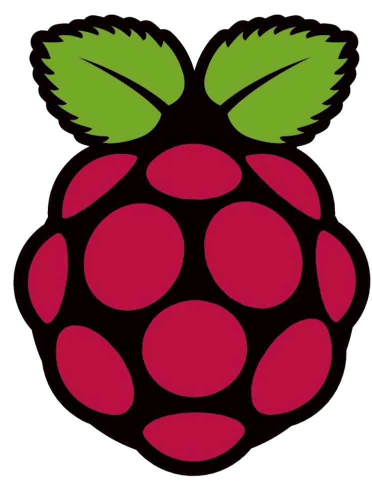

Encryption and Decryption with a Raspberry Pi Device

Winning Entry in Fall 2020 RasPi Programming Competition

Taylor Powell - Old Dominion University
Competition Sponsors - Dr. Ayman Elmesalami & Dr. Soad Ibrahim

Introduction

In this project, I aimed to design a program which allows a user to securely encrypt and decrypt messages using a variety of historically significant techniques in the development of cryptology. The specific encryption methods I decided to explore are:

1. A Caesar Cipher
2. A Modified Vigenère Square
3. A Stream Cipher
4. Steganography

In addition, I aimed to provide the user with the ability to choose from a combination of these encryption techniques as well as input and output methods.

Caesar Cipher

The Caesar Cipher is based on the principle of shifting the letters of the message being encrypted a set number of "places" at a time (e.g. shifting "ABC" by 2 produces "CDE").

A Caesar Cipher is one of the earliest known encryption techniques since it is fairly simple to employ in written communications and give the user some minor degree of security since the original message is not immediately obvious. However, even simple decryption techniques can readily break this code, and it is not at all suitable for modern encryptions.

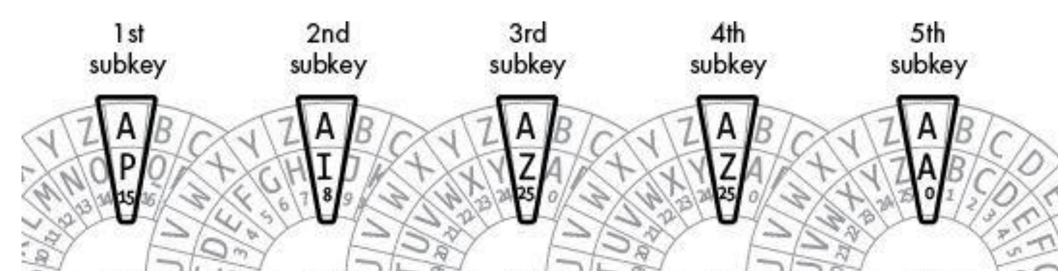

Figure 1: Visualizing the Vigenère Cipher

Vigenère Square Cipher

The Vigenère Cipher is similar in principle to the Caesar Cipher, however, the shift is determined by a code word or phrase.

As an example, if you wish to encrypt the phrase "ABC" with the code word "CAT" by hand, the original Vigenere Square would give an encrypted phrase of "DCW."

The Vigenère Square Cipher was originally implemented using a 26x26 grid of the alphabet. A user would take the next letter to be encrypted as well as the corresponding letter from the code word, and find their intersection on the grid to encrypt the message. This code was significantly more secure than the Caesar Cipher, however, when the code word or phrase is significantly shorter than the message, frequency analysis methods can allow a third party to decrypt the message with enough time and encoded text to examine.

In the particular case that you choose a code phrase equal in length to the message to be encrypted and use that code phrase only once, this method becomes a variant of the one-time pad method, and it unbreakable as long as the code phrase is kept secret.

The Program

For my program, I used the Python programming language inside the Raspberry Pi specific development environment Thonny. The program prompts the user to choose to either encrypt or decrypt a message, and to decide whether to use the Caesar, Vigenère , or Stream Cipher. The program then splits based on the user's selects and implements accordingly.

Encryption

For encryption, the user can choose between three output options: text in the execution window, outputting to a .txt file, or encoding the information into a user-selected picture (steganography).

For both the Caesar and Vigenère Ciphers, the encryption is done by converting letters in the message (and the code word/phrase for Vigenere) to their representative ASCII values. For the Caesar Cipher, the shift value is added to the ASCII value for the message, and the resulting value is converted back into its character representation. The program allows for positive and negative values, and the program acts to ensure the created value is within the printable character set. For the Vigenère Cipher, both the message letter and the code letter are converted into ASCII values and added together, resulting in a new ASCII value which is handled similarly to the Caesar Cipher case.

For the Stream Cipher, I convert the message into its bitwise representation. I then use Python's 'random' library to generate a bit string encryption key of equal length. The two are then added together in a XOR operation (as shown to the right), resulting in an encrypted bit string.

If the user encodes the encrypted message into a photo, the program prompts them to select a photo file. For the Caesar and Vigenère Ciphers, the encrypted message is broken into its bitwise representation. For the Stream Cipher, the encryption key is appended to the encrypted bitstring to create the string to be encoded. The program also generates a 64 bit string to represent the bit-length of the encoded message to assist with decoding the photo. This bit string is appended to the front of the encrypted bit message.

Each bit is encoded into one pixel's red color value. In order to do so, I alter the least significant bit of the pixel's red color value to match the bit for the bit being encoded. Using this process, the entire bit string is encoded into the photo file (assuming there is adequate space to do so).

For a standard iPhone X photo with 4032x3024 pixels, a single photo could hold over 500 pages of single-spaced 12pt writing encoded into its pixels using this technique!

Original Message -	`Hello World!`
Code Phrase -	`QWERTY123!@#`
Encrypted Message -	`:]R_dy)BF.ED`

Figure 2: Example of Vigenère Cipher Encryption

Decryption

For decryption, many of the processes previously mentioned are simply done in reverse-order.

The user informs the program whether the encrypted message is stored: direct input to the execution window, a .txt file, or encoded into a photo. In the case of the Stream Cipher, the program assumes the message is in the stored format provided by encryption (i.e. the encryption key is appended to the encrypted message). For either the Caesar or Vigenère Ciphers, the program additionally prompts the user for the shift value or the code word/phrase.

Since the Caesar and Vigenère Ciphers are additive ciphers, the decryption is simply done using subtraction instead of addition.

Since the Stream Cipher is a symmetric-key cipher, the encrypted message and the encryption key are again added together using an XOR operation to generate the unencrypted message.

To decode a photo, the program first retrieves the values from the first 64 pixels to determine the length of the message. The program then uses the recovered length to retrieve the appropriate number of additional pixel values to recover the entire encrypted bit string from the encoded file.

Future Work

In portions of the code, this initial project was approached as a proof-of-concept design, and thus is not as secure as possible. There are therefore a few aspects I intend to alter in future iterations:

1. Implement use of Python's library 'secrets' instead of 'random' to generate the bit string for the Stream Cipher
2. Encode the photos with a fully encrypted 64 bit string for the message length, as well as an additional bit string to represent a starting location for the encrypted message.
3. Separate the encrypted message from the encryption key for the Stream Cipher while encoding the photos to make the information more secure.
4. Implement a public-key type encryption algorithm to enhance security
5. Work with different media types for steganography (i.e. audio and video files, puzzle-type encoding techniques, etc)

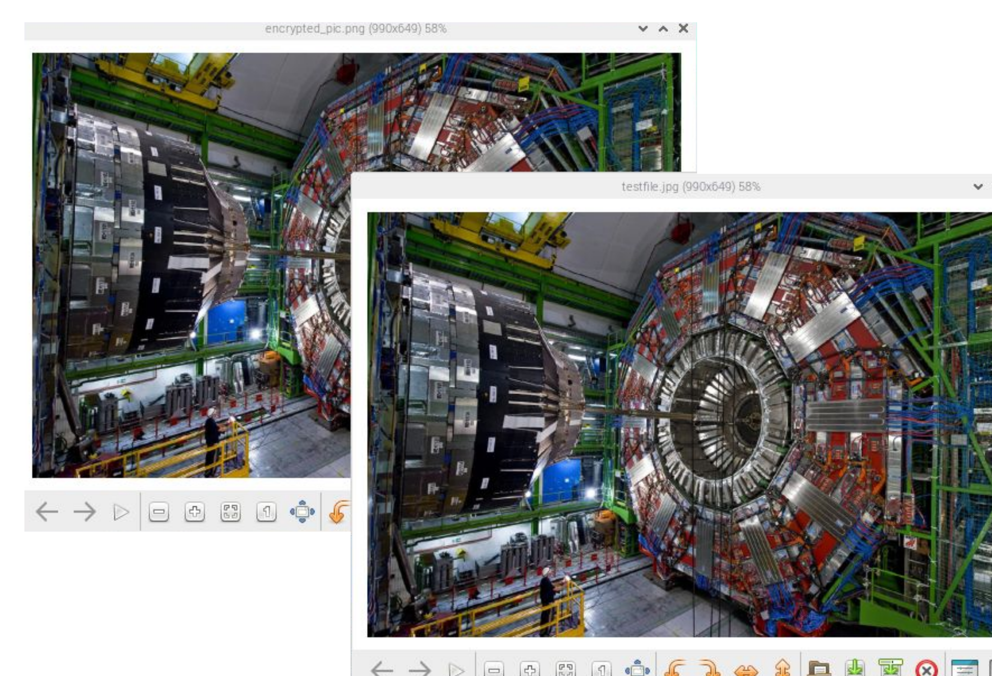

Figure 3: An original and encrypted photo side-by-side

Stream Cipher

The Stream Cipher works on the principle of bit-by-bit encryption. For a given message, the individual letters are broken into their bit-representations. The encryption program will then generate a "random" bit string (comprised of 0's and 1's) equal in length to the message's bit string. The two bit strings are then combined using an XOR operation.

The Stream Cipher is a symmetric-key encryption. This means that once the message is encrypted, the user attempting to decrypt the message will use the same random bit string to conduct the decryption.

Since the Stream Cipher is based on bitwise operations, it is a technique which was developed early in the computer era. The main potential issue for security is whether the generated bit string is truly random. Most programming languages are either deterministic or only pseudo-random, thus third parties with knowledge of the techniques used to encrypt the message could be able to replicate the encryption key and break the code.

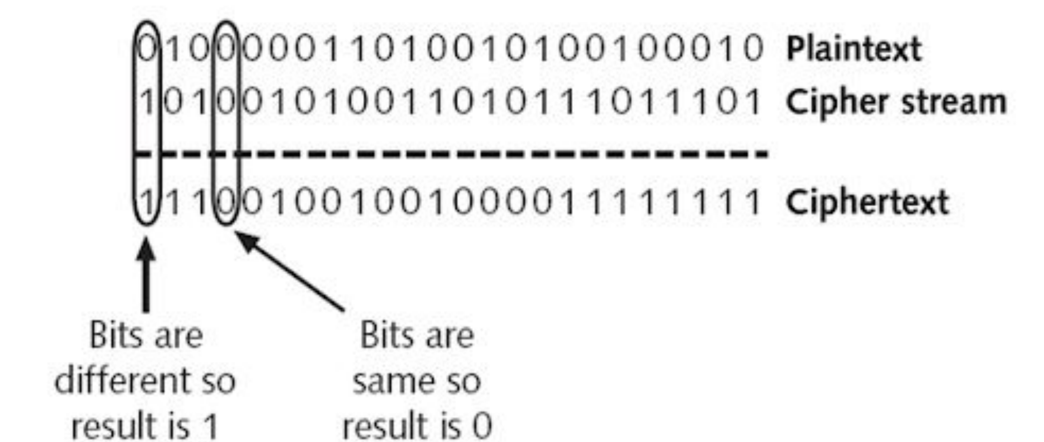

Figure 4: Visualizing the XOR Operation

Steganography

Steganography is the practice of hiding information in media. Steganography can take many different forms, and has been in practice in various ways for centuries. A classic example of steganography is writing in "invisible ink" which requires some special process to reveal.

In digital media, information can be hidden in innumerable ways: manipulating pixel values, embedded information in video or audio files, or even manipulating rate at which network data is received on a computer. Using modern applications, the ability to hide encrypted information in plain sight is only limited by the programmer's technical capabilities and imagination.

Of particular interest for this project, I worked with manipulating the least significant bit of the byte of information representing the color values of individual pixels.

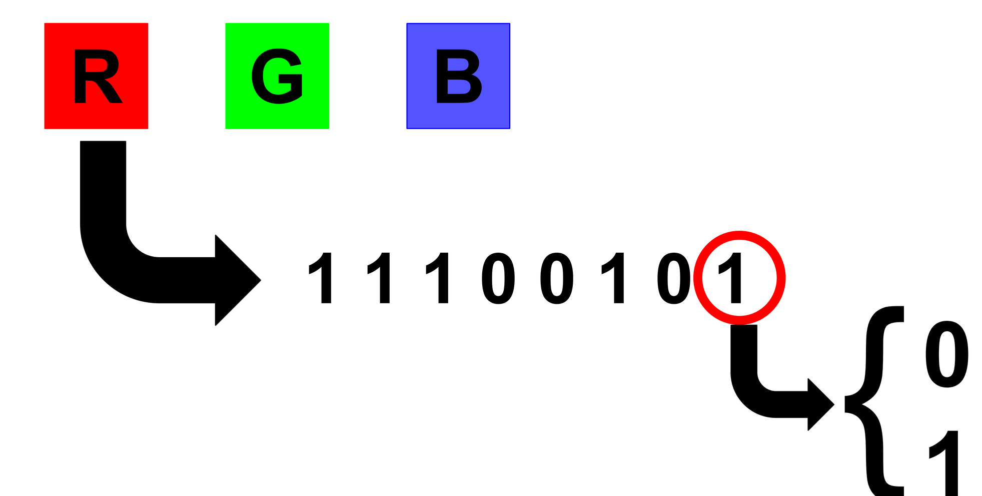

Figure 5: Visualizing the encoding process for a pixel