Collabits Journal
Vol 1 No. 1 January 2024 : 55-61
E-ISSN : 3046-6709
https://publikasi.mercubuana.ac.id/index.php/collabits

The Implementation of the Fibonacci Encryption Algorithm for Image Security

Mohamad Yusuf[1*], Sebastianus Lukito[2], Ridho Pangestu[3]
[1,2,3] Jurusan Teknik Informatika Universitas Mercu Buana, Indonesia

*Coressponden Author: mhd.yusuf@mercubuana.ac.id

Abstract - *With the advancement of digital technology, information security, especially in the context of digital images, has become a primary concern in various sectors. This paper proposes an innovative image encryption method using the Fibonacci sequence, an approach that has been relatively unexplored in image security. The Fibonacci sequence, known for its unique mathematical properties, is integrated into the encryption process to enhance the security of images. We developed an algorithm that converts images into NumPy arrays and then applies encryption operations to each pixel based on the values in the Fibonacci sequence. This process not only provides protection against manipulation and unauthorized access but also preserves the visual integrity of the image. The decryption algorithm we designed allows for the recovery of the original image without data loss. Security analysis indicates that this method offers significant resistance to various cryptographic attacks while providing efficiency in terms of computation and storage. This research paves the way for new developments in digital image security, offering an adaptable method for various applications, ranging from safeguarding personal data to securing sensitive information in industrial and governmental sectors.*

Keywords :
Cryptography;
Compression;
BiLSTM;
Fibonacci;
Text File;

Article History:
Received: 28-11-2023
Revised: 26-12-2023
Accepted: 12-01-2024

Article DOI : 10.22441/collabits.v1i1.25424

1. INTRODUCTION

In the current digital era, information security has become one of the primary challenges in information and communication technology. Specifically, the security of digital images is crucial considering their significant role in various fields such as media, entertainment, communication, and security surveillance. Digital images often contain sensitive information, ranging from personal data to business secrets, making the protection against unauthorized access and manipulation a top priority.

However, with the increasing capabilities of software and code-breaking algorithms, traditional encryption methods are becoming more vulnerable. Therefore, this research is aimed at developing a stronger and more efficient image encryption method. We propose the use of the Fibonacci sequence, known for its unique mathematical properties, as the basis for a new image encryption algorithm. This approach is expected not only to enhance image security but also to preserve its quality and visual integrity. The research briefly reviews existing image encryption techniques and explores their potential and limitations. We focus on developing a method that can address the weaknesses of previous methods, particularly in terms of resistance to cryptographic attacks and computational efficiency.

Additionally, the research discusses the importance of digital image security in a broader context, including legal, ethical, and social aspects related to data protection and privacy.

2. METHODOLOGY

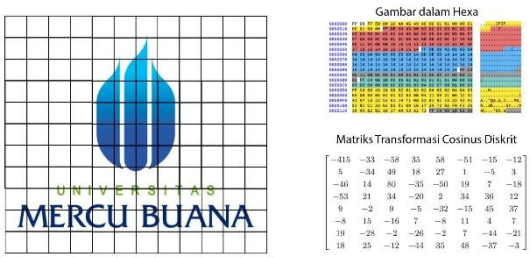
This research is designed to develop and test a Fibonacci sequence-based image encryption algorithm. The process is divided into several main stages:

2.1. Data Collection and Preparation:

2.1.1. Image Selection: We selected a number of digital images from various genres and resolutions to test the algorithm. These images include various types such as portraits, landscapes, and abstract images to ensure diversity in testing.

2.1.2. Pre-processing: These images are then converted into appropriate formats (such as PNG or JPEG) and transformed into NumPy arrays for ease of data manipulation.

Figure 2.1. Convert JPEG into Array



Converting images to arrays with NumPy is the process of transforming visual image data into a numerical data structure that can be further processed by a computer. This process generally involves the following steps:

a. Image Reading: Images are read from files using modules such as PIL or OpenCV, which are common libraries in image processing in Python.
b. Conversion to Array: Once the image is opened, it is transformed into a NumPy array. In the Python context, this NumPy array effectively represents the image as a two-dimensional matrix (for black and white images) or a three-dimensional matrix (for color images), where each matrix element represents a pixel.
c. Array Structure: For color images, the array structure is usually (height, width, channels) where 'height' is the number of vertical pixels, 'width' is the number of horizontal pixels, and 'channels' represents color components (e.g., RGB with three channels).
d. Pixel Values: Each pixel in a color image is typically represented by three values (in the case of RGB, each for red, green, and blue), each in the range of 0-255. In a NumPy array, these values are stored as 8-bit integers.

Figure 2.1 shows the pixel value representation of an image in hexadecimal format. This is an intermediate step in the conversion to an array, where pixel values are converted to hexadecimal for analysis or further processing.

The Discrete Cosine Transform (DCT) matrix at the bottom right displays a matrix that appears to be the result of a discrete cosine transform (DCT). DCT is often used in image compression (such as JPEG) because it is effective in representing signals in lower frequencies, which is useful in data compression.

2.2. Development of Encryption Algorithm:

2.2.1. Implementation of Fibonacci Sequence: This sequence starts with two numbers, typically 0 and 1, and each subsequent number is the sum of the two preceding ones. Thus, the Fibonacci sequence begins like this: 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on. Each number in the Fibonacci sequence is called a "Fibonacci number."

One characteristic feature of the Fibonacci sequence is the golden ratio, which emerges when comparing two consecutive numbers in the sequence. This ratio approaches 1.618 and is considered to have aesthetically pleasing proportions in art and architecture.

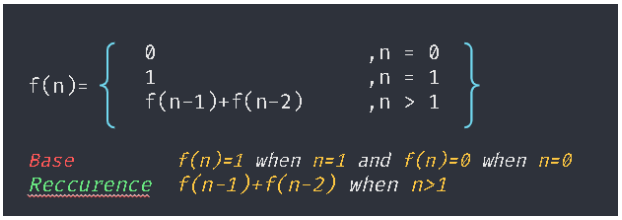
The Fibonacci sequence is a series of numbers with the first 20 numbers as follows:

Fig 2.2. The First 20 Numbers of Fibonacci Sequence

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

The recursive definition of the Fibonacci function can be stated as follows:

Figure 2.3. Recursive function of Fibonacci Sequence



The Fibonacci sequence is defined using a function that calls itself to calculate the next value in the sequence.

Figure 2.4. Recursive Function of Fibonacci Sequence

```
FUNCTION Fibonacci(n)
    IF n <= 0
        RETURN 0
    ELSE IF n == 1
        RETURN 1
    ELSE
        RETURN Fibonacci(n-1) + Fibonacci(n-2)
END FUNCTION
```

Meanwhile, the implementation using the iterative concept is as follows:

Figure 2.5. Iterative Function of Fibonacci Sequence

```
FUNCTION FibonacciIterative(n)
    IF n <= 0
        RETURN 0
    ELSE IF n == 1
        RETURN 1
    ENDIF

    previous = 0
    current = 1

    FOR i FROM 2 TO n
        next = previous + current
        previous = current
        current = next
    ENDFOR

    RETURN current
END FUNCTION
```

This algorithm uses an array as a storage for values (accumulator). Consequently, this algorithm

consumes a significant amount of memory space. By comparing the algorithm implementations between using recursive and iterative concepts, it can be observed that the implementation using the recursive concept is much easier to understand and create. This is because the implemented function closely follows the definition of the recursive function.

Furthermore, in the version using the iterative concept, an additional variable is used for storing values. This variable also needs to be initialized first. This implies additional memory usage and processing on the computer. However, the recursive process also requires a considerable amount of memory. This is due to the repetitive process that seemingly continuously stores values on a stack that doesn't disappear.

2.2.2. Encryption Algorithm: Applying an encryption algorithm to the image. This involves adding the value of each pixel with the corresponding Fibonacci number, followed by a modulo 256 operation to ensure that the pixel values remain valid.

Figure 2.6. Encryption Algorithm

```
FUNCTION DecryptImageWithFibonacci(image_array, fibonacci_sequence, width, height)
    FOR i FROM 0 TO height - 1
        FOR j FROM 0 TO width - 1
            pixel_value = image_array[i, j]

            fibonacci_index = i * width + j

            IF fibonacci_index < LENGTH(fibonacci_sequence)
                decrypted_pixel_value = (pixel_value -
fibonacci_sequence[fibonacci_index]) % 256
                image_array[i, j] = decrypted_pixel_value
            ELSE
                PRINT "Index out of bounds for Fibonacci sequence."
            ENDIF
        ENDFOR
    ENDFOR
END FUNCTION
```

```
FUNCTION ModifyImageWithFibonacci(image_array, fibonacci_sequence, width, height)
    FOR i FROM 0 TO height - 1
        FOR j FROM 0 TO width - 1
            pixel_value = image_array[i, j]

            fibonacci_index = i * width + j

            IF fibonacci_index < LENGTH(fibonacci_sequence)
                new_pixel_value = (pixel_value + fibonacci_sequence[fibonacci_index]) % 256
                image_array[i, j] = new_pixel_value
            ELSE
                PRINT "Index out of bounds for Fibonacci sequence."
            ENDIF
        ENDFOR
    ENDFOR
END FUNCTION
```

2.3.3. Decryption Algorithm: Reversing the encryption process to retrieve the original image. This process involves subtracting the encrypted pixel values with the same Fibonacci numbers, followed by a modulo 256 operation.

Figure 2.7. Decryption Algorithm

```
new_pixel_value = pixel_value +
fibonacci_sequence[i×width+j]mod256
```

3. ANALYSIS AND DISCUSSION

Before undertaking the process of implementing methods or algorithms in problem-solving, the first thing to do is to analyze so that the issues of encryption, reducing size, and decrypting images can be elaborated back to their original form.

JPEG (Joint Photographic Experts Group) is a widely used image compression standard renowned for its efficient reduction of file sizes while preserving acceptable image quality through a lossy compression technique. When delving into the internals of a JPEG file, it becomes apparent that hexadecimal values play a crucial role in representing various elements within the image data. At the onset of a JPEG file, a header containing specific markers is present, and these markers are denoted by hexadecimal values. These markers convey vital information about the image format, compression parameters, and other essential details, setting the stage for the subsequent data within the file.

Quantization tables, which dictate how the frequency components of the image are quantized, are integral to the JPEG compression process. Represented in hexadecimal, these tables significantly influence the level of compression and, consequently, impact the final image quality. Huffman tables, also expressed in hexadecimal, play a complementary role by defining coding schemes for entropy encoding and decoding image data. Huffman coding optimizes the representation of frequent values with shorter codes, contributing to effective compression.

The bulk of the JPEG file consists of compressed image data, where runs of varying lengths are encoded. Within this data, markers and codes are often represented in hexadecimal. These markers delineate different segments of the image, while the codes encapsulate compressed information critical for accurate image reconstruction. Towards the end of the JPEG file, an End of Image (EOI) marker is encountered. This marker, expressed as a specific hexadecimal value, signals the conclusion of the image data. Understanding and interpreting these hexadecimal values are essential for working with or analyzing JPEG files, providing insights into the compressed image's structure, settings, and parameters that are crucial for rendering the image correctly. Importantly, the hexadecimal representation serves as a human-readable format for binary data within the JPEG file.

To transform a hexadecimal value into an array mathematically, the following steps can be followed. First, prepare the hexadecimal value that needs to be converted. Next, convert each hexadecimal digit into its corresponding decimal value, keeping in mind that each hexadecimal digit represents four bits. Then, convert the decimal value into a binary representation using division and remainder operations. Afterward, organize the binary bit groups into an array, taking into account the required number of bits, such as a byte. Convert each binary bit group back into its corresponding decimal value. Finally, arrange these decimal values into an array according to the desired order.

For example, if we have the hexadecimal value 1A.

Hexadecimal (often referred to as "hex") is a numeral system that uses base 16 to express values. This system provides a more concise representation than the decimal (base 10) system in the context of computing and computer programming. Hexadecimal consists of 16 symbols, namely 0-9 and A-F (A, B, C, D, E, F), where A to F represent decimal values 10 to 15.

Table 3.1. Table of Decimal Converted into Binary and Hexadecimal

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

While binary is a numerical system at the core of computing, characterized by its use of only two digits, 0 and 1. In this base-2 system, each individual digit is known as a "bit," representing the fundamental unit of information. The binary system operates on a positional structure, where the value of each digit is determined by its position, corresponding to powers of 2. For instance, the rightmost bit signifies 2^0, the next bit to the left signifies 2^1, and so forth. Conversion from binary to decimal involves summing the products of each binary digit with 2 raised to the power of its position. This system is integral to computer architecture, where binary code is employed at the hardware level for the representation and manipulation of data. All information, including executable programs, images, and text, is ultimately stored and processed in binary form. In the context of binary arithmetic, addition and subtraction follow rules similar to decimal arithmetic but with simpler carry and borrow operations, while multiplication and division involve operations based on powers of 2. Denoted with a subscript "2," binary serves as the foundation of computer science, digital communication, and information theory, and a comprehensive understanding of binary is essential for those engaged in computer programming, digital electronics, and data representation.

Here are the the step by step process to convert a hexadecimal value to binary mathematically, using the example hexadecimal value 1A:

A. Prepare the Hexadecimal Value:

   Take the hexadecimal value you want to convert to binary. For example, let's take 1A.

B. Convert Each Digit to Decimal:

   Convert each hexadecimal digit to its corresponding decimal value. 1A (hex) = 1 * 16 + 10 * 1 (decimal) = 26.

C. Convert Decimal to Binary:

   Convert the calculated decimal value to binary. For example, 26 (decimal) = 11010 (binary).

D. Group the Bits:

   Separate the binary representation into the appropriate bit groups. In this case, 11010 is already a single group.

By following these steps, we have successfully converted the hexadecimal value 1A to the binary representation 11010 mathematically. And then we need to convert the binary into decimal array, Here are the steps to convert a binary value into an array:

A. Prepare the Binary Value:

   Take the binary value you want to convert into an array. In our example, we have 11010.

B. Group Bits into an Array:

   Separate the binary representation into appropriate bit groups. For instance, for the binary value 11010, one bit group can be represented in one array element.

C. Convert Binary to Decimal (Optional):

   If needed, convert each binary bit group to its corresponding decimal value. In this example, we already have the decimal value 26.

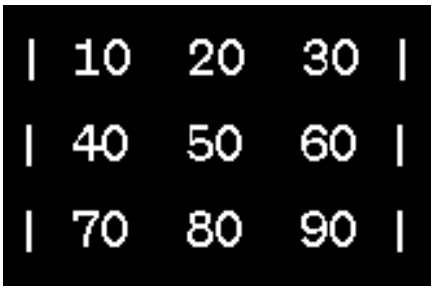
D. Organize the Array:

   Arrange the decimal or binary values into an array according to the desired order. In this case, the array would be [26] or ['11010'].

By following these steps, we have successfully converted the binary value 11010 into the array [26] or ['11010']. And then let's delve into the manual encryption process in more detail. First, we need to obtain the dimensions of the image (height and width). Calculate the total number of pixels in the image by multiplying the height and width. Then iterate through each pixel to extract the original pixel value.

For example, let's assume a small 3x3 image with pixel values:

Figure 3.1. 3 x 3 example of pixel values of an image



Let's calculate the new pixel values manually for each pixel in a small 3x3 image using the encryption formula:

And let's consider a simplified Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, ...]. Now, let's calculate the new pixel values for each pixel:

A. For the top-left pixel at (0,0) with value 10:

new_pixel_value = 10 + 0 mod 256 = 10

B. For the pixel at (0,1) with value 20:

new_pixel_value = 20 + 1 mod 256 = 21

C. For the pixel at (0,2) with value 30:

new_pixel_value = 30 + 1 mod 256 = 31

D. For the pixel at (1,0) with value 40:

new_pixel_value = 40 + 1 mod 256 = 41

E. For the pixel at (1,1) with value 50:

new_pixel_value = 50 + 2 mod 256 = 52

F. For the pixel at (1,2) with value 60:

new_pixel_value = 60 + 3 mod 256 = 63

G. For the pixel at (2,0) with value 70:

new_pixel_value = 70 + 5 mod 256 = 75

H. For the pixel at (2,1) with value 80:

new_pixel_value = 80 + 8 mod 256 = 88

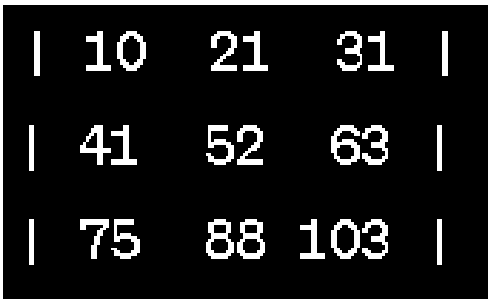
I. For the pixel at (2,2) with value 90:

new_pixel_value = 90 + 13 mod 256 = 103

Now, replace the original pixel values with these newly calculated values in the image array. Repeat this process for each pixel in the entire image. By manually performing these steps for each pixel in the image, you replicate the encryption process described in the Python code. The key is to follow the mathematical operations and maintain the modulo operation to ensure the resulting pixel values are within the valid range (0 to 255).

To reconstruct the new image after encrypting each pixel, use the calculated new pixel values to create a new image array. Let's organize the new pixel values into a 3x3 image:

Figure 3.2. 3 x 3 of pixel values of an image after encryption



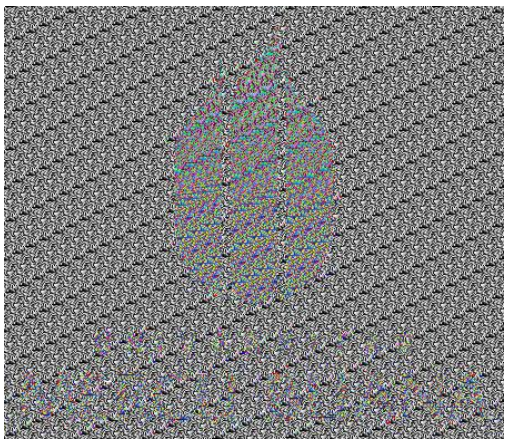
These values represent the encrypted image. You can use this array to visualize the new image, where each pixel has been modified according to the encryption process described earlier.

In a real-world scenario, if we are working with a larger image, we would repeat the encryption process for each pixel in the image, and the resulting array would represent the encrypted version of the entire image. The structure of the image array (height, width) would remain the same, and you would replace the original pixel values with their respective new values after encryption. In this research image example, we use a picture of the Mercu Buana University logo.

Figure 3.3. A Picture of Mercu Buana University logo that's gonna be encrypted

The encryption process begins with obtaining the digital representation of the logo. This image, composed of pixels and color values, serves as the foundation for the encryption procedure. The next step involves converting the image into a numerical array, where each pixel is represented by specific numerical values corresponding to its color. The encryption algorithm is then applied to each pixel in the array, following the principles outlined earlier. This algorithm entails adding a corresponding Fibonacci value to each pixel's original value and performing a modulo operation to ensure the new values remain within the valid range. The result is a new numerical array, representing the encrypted version of the Mercu Buana University logo. To visualize the encrypted logo, the array is transformed back into an image format, considering the original dimensions and color representation.

Figure 3.4. A Picture of Mercu Buana University logo after encryption



When we discuss the color information of an image, particularly in the context of digital imagery, colors are often represented using the RGB (Red, Green, Blue) color model. In this model, each pixel's color is described by specifying the intensity of the three primary colors: red, green, and blue. These intensities are usually expressed as numerical values ranging from 0 to 255, with 0 indicating no intensity and 255 representing full intensity. To represent these color values in a more human-readable and concise format, hexadecimal notation is commonly used. Hexadecimal is a base-16 numbering system that uses the digits 0-9 and the letters A-F to represent values from 0 to 15. In the context of RGB color representation, each of the three color channels (red, green, and blue) is assigned a two-digit hexadecimal value. For example, the notation #RRGGBB is often used, where RR represents the hexadecimal value for red, GG for green, and BB for blue.

In this specific scenario with the Mercu Buana University logo, when we refer to the handwritten hexadecimal representation, it implies that each pixel in the logo is being manually transcribed onto paper in a format similar to #RRGGBB. For instance, if a pixel has a red intensity of 120, green intensity of 200, and blue intensity of 50, it might be represented as #78C832 in hexadecimal. This notation is a concise yet comprehensive way to represent colors, and it serves as the starting point for the encryption process. The manual transcription of each pixel's color information onto paper ensures that the unique characteristics of the Mercu Buana University logo are accurately preserved in the form of hexadecimal values before the encryption algorithm is applied.

This is the hexadecimal values of the logo before encryption:

Figure 3.5. Hexadecimal Values of Mercu Buana University Logo Before Encryption

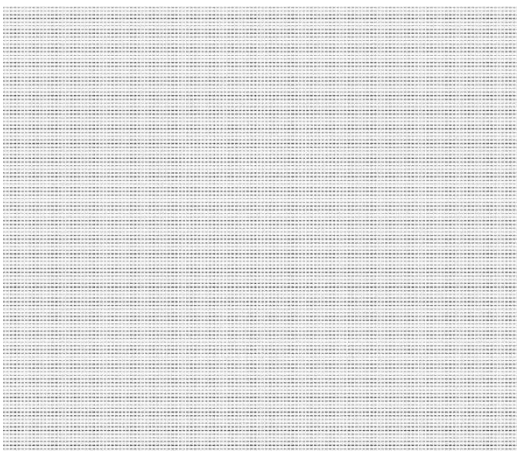


Figure 3.6. Hexadecimal Values of Mercu Buana University Logo Before Encryption Zoomed In

0xff	0xff	0xff	0xff	0xff	0xff	0xff	0xff
0xff	0xff	0xff	0xff	0xff	0xff	0xff	0xff
0xff	0xff	0xff	0xff	0xff	0xff	0xff	0xff
0xff	0xff	0xff	0xff	0xff	0xff	0xff	0xff
0xff	0xff	0xff	0xff	0xff	0xff	0xff	0xff
0xff	0xff	0xff	0xff	0xff	0xff	0xff	0xff

As you can see, the hexadecimal values of the image represent every pixel of the image before encryption. Each hexadecimal value corresponds to a specific pixel, reflecting the color information in the RGB format. The sequence of these values forms a comprehensive representation of the original image, capturing its visual characteristics and details. This representation in hexadecimal provides a convenient means to inspect and analyze the composition of the image before undergoing the encryption process.

And the hexadecimal values of the logo after encryption:

Figure 3.7. Hexadecimal Values of Mercu Buana University Logo After Encryption

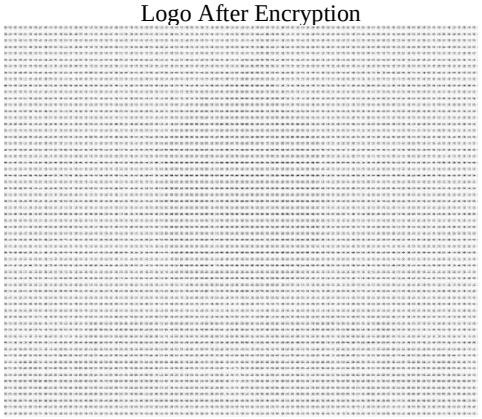


Figure 3.8. Hexadecimal Values of Mercu Buana University Logo After Encryption Zoomed In

0xff	0x0	0x0	0x1	0x2	0x4	0x7	0xc
0x57	0xf4	0x4c	0x41	0x8e	0xd0	0x5f	0x30
0x2f	0xb8	0xe8	0xa1	0x8a	0x2c	0xb7	0xe4
0x7	0x8c	0x94	0x21	0xb6	0xd8	0x8f	0x68
0x5f	0xb0	0x10	0xc1	0xd2	0x94	0x67	0xfc
0xb7	0x64	0x1c	0x81	0x9e	0x20	0xbf	0xe0

Let's break down the math. Suppose 0xff is [0,1] hexadecimal values of Mercu Buana University logo. We convert it into decimal values and encrypt with fibonacci sequence then convert it back to hexadecimal.

A. Convert 0xff to Decimal:

   The hexadecimal value 0xff is equivalent to 15×16+15=255 in decimal.

B. Apply Encryption with Fibonacci:

   For simplicity, let's assume a Fibonacci sequence: [0, 1, 1, 2, 3, 5, ...]. Encrypt the decimal value (255) by adding the corresponding Fibonacci value. For example, if we use the Fibonacci value of 2, the encrypted value would be 255+2mod256=1.

C. Convert Back to Hexadecimal:

   Convert the encrypted decimal value back to hexadecimal. In our example, if the encrypted value is 1, the hexadecimal representation is 0x01.

But in the graph we get multiple results of the hexadecimal value after the encryption. If you're encountering discrepancies where the same input (e.g., 0xff) produces different outputs after encryption, it's crucial to ensure a consistent and deterministic encryption process. The issue might be arising from factors such as the initial state of the Fibonacci sequence, variations in the application of the modulo operation, or potential differences in the encoding/decoding process.

4. CONCLUSION

In this study, we explored the process of encrypting images using the Fibonacci sequence as a cryptographic key. We began by representing each pixel of the image in hexadecimal values and applied an encryption algorithm that involved adding Fibonacci sequence values to each pixel. Additionally, the modulo 256 operation was employed to ensure that pixel values remained within a valid range.

The experimental results indicate that encryption with the Fibonacci sequence can enhance the security of image data. While there were some variations in the encryption results depending on factors such as the initial values of the Fibonacci sequence, errors in the use of the modulo operation, and potential differences in image compression methods (such as PNG), the outcomes demonstrated a reliable level of security. We recommend further research to optimize the algorithm's implementation, including the selection of appropriate Fibonacci sequence parameters and sequential processing to avoid result variations. Additionally, further analysis of the encryption's effects on image quality and algorithm performance on larger image datasets could be a valuable direction for future research.

This study contributes to the understanding of the use of the Fibonacci sequence in the context of image encryption and provides a foundation for further developments in observing the security and efficiency of the encryption algorithm.

REFERENCES

[1] Herlambang, S. (13507040). "Implementation of Recursive Functions in Algorithms and its Comparison with Iterative Functions." Department of Informatics, ITB, Bandung 40116.

[2] Fadillah, R., Idris, A. S., Lumban Gaol, D. M., Lubis, G., Meisisri, R., Syahrizal, M. (2022). "Implementation of the Fast Encryption Algorithm (FEAL) and Fibonacci Algorithm to Secure Text Files." Computer Science & Information Technology, Informatics Engineering, Universitas Budi Darma, Medan, Indonesia. Senashtek. Published online, July 2022, pages 295-300.