

ISTANBUL KULTUR UNIVERSITY
ENGINEERING FACULTY
COMPUTER ENGINEERING DEPARTMENT

CSE0421
CRYPTOGRAPHY

TERM PROJECT

Encryption and Decryption of Data
Embedded in Visual Content

GROUP MEMBERS:

Sueda Nur Kalaz

Semir Kimyonşen

Abstract

Digital medical images are assuming an increasingly important role in the diagnosis and treatment of diseases within the medical field. However, these medical images often contain sensitive patient information which makes their exchange over networks susceptible to numerous security threats and privacy breaches. One of the methods to prevent such privacy risks involves encrypting and embedding information within images. The encryption of information embedded within an image serves as a deterrent against attacks, hindering unauthorized retrieval due to the complexity of algorithms used and the way the information is embedded. This fortifies the protection of encrypted data within the image, thwarting potential attacks and preventing unauthorized access.

Introduction

Our project is to provide a secure program combining cryptographic techniques and information hiding which will hide the personal information on a gray scale medical image.

There are two main problems that need to be addressed in developing the described program the first of these is to embed the personal information on the gray scale medical image without altering the overall look of the image so that unless known before it will not be clear to the person whether the said image contains hidden information and the latter one is to retrieve the embedded information back from this image.

From these point on these 2 main problems branches onto several design issues which can be defined as below:

1. Embedding the personal information on a gray scale image

For this problem, first of all the program has to choose which image format it will use to hide information on. After that comes the question which personal information it should encrypt and embed onto image?

From there comes the backbone of the program that is which algorithm should it use to encrypt the personal information. It is very important inasmuch as this is the part in which the security of the program is provided.

The next problem is to hide the encrypted message onto the gray scale medical image. It must be done with utmost care since while embedding the ciphertext the program must not alter the image in such a way that it will distort the look of the image.

These are the main problems in embedding the personal image on a gray scale image.

2. Retrieving the information that had already been hidden by the program

This part naturally shares almost all the problems that has been explained before in embedding the information on the image part. Although to execute this part of the project there are some questions that should be answered as well such as

The developed program should be able to retrieve ciphertext according to the technique which had used to embed the information on the gray scale medical image.

Also, it should be able to decrypt the ciphertext that it retrieved and print the output to the user.

Implementation

To implement the solution of the project, we will use the steps that given below. More detailed information about these steps is provided in p.

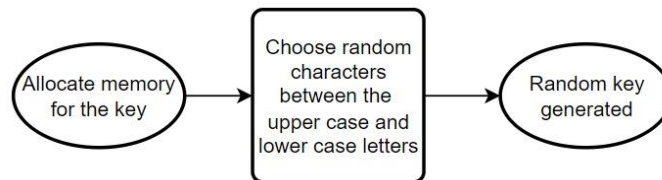
1. Selecting the personal information that will be hidden into the gray scale medical image
 - We have carefully chosen specific personal details name, surname, social security number, birthday, gender, and the date and time of the operation to embed within the medical image. This information is crucial for identification and record keeping purposes within medical contexts.
2. Selecting cipher algorithm to be used
 - After considering various encryption methods, we opted for the Vernam cipher algorithm due to its robust security features, including its ability to provide perfect secrecy when implemented correctly.
3. Algorithm to embed the data
 - Our choice of least significant bit (LSB) steganography stems from its efficiency in hiding data within image files without perceptible visual changes. The LSB method was selected for its balance between effectiveness and imperceptibility.
4. Deciding the format of the image file
 - For the successful implementation of LSB steganography, we determined that the bitmap (.bmp) format offers optimal support for the embedding and extraction of data within the image. Also it is suitable for LSB thanks to its uncompressed format, direct pixel access, lossless quality and wide compatibility characteristics.
5. Algorithm to extract the data
 - The process of extracting embedded data involves reversing the steps used for embedding. This process ensures the retrieval of the hidden information from the image while maintaining data integrity and accuracy.
6. Displaying the decrypted data
 - Upon successful extraction and decryption, the concealed personal data will be presented to the user in a clear and organized manner, ensuring accessibility and comprehensibility.

Discussion

Proper encryption forms a crucial part of the proposed project schema, employing a multi-step approach for both encryption and decryption processes. Among the encryption algorithms utilized, the Vernam Cipher stands out due to its provision of perfect secrecy. This cipher operates by utilizing the XOR (exclusive OR) operation alongside randomly generated keys to encrypt textual data. Essentially, each character undergoes encryption by undergoing an XOR operation with a specific key. To decrypt the encrypted text, the same key is employed in an XOR operation to revert the characters to their original state.

The encryption process involves three significant steps, ensuring a robust and secure transformation of data.

In the first step, to encrypt the data, a random key is generated. The foundation of Vernam Cipher lies in generating random keys used for encryption. These keys are necessary for deciphering the encrypted text and the security of the encryption process relies on these keys being random and used only once.



```
//===== generateRandomKey function -START =====
char* generateRandomKey(int length) {
    char* key = (char*)malloc((length + 1) * sizeof(char));
    if (key == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    srand((unsigned int)time(NULL));

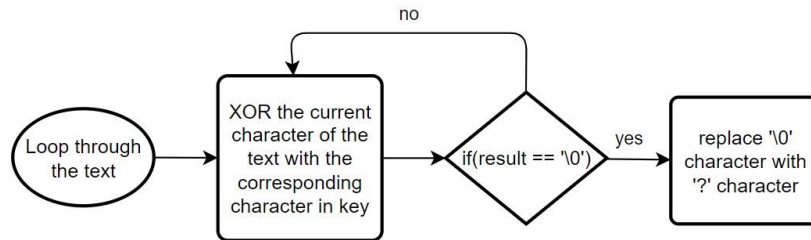
    const char charset[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

    for (int i = 0; i < length; ++i) {
        int index = rand() % (sizeof(charset) - 1);
        key[i] = charset[index];
    }
    key[length] = '\0';

    return key;
}
//===== generateRandomKey function END =====
```

After generating a random key, data is encrypted. At the bit level, each character is converted into its binary representation using character encodings like ASCII or Unicode. Randomly generated keys, which are of the same length as the text, undergo XOR operations with the binary representation of the text. For instance, each bit of the text is XORed with the corresponding bit in the key. The result of this XOR operation yields the encrypted text at the bit level. The encrypted text, when viewed at the bit level, appears drastically altered from the original and lacks coherent patterns without decryption. However,

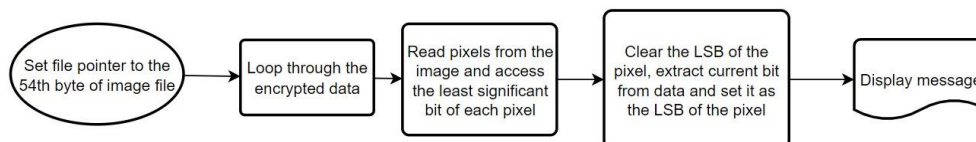
there are some issues while encrypting the data. While encrypting the data, NULL value corresponds to 0, causing issues when extracting data. We replace it with the question mark, represented by the value 63 in the ASCII table.



```

//===== vernamEncrypt function -START =====
void vernamEncrypt(char* text, const char* key, const short size, const short keyLen) {
    for (size_t i = 0; i < size; ++i) {
        text[i] = text[i] ^ key[i % keyLen];
        if (text[i] == '\0') {
            text[i] = '?'; /*NULL value corresponds to 0.
                           There is a problem when extracting the data.
                           We replace it with a question mark
                           corresponding to the value 63 in the ASCII table.*/
        }
        i++;
    }
}
//===== vernamEncrypt function END =====
  
```

In the third part, the random key and the cipher text is embedded into the bitmap medical image file. We apply the least significant bit (LSB) steganography method for embedding the data into medical image. The hiding process involves altering the least significant bit of each byte of data in the medical image. This modification is so subtle that it remains imperceptible to the human eye when observing the resulting stego (embedded) image. While embedding the data into image, there are some issues that should be concerned. The first 54 bytes of an image in BMP format, from position 0 to 53, are allocated for header information, and no message is embedded within these bytes. So, while embedding the data, these bytes will be skipped. Additionally, the embedded data and the random key should not be embedded on top of each other. To do that, while embedding the key into image, after skipping first 54 bytes, we will also skip some more pixels to embed it so they will not overlap.



```

//===== embedMessage function -START =====
void embedMessage(FILE* image, const char* message, const short size) {

    fseek(image, BMP_HEADER_SIZE, SEEK_SET);

    for (size_t i = 0; i < size; ++i) {
        char ch = message[i];

        for (int j = 7; j >= 0; --j) {
            uint8_t pixel;
            fread(&pixel, sizeof(uint8_t), 1, image);

            pixel = (pixel & 0xFE) | ((ch >> j) & 1);
            fseek(image, -1, SEEK_CUR);
            fwrite(&pixel, sizeof(uint8_t), 1, image);
        }

        printf("\nMessage successfully embed!");
    }

    //===== embedMessage function END =====

//===== embedRandomKey function -START =====
void embedRandomKey(FILE* image, const char* data, const int startPixel, const int length) {
    fseek(image, BMP_HEADER_SIZE + startPixel * 3, SEEK_SET);

    for (int i = 0; i < length && data[i] != '\0'; ++i) {
        char ch = data[i];

        for (int j = 7; j >= 0; --j) {
            uint8_t pixel;
            fread(&pixel, sizeof(uint8_t), 1, image);

            pixel = (pixel & 0xFE) | ((ch >> j) & 1);
            fseek(image, -1, SEEK_CUR);
            fwrite(&pixel, sizeof(uint8_t), 1, image);
        }
    }

    //===== embedRandomKey function END =====

```

Like the encryption process, decryption process also involves three significant steps, ensuring a robust and secure transformation of data.

In the first step, to decrypt the data, the random key is extracted from the image. Starting from the special pixel that is used to embed it, also its extraction is done.

```

//===== extractRandomKey function -START =====
void extractRandomKey(FILE* image, char* extractedData, const int startPixel, const int length) {
    fseek(image, BMP_HEADER_SIZE + startPixel * 3, SEEK_SET);

    for (int i = 0; i < length; ++i) {
        char ch = 0;

        for (int j = 7; j >= 0; --j) {
            uint8_t pixel;
            fread(&pixel, sizeof(uint8_t), 1, image);

            ch = (ch << 1) | (pixel & 1);
        }

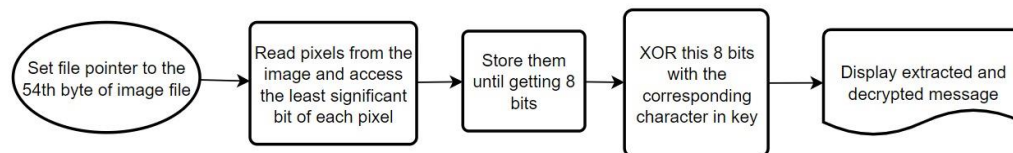
        extractedData[i] = ch;
    }

    extractedData[length] = '\0';
}

//===== extractRandomKey function END =====

```

After retrieving the key from the image, the extraction process proceeds to obtain the data starting from the 54th byte of the image. Retrieving the key from the image involves reversing the operation used to embed it initially. Initially, the Least Significant Bits (LSBs) of the pixels were manipulated. As every 8 bits correspond to a single character, the decryption function was invoked for each set of 8 bits to decrypt the character, consequently generating the cipher text.



```

//===== extractMessage function -START =====
void extractMessage(FILE* image, const short size, const char* key) {
    char extractedMessage[MESSAGE_SIZE];
    memset(extractedMessage, 0, MESSAGE_SIZE);

    fseek(image, BMP_HEADER_SIZE, SEEK_SET);

    int index = 0;
    char ch = 0;

    while (1) {
        for (int j = 7; j >= 0; --j) {
            uint8_t pixel;
            fread(&pixel, sizeof(uint8_t), 1, image);

            ch = (ch << 1) | (pixel & 1);

            if (++index % 8 == 0) {
                if (ch == '\0') {
                    vernalDecrypt(extractedMessage, size, key, 11);

                    printSeparatedData(extractedMessage);
                    return;
                }
                extractedMessage[index / 8 - 1] = ch;
                ch = 0;
            }
        }
    }
}

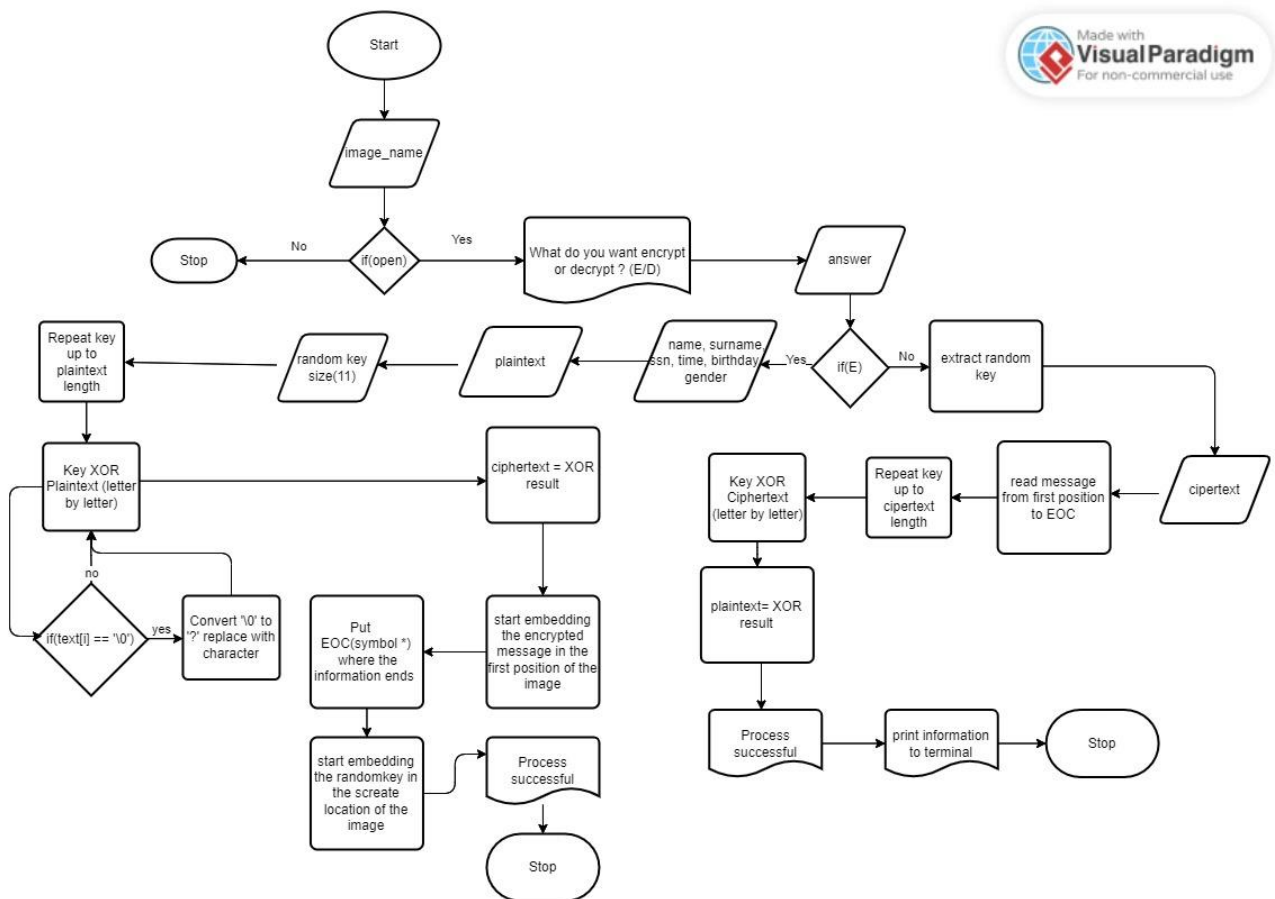
//===== extractMessage function END =====,

```

Lastly data is decrypted and separated into parts according to special character, which is + in this case. At the binary level, the decryption process involves reversing the encryption steps conducted during the encryption phase. To decrypt the encrypted text, the inverse XOR operation is performed between the encrypted text and the key, restoring the original binary representation of the text. Subsequently, this binary data is converted back into characters using ASCII or Unicode encoding. During decryption, any bit sequence equivalent to the ASCII value of a question mark (63 in ASCII) is substituted with the NULL value (\0) to accurately reconstruct the original data. The decrypted characters are then reassembled, forming the coherent and original message by undoing the drastic alterations observed in the encrypted text at the bit level, thereby restoring its coherence and readability.

```
//===== vernamDecrypt function -START =====
void vernamDecrypt(char* text, const short size, const char* key, const short keyLen) {
    for (size_t i = 0; i < size; ++i) {
        if (text[i] == '?') {
            text[i] = '\0'; // ? with ASCII code 63 We replace the character with the NULL character whose ASCII code is 0.
            text[i] = text[i] ^ key[i % keyLen];
        }
        else {
            text[i] = text[i] ^ key[i % keyLen];
        }
    }
}
//===== vernamDecrypt function END =====
```

According to these, our project's flowchart is given below:



Our application has two modes, developer mode and user mode. If the program is initiated in developer mode, a slight adjustment within the code is required for this mode to be activated. The determination of whether the individual is a developer or not relies on the correct entry of a username and password. If the credentials are entered correctly, the developer mode will be accessible, enabling specific operations. However, if incorrect details are entered, the system will continuously deny access and prevent any actions. This addition allows our program to be tested by different developers without sharing the source code, offering a layer of security against unauthorized access by individuals who do not have the correct username and password. This feature is not available to regular users for added security measures.

When the application is launched with the developer mode, it will first inquire whether the user is an administrator. If the user is an administrator, they will be prompted to enter the admin account password. Upon entering the correct password, the user will be granted permission to perform encryption or decryption tasks. If the password is incorrect, the program will terminate.

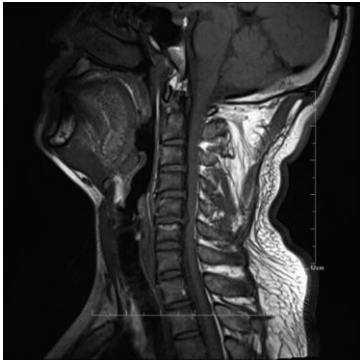


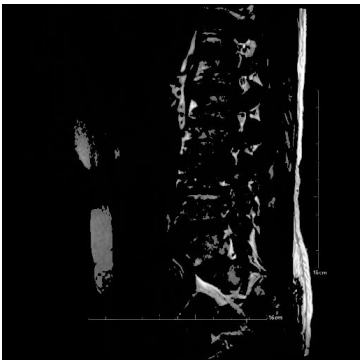


In case the user selects 'encryption,' the program will generate a random key. This generated key will then be embedded into a specific pixel of the image, designated by us, for use in our decryption algorithm. Subsequently, the patient information entered by the user will be combined into a single string. This string, terminated with an End-of-Communication (EOC) character as per our specification, will be encrypted using XOR with the random key. The resulting ciphertext will be embedded into the least significant bit of each pixel in the image. After completing the embedding process, the user will be notified, and the program will terminate.

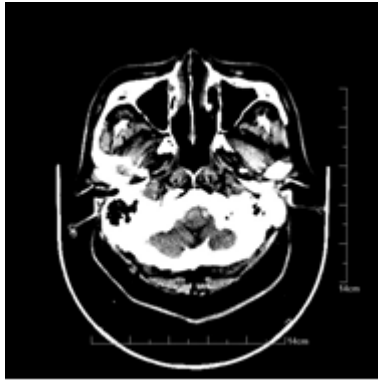
If the user selects 'decryption,' the program will first extract the key embedded in the specified pixel of the image. Following this, starting from the 54th pixel, the program will begin extracting the ciphertext that we embedded earlier and decrypt it character by character by performing XOR operations. Finally, the extracted data from the image will be displayed on the screen for the user, and the program will terminate.

Results

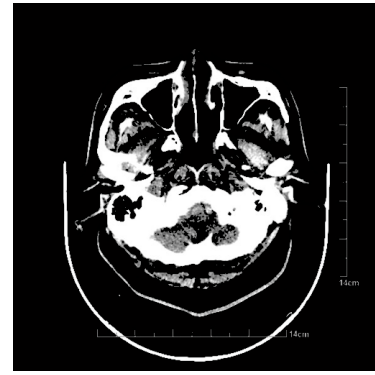
After implementing the steps outlined in the discussion section, we attempted to embed patient information into several bitmap medical images within our developed project. Subsequently, we compared these images to observe the original and encrypted versions containing the concealed data and report them.

Our projects results are given below:

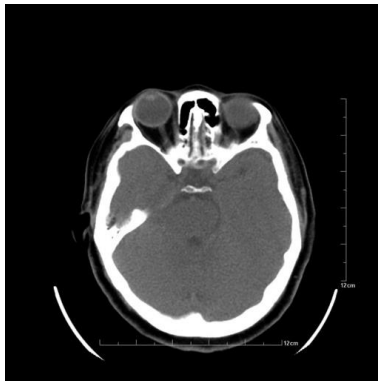
Original image	Encrypted data embedded image
<div><p>MR_Cervical_vertebra.bmp</p></div>	<div><p>MR_Cervical_vertebra.bmp</p></div>
<div><p>MR_Waist.bmp</p></div>	<div><p>MR_Waist.bmp</p></div>
<div><p>X_Lungs_2.bmp</p></div>	<div><p>X_Lungs_2.bmp</p></div>



CT_Head_1.bmp



CT_Head_1.bmp



CT_Paranasal_sinus.bmp



CT_Paranasal_sinus.bmp

Thanks to the advantages provided by the LSB algorithm, when embedding sensitive information such as patient records into medical images, there is no significant distortion or blurring; however, some minimal alterations may occur. While there are certain differences between the original and encrypted versions of the image, if these changes are carefully managed and kept imperceptible, distinguishing a picture embedded with encrypted information from a regular one can be quite challenging. Consequently, determining whether data has been concealed in images using this method becomes difficult, complicating the identification of the presence of embedded information. Furthermore, thanks to the one-time pad offered by the Vernam Cipher algorithm, extracting information from images becomes exceedingly difficult.

Conclusions

In conclusion, medical images hold a significant place in the healthcare sector, and ensuring the confidentiality of information embedded within these images is crucial. Our project focused on embedding personal information into grayscale medical images securely. The application's functionality,

allowing encryption and decryption based on administrator authorization, further enhances the security and control over the embedded information. The program's ability to embed and extract data while preserving the image's integrity demonstrates its efficacy in secure data transmission. However, challenges such as ensuring perfect concealment without image distortion and managing the embedding of keys and data remain. Additionally, enhancing the application to handle various image formats and scaling the approach for larger datasets can be considered for future improvements.

For developers:

Main Function (main)

- It allows the user to enter the image file name and operation type to add or remove embedded data.
- **encryptData** function extracts personal information from the user, encrypts it and stores it embedded in the image.
- **extractMessage** function extracts the encrypted data from the image, decrypts it and displays it to the user.

Data Encryption and Decryption Functions (vernamEncrypt, vernamDecrypt)

- **vernamEncrypt** function encrypts the text using the Vernam encryption algorithm.
- **vernamDecrypt** function decrypts the encrypted text.

Image Processing Functions (embedMessage, extractMessage, embedRandomKey, extractRandomKey)

- **embedMessage** function embeds the encrypted text into the image.
- **extractMessage** function extracts and decrypts the ciphertext from the image.
- **embedRandomKey** and **extractRandomKey** functions embed and extract the random key from the image.

Other Helper Functions (printHex, stringLength, printSeparatedData, generateRandomKey)

- The **printHex** function prints data in hexadecimal format.
- **stringLength** function calculates the length of a string.
- **printSeparatedData** function prints separated data to the screen.
- **generateRandomKey** function generates a random key of a given length.

Sure, here are the descriptions of the functions used in **ADMIN_DEBUG** and the features they provide to developers:

1. **vernamEncrypt** and **vernamDecrypt** Functions

- **Feature:** These functions encrypt or decrypt data using the Verbatim encryption algorithm
- **Developer Feature:** Using **ADMIN_DEBUG**, you can monitor the stages of the encryption or decryption process in detail. **printf** statements show the value of each character before and after encryption. This can be useful to check the correctness of encryption and decryption.

2. **embedMessage** and **extractMessage** Functions

- **Feature:** These functions embed or extract ciphertext from the image.
- **Developer Feature:** Using **ADMIN_DEBUG**, you can display the message embedded and extracted by **printf** statements in hexadecimal format. This can be useful to check the correctness of the displayed values of the message.

3. **embedRandomKey** and **extractRandomKey** Functions

- **Feature:** These functions allow the random key to be embedded in or extracted from the image.
- **Developer Feature:** Using **ADMIN_DEBUG**, you can view the values of the embedded or extracted random key. This can be useful to verify that the random key was processed correctly.

4. Other Auxiliary Functions

- **Feature:** functions like **printHex**, **stringLength**, **printSeparatedData**, **generateRandomKey** perform auxiliary operations.
- **Developer Feature:** **Printf** statements have been added that indicate the internal state of certain steps or data using **ADMIN_DEBUG**. In this way, it can be easier to understand if any step is in an unexpected state and to understand the running process of the code.

This **ADMIN_DEBUG** feature can be used to help developers in the process of tracing code, debugging and validating functions. However, in the normal user environment such detailed output is usually disabled, as extra output can affect performance and pose a security risk.

References

1. Shannon, Claude E. "Communication Theory of Secrecy Systems." Bell System Technical Journal 28.4 (1949): 656-715.
2. Singh, Simon. The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography. Anchor, 2000.
3. Stinson, Douglas R. Cryptography: Theory and Practice. CRC press, 2005.
4. Paar, Christof, Jan Pelzl, and Bart Preneel. Understanding Cryptography: A Textbook for Students and Practitioners. Springer, 2010.
5. B. Baysan And S. Özekes, "DLSB - The distanced least significant bit steganography DLSB - Uzaklaştırılmış en önemsiz bit steganografi," *Journal of the Faculty of Engineering and Architecture of Gazi University*, vol.38, no.3, pp.1725-1735, 2023

6. [Steganography/Steganography_Project at master · pavankumar5963/Steganography · GitHub](#)
7. [GitHub - jsevilla274/isteg: Simple Image Steganography using LSB encoding](#)
8. [GitHub - w-i-l/lbsb-method-bmp: A steganography program to hide a message with LSB-method in a BMP file rgb](#)
9. [GitHub - AlamHasabie/multiple-lsb-steganography: Multiple LSB text steganography on a .bmp/.png formatted file. The text is also encrypted with extended \(256 characters\) Vigenere Cipher beforehand. IF4020 Cryptography class project](#)
10. [GitHub - Johnson-Su/Steg-Hide: A C based program that uses LSB steganography to hide text in images](#)