space complexity is **O(1).**

FileWriter: It has a space complexity of O(1) because it uses a fixed amount of memory for buffering. It does not store the entire file in memory at once, so memory usage remains constant regardless of the file size.

Image Data: The `BufferedImage` class stores pixel data for each point. Its size depends on the image's dimensions, requiring space proportional to the total number of image elements. Consequently, memory usage is O(width * height), as it holds information for every picture element.

Loop (Iterating over image pixels): The for loops process each element, utilizing fixed space per element for variables such as pixel, color, red, green, blue, r, g, b, rstr, gstr, and bstr. Each of these consumes O(1) space. Consequently, the total space complexity is O(N), where N represents the number of elements in the image.

File writing: Each call appends the encrypted RGB characters to the file. The memory usage of the writer remains constant, independent of the image size, while the final file size on disk is proportional to the number of pixels and characters per pixel, i.e., O(W * H).

Therefore, the final space complexity of the decryption process is **O(W * H)**, where W is the width and H is the height of the image.

Best Case: The best case occurs when the input image has a small resolution, resulting in a relatively small memory footprint. However, the space complexity remains **O(W * H).**

Worst case: The worst case arises when the image has a large resolution, which increases the memory requirement to store the image. In this case, the space complexity is O(W * H), and the memory usage can become significant**.**

Average case: The space complexity is **O(W * H)**, assuming typical image sizes. It will depend on the dimensions of the input image but will generally scale linearly with the number of pixels.

6.3.3 MRSA Encryption

To analyze the space complexity, we will break it down step by step, examining memory usage at each stage.

- The import statements do not significantly impact space complexity, as they simply include necessary libraries for GUI management, image processing, and cryptography. These libraries provide essential functions without allocating substantial memory on their own.
- The encrypt method, which handles RSA encryption, does not contribute much to space complexity. It relies on the modPow method, an efficient modular exponentiation algorithm, and does not allocate additional memory for intermediate results or large data structures. Therefore, its memory impact is minimal.
- The constructor invokes the encrypt() method, but it does not cause significant memory consumption except for the space the GUI components occupy. The constructor mainly sets up necessary functionality, keeping the memory footprint small.
- The GUI elements, including the JFrame, buttons, text fields, and labels, occupy a set amount of memory. We fix the number of GUI elements, such as the JFrame, buttons, text fields, and labels, in advance, and their memory usage remains steady as they do not change dynamically. Although this contributes to the overall memory usage, it does not scale with input size, making its space complexity constant.
- The primitive variables consist of seven fixed parameters. Each element allocates memory based on its size, but since it holds static values, the storage consumption stays consistent. The input text requires space proportional to its length, resulting in a space complexity of O(plength).
- Three arrays handle data for encryption and steganography. The ASCII array stores ASCII values, sized according to plength, while the encrypted array either holds one more element than the plaintext length or adjusts to be a multiple of three. The r, g, and b arrays store RGB values, with their dimensions depending on the input text length (plength). These arrays thus contribute O(plength) to the overall space complexity.
- The image has dimensions of imglength x imglength. Since imglength is roughly proportional to plength, the image's space complexity is O(plength^2).

As a result, image storage dominates the total space complexity, contributing O(plength^2).

Best Case: This occurs when the input length is minimal (e.g., one or two characters). In this scenario, the space complexity remains O(plength^2) because the image storage primarily drives memory consumption. Therefore, the best-case space complexity is O(plength^2).

Worst Case: The worst-case scenario mirrors the best case, as the space complexity depends largely on the image size and input length. Even if the input length increases, the image storage still requires O(plength^2) space.

Average Case: Since the image memory usage scales with the square of the input length and the number of pixels increases quadratically with the input size, the space complexity in the average case remains O(plength^2).

6.3.4 MRSA Decryption

Variables: The variables p, q, r, s, n, phi, m, and d represent values related to the decryption process. These variables occupy constant space due to their small, fixed sizes.

FileWriter: It occupies constant space for file output. This is because it handles the file output process without any dependencies on the size of the image or decrypted content.

Color Information: The Color object stores the pixel color, while the variables red, green, and blue save the individual RGB values of each image component. Additionally, these variables need a certain amount of space for every pixel.

Image Dimensions and Pixel Storage: The constant values in the variables (pixel, width) determine the size of the picture. The image's storage space complexity is **O(width * height)**, as each pixel has three RGB color components. Each element takes up a predetermined amount of space for its color data, increasing the overall number of units.

Decryption Process: It uses three BigInteger objects per pixel (bigred, biggreen, and bigblue), with space complexity proportional to the number of pixels. After decryption, the program converts the BigInteger values (rr, gg, bb) into integers and characters, which use constant space.

Final Output: The FileWriter writes decrypted characters (rsaR, rsaG, rsaB) to the file, using constant space relative to the data being written.

Therefore, The Final Space Complexity is **O(widthimg*heightimg).** This reflects the space the image requires to store, which dominates all other space requirements.

In the **best case**, even if the image is very small (e.g., 1x1), the space complexity is still proportional to the size of the image.

In the **worst case**, with a very large image, the space complexity still depends on the size of the image.

On average, the image size is not unusually small or large, the space complexity is still proportional to the image's dimensions.

6.5 CPU Utilization

Figure 6.12 presents a comparison of the execution time for encryption between two schemes. It shows that for smaller input bit lengths, the two algorithms exhibit nearly identical execution times. However, as the bit length increases, the disparity between the execution times of the two algorithms grows significantly, with the difference becoming more pronounced as the input size expands.

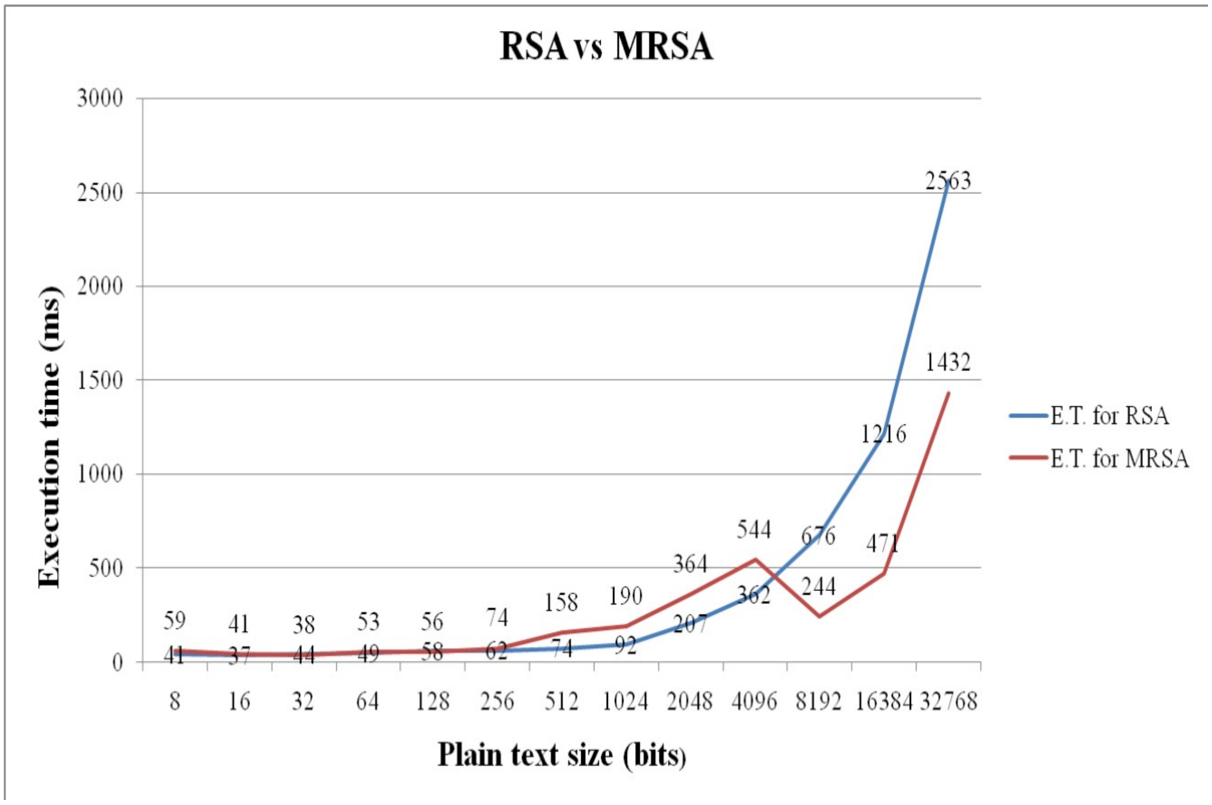


Figure 6.12: Execution Time Comparison for two Schemes

CHAPTER SEVEN
CONCLUSION

In this study, we combine RSA encryption and steganography, two crucial methods for securing data. We use RSA encryption to protect privacy by applying a public-private key pair for asymmetric encryption, which makes unauthorized decryption nearly impossible. At the same time, we use steganography to hide the encrypted data within an image by altering its RGB color values. By merging these two techniques, we create a multi-layered security system: encryption keeps the data safe from interception and unauthorized access, while steganography hides the message itself, greatly reducing the chance of detection by potential attackers.

The main advantage of this study is the increased security and privacy provided through encryption and data concealment. By embedding sensitive information within an image, we ensure that anyone trying to track it cannot uncover the hidden message unless they have the decryption key. This two-layered approach is valuable in fields like secure communication, digital forensics, and private information exchange, where it is crucial to protect confidential data in an increasingly connected world. Additionally, we offer a practical teaching tool that shows how modern cryptography and steganography work, highlighting their effectiveness in protecting information against growing cybersecurity threats.

.

REFERENCES

[1] W. E. Al-Ahmadi, A. O. Aljahdali, F. Thabit, and A. Munshi, “A secure fingerprint hiding technique based on DNA sequence and mathematical function,” *PeerJ Comput. Sci.*, vol. 10, p. e1847, Mar. 2024, doi: 10.7717/peerj-cs.1847.

[2] M. A. Majeed, R. Sulaiman, Z. Shukur, and M. K. Hasan, “A Review on Text Steganography Techniques,” *Mathematics*, vol. 9, no. 21, p. 2829, Nov. 2021, doi: 10.3390/math9212829.

[3] R. Imam, Q. M. Areeb, A. Alturki, and F. Anwer, “Systematic and Critical Review of RSA Based Public Key Cryptographic Schemes: Past and Present Status,” *IEEE Access*, vol. 9, pp. 155949–155976, 2021, doi: 10.1109/ACCESS.2021.3129224.

[4] F. H. Rabevohitra and Y. Li, “Text Cover Steganography Using Font Color of the Invisible Characters and Optimized Primary Color-intensities,” in *2019 IEEE 19th International Conference on Communication Technology (ICCT)*, Xi’an, China: IEEE, Oct. 2019, pp. 1704–1708. doi: 10.1109/ICCT46805.2019.8947188.

[5] Md. Khairullah, “A Novel Text Steganography System Using Font Color of the Invisible Characters in Microsoft Word Documents,” in *2009 Second International Conference on Computer and Electrical Engineering*, Dubai, UAE: IEEE, 2009, pp. 482–484. doi: 10.1109/ICCEE.2009.127.

[6] T.-Y. Liu and W.-H. Tsai, “A New Steganographic Method for Data Hiding in Microsoft Word Documents by a Change Tracking Technique,” *IEEE Trans. Inf. Forensics Secur.*, vol. 2, no. 1, pp. 24–30, Mar. 2007, doi: 10.1109/TIFS.2006.890310.

[7] L. Xiang, W. Wu, X. Li, and C. Yang, “A linguistic steganography based on word indexing compression and candidate selection,” *Multimed. Tools Appl.*, vol. 77, no. 21, pp. 28969–28989, Nov. 2018, doi: 10.1007/s11042-018-6072-8.

[8] “Text Steganography Based on Ci-poetry Generation Using Markov Chain Model,” *KSII Trans. Internet Inf. Syst.*, vol. 10, no. 9, Sep. 2016, doi: 10.3837/tiis.2016.09.029.

[9] Y. Luo and Y. Huang, “Text Steganography with High Embedding Rate: Using Recurrent Neural Networks to Generate Chinese Classic Poetry,” in *Proceedings of the 5th ACM Workshop on Information Hiding and Multimedia Security*, Philadelphia Pennsylvania USA: ACM, Jun. 2017, pp. 99–104. doi: 10.1145/3082031.3083240.

[10] Z. Yang, P. Zhang, M. Jiang, Y. Huang, and Y.-J. Zhang, “RITS: Real-Time Interactive Text Steganography Based on Automatic Dialogue Model,” in *Cloud Computing and Security*, vol. 11065, X. Sun, Z. Pan, and E. Bertino, Eds., in Lecture Notes in Computer Science, vol. 11065. , Cham: Springer International Publishing, 2018, pp. 253–264. doi: 10.1007/978-3-030-00012-7_24.

[11] Z.-L. Yang, X.-Q. Guo, Z.-M. Chen, Y.-F. Huang, and Y.-J. Zhang, “RNN-Stega: Linguistic Steganography Based on Recurrent Neural Networks,” *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 5, pp. 1280–1295, May 2019, doi: 10.1109/TIFS.2018.2871746.

[12] N. Y. Goshwe, “Data Encryption and Decryption Using RSAAlgorithm in a Network Environment”.

APPENDIX

Encryption for RSA

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;
public class javaConstructor {
    JFrame frame = new JFrame("Merging Steganography With RSA");
     JTextField plainText= new JTextField(15);
     JButton encrypt= new JButton("Encrypt");
     JTextField outputText= new JTextField(15);
     static int power(int x, int y, int p)
     {
       int res = 1;
       x = x % p;
if (x == 0)
         return 0;
    while (y > 0)
       {
         if ((y & 1) != 0)
           res = (res * x) % p;
         y = y >> 1;
         x = (x * x) % p;
       }
       return res;
     }
    public javaConstructor() throws IOException{
        initialize();
    }
    public void initialize() throws IOException{
    int p=17,q=11;
    int n=p*q;
    int phi=(p-1)*(q-1);
     frame.setVisible(true);
     frame.setSize(400, 400);
     frame.setLocation(0,0);
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.getContentPane().setBackground(Color.BLUE);
     encrypt.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
  long startTime = System.nanoTime();
        int f=13;
        double d=1.0;


              d=(1+3*phi)/f;
         String output=plainText.getText();
       int l=output.length();
       int bitsize= 8*l;
       System.err.println(bitsize);
     int arr[]=new int[l+1];
     int length;
     if(l%3==0){
       length=l;
     }
     else{
      if(l%3==1){
        length=l+2;
      }
      else{
        length=l+(3-l%3);
      }
     }
    int r[]=new int[length];
     int g[]=new int[length];
     int bcol[]= new int[length];
     double cypher[]=new double[l+1];
     int multiply=1;
     int s=0,b=0,j=0,k=0,flag=0;
     System.out.println("The ASCII values are");
         for(int i=0;i<l+1;i++){
          if(i==l){
            arr[i]=32;
          }
```

```
          else{
            arr[i]= Integer.valueOf(output.charAt(i));
            System.out.print(arr[i]+" ");
            cypher[i]=power(arr[i], f, n);
           arr[i]=(int)cypher[i];
          }
           }
        System.out.println("After Implementin RSA, the cypher text is");
        for(int z=0;z<arr.length;z++){
          System.out.print(arr[z]+" ");
        }
    JFrame newframe= new JFrame("Cypher Text");
      newframe.setVisible(true);
      newframe.setSize(400, 400);
      newframe.setLocation(0,0);
      newframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JPanel newJPanel= new JPanel();
      if(l%3==0){
  do{
    if(s<l-2){
      b=arr[s];
    }
    else{
      break;
    }
    if(s+1<l-1){
    j=arr[s+1];
    }
    else{
      j=0;
    }
    if(s+2<l){
      k=arr[s+2];
    }
    else{
      k=0;
      }
  r[flag]=b;
  g[flag]=j;
  bcol[flag]=k;
```

```
s=s+3;
flag++;
      multiply*=r[flag];
      JButton rgb = new JButton();
      rgb.setForeground(new Color(250,250,250));
      rgb.setBorderPainted(false);
      rgb.setPreferredSize(new Dimension(20,20));
     rgb.setBackground(new Color(b,j,k));
     newJPanel.add(rgb);
     newJPanel.setBackground(new Color(159, 200, 214));
}while(s<l);
 }
 else{
  if(l%3==1){
    do{
      b=arr[s];
      if(s+1<l-1){
      j=arr[s+1];
      }
      else{
        j=0;
      }
      if(s+2<l-1){
        k=arr[s+2];
      }
      else{
        k=0;
        }
   r[flag]=b;
    g[flag]=j;
    bcol[flag]=k;
        s=s+3;
         multiply*=r[flag];
         flag++;
         JButton rgb = new JButton();
         rgb.setForeground(new Color(250,250,250));
         rgb.setBorderPainted(false);
         rgb.setPreferredSize(new Dimension(20,20));
        rgb.setBackground(new Color(b,j,k));
        newJPanel.add(rgb);
```

```
            newJPanel.setBackground(new Color(159, 200, 214));
      }while(s<l);
      }
      else{
        do{
          b=arr[s];
             if(s+1<l){
          j=arr[s+1];
          }
          else{
            j=0;
          }
          if(s+2<l){
            k=arr[s+2];
          }
          else{
            k=0;
             }
       r[flag]=b;
        g[flag]=j;
        bcol[flag]=k;
            s=s+3;
             multiply*=r[flag];
             flag++;
             JButton rgb = new JButton();
             rgb.setForeground(new Color(250,250,250));
             rgb.setBorderPainted(false);
             rgb.setPreferredSize(new Dimension(20,20));
            rgb.setBackground(new Color(b,j,k));
            newJPanel.add(rgb);
            newJPanel.setBackground(new Color(159, 200, 214));
      }while(s<l+1);
      }
     }
for(int i=0;i<l;i++){
  System.out.print("r="+r[i]);
}
for(int i=0;i<l;i++){
  System.out.print("g="+g[i]);
}
```

```
for(int i=0;i<l;i++){
  System.out.print("bcol="+bcol[i]);
}
System.out.println();
       int rlength= r.length;
       System.out.println("Length of r"+rlength);
      try{
           BufferedImage img = new BufferedImage(
               rlength,rlength, BufferedImage.TYPE_INT_RGB );
           File myFile = new File(multiply+"MyFile.png");

           for(int x = 0; x <rlength; x++){
             int col = (r[x] << 16) | (g[x] << 8) | bcol[x];
               for(int y = 0; y <rlength; y++){
                   img.setRGB(x, y, col);
               }
           }
           ImageIO.write(img, "PNG", myFile);
           System.out.println("File is created.");
        }
        catch(Exception e1){
           System.out.println("Couldn't create sorry!");
        }

       newframe.add(newJPanel);
       long endTime = System.nanoTime(); // End timer
               long duration = (endTime - startTime); // Calculate duration
               System.out.println("Execution time in nanoseconds: " + duration);
               System.out.println("Execution time in milliseconds: " + duration / 1000000 +  "ms");
      }
     });
    JPanel open = new JPanel();
    encrypt.setBackground(new Color(123, 155, 166));
    encrypt.setForeground(new Color(250,250,250));
    encrypt.setBorderPainted(false);
    plainText.setBorder(null);

     open.add(plainText);
     open.add(encrypt);
     open.setBackground(new Color(159, 200, 214));
     frame.add(open);
    }
    public static void main(String[] args) throws IOException {
       javaConstructor a = new javaConstructor();
    }}
```

Decryption for RSA

```
import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.*;
public class ColorPicky {
  static int power(int x, int y, int p)
  {
    int res = 1;
    x = x % p;
    if (x == 0)
      return 0;
    while (y > 0)
    {
      if ((y & 1) != 0)
        res = (res * x) % p;
       y = y >> 1;
      x = (x * x) % p;
    }
    return res;
  }
   public static void main(String[] args) throws IOException {
     long startTime = System.nanoTime();
    JFileChooser chooser= new JFileChooser();
    chooser.showOpenDialog(chooser);
   File selectedFile = chooser.getSelectedFile();
  String path = selectedFile.getAbsolutePath();
int p=17,q=11;
int n=p*q;
int phi=(p-1)*(q-1);
int f=13;
int d=(1+3*phi)/f;
        FileWriter writer = new FileWriter("pixel_values.text");
        File file = new File(path);
        BufferedImage img = ImageIO.read(file);
        for (int y = 0; y < img.getHeight(); y++) {
          for (int x = 0; x <img.getWidth(); x++) {
            int pixel = img.getRGB(x,y);
            Color color = new Color(pixel, true);
            int red = color.getRed();
            int green = color.getGreen();
            int blue = color.getBlue();
            System.out.println("r="+red+",g="+green+",b="+blue);
            int r=power(red, d, n);
            int g=power(green, d, n);
            int b=power(blue, d, n);
           char rstr=(char)r;
            char gstr=(char)g;
            char bstr=(char)b;
            System.out.println(rstr+gstr+bstr+"");
            writer.append(rstr);
            writer.append(gstr);
            writer.append(bstr);
              writer.flush();
          }
          break;
        }
        long endTime = System.nanoTime();
                long duration = (endTime - startTime);
                System.out.println("Execution time in nanoseconds: " + duration);
                System.out.println("Execution time in milliseconds: " + duration / 1000000 +  "ms");
        writer.close();
        System.out.println("RGB values are stored in the file.");
    }
}
```

Encryption for MRSA

Main Class

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.geom.RoundRectangle2D;
import java.awt.image.*;
import java.io.*;
import java.math.BigInteger;
import javax.imageio.*;
import javax.swing.*;
public class Main extends JFrame{
  public static BigInteger encrypt(BigInteger message, BigInteger m, BigInteger n) {
    return message.modPow(m, n);
}
public Main() throws IOException{
  super("Modified RSA");
  encrypt();
}
   public void encrypt() throws IOException {
  BigInteger m = new BigInteger("5");
    setUndecorated(true);
    setLocation(400,150);
    setSize(400,470);
    setShape(new RoundRectangle2D.Double(0,0,400,470,40,40));
    setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));
    newS form =new newS();
    add(form);
   JTextField text= new JTextField(15);
   JTextArea title = new JTextArea();
   JLabel t = new JLabel("RGB Steganography With Modified RSA");
   JButton en= new JButton("Hide");
   JPanel aka = new JPanel();
  JPanel ti=new JPanel();
  en.setBackground(new Color(70, 232, 78));
  en.setOpaque(true);
  en.setFocusPainted(false);
   en.setForeground(new Color(250,250,250));
   en.setBorderPainted(false);
   text.setLayout(new GridLayout());
   text.setBorder(BorderFactory.createMatteBorder(0, 0, 2, 0, Color.DARK_GRAY));
   text.setBounds(0,27,300,30);
   text.setOpaque(false);
  t.setForeground(new Color(250,0,250));
  t.setLayout(new BorderLayout());
  t.setOpaque(false);
  t.setBounds(0,27,300,30);
  aka.add(Box.createRigidArea(new Dimension(0, 80)));
  ti.add(Box.createRigidArea(new Dimension(10, 0)));
  ti.setOpaque(false);
  ti.add(t);
  aka.add(Box.createVerticalStrut(10));
  aka.add(text);
  aka.add(Box.createVerticalStrut(20));
  aka.setLayout(new BoxLayout(aka, BoxLayout.Y_AXIS));
  aka.setOpaque(false);
  aka.add(en);
  form.add(ti);
  form.add(aka);
        BigInteger p = new BigInteger("2");
        BigInteger q = new BigInteger("3");
        BigInteger r = new BigInteger("5");
        BigInteger s = new BigInteger("7");
        BigInteger n = p.multiply(q).multiply(r).multiply(s);
        BigInteger phi = (p.subtract(BigInteger.ONE))
                .multiply(q.subtract(BigInteger.ONE))
                .multiply(r.subtract(BigInteger.ONE))
                .multiply(s.subtract(BigInteger.ONE));
         while (phi.gcd(m).compareTo(BigInteger.ONE) > 0) {
            m = m.add(BigInteger.ONE);
        }
        BigInteger d = m.modInverse(phi);
        System.out.println("Public Key: (n = " + n + ", e = " + m+ ")");
        System.out.println("Private Key: (n = " + n + ", d = " + d + ")");
en.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    BigInteger m = new BigInteger("11");
String plaintext=text.getText();
System.out.println("PlainText="+plaintext);
int plength=plaintext.length();
BigInteger ascii[]=new BigInteger[plength];
BigInteger encrypted[]=new BigInteger[plength+1];
int remainder= plength%3;
int l;
```

```
if(remainder==1){
 l=plength+2;
}
if(remainder==0){
  l=plength+1;
}
else{
  l=plength+(3-remainder);
}
int r[]=new int[l];
int g[]=new int[l];
int b[]=new int[l];
for(int i=0; i<plength+1;i++){
  if(i==plength){
BigInteger space = new BigInteger("128");
    encrypted[i]=space;
  }
  else{
    ascii[i]= BigInteger.valueOf(plaintext.charAt(i));
    BigInteger encryptedMessage = encrypt(ascii[i], m, n);
    encrypted[i]=encryptedMessage;
  }
  System.out.println("Encrypted Message: " + encrypted[i] + ",");
}
int z=0,x=0,y=0;
int u=0,flag=0;
System.out.print("x=");
System.out.print("y=[");
        if(plength%3==0){
          do{
            if(u<l-1){
              z=encrypted[u].intValue();
            }
            else{
              break;
            }
            if(u+1<l-1){
            x=encrypted[u+1].intValue();
            System.out.print(x+",");
            }
            else{
              x=0;
            }
            if(u+2<l-1){
              y=encrypted[u+2].intValue();
            }
            else{
              y=0;
              }
         r[flag]=z;
          g[flag]=x;
          b[flag]=y;
              u=u+3;
              flag++;
         }while(u<l);
        }
        else{
          int newplength;
if(remainder==1){
newplength=plength+2;}
else{
   newplength= plength + (3-plength%3);}
    do{
      z=encrypted[u].intValue();
     if(u+1<newplength-1){
     x=encrypted[u+1].intValue();
     }
     else{
       x=0;
     }
     if(u+2<newplength-1){
       y=encrypted[u+2].intValue();
       System.out.print(y+",");
     }
     else{
       y=0;
       System.out.print(y+",");
       }
  r[flag]=z;
   g[flag]=x;
  b[flag]=y;
      u=u+3;
      flag++;
}while(u<newplength);
        }
System.out.println(" ");
System.out.print("r=[");
for(int k=0;k<r.length;k++){
System.out.print(r[k]+",");
}
System.out.println("]");
System.out.print("g=[");
for(int k=0;k<r.length;k++){
 System.out.print(g[k]+",");
}
System.out.println("]");
System.out.print("b=[");
for(int k=0;k<r.length;k++){
  System.out.print(b[k]+",");
}
System.out.println("]");
int imglength = r.length;
try{
  BufferedImage img = new BufferedImage(
      imglength,imglength, BufferedImage.TYPE_INT_RGB );
      int name = (int)(Math.random() * 101);
        File myFile = new File(name+"MyFile.png");
        for(int a = 0; a <r.length; a++){
          int rgb =(r[a] << 16) | (g[a] << 8) | (b[a]);
          System.out.println("RGB:"+ rgb);
            for(int c= 0; c <imglength; c++){
              img.setRGB(a, c, rgb);
            }
        }
        ImageIO.write(img, "PNG", myFile);
        System.out.println("File is created.");
      }
      catch(Exception e1){
        System.out.println("Couldn't create sorry!");
      }
```

```
    }
});
   }
    public static void main(String[] args) throws IOException {
      new Main().setVisible(true);
    }
}
```

News Class

```
import javax.swing.*;
import java.awt.*;
public class newS extends JPanel{
public newS(){
    }
protected void paintComponent(Graphics g){
    super.paintComponent(g);
    Graphics2D g2d= (Graphics2D) g;
g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,RenderingHints.VALUE_ANTI
ALIAS_ON);
   int w=getWidth(), h=getHeight();
   GradientPaint gp = new GradientPaint(0, 0, Color.CYAN, 0, 1000, Color.WHITE);
   g2d.setPaint(gp);
   g2d.fillRoundRect(0,0,w,h,40,40);
}
}
```

Decryption for MRSA

```
import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.math.BigInteger;
import javax.imageio.ImageIO;
import javax.swing.JFileChooser;
public class decrypt {

    public static BigInteger decrypt(BigInteger c, BigInteger d, BigInteger n) {
  return c.modPow(d, n);
}

public static void main(String[] args) throws IOException {
    JFileChooser folders= new JFileChooser();
    int select = folders.showOpenDialog(folders);
    File selectedFile = folders.getSelectedFile();
    String filepath = selectedFile.getAbsolutePath();
 BigInteger p = new BigInteger("2");
 BigInteger q = new BigInteger("3");
 BigInteger r = new BigInteger("5");
 BigInteger s = new BigInteger("7");
 BigInteger n = p.multiply(q).multiply(r).multiply(s);
BigInteger phi = (p.subtract(BigInteger.ONE))
        .multiply(q.subtract(BigInteger.ONE))
        .multiply(r.subtract(BigInteger.ONE))
        .multiply(s.subtract(BigInteger.ONE));
        BigInteger m = new BigInteger("11");
  while (phi.gcd(m).compareTo(BigInteger.ONE) > 0) {
      m = m.add(BigInteger.ONE);
 }
 BigInteger d = m.modInverse(phi);

 FileWriter write = new FileWriter("decrypted.text");
       File file = new File(filepath);
       BufferedImage img = ImageIO.read(file);
       int widthimg=img.getWidth();
   int y=0;
           for (int x = 0; x <widthimg; x++) {
             System.out.println("Y="+y+","+"X="+x);
             int pixel = img.getRGB(x,y);
             Color color = new Color(pixel, true);
             int red = color.getRed();
             int green = color.getGreen();
             int blue = color.getBlue();
             BigInteger bigred = BigInteger.valueOf(red);
             BigInteger biggreen = BigInteger.valueOf(green);
             BigInteger bigblue = BigInteger.valueOf(blue);
             System.out.println("R="+red+", G="+green+", B="+blue+",");
             BigInteger rr=decrypt(bigred, d, n);
             BigInteger gg=decrypt(biggreen, d, n);
             BigInteger bb=decrypt(bigblue, d, n);
             System.out.println("rr="+rr+", gg="+gg+", bb="+bb+",");
             int rri= rr.intValue();
             int ggi = gg.intValue();
             int bbi = bb.intValue();
            char rsaR=(char)rri;
            char rsaG=(char)ggi;
            char rsaB=(char)bbi;
            System.out.println("rsaR="+rsaR+", rsaG="+rsaG+", rsaB="+rsaB+",");
              write.append(rsaR);
              write.append(rsaG);
              write.append(rsaB);
              write.flush();
           }
        System.out.println("RGB values are stored in the file.");
    }
}
```