**An Improvised GGH Cryptosystem Implementation on Images through Encoded Strings**

**A Project Report**

***Submitted by***

**YARALA HRUTHIK REDDY – 1700289C203**

**KURUBA KIRAN KUMAR – 1700228C203**

**KEESARA SURYA TEJA GOUD – 1700222C203**

**Batch - 2017 CSE 1**

***Course***

**Cryptography**



**SCHOOL OF ENGINEERING & TECHNOLOGY**

**BML MUNJAL UNIVERSITY**

**April 2020**

***Abstract-*** *Image encryption is already an important part of cloud storage and various applications related to medical, military and any secure image rendering applications. The possibility of quantum computing soon being a reality, many researchers are looking forward to post-quantum cryptography. Lattice-based cryptography is the leading candidate with the Goldreich–Goldwasser–Halevi (GGH) cryptosystem showing a practical alternative implementation. We propose an improvised GGH cryptosystem that works on images through encoding concepts to ensure better security than other ciphers present. Lattices enable for computationally fast cryptosystems and in our results, we were able to contrast the encryption execution times of our algorithm with RSA algorithms encryption times. We coded our algorithm in python programming language with few basic library imports. We were able to achieve strong image encryption through implementing the GGH cryptosystem by encoding the images to a text file of characters. We also performed decryption to show the working of the complete cryptosystem.*

***Keywords-*** *Encoding, GGH Cryptosystem, Image Encryption, Lattice-Based Cryptography, Post-Quantum Cryptography.*

1. **INTRODUCTION**

Images encryption is a crucial part of cryptography and in daily life where millions of images are exchanged every day. Applications such as medical imaging systems, military imaging systems or any image rendering applications require security from being leaked. Maintaining confidentiality of the images is more important than protecting the other two security goals i.e., integrity and availability. Many classical encryption methods are mostly used for text encryption as dealing with images files comes with high computational expense.

A digital image can be represented in various formats, and every image is made up of independent pixels. This enables many algorithms to perform the encryption methods on these pixels. There is a possible chance of information leakage in these methods as the encrypted images often show visual signs of the original image and result in various crypto attacks on the encrypted images. A better way to encrypt images without leaking them visually is to encode them into other formats such as text files. Even if there is leakage in this information, it is difficult to know the original format of the encrypted message.

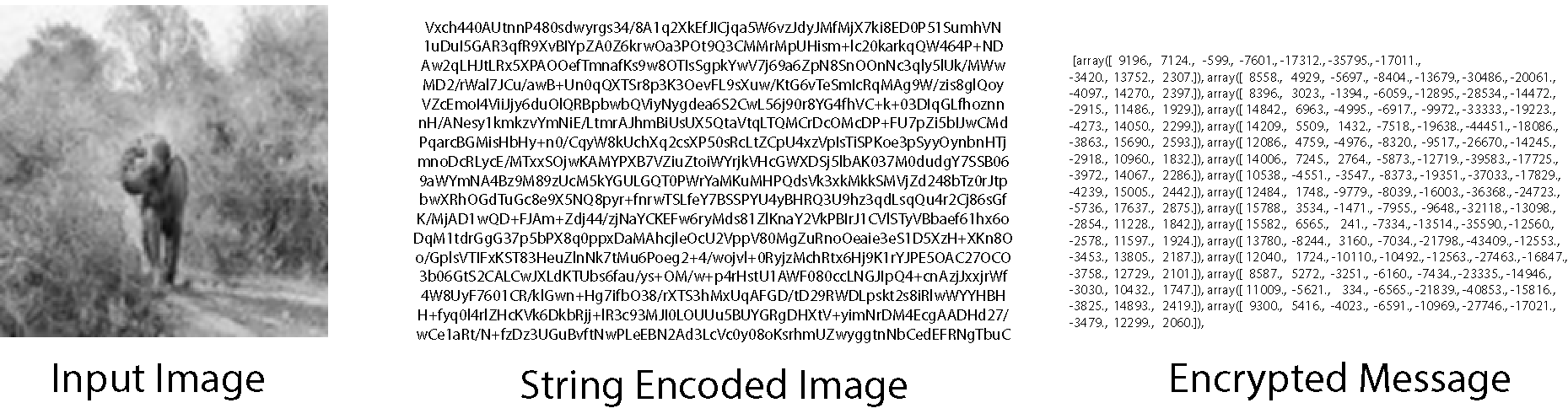


Figure 1. Image at various stages during our proposed encryption.

Also, improvements in technology are coming with its cons. Quantum Computing is going to change a lot of things. One of those is in the field of cryptography. The current public key cryptosystems work based on the number theory problems. The complete security comes from the toughness of these problems. Though there are not many algorithms to solve these tough problems efficiently on classical computers. However, Shor’s algorithm can solve these problems in polynomial time [1]. As soon as these quantum computers become a reality, these public key cryptosystems will no longer be secure enough. Consequently, cybersecurity researchers are looking towards post-quantum cryptography, as it is believed to be even quantum-proof.

Out of all the candidates for post-quantum cryptography, lattice-based cryptography is making a good potential when compared to the rest and the Goldreich–Goldwasser–Halevi (GGH) cryptosystem places an implementable alternative to the number theory. The algorithm GGH employs is a tough one to break even in the average cases making it an extraordinary choice as one of the best lattice-based cryptosystems. GGH cryptosystem makes use of a private key and a public key to encrypt data which makes it a public key cryptosystem. GGH cryptosystem is a good method for encrypting images as its drawback of producing errors during decryption is not significant in the case of images as the errors happen in very few pixels and this change in pixels isn’t visually changing the image [2]. We propose an improvised GGH cryptosystem algorithm to encrypt images through encoded data.

1. **LITERATURE SURVEY**

In the past few years, the world has seen rapid growth of cloud services being provided to them by some of the companies. So, is the amount of digital content produced in the form of images, audio, video, etc. is extremely high. Majority of this digital data contains valuable or even classiﬁed information. Hence, preventing the leakage of information makes an important issue for individuals and organizations [3] [4]. The most secure way to do it is by using encryption methods made especially for images. Generally, encryption algorithms are based on toughest problems. However, one of the domains which offer some of the toughest problems to solve is Lattice Based Cryptography.

The concept of lattices is not something that is new. In 1996, the first lattice-based cryptographic construction was introduced by Ajtai and the security was based on the hardness of some well-studied lattice problems [5]. The good thing about these problems is that the worst-case problems of number theory problems is equivalent to the lattice-based problems of average cases. But it was Goldreich–Goldwasser–Halevi (GGH) cryptosystem that put a practical alternative on the table for number theory [6] [7].

A lattice is generally defined as any grid of points that are regularly spaced and stretched out to infinity. These points are also called vectors. Therefore, an even more technical definition for lattice is as follows, a collection of evenly spaced vectors. Any lattice consists of a special vector known as origin. An origin has all its coordinates set to 0. Lattices are infinitely large entities and computers which technically have limited amounts of memory cannot represent these entities straightforward. So, the concept of basis was introduced to solve this issue. A basis is defined as a small collection of vectors that reproduce any point on the grid that makes a lattice.

In mathematical language, we define a lattice as a discrete subgroup of Ｒ*n* , or the set *L*(*b*1, *b*2, *b*3, ......*bn*) of all linear combination Σ*nibi* where *ni* ε *Z*, and the *bi* ’s are linearly independent vectors over Ｒ, where (*b*1, *b*2, *b*3, ......*bn*) are basis vectors [8].

L = { }



Figure 2 Lattice Representation by basis [x1, x2] and [b1, b2]. [9]

Some lattice problems over which the security has been constructed are:

* Closest-Vector Problem (CVP)
* Shortest-Vector Problem (SVP)
* Shortest Independent Vector Problem (SIVP)
* Shortest Integer Solution Problem (SIS)

The choice of basis changes the difficulty of these problems. Making them easy to solve with a good basis and hard to solve in case of a bad basis. The GGH cryptosystem has been designed to be used with the hard lattice problem to find the closest vector in lattice. The CVP states that for a given vector ‘c’ and a basis of a lattice, identify a vector ‘V∈L’ that is closest to the vector ‘c’ among all possible points of L.

1. **PROPOSED ALGORITHM AND CODE**

We propose a different way to implement GGH cryptosystem by initially encoding the data to achieve multi-level security. The encryption was done on grayscale images which have been encoded into various formats in the process. The flow chart of the process is given below:

A picture containing flower

Description automatically generated

Figure 3. Flowchart of the process involved.

The security of the cryptosystem we propose depends on a single parameter which is the size of the dimensions of the matrices used ‘n’. An optional noise can be added which will increase the security greatly.

**The proposed algorithm:**

* 1. **Key Generation:**

Person-1 starts generating a private key ‘V’, which can be any random integer n-by-n square matrix whose Hadamard Ratio is close to 1.

A public key ‘W’ can be generated by calculating,

Here, ‘U’ is a random integer n-by-n unimodular matrix.

The public key ‘W’ is accepted only if its Hadamard Ratio is close to 0.

* 1. **Encryption:**
     1. Initial Encryption through Encoding:

Image files are encoded into a Base64 string of characters and the Base64 strings into binary data. The binary data is then converted to encrypted integers for the computation purpose.

* + 1. Second Encryption:

The encrypted integers ‘M’ is divided into 1-by-n matrices, and any residue bits are either ignored or saved depending on the security level needed.

All the 1-by-n matrices are iteratively multiplied with the public key ‘W’ and to generate the respective cipher text block ‘C’.

Here, ‘i’ is iteratively increased from 0 to ‘m’, where ‘m’ is the number of 1-by-n matrices. An optional noise ‘E’, a random 1-by-n matrix, can be added for further security.

* 1. **Decryption:**
     1. Initial Decryption:

To decrypt the cipher text ‘C’ we compute,

Here, ‘round()’ function rounds the values to their nearest integer values.

* + 1. Final Decryption through Decoding:

The message is decoded to the Base64 string back which is then converted back to the image file.

**The code for our cryptosystem implementation:**

'''

Authors: Yarala Hruthik Reddy, Kuruba Kiran Kumar, Surya Keesara

Title: GGH Cryptosystem Implementation on Image Files

'''

from PIL import Image

import random

import numpy as np

import base64

import binascii

import os

# This function generates a private and public key

def keyGen(dimension):

    privateKey = []

    print("Searching for a private key")

    ratio = 1

    '''

    while True:

        privateKey = np.random.randint(-10, 10, size=(dimension,dimension))

        ratio = hadamardRatio(privateKey, dimension)

        if(.9 <= ratio <= 1):

            print(privateKey)

            break

    '''

    privateKey = np.identity(dimension)

    print(privateKey)

    print("Searching for a public key")

    while True:

        uniMod = randUniMod(dimension)

        temp = np.matmul(uniMod, privateKey)

        ratio = hadamardRatio(temp, dimension)

        if ratio <= .1:

            publicKey = temp

            break

    print(publicKey)

    return privateKey, publicKey, uniMod

# This function returns the Hadamard Ratio of a matrix

def hadamardRatio(matrix, dimension):

    detOfLattice = np.linalg.det(matrix)

    detOfLattice = detOfLattice if detOfLattice > 0 else -detOfLattice

    mult = 1

    for v in matrix:

        mult = mult \* np.linalg.norm(v)

    hadRatio = (detOfLattice / mult) \*\* (1.0/dimension)

    return hadRatio

# This function returns a Random Unimodular matrix

def randUniMod(dimension):

    random\_matrix = [[np.random.randint(-10, 10,)

                      for \_ in range(dimension)] for \_ in range(dimension)]

    upperTri = np.triu(random\_matrix, 0)

    lowerTri = [[np.random.randint(-10, 10) if x <

                 y else 0 for x in range(dimension)] for y in range(dimension)]

    #Creating an upper trianglular and lower triangular matrices with diagonals as +1 or -1

    for r in range(len(upperTri)):

        for c in range(len(upperTri)):

            if(r == c):

                if bool(random.getrandbits(1)):

                    upperTri[r][c] = 1

                    lowerTri[r][c] = 1

                else:

                    upperTri[r][c] = -1

                    lowerTri[r][c] = -1

    uniModular = np.matmul(upperTri, lowerTri)

    return uniModular

# Converts images to black and white

def black\_and\_white(input\_image\_path, output\_image\_path):

   color\_image = Image.open(input\_image\_path)

   bw = color\_image.convert('L')

   bw.save(output\_image\_path)

# Loads images and copies the encoded image in a string

def loadImage(file\_path):

    with open(file\_path, "rb") as img\_file:

        my\_string = base64.b64encode(img\_file.read())

    print("Image Loaded")

    return my\_string

# Writes the encoded string to a file

def writeImageToString(filePath, string):

    newString = string.decode("utf-8")

    file = open(filePath, "w")

    file.write(str(newString))

    file.close()

# String to Image Converter

def writeStringToImage(filePath, imagePath):

    with open(filePath, "r") as f:

        string = f.read().replace("\n", "")

    imageData = base64.b64decode(string)

    with open(imagePath, "wb") as f:

        f.write(imageData)

# Writes text files

def writeTextBlocks(filePath, string):

    file = open(filePath, "w")

    file.write(str(string))

    file.close()

# Encoded text to Encrypted Ints

def encodedToBinaryToEncrypted(seekVal):

    encoded = []

    with open("ImageString\\imageString.txt", 'r') as f:

        encText = f.seek(seekVal)

        encText = f.read(10)

        for c in encText:

            encoded.append(base64.b64encode(bytes(c, "utf-8")))

    binaryMessage = []

    for i in range(len(encoded)):

        binaryMessage.append(binascii.a2b\_base64(encoded[i]))

    encryptedInts = []

    for i in range(len(encoded)):

        encryptedInts.append(int.from\_bytes(binaryMessage[i], byteorder='little'))

    return encryptedInts

# Encryption Function

def encrypt(encryptedInts, publicKey):

    cypherText = []

    for i in range(len(encryptedInts)):

        cypherText = np.matmul(encryptedInts, publicKey)

    return cypherText

# Decryption Function

def decrypt(cypherText, privateKey, uniModular):

    A = privateKey

    x = cypherText

    BPRIME = np.linalg.inv(A)

    BB = np.matmul(BPRIME, x)

    uniModularInv = np.linalg.inv(uniModular)

    m = np.round(np.matmul(BB, uniModularInv)).astype(int)

    return m

# Misc Methods

# Writes decrypted message to a file

def writeDecryptedMessage(filePath, message):

    file = open(filePath, "a+")

    for i in range(len(message)):

        letter = chr(abs(message[i]))

        file.write(letter)

    file.close()

'''

# Writes encrypted message to a file

def showEncryptedMessage(filePath, message):

    file = open(filePath, "a+")

    for i in range(len(message)):

        letter = chr(abs(int(message[i])))

        file.write(letter)

    file.close()

'''

# Writes Public key to a file

def writePublicKey(publicKey):

    file = open('PublicKey\\ggh\_block.txt', 'w')

    file.write("------BEGIN GGH PUBLIC KEY BLOCK -----\n")

    for row in range(len(publicKey)):

            for col in range(len(publicKey)):

                encoded = base64.b64encode(publicKey[row][col])

                file.write(str(encoded)[8:13])

            file.write("\n")

    file.write("\n--------END GGH PUBLIC KEY BLOCK -------")

    file.close()

# Writes the residue to the final message

def residueAdder(filePath, residueString):

    file = open(filePath, "a")

    for i in range(len(residueString)):

        letter = residueString[i]

        file.write(letter)

    file.close()

# Main Function

def main():

    dirs = ['BWImages', 'Decrypted', 'Encrypted', 'ImageString', 'PublicKey', 'Residue']

    for i in dirs:

        if not os.path.exists(i):

            os.mkdir(i)

            print("Directory ", i,  " Created ")

        else:

            print("Directory ", i,  " already exists")

    alice = keyGen(10)

    writePublicKey(alice[1])

    cypherTextFile = []

    inputFileName = 'InputImages\\' + input("Enter Input file name with relative path (InputImages\\fileName.jpg or .png): ")

    OutputMessageFile = 'Decrypted\\' + input("Enter the file name with relative path (Decrypted\\fileName.txt) to output the decrypted message to: ")

    black\_and\_white(inputFileName, 'BWImages\\bwImage.jpg')

    stringEncoded = loadImage("BWImages\\bwImage.jpg")

    writeImageToString("ImageString\\imageString.txt", stringEncoded)

    print(len(stringEncoded))

    print("\n-------------------------Encrypting------------------------\n")

    seekVal = 0

    while True:

        encryptedInts = encodedToBinaryToEncrypted(seekVal)

        if (seekVal > len(stringEncoded) - 10):

            residueString = stringEncoded[-(len(stringEncoded) - seekVal):]

            break

        else:

            seekVal = seekVal + 10

        bob = encrypt(encryptedInts, alice[1])

        cypherTextFile.append(bob)

    residueString = residueString.decode("utf-8")

    writeTextBlocks("Encrypted\\cypherTextFile.txt", cypherTextFile)

    writeTextBlocks("Residue\\residueText.txt", residueString)

    '''

    print("\n----------------Encrypted Message Print-------------------\n")

    seekVal = 0

    while True:

        if (seekVal >= len(cypherTextFile)):

            break

        else:

            showEncryptedMessage("Encrypted\\encryptedMessage.txt", cypherTextFile[seekVal])

        seekVal = seekVal + 1

    residueAdder("Encrypted\\encryptedMessage.txt", residueString)

    writeStringToImage("Encrypted\\encryptedMessage.txt", "Encrypted\\encryptedImage.jpg")

    '''

    print("\n------------------------Decrypting---------------------------\n")

    seekVal = 0

    decryptedTextFile = []

    while True:

        if (seekVal >= len(cypherTextFile)):

            break

        else:

            aliceReceives = decrypt(cypherTextFile[seekVal], alice[0], alice[2])

            decryptedTextFile.append(aliceReceives)

        seekVal = seekVal + 1

    writeTextBlocks("Decrypted\\decryptedTextFile.txt", decryptedTextFile)

    print("\n---------------Decrypted Message Printing-------------------\n")

    seekVal = 0

    while True:

        if (seekVal >= len(decryptedTextFile)):

            break

        else:

            writeDecryptedMessage(OutputMessageFile, decryptedTextFile[seekVal])

        seekVal = seekVal + 1

    residueAdder("Decrypted\\decryptedMessage.txt", residueString)

    writeStringToImage(OutputMessageFile, "Decrypted\\decryptedImage.jpg")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Libraries:**

Image library to open and load images. Random library to generate random numbers. NumPy library for linear algebra calculations such as matrix multiplications. Base64 library for encoding and decoding strings. Binascii library to convert characters to integers. OS library for directory and file path handling.

**Key Generation:**

There was an attempt to generate a nearly orthogonal basis as a private key (Commented Code in the **keyGen()** function) but the success rate of decrypting back the original message was around 40% due to floating point computation errors. Instead, we chose a perfectly orthogonal basis i.e., Identity matrix of respective dimension as the private key. The public key was generated by multiplying a random unimodular matrix with the private key. The public key is accepted only if its Hadamard Ratio is close to zero or if the public key is a nearly parallel basis. The function returns the private key, the public key and the unimodular matrix used.

**Image Handling:**

Any input images were converted to grayscale to reduce the size of the image, **black\_and\_white()** function does this operation. The grayscale images were then read and encoded into a string, **loadImage()** function does this operation.

**Encryption:**

The encoded text was converted to encrypted integers by through ASCII conversions, **encodedToBinaryToEncrypted()** function achieves this operation. The encrypted integers were then sent to the **encrypt()** function to generate the final cypher text.

**Decryption:**

The cypher text along with the private key and the unimodular matrix were sent to the **decrypt()** function. The inverses of the unimodular matrix and the private key are generated, and the inverse of the private key was multiplied with the cypher text. This was then multiplied with the inverse of unimodular matrix to get back the original matrix.

**Helper Functions:**

**hadamardRatio()** and **randUniMod()** functions are the helper functions for the **keyGen()** function. **writeImageToString()** function writes the image string to a file and the **writeStringToImage()** function writes the string back to the image. **writeTextBlocks()** functions writes the encrypted and decrypted integer messages to text files. **writeDecryptedMessage()** function converts the decrypted integers into a string and writes them in a text file. **writePublicKey()** function writes the public key to a text file. **residueAdder()** function adds the residue characters back to the original message.

**NOTE:**

* Image input should be of size of 100 width and 100 height. Only grayscale encryption and decryption are done. Could be extended for colour images too but will be computationally heavy.
* Image string was broken down into individual strings of 10 characters each as the key dimensions were also 10 in this case. Higher keys and higher individual length of strings can be used but it is computationally heavy. We found 10 sized strings and 10 sized keys to be perfect for our computations.
* The residueAdder() is an optional function to take care of the residue bits, but the loss of those bits will not impact the image significantly.
* We tried to visually show the encrypted message as an image file but the mapping of the encrypted message integers to the respective ASCII characters is not possible in most of the cases and results in errors rendering the encrypted image.

1. **SOFTWARE AND HARDWARE ENVIRONMENT**

We used the cloud environment of Google Colab for all our final code executions.

|  |  |  |
| --- | --- | --- |
| **EXPERIMENTAL ENVIRONMENTS** | | |
| Hardware | CPU  MEMORY  GRAPHICS CARD  HARD DISK | Intel(R) Xeon(R) CPU @ 2.30GHz  12.72 GB  Tesla K80 GPU  68.40 GB |
| Software | OPERATING SYSTEM | Windows 10 Pro-64bit |
| GOOGLE COLAB |  |
| Algorithms | GGH CRYPTOSYSTEM | |
| RSA | |

1. **RESULT ANALYSIS**

The proposed GGH cryptosystem of image encryption and decryption was implemented in Python. We used various encoding libraries like: base64, binascii, and many more. We found a trend that as the image size increases, more is the time taken to encrypt and decrypt. Generally, GGH cryptosystem’s time complexity is of NP-hard problems. But the probable time complexity of the algorithm if both the private key and public key is known is O(n), where ‘n’ is the length of encoded string. So, if the image size is going to increase more is the time taken to perform base64 encoding. Thereafter, more will be the time taken to extract integers from the base64 encoding. We ran our proposed algorithm against RSA algorithm for the same input images and have noted the computational times of both the algorithms. The following tables presents some of our executions of various grayscale images:

Table 1. Our Proposed GGH Cryptosystem Code Results

|  |  |  |  |
| --- | --- | --- | --- |
| **Image Size**  **(in KB)** | **Encryption Time**  **(in Seconds)** | **Decryption Time**  **(in Seconds)** | **Total time**  **(in Seconds)** |
| elephant.jpg (6) | 0.34805750846 | 0.741650581359 | 1.0903983116 |
| cat.jpg (6) | 0.76420259475 | 1.550694227218 | 2.3156971931 |
| tulip.jpg (5) | 0.34034093856 | 0.723017215728 | 1.0633581542 |
| roseflower.jpg (91) | 3.37726140022 | 7.980527639389 | 11.358075618 |

Figure 4. Encryption Time vs Size Graph of our proposed GGH cryptosystem

Figure 5. Decryption Time vs Size Graph of our proposed GGH cryptosystem

Figure 6. Total Time (Encryption and Decryption) vs Size Graph of our proposed GGH cryptosystem

Table 2. RSA Algorithm Results on the Same Images

|  |  |  |  |
| --- | --- | --- | --- |
| **Image Size**  **(KB)** | **Encryption Time**  **(in sec)** | **Decryption Time**  **(in sec)** | **Total time**  **(in sec)** |
| elephant.jpg (6) | 1.92438197 | 0.097889184 | 2.0226073 |
| cat.jpg (6) | 4.12467908 | 0.765068054 | 4.8897950 |
| tulip.jpg (5) | 2.67688369 | 0.112004756 | 2.7889254 |
| roseflower.jpg (91) | 6.30204105 | 3.767927408 | 10.070017 |

Figure 7. Encryption Time vs Size Graph of the RSA algorithm.

Figure 8. Decryption Time vs Size Graph of the RSA algorithm

Figure 9. Total Time (Encryption and Decryption) vs Size Graph of the RSA algorithm

Figure 10. Comparing encryption graphs together.

Figure 11. Comparing decryption graphs together.

Figure 12. Comparing both the systems together.

Here, we can see that time taken by RSA to encrypt is exceptionally long when compared to our GGH cryptosystem. Decryption times were faster in RSA relatively to our GGH cryptosystem. So, our proposed GGH cryptosystem is a decent choice for encryption and decryption when compared to the other present encryption algorithms.

1. **CONCLUSION AND FUTURE SCOPE**

We implemented the new GGH cryptosystem by encoding the input grayscale images into strings and integers to ensure better security when sending encrypted files over the network. It is difficult to know the original format of the message to decrypt it though brute force attempt over common file formats might reveal the message. Still breaking the secondary encryption of the GGH cryptosystem is not a computationally easy task. A constraint of our model was in the generation of the private key. We had to choose a perfectly orthogonal basis as the private key which is not necessarily required. The success rate of our model when we chose a nearly orthogonal basis as the private key was in the range of 50% to 70%. We implemented the cryptosystem on grayscale images of size 100-by-100 without any problems. The model could be extended to work on colored images and other media formats. The code can also be improved with the usage of better libraries or a more suited programming language.

# References

|  |  |
| --- | --- |
| [1] | P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," *Proceedings 35th Annual Symposium on Foundations of Computer Science,* 1994. |
| [2] | M. Sokouti, A. Zakerolhosseini and B. Sokouti, "Medical Image Encryption: An Application for Improved Padding Based GGH Encryption Algorithm," *The Open Medical Informatics Journal,* vol. 10, 2016. |
| [3] | M. Zanin and A. N. Pisarchik, "Gray code permutation algorithm for high-dimensional data encryption," *Elsevier Information Sciences,* vol. 270, pp. 288-297, 2014. |
| [4] | Zhang, Ying-Qian, Wang and Xing-Yuan, "A symmetric image encryption algorithm based on mixed linear–nonlinear coupled map lattice," *Elsevier Information Sciences,* vol. 273, pp. 329-351, 2014. |
| [5] | M. Ajtai, "Generating hard instances of lattice problems," *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing,* 1996. |
| [6] | O. Goldreich, S. Goldwasser and S. Halevi, "Public-key cryptosystems from lattice reduction problems," in *CRYPTO ’97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, London: Springer-Verlag, 1997. |
| [7] | H. Nejatollahi, N. Dutt and R. Cammarota, "Special Session: Trends, Challenges and Needs," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2017. |
| [8] | P. Q. Nguyen and J. Stern, "The Two Faces of Lattices in Cryptology," in *In International Cryptography and Lattices Conference*, 2001. |
| [9] | P. K. Pradhan, S. Rakshit and S. Datta, "LATTICE BASED CRYPTOGRAPHY: Its Applications, Areas of Interest & Future Scope.," in *Third International Conference on Computing Methodologies and Communication (ICCMC)*, 2019. |