Project ID: RI11-JB2-0012

Project Title: Image-based Encryption System

Team Work

Student ID	Student name	Session
10604707	Abdulrahman Al-Othman	11122001
10706171	Motasim Alsayed	11122002

Supervisor: Dr. Khaled Suwais

2012 Riyadh

Dedication

To my family, who has always supported me in completing my education. To all the people who have helped me through the difficult times in my life.

Abdulrahman Al-Othman

To my lovely mother, who has always believed in me. I wouldn't be where I am today if it wasn't for you.

Motasim Alsayed

Acknowledgments

We would like to thank our supervisor Dr. Khaled Suwais for his kind and continuous support and feedback.

We would also like to thank Bernd de Graaf and Yvo van Beek from CodeGazer.com for their time and for open sourcing PixelCryptor to help us analyze and understand the system.

Finally, we would like to thank the community of StackOverflow.com for helping us in optimizing the system.

Table of Contents

List of Figures

List of Tables

Abstract

Password have been widely used in symmetric encryption algorithms, but there are many problems related to passwords, including the difficulty for humans for remembering secure passwords, and the easiness of cracking the non-secure ones using a brute force, a dictionary or a rainbow table attack.

In this report, we introduce and describe a symmetric stream cipher encryption system that is based on images instead of password. The performance of this system depends on the keystream generator, which in turn depends on the number of pixels in the image and the key size specified. The throughput of the stream cipher varies from 80 KB per second using a key size of 512-bits and an image with 60 thousand pixels, down to 144 bytes per second using a key size of 128-bits and an image with 15 million pixels.

Chapter 1: Introduction

1.1 Problem Statement

A common approach for generating encryption keys is using a password and applying a Password-Based Key Derivation Function (PBKDF) to it[1]. This approach has some problems, most of which are related to the nature of passwords themselves.

Such problems include vulnerability to brute force, dictionary and rainbow table attacks[1]. There is also a conflict between the users' requirements and the security requirements for specifying a password. Typically, users want short, easy to remember passwords. Security, on the other hand, requires long passwords that have enough randomness. Hence, a password that satisfies the users' requirements will be easy to crack, and a password that satisfies the security requirements will be difficult for humans to remember.

1.2 Aims and Objectives

Everything in our digital life requires an authentication mechanism to establish the identity of the user and protect his/her privacy. Since passwords are the most common form of authentication, our aim is to provide an alternative form that is not susceptible to the security risks and problems associated with passwords.

Our objectives are:

- Develop a synchronous stream cipher system that uses images as the secret keys to encrypt or decrypt text.
- Develop a graphical user interface and integrate it with the stream cipher to enable the user to encrypt/decrypt text and to present time

measurements and some security and other facts related to the inputs and the parameters specified.

1.3 Research Methodology

After analyzing and understanding the system's requirements, we use a divide and conquer design strategy by dividing each function into smaller sub-functions and then design, implement and test each sub-function independently, and finally integrate all of these functions and then perform an integration test.

For each function that is related to the security of the system, we prototype it on paper based on our research and on our understanding of the requirement, and then consult our supervisor on this prototype.

When we agree on a prototype for a specific function, we follow the Test-Driven Development process, in which we start by writing test cases based on this prototype before writing any code. After writing the unit tests for these test cases we make sure that they are failing before writing the function's code, this is to make sure that they don't mistakingly pass without requiring any new code, and also to test the test case itself and make sure it is not worthless. We may then design the function using UML sequence diagrams and then start coding, or we may start coding directly, based on the complexity of the function. As we are coding, we check the unit tests and continue until they pass. This process is illustrated in figure 1.

This process helps us in focusing on the requirements and increases our confidence in the system. It also makes it easier to refactor the code or modify some functionality based on changes in the requirements.

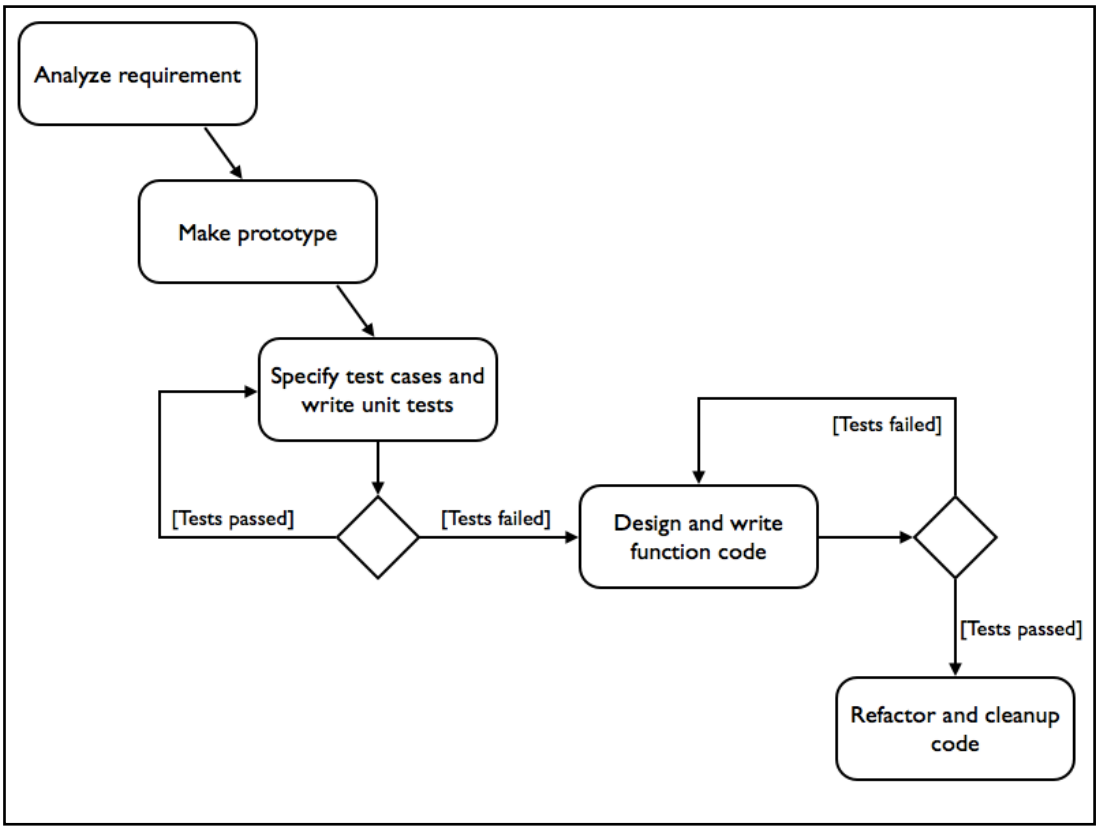

Figure 1. The test-driven development process.

1.4 Tools and Technologies

We will be using Java as the programming language for implementing the system. The reasons for choosing Java is that it is widely adapted, so the system could be easily read and can be easily ported to other languages in the future, and it is portable so the system can be run on any platform that has Java installed. While Java isn't the best language for performance-critical systems, we thought it would be acceptable for the use cases that will be performed using the graphical user interface that we will develop.

We will use the Netbeans IDE for many tasks including code editing, profiling and analyzing the system, compiling the source code, testing and debugging. Netbeans offers nice features for code editing, including auto completion, auto-formatting, dynamic error highlighting, etc. It also has a nice profiler that helps us analyze particular parts of the system, and helps in analyzing the overall memory consumption by the system. It also

offers a great debugger that can help us step into the source code while the system is running so we can inspect the state of the system and figure out the source of the bug.

We will also be using Git as the source control system for the project. It will help us prototype many different approaches for solving a problem without the need for dealing with the file system (such as duplicating a file or making a backup or something similar).

Finally, we will use Microsoft Word and Pages for editing and writing any project related documents, and we will also use Keynote for creating graphs and other UML and non-UML diagrams.

1.5 Report Organization

In chapter 2, we will write about the related works done. Then, we will write the use cases, the functional and non-functional requirements of the system in chapter 3. Then we will give a detailed description of the system in chapter 4. In chapter 5, we will describe the graphical user interface, and we will write requirements to run the system. We will next give a detailed analysis of the system's performance in chapter 6. Finally, we will write a conclusion and our recommendations and the scope for future work in chapter 7.

Chapter 2: Related Work

Cryptographic algorithms are usually categorized into two categories: asymmetric and symmetric cryptographic algorithms[2]. Asymmetric cryptography, also called public-key cryptography, requires two keys, in which one is used for encryption and the other is used for decryption[2]. There are many asymmetric cryptographic algorithms including RSA, Rabin, NTRUEncrypt and ECC. Symmetric cryptography, on the other hand, requires only one key for both encryption and decryption[2].

Symmetric cryptography algorithms are further divided into block and stream ciphers[2]. A block cipher encrypts a message by breaking it down into blocks, and then encrypting each block individually[3]. There are many modes in which a block cipher may operate in. The simplest is the Electronic Code Book (ECB) mode, in which each plaintext block is encrypted independently of the other blocks, which means that identical plaintext blocks give identical ciphertext blocks[4]. Another mode of operation is the Cipher Block Chaining (CBC) mode, in which a pseudo-random seed is XORed with the first plaintext block before encrypting the plaintext block, and the ciphertext block is then XORed with the next plaintext block before encrypting it, and so on[5]. There are many block ciphers, one of the most popular block ciphers is the Advanced Encryption Standard (AES), other algorithms include RC5, IDEA and Blowfish.

A stream cipher is a type of symmetric key algorithm that generates a pseudo-random sequence of bits, called the keystream, using a secret key, and then combines these bits with the plaintext bits (usually using exclusive-or) to generate the ciphertext bits[6]. The decryption process in stream ciphers is exactly the same as the encryption, in which the cipher

generates the same keystream that was generated during the encryption, and then combines its bits with the ciphertext bits to generate the plaintext bits[6].

There are many stream ciphers, including RC4 which is used in the Secure Sockets Layer (SSL) protocol for securing internet traffic[7] and in the Wired Equivalent Privacy (WEP) security algorithm for securing wireless networks[8]. Another popular stream cipher is A5/1 which is used for encrypting the traffic in the GSM cellular network [9] and is based on three linear feedback shift registers. Also there is the E0 stream cipher that is used in the Bluetooth standard [10].

Symmetric encryption algorithms are usually based on passwords. In order for these ciphers to eliminate some of the security risks associated with passwords, they usually use a Password-Based Key Derivation Function (PBKDF) to derive a secure secret key from the provided password[1].

Other symmetric encryption ciphers, such as ours, use images as secret keys. One such software system that uses images as secret keys is PixelCryptor. PixelCryptor is an open source file encryption system that is based on images, in which the user is asked to select an image from his or her PC to be used as the secret key, and then it encrypts or decrypts the selected files or folders using the selected image.

Figure 2. A screenshot of PixelCryptor's graphical user interface.

PixelCryptor is a stream cipher, the keystream simply consists of the combinations of the RGB colors of all the pixels in the image, where each color is represented by a single byte, as illustrated in figure 3. This means that the period of the keystream in bits is defined by multiplying the number of pixels by 24 (the number of bits per pixel).

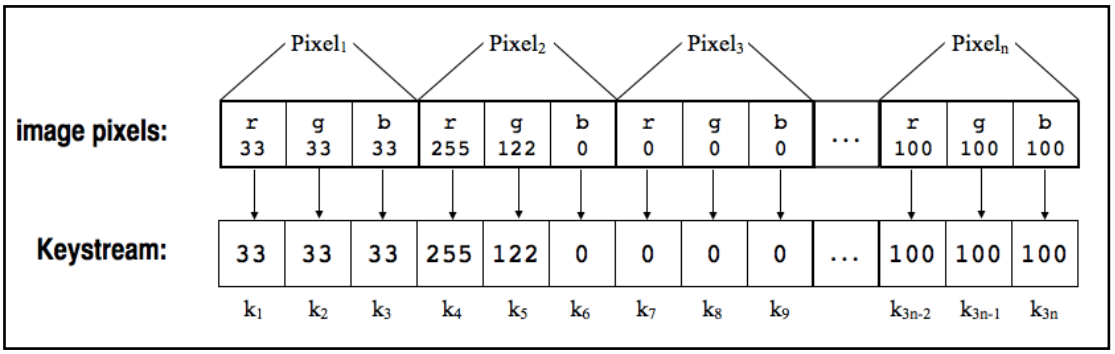

Figure 3. The keystream generated by PixelCryptor.

The encryption is done in a byte-by-byte bases, in which the content byte value is subtracted from the keystream byte value. In cases where the content byte value is greater than the value of the keystream byte, which results in a negative integer between -255 and -1 inclusive, the right most

8-bits of this integer will be used as the byte value, and the other bits will be discarded[11]. Our experiments have shown that the result of this truncation of the high-order bits is equal to adding 256 to the subtraction result. Decryption will be done in a similar manner, in which the encrypted byte value is subtracted from the keystream byte value, which will give the original content byte value. This is illustrated in figure 4.

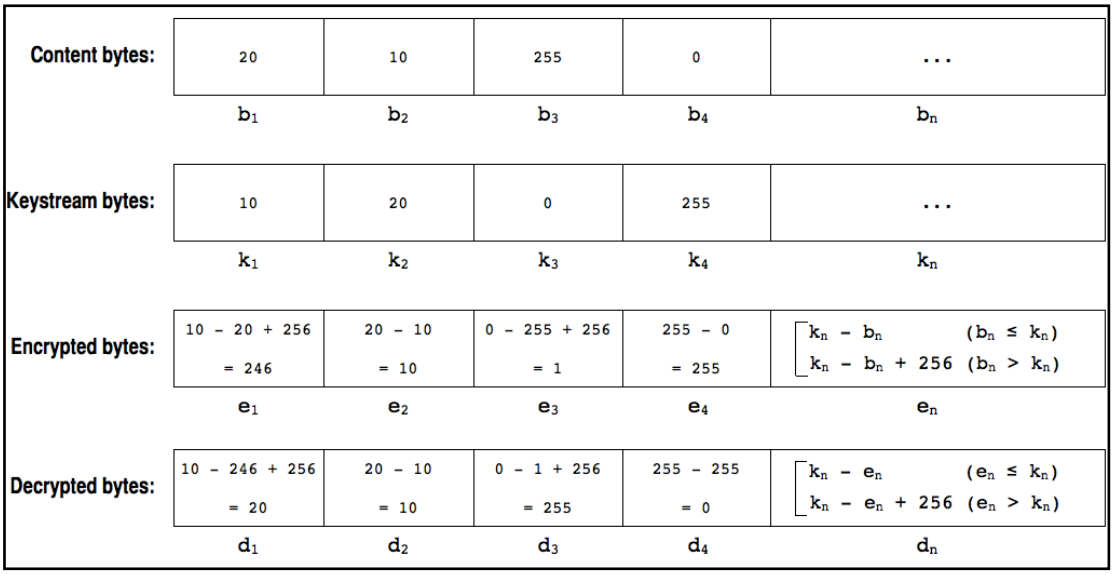

Figure 4. The encryption and decryption done by PixelCryptor.

The documentation suggests the image sizes listed in table 1 for the best security and performance based on the size of the content that will be encrypted.

Content size	Image resolution
0 - 20 MB	At least 100x50
20 - 200 MB	At least 200x200
200 - 2500 MB	At least 600x400
Over 2500 MB	At least 800x600

Table 1. PixelCryptor's recommended image size based on the encrypted content size.

Based on our analysis and the available documentation, we noted some advantages and disadvantages compared to our system. The main

advantage it has is its high performance for encrypting large files, especially when images with less than 1 million pixels are used. The reason for its high performance is that the keystream bits are not computed, they are taken directly from the image's bits. Another advantage is that the system recognizes the image that was used in the encryption when one needs to decrypt the files, this is done by appending some data that identifies the image to the encrypted data.

We noted two main disadvantages in the system. The first is the short period of the keystream. Based on the recommendations in table 1, if we are going to encrypt 20MB of data using an image with 5000 pixels, the period is 120,000 bits, which means that the keystream is going to repeat more than 1398 times during the encryption.

The second main disadvantage is that the randomness of the bits in the keystream depends solely on the chosen image. Since the keystream is directly taken from the color values without any modification, this means that it may consist of all 1s if the chosen image consisted of only a white color for all the pixels, which makes the keystream very weak. So one needs to choose an image that have a great variation of RGB values, and these colors need to be distributed evenly to ensure that the bits in any keystream block have enough randomness.

Other disadvantages include the long time it takes for initialization when using large images. Also, one cannot choose an image file that does not have an image extension, so if one wants to hide the image by removing the extension, he or she needs to re-add the extension in order to use the image in the system.

Chapter 3: System Analysis and Design

3.1 Use Cases

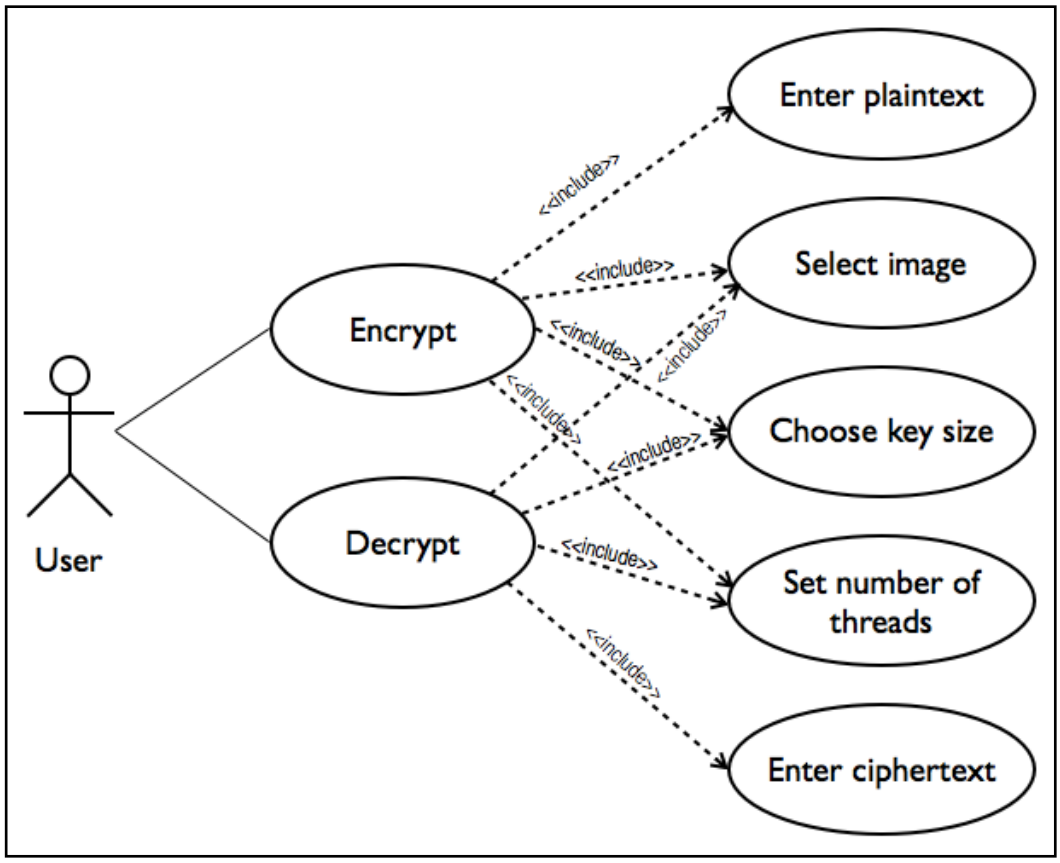

Figure 5. Use case diagram.

1. Encrypt. The user selects an image and enters some plaintext and specifies a key size and the number of threads, the system will then encrypt the text using the image's data by the key size specified on the number of threads specified, and then returns the ciphertext in a hexadecimal format.
2. Decrypt. The user selects an image and writes some ciphertext and specifies a key size and the number of threads, the system will then decrypt the text using the image's data by the key size specified on the number of threads specified, and then returns the plaintext as it was before encryption.

3. Select image. The user chooses an image file from his or her computer, and the system then will display a feedback to the user to indicate that the image has been selected.
4. Choose key size. The user chooses a key size in bits from the sizes 128, 256 and 512.
5. Set number of threads. The user enters a number to specify the number of threads to run the algorithm on.
6. Enter plaintext. The user enters some text that he/she wants to encrypt, and the system displays the entered text.
7. Enter ciphertext. The user enters hexadecimal characters that he/she wants to decrypt, and the system will display these characters.

3.2 Functional Requirements

The system should:

1. Allow the user to select an image from his/her PC.
2. Allow the user to choose a key size in bits from the set {128, 256, 512}.
3. Allow the user to specify the number of threads that will be used to carry out the encryption or decryption process.
4. Allow the user to enter a plaintext or ciphertext.
5. Allow the user to encrypt or decrypt text using a specific image and a key size.
6. Calculate and present the time taken for each task it performs and the total time it takes for the encryption or decryption process.

3.3 Non-functional Requirements

The system should:

1. Generate different keystreams for different images, even if the difference was in only one pixel.

2. Interact with the user using a graphical user interface.
3. Run on and support the latest versions of Windows and Mac OS X.
4. Support ASCII and unicode characters.
5. Support RGB, grayscale and indexed images.
6. Support the following image formats: png, jpg, tiff and gif.

Chapter 4: Proposed Algorithm

4.1 Overview

The system is an encryption system that is based on a synchronous stream cipher that uses images, instead of passwords, as the secret key. A synchronous stream cipher is a type of symmetric key algorithm that generates a pseudo-random sequence of bits, called the keystream, independent of the plaintext and ciphertext[6]. These bits are then combined with the plaintext bits (usually using exclusive-or) to produce the ciphertext, and then they may be combined with the ciphertext bits to produce the plaintext bits again.

The system starts by loading the image into memory and getting the input text bytes, and then building a vector by applying a transformation function to the image's pixels to be used later as the secret key. The system will then generate the keystream by combining multiple keys together. A single key is generated by a sequence of bit-shifting the image vector, then hashing it (using one of the Secure Hash Algorithms) and finally performing an exclusive-or between the image vector and the hash value, which will be described in detail in section 5 of this chapter. After generating all the keys required so that their combined bytes are equal to or greater than the input text bytes, the remaining process is simply performing an exclusive-or operation between each keystream byte with the input text bytes. The system will then represent the resulting bytes by a readable form, which may be the hexadecimal values of the encrypted bytes in the case of encryption, or a simple mapping of the decrypted bytes to their respective characters in the case of decryption. This process is illustrated in figure 6.

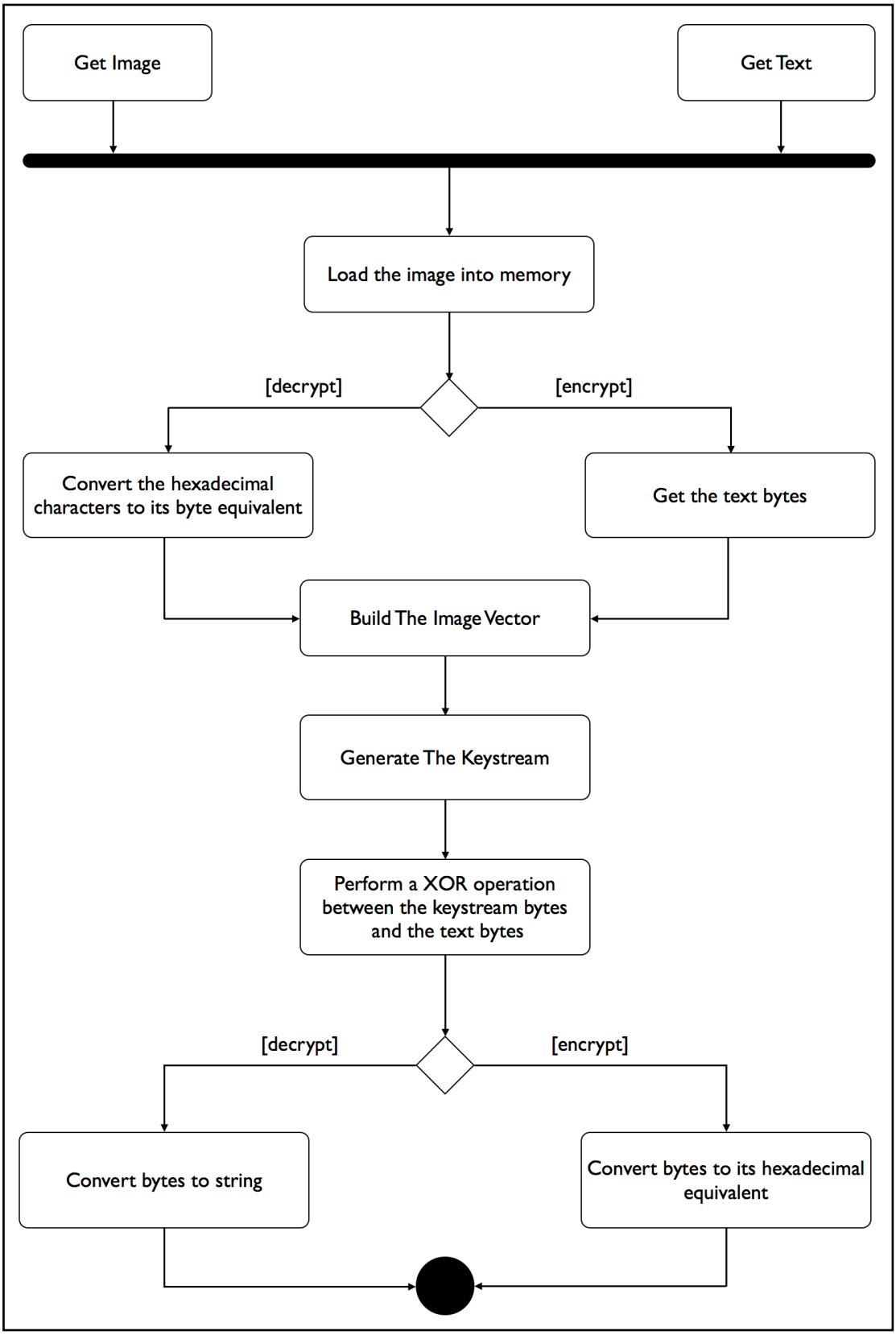

Figure 6. An activity diagram of the system.

We implemented a class Coordinator that will be responsible for taking inputs from the user interface and then coordinating these tasks to achieve the desired output, and finally returning the result back to the user interface. We decided to wrap each task in a single utility class to make the code easier to read and debug. The class diagram for the core of the system is illustrated in figure 7.

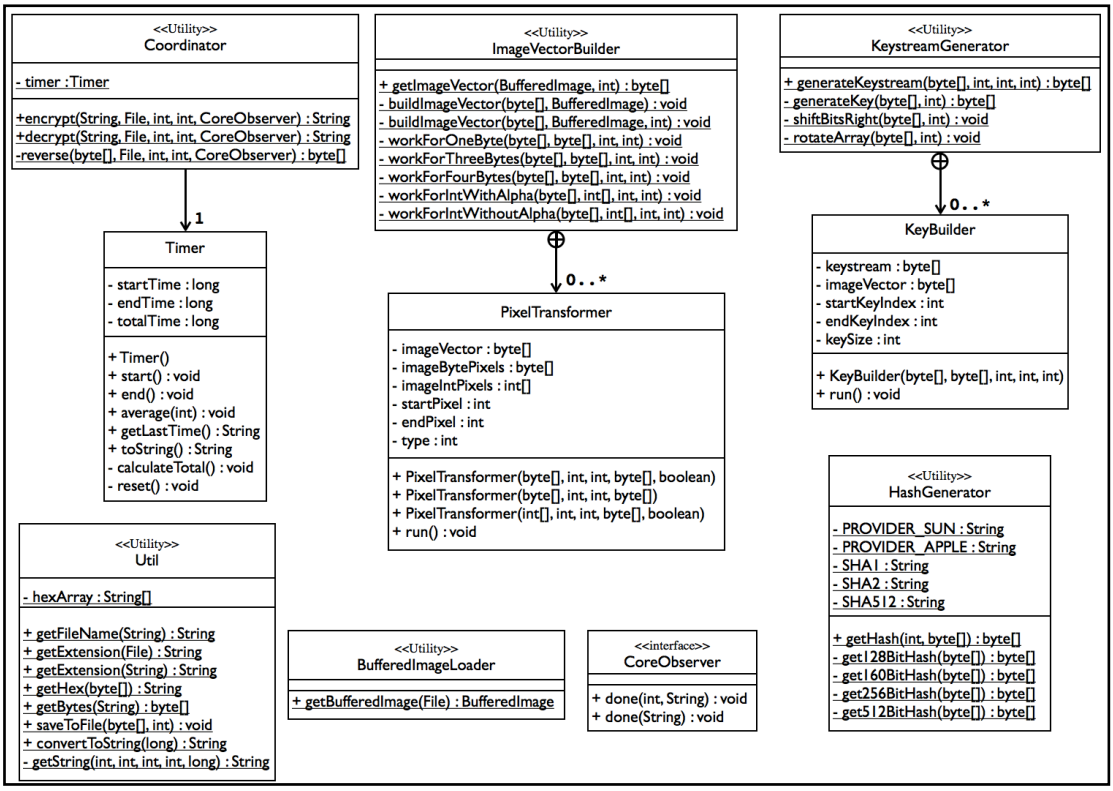

Figure 7. The class diagram for the core of the system.

4.2 Getting The Input Text Bytes

Getting the underlying bytes of string objects is an easy task in Java, which can be done by calling the instance method getBytes(String) of the String class. This method takes an optional string argument that can be used to indicate the encoding type and returns an array of bytes that represent the underlying bytes of the string text. We are always using UTF-8 as the encoding argument to make sure we support unicode characters in the system. The following line of code is used to access the underlying bytes of a string object:

```
byte[] plainTextBytes = plainText.getBytes("UTF-8");
```

We are using the above approach only in the case of encryption, that is, the input text is plaintext. In the case of decryption, we are assuming that the input text will be a sequence of hexadecimal characters, because that is the form we are using in the system to represent ciphertext, and so we used the following method[12] that converts hexadecimal values into an array of bytes:

```
static byte[] getBytes(String hex) {
 if (hex == null || hex.isEmpty())
  return null;

 int length = hex.length();
 byte[] data = new byte[length / 2];
 for (int i = 0; i < length; i += 2) {
  data[i/2] =(byte)((Character.digit(hex.charAt(i), 16)
                                                    << 4)
                + Character.digit(hex.charAt(i + 1), 16));
 }
 return data;
}
```

The number of text bytes depends on the input, if the input was plaintext and contains only ASCII characters then the number of bytes equals the number of characters in the input text. And if it was plaintext and it contained unicode characters then the number of bytes can be calculated as the number of unicode characters multiplied by 2, plus the number of ASCII characters. So an ASCII character is represented by a single byte in Java, and a unicode character is represented by 2 bytes.

If the input text was a sequence of hexadecimal characters then the number of bytes will be the number of hexadecimal characters divided by 2, this is because each byte is always represented by 2 hexadecimal characters, starting from the 00 hexadecimal value up to the FF hexadecimal value to represent the numbers from 0 to 255 respectively.

In either case, the number of text bytes can be easily determined by checking the number of elements in the bytes array as following:

```
int numberOfInputTextBytes = bytesArray.length;
```

4.3 Loading The Image

Loading and decoding images in Java is usually done using the ImageIO class located in the javax.imageio package. This class has a reliable static method read(File) that handles all the low-level work needed to decode the image. This method takes a file object that represents the image file and returns an instance of type BufferedImage, which we will be using later to gain access to the image's pixels and dimensions.

There are other ways for loading images, one way uses the JAI class from the Java Advanced Imaging API. This class is also convenient and requires only two lines of code to load the image and return a BufferedImage object, but based on the benchmarks that we performed (see section 5.1), we decided to use the ImageIO class, but we also fallback to the JAI class in cases where the ImageIO class cannot handle the input image's format (which is the case for tiff images).

We wrapped the task of loading images into a class called BufferedImageLoader which has a single static method that takes a File instance that represents the image file, and returns a BufferedImage instance as following:

```
static BufferedImage getBufferedImage1(File aFile)
                                          throws Exception {
 BufferedImage result = null;
 try {
  result = ImageIO.read(aFile);
 } catch (IOException ex) {
  result = null;
 }
```

```
 if (result == null) {
  // fallback to the JAI API
  PlanarImage pi =JAI.create("fileload",aFile.getPath());
  if (pi != null)
    result = pi.getAsBufferedImage();

  if (result == null)
    throw new Exception("Couldn't read the image at "+
                        Util.getFileName(aFile.getPath()));
 }
 return result;
}
```

Having loaded the image, we will next start building the image vector that will be used later as the secret key for generating the keystream.

4.4 Building The Image Vector

Having loaded the image in a BufferedImage instance, we are now going to build the image vector by applying a transformation function on the image's pixels. The final result of this step is a byte vector of length equal to the number of pixels, and this vector will represent the secret key in the coming steps. The transformation function is (red + green + blue) modulo alpha, and it is illustrated in figure 8.

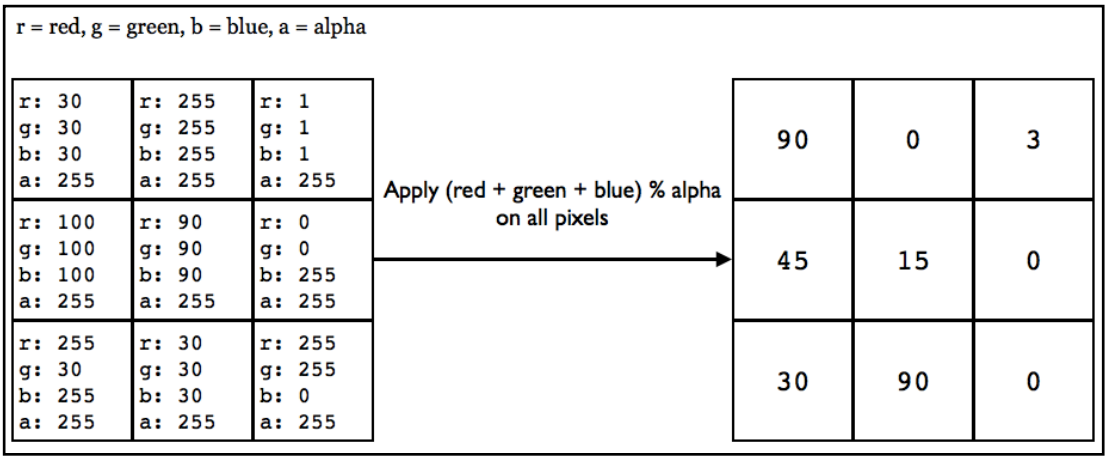

Figure 8. Applying the transformation function on the pixels of a 3X3 image.

To do the transformation, we will first need to gain access to the pixel's underlying values (i.e. the red, green, blue and alpha values). We found two ways in which we can do this using the BufferedImage instance.

The first and the most convenient way is to use the getRGB(int, int) instance method of the BufferedImage class, this method takes the x and y positions of a pixel, and returns the ARGB values combined into one 32-bit integer, which we can then separate into four bytes to get access to the individual alpha, red, green and blue values. For example, the following code can be used to get the ARGB values of the pixel at (0, 0) which represents the upper-left corner of the image:

```
int argb   = aBufferedImage.getRGB(0, 0);
byte alpha = (byte) ((argb >> 24) & 0xff);
byte red   = (byte) ((argb >> 16) & 0xff);
byte green = (byte) ((argb >> 8 ) & 0xff);
byte blue  = (byte) (argb         & 0xff);
```

The getRGB(int, int) method will always return all the ARGB values, even if the image does not have an alpha channel, in which case it will assume a full opacity and return an alpha value of 255. And for grayscale images that use a single byte for each pixel, it will return the same values for red, green and blue with full alpha. Finally, for indexed images, it will return the red, green and blue combinations for each particular index with an alpha value that may or may not be 255, depending on the image.

The second way to access the pixel's values is to access the data array that is associated with the DataBuffer object that is associated with the Raster object which is in turn associated with our BufferedImage object as in the following code:

```
WritableRaster raster = aBufferedImage.getRaster();
DataBuffer buffer     = raster.getDataBuffer();
byte[] pixels = ((DataBufferByte)buffer).getData();
```

The array will contain all the image pixels' values, which may or may not contain an alpha value depending on the image. So a single pixel may be represented by four adjacent elements in this array for images with an alpha channel, or it may be represented by three array elements for images without an alpha channel, or it may be represented by a single element for indexed or grayscale images.

Obviously, the second approach requires more work on our part as opposed to the first one, because we will need first to check the image's type to know how a single pixel is represented in the array, and then we will need an accurate way to process each of the pixel values individually based on the image type. The first method on the other hand requires only a nested for-loop that loops through all the x and y values of the image, and we will write code inside these loops similar to the one written above in the getRGB(int, int) example.

Since speed is a significant requirement of the system, we did a benchmark to compare the two approaches in processing an image (see section 5.1), to make sure that we select the one that is more efficient, rather than the more convenient to write, and based on the results we got, we decided to use the second approach to process the image's pixels.

Fortunately, the BufferedImage instance has a type instance variable that indicates how a single pixel is represented (e.g. is it a single byte or four bytes of alpha, red, green and blue?). We used this fact and implemented different methods to process the image’s pixels based on the image's type. The different types that an instance of the BufferedImage may have if it has been loaded using our approach are described in table 2.

Type	Description
TYPE_3BYTE_BGR	Represents an image with 8-bit RGB color components, corresponding to a Windows-style BGR color model) with the colors Blue, Green, and Red stored in 3 bytes.
TYPE_4BYTE_ABGR	Represents an image with 8-bit RGBA color components with the colors Blue, Green, and Red stored in 3 bytes and 1 byte of alpha.
TYPE_4BYTE_ABGR_PRE	Represents an image with 8-bit RGBA color components with the colors Blue, Green, and Red stored in 3 bytes and 1 byte of alpha. The color data in this image is considered to be premultiplied with alpha.
TYPE_BYTE_BINARY	Represents an opaque byte-packed 1, 2, or 4 bit image.
TYPE_BYTE_GRAY	Represents a unsigned byte grayscale image, non-indexed.
TYPE_BYTE_INDEXED	Represents an indexed byte image.
TYPE_CUSTOM	Image type is not recognized so it must be a customized image.

Table 2. The types of a BufferedImage instance and their descriptions[13].

We supported all the types in table 2, except for TYPE_BYTE_BINARY and TYPE_CUSTOM. We didn't support the binary type because we couldn't create or find an image of this type so we could test the code that deals with it. And we didn't support custom image types because we don't know how their pixels will be represented in the array.

If the image is of type TYPE_3BYTE_BGR, it will be processed by a method that assumes each pixel is represented by 3 elements in the array, and it will always use an alpha value of 255. The transformation done by this method is illustrated in figure 9, and the following code is used to do this transformation:

```
final int alpha = 255;
final int pixelLength = 3;
final int maxPixel = imgPixels.length - pixelLength;
for (int pixel=0, j=0;pixel<=maxPixel;pixel+=pixelLength,
                                                    j++){
   int rgbSum = 0;
```

```
    // sum the RGB color values of this pixel
    for (int i = 0; i < pixelLength; i++) {
        rgbSum += ((int) imgPixels[pixel + i] & 0xff);
    }
    imageVector[j] = (byte) (rgbSum % alpha);
}
```

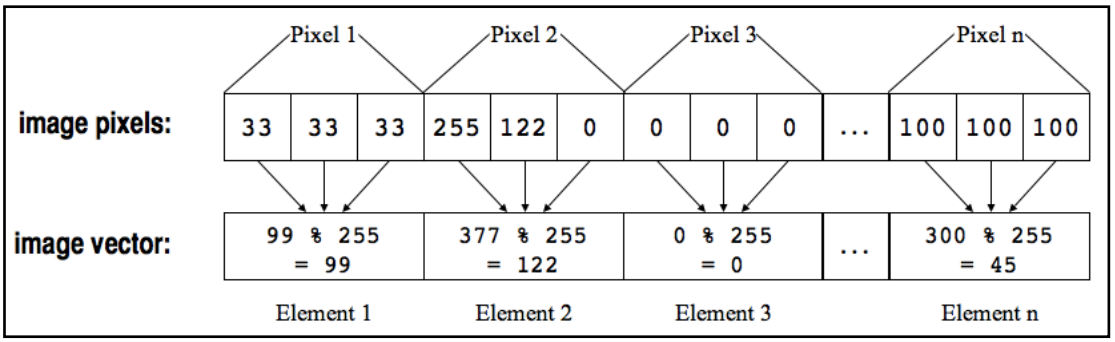

Figure 9. The transformation applied to an image of type TYPE_3BYTE_BGR.

The same transformation will be done on other image types, except that the pixel length (the number of array elements to represent a pixel) will be different, and the value of alpha may be different depending on whether the image has an alpha channel or not.

Grayscale images have one color for each pixel, and so what we do to get the RGB values is to set the grayscale value as the value of all RGB colors, as this is the common way for converting a grayscale image into an RGB image.

Finally, indexed images will need a different mechanism for accessing the RGB values, because the pixels' array contains the indices of the colors, not the color values. So we are using the getRGB(int) method of the ColorModel instance that is associated with our BufferedImage instance to get the RGB colors of a particular index, and then convert the indices array into an array similar to the one associated with TYPE_4BYTE_ABGR if the image has an alpha channel, or otherwise into an array similar to the one for TYPE_3BYTE_BGR.

4.5 Generating The Keystream

The keystream consists of multiple keys combined into one byte array. These keys have the same size, which may be 128-, 256- or 512-bits, which is specified by the user from the user interface. The number of keys needed is determined by the number of the input text bytes and the key size specified, and can be calculated using the following pseudocode:

```
if numOfBytes mod keySizeInBytes == 0
 numOfKeys = numOfBytes / keySizeInBytes
else
 // We need one more key for the remaining bytes
 numOfKeys = 1 + (numOfBytes / keySizeInBytes)
```

Having determined the number of keys needed, we start by generating the first key, which consists of two steps. The first is to get the hash value of the image vector using one of the Secure Hash Algorithms (SHA), depending on the key size specified. The second step is to combine the image vector elements with the corresponding hash elements using exclusive-or. The hash elements may be combined with more than one element from the image vector if the image vector's length is greater than the hash length. For example, if there are 17 elements in the image vector and 16 elements in the hash, the first element in the hash will first be XORed with the first element in the image vector, and the result will be XORed again with the 17th element in the image vector. The final result of the XOR operations is the first key in the keystream. This process is done using the following code:

```
// 1. hash imageVector
byte[] hash = HashGenerator.getHash(keySize,imageVector);

// 2. XOR the hash with imageVector
for (int i = 0, j = 0; i < imageVector.length; i++, j++)
{
   if (j == hash.length)
      j = 0;
   hash[j] = (byte) (hash[j] ^ imageVector[i]);
}
return hash;
```

The remaining keys will be generated in a similar manner, except that we will first perform a 1-bit circular right-shift to the image vector before getting its hash value. The process of generating the full keystream is illustrated in figure 10.

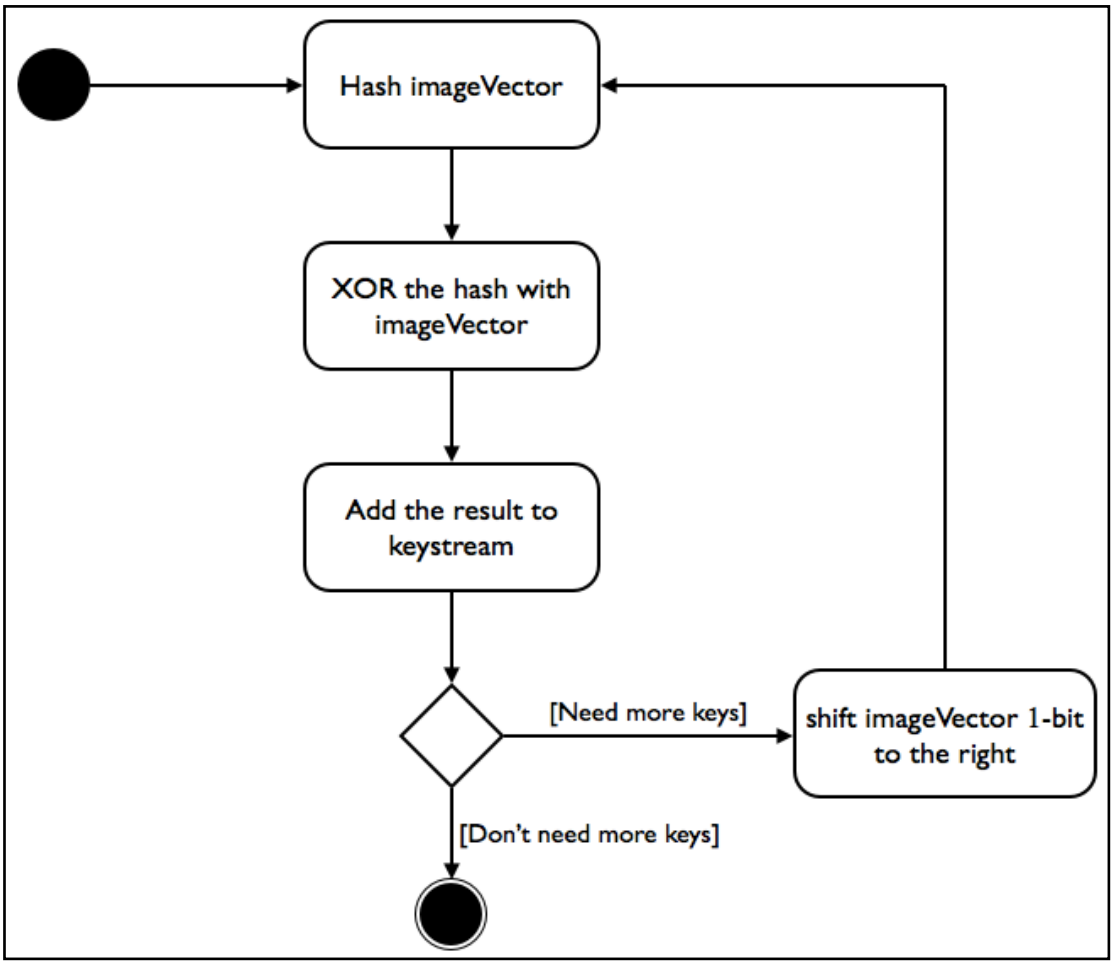

Figure 10. Generating the keystream.

A bit-wise circular right-shift simply means moving the bits in the array one step to the right, and the right-most bit becomes the left-most bit in the array. This is done to ensure that each key is different from the other keys in the keystream, because a 1-bit shift gives us new values for the elements in the image vector, which in turn gives us new keys to add to the keystream in each round. The code used for shifting the bits of the image vector is the following:

```
// 1. make the right-most bit become the left-most bit.
byte previousByte = imageVector[0];//keep the byte before
                                   //shifting its bits
imageVector[0] = (byte) (((imageVector[0]&0xff)>>1) |
          ((imageVector[imageVector.length-1]&0xff)<<7));

// 2. shift the remaining bits.
for (int i = 1; i < imageVector.length; i++) {
  byte tmp = imageVector[i];
  imageVector[i] = (byte) (((imageVector[i]&0xff)>>1) |
                                ((previousByte&0xff)<<7));
  previousByte = tmp;
}
```

Getting a hash value in Java is achieved using the MessageDigest class in the java.security package. This class has a static method getInstance(String, String) that takes the name of the algorithm required and an optional argument for the name of the provider of the algorithm (e.g. Sun, Bouncy Castle, etc.) and returns a MessageDigest instance that can be used to get hash values. We will be using SHA-1, SHA-2 and SHA-512 as the algorithm name for the key sizes 128-bits, 256-bits and 512-bits respectively. The SHA-1 algorithm returns a 160-bit hash, so we will truncate this hash to 128-bits when the key size is 128-bits. As for the provider argument, it specifies the name of the provider of the implementation of the algorithm. We found three implementations for the SHA-1 algorithm, they are provided by Apple, Bouncy Castle and Sun, for the other algorithms we will use the default implementation provided by Sun. There are slight differences in the performances of the different implementations of SHA-1 as we will see in section 5.2, and based on the results we obtained, we chose to use Apple's implementation as the default one, and we fallback to Sun's implementation if Apple's implementation wasn't installed on the machine that runs the system.
We implemented a class HashGenerator that wraps the hashing functionality. It has a single static method that takes a key size in bits and

an array of bytes, and then it will hash the array using the algorithm for the specified key size, and then returns that hash.

4.6 Encryption / Decryption

Since we are implementing a stream cipher, the final encryption or decryption processes are exactly the same. We combine the input bits with the keystream bits using an exclusive-or operation to produce the final bits as illustrated in figure 11. The input bits may represent the plaintext in case of encryption or the ciphertext in case of decryption, and the final bits will represent either the ciphertext in the case of encryption or the plaintext in the case of decryption.

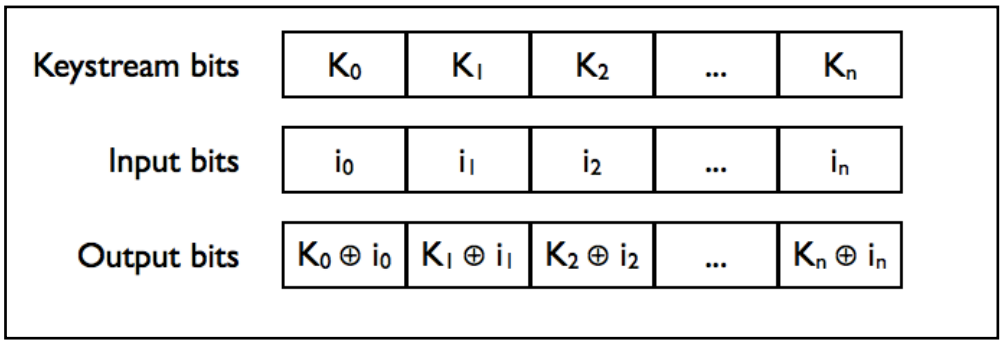

Figure 11. Performing an exclusive-or between the keystream and the input bits.

The exclusive-or operation is simply done by looping through all the input text bytes, and in each loop we perform a XOR operation between the current text byte with a byte in the keystream. This is done by the following code:

```
for (int i = 0; i < inputBytes.length; i++) {
   inputBytes[i] = (byte) (keystream[i] ^ inputBytes[i]);
}
```

We are simply looping through all the input text bytes, and in each loop, we are changing the input text bytes to the result of the XOR operation between the keystream byte and the input text byte. We could have

allocated new bytes for the result, but that would unnecessarily add to the memory requirements of the system.

4.7 Converting The Bytes Back To A String

After the XOR operation, we need to convert these final bytes into a readable form. In the case of encryption, since we cannot guarantee that the ciphertext bytes can be mapped into valid ASCII or unicode characters, we need to represent these bytes by a readable format that can be easily transferred, and can be easily converted back to its byte equivalent. We chose to use the hexadecimal values of these bytes as the representation of the ciphertext, and we implemented a simple look-up table to do this conversion as fast as possible by the following method:

```
static String getHex(byte[] b) {
 StringBuilder hexString = new StringBuilder(b.length*2);
 for (int i = 0; i < b.length; i++) {
   hexString.append(hexArray[b[i]&0xff]);
 }
 return hexString.toString();
}
```

Where hexArray is a lookup table of length 256, where the element at index 0 is the hexadecimal value 00, and the element at index 255 is the hexadecimal value FF.

In the decryption case, we will assume that the resulting bytes are valid ASCII or Unicode characters (i.e. the user has chosen the same image used for the encryption), and so we will map these bytes to their respective characters. This mapping is done in Java using a constructor of the String class that takes a byte array and the encoding name as the arguments, and returns a string object that contains the original characters. If the bytes cannot be mapped into valid characters (e.g. if the user is trying to decrypt the text using a different image that has been

used in the encryption) then the string will contain some unreadable characters or a string that doesn't make sense. The conversion of the plaintext bytes into a string is simply done by the following code, where originalBytes represents the resulting bytes of the XOR operation:

```
String plainText = new String(originalBytes, "UTF-8");
```

4.8 Exploiting Multithreading

We identified two parts of the system that we may speedup using multithreading. The first is building the image vector, and the second is generating the keystream. The unit of work that a single thread can work on in the image vector building step is a pixel, so the number of threads specified cannot exceed the number of pixels in the image, and the unit of work in the keystream generation that a single thread can work on is generating a single key, so the number of threads cannot exceed the number of keys that will be generated.

The division of work among the threads is simply done by dividing the remaining work (unprocessed pixels or not-generated keys) over the remaining number of threads that haven't started yet. So in building the image vector, if the image has 17 pixels and we wanted them to be processed by 3 threads for example, the first thread will work on the first 5 (17 over 3) pixels, the second thread will work on the next 6 (12 over 2) pixels, and finally the last thread will work on the last 6 (6 over 1) pixels as illustrated in figure 12, this is done by the following code:

```
int remainingWork = imageVector.length;
int endRange = -1;
for (int i = 0; i < numOfThreads; i++) {
  //num of threads that haven't started == numOfThreads-i
  int amountOfWork = remainingWork / (numOfThreads - i);
  remainingWork -= amountOfWork;
  int startRange = endRange + 1;
  endRange       = startRange + amountOfWork - 1;
  PixelTransformer pt=new PixelTransformer(imgPixels,
                       startRange, endRange, imageVector);
```

```
  threads[i] = new Thread(pt);
  threads[i].start();
}
```

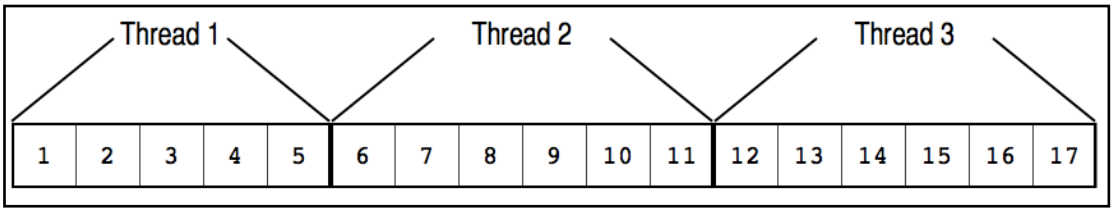

Figure 12. Dividing the image vector building work among 3 threads.

Generating a key in the keystream generation step is not a purely independent task, because in generating each key we are bit-shifting the image vector, so the next key we generate will depend on the state of the image vector in the current round.

However, since the number of bit-shifts that has been performed on the original image vector equals the number of key that we want to generate minus one, we can use this fact and make the non-first threads start by performing the required bit-shifts on the original image vector first and then continue working normally on the remaining keys.

For example, if we needed to generate 15 keys and we wanted to generate them on 3 threads, we will divide the work in a similar manner to the work division done in building the image vector. Now when the first thread starts working, it will work normally as if we were running this step on a single thread, but the second and third threads will have a small initialization to do. Since the second thread will start working on the sixth key, we know that the original image vector should now be shifted by a total of 5 bits, so it starts by performing this number of bit-shifts and then continues to work on the remaining keys normally. The third thread will do a similar thing, since it starts on generating the eleventh key, we know that the original image vector should now be shifted by 10 bits, so it will

start by performing these bit shifts on the image vector and then continues working normally.

If a thread starts by generating the 90th key in the keystream, it will need to perform 89 bit shifts on the image vector, which makes the whole point of multithreading pointless if the phrase was taken literally. But since performing 8 bit shifts on an array of byte elements is the same as rotating the array by one element, we can use this fact and make a big shortcut to the process. Instead of performing 89 bit shifts to the array, we will first rotate it by 11 (89 / 8) elements in a one method call, and then perform 1 (89 % 8) bit shift to the image vector.

The approach we've taken for multithreading the keystream generation step has the implication that each thread must have its own copy of the image vector, and this will increase the memory used by the system.

Since the image vector is a byte array that has a number of elements equal to the number of pixels in the image, this array will usually occupy more than 1 megabytes of memory, and so we need to make sure that we are not requesting more than the available memory that we can use in the java virtual machine, otherwise an OutOfMemoryError will be raised. To do this we are checking the available memory using the Runtime class in Java, and then we may decrease the number of threads for generating the keys based on the number of elements in the image vector and based on the available memory. The following is the code we are using to perform this check:

```
//invoke the garbage collector to make accurate estimates
System.gc();
Runtime runtime = Runtime.getRuntime();
long usedMem = runtime.totalMemory()-runtime.freeMemory();
long availableMemory = runtime.maxMemory() - usedMem;
// add 1 thread to max because the first thread will work
```

```
// on the original copy that has been already allocated
int maxThreads  =(int)(availableMemory/imageVector.length
                                                    + 1);
if (numOfThreads > maxNumOfThreads) {
   numOfThreads = maxNumOfThreads;
}
```

We could have applied multithreading on other parts of the system, but these parts are lightweight tasks, and can be performed very fast on a single thread, and so applying multithreading to these tasks may actually slow them down instead.

To explain what a heavy task is, we will give an example of a non-heavy or a lightweight task. One such task in our system is the encryption / decryption step. We are expecting in our system to deal with input text bytes that are at maximum 500 thousand, which represent 500 thousand ASCII characters or 250 thousand unicode characters, and the average time to perform this step on 1 million bytes on a single thread is 1 millisecond as per the benchmarking we've done (see section 5.3). If we tried to multithread this task, the task will take longer than if it was performed on a single thread. We will write more on this subject in section 5.3.

Chapter 5: Implementation

5.1 System Requirements

The system requires:

- any operating system that has Java 6 or later installed;
- at least 1 GHz of clock speed.
- at least 128 MB of RAM.
- 10GB of free disk space.

5.2 The Graphical User Interface

The GUI that we developed is a simple interface to demonstrate the system and its performance (the time it takes to perform a particular task).

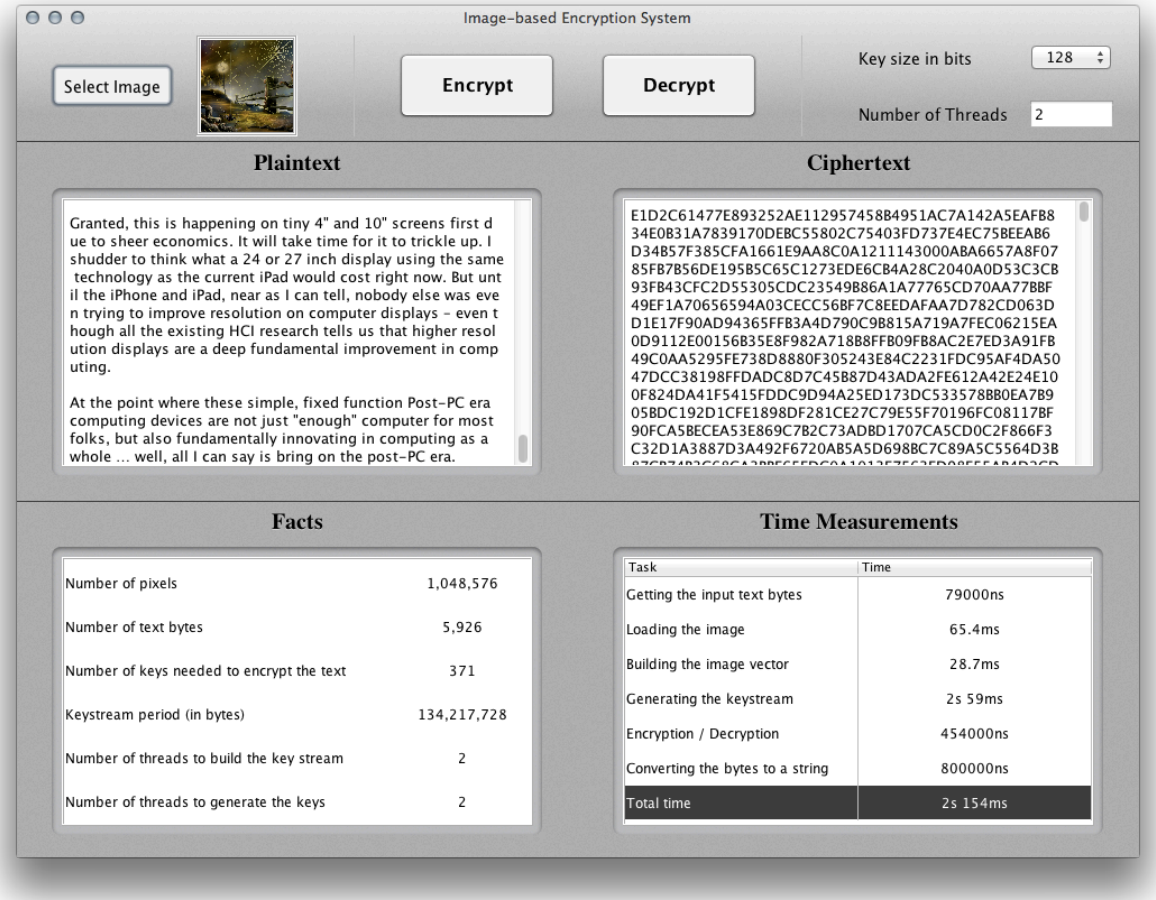

Figure 13. A screenshot of the user interface after encrypting some text.

The user starts by selecting an image from his/her computer, and enters some plaintext in the plaintext area, or ciphertext in the ciphertext area, and specifies a key size and number of threads and then selects either the encrypt or decrypt button.

The key size parameter will be used to identify which SHA algorithm to use, and consequently, to specify the length of one key in the keystream. So for the key size 128-bits, SHA-1 will be used, and for the key size 256-bits, SHA-2 will be used, and finally, SHA-512 will be used if 512-bits was specified as the key size.

The number of threads will be used to determine the number of threads that will be used in building the image vector (see section 4.4) and in generating the keystream (see section 4.5). The number of threads specified is not necessarily the same as the actual number used in those steps, because as we described in section 4.8, the number of threads cannot exceed neither the number of pixels nor the number of keys that will be generated, so in these cases the algorithm may reset the number of threads to the number of pixels or number of keys respectively. Another case where the number of threads will be changed is in the key generation step, because each thread must have its own copy of the image vector, the number of threads may be reduced based on the available runtime memory. Thus, we included the actual number of threads that will be used in the facts table.

The facts table displays two facts about the keystream that will be generated. The first is the number of keys that need to be generated to build the full keystream, this depends on two factors: the key size specified, and the number of text bytes entered. The second fact is the period of the keystream in bytes, which is the number of bytes in the

keystream before it starts repeating again. The period of the keystream depends on the key size specified and the number of pixels in the image. The number of bits in the period can be calculated by multiplying 8 by the number of pixels by the key size in bits.

When the user clicks either the encrypt or decrypt buttons, the UI will first validate the inputs before asking the Coordinator to perform the required task. The validation consists of checking that an image has been selected and the corresponding text area is not empty. In the case of decryption, it will also inspect the ciphertext to confirm that it consists of only hexadecimal characters (using a regular expression), and that the number of characters are multiples of two (since a byte is represented by 2 hexadecimal characters). And finally it will check the number of threads entered to confirm that it is an integer greater than 0.

If the input has passed the validation process, the UI will ask the Coordinator class to perform the required task based on the inputs on a background thread (to keep the UI responsive). Otherwise it will notify the user that the input(s) are not valid with a specific reason.

In order for the UI to keep track of the work done by the Coordinator class and the time taken for each task, we used the Observer design pattern. We defined an interface called CoreObserver that defines two methods, the first will be used for notifications about single tasks (such as loading the image), and the second will be used for notifications about the overall work. We made the UI create an object that implements this interface, and then pass a reference to this object to the Coordinator, and the Coordinator will then use the methods of the interface to notify this object about its progress.

Chapter 6: Performance Analysis

6.1 Overall System Performance

The machine we ran the benchmarks on has a 1.86 GHz Intel Core 2 Duo processor that runs a Mac OS X with 4 GB of memory. We always use the JVM arguments -Xms2048MB and -Xmx2048MB which specify an initial and a maximum of 2 GB of memory for the heap, this is intended to try to minimize the number of times in which the garbage collector does a full garbage collection, which affects the benchmark results.

In chapter 4, we mentioned that we made some decisions based on performance reasons in two components of the algorithm. The first was the decision of loading images using the ImageIO class over the JAI class, and as you can see in figure 14, the ImageIO class is about 30% faster than the JAI class. The second decision was to access the image pixels using the underlying data buffer instead of using the getRGB(int, int) instance method of the BufferedImage class, and as we can see in figure 15, accessing the underlying data buffer is about 90% faster than using the getRGB(int, int) instance method of the BufferedImage class.

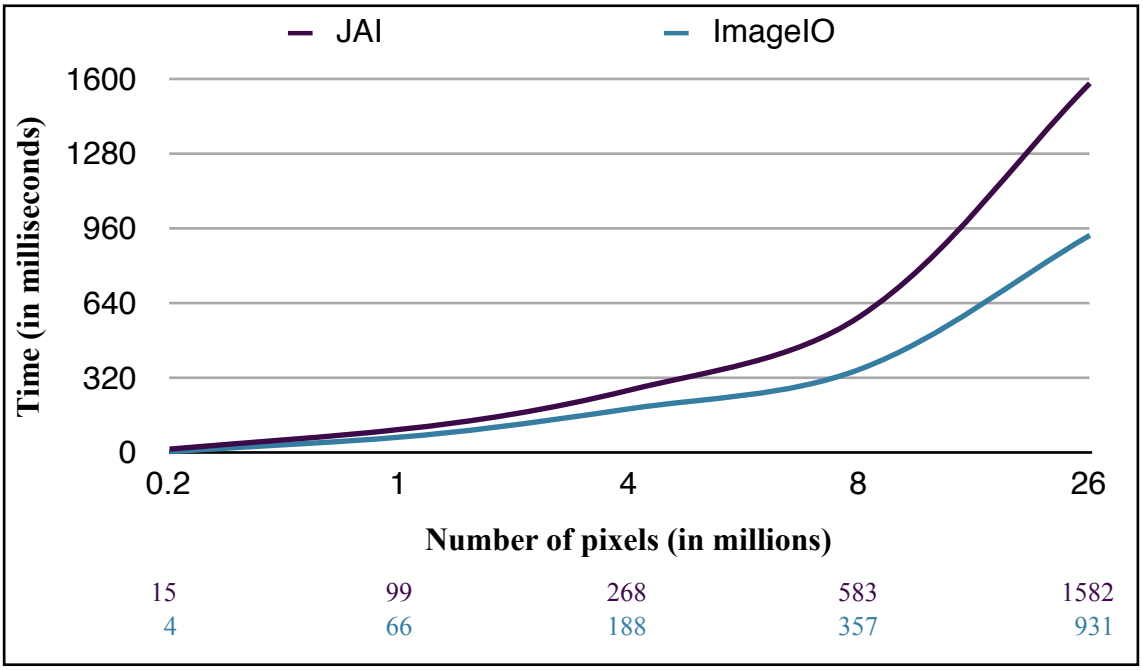

Figure 14. Performance comparison between the ImageIO and the JAI classes

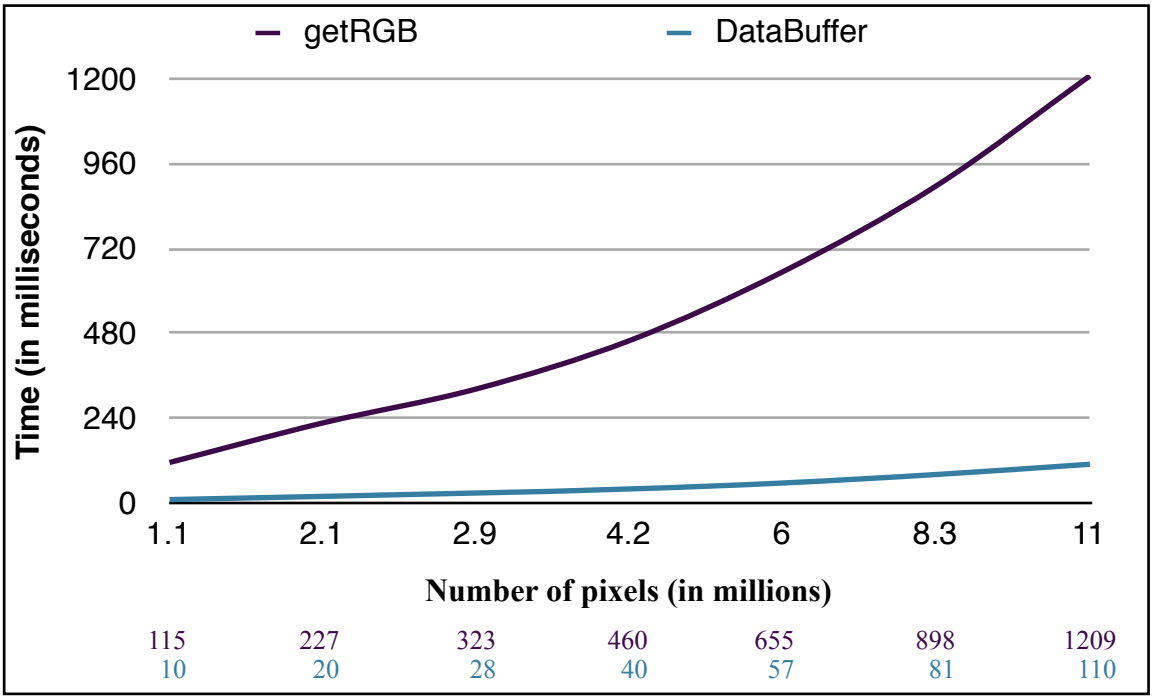

Figure 15. Building the image vector using getRGB versus accessing the data buffer directly.

To analyze the current performance of the system, we used the profiler that comes with the Netbeans IDE to find out the hotspots (i.e. the methods) in the algorithm, in which most of the time is spent.

To find out where does the system spend most of its time, we profiled the process of encrypting 10KB of text using an image with about 26 million pixels and a 128-bit key size on one thread, and as we can see in table 3, 99% of the time was spent on only three methods, and they are all parts of generating the keystream. About 51% of the time was spent in the static method get128BitHash(byte[]) in the HashGenerator class which gets a 128-bit hash value of the image vector, and about 27% of the time was spent in the XOR(byte[], byte[]) static method of the KeystreamGenerator class which performs an exclusive-or between the hash value and the image vector, and about 21% of the time is spent in the shiftBitsRight(byte[], int) static method which performs a circular right-shift to the bits of the image vector.

Class name	Method signature	%
HashGenerator	get128BitHash(byte[])	51.4%
KeystreamGenerator	XOR(byte[], byte[])	26.9%
KeystreamGenerator	shiftBitsRight(byte[], int)	20.5%
BufferedImageLoader	getBufferedImage(File)	0.8%
ImageVectorBuilder	workForThreeBytes(byte[], byte[], int, int)	0.3%
ImageVectorBuilder	getImageVector(BufferedImage, int)	0.01%
KeystreamGenerator	generateKeystream(byte[], int, int, int)	0.01%

Table 3. The percentage of time spent on the core methods for encrypting 10KB of text using an image with 26 million pixels and a key of size 128-bits on 1 thread.

In tables 4 through 7 we included more benchmark results that support this claim, we excluded from these tables the other algorithm components because they always take less than 0.1% of the time when encrypting 10KB of text. In all of these benchmarks we used three different images to encrypt 10 kilobytes of text, but we changed the key size and number of threads parameters between these benchmarks to illustrate their effect on the total time taken.

In tables 4 and 5, we used a key size of 128-bits in both benchmarks, but we used 1 thread in the results in table 4, and 2 threads in table 5. We can see here the effect of multithreading in the total time taken in these tables. For the image with 1 million pixels, the total time taken using 1 thread was 4.968 seconds, and for 2 threads it was 2.789 seconds, which is faster by 44%. The same applies for the other images, the total time taken was reduced by 44% when we used 2 threads for an image with 26 million pixels, and it was reduced by 45% for the image with 144 million pixels.

Number of pixels	1,048,576		26,214,400		144,000,000	
Algorithm component	**Time (s)**	**%**	**Time (s)**	**%**	**Time (s)**	**%**
Loading the image	0.062	1.2	0.938	0.7	4.873	0.6
Building the image vector	0.015	0.3	0.416	0.3	3.558	0.4
Generating the keystream	4.891	98.5	131.96	99.0	796.18	99.0
Total	4.968	100.0	133.32	100.0	804.61	100.0

Table 4. Profiling results for encrypting 10KB of text on 1 thread using 128-bit keys.

Number of pixels	1,048,576		26,214,400		144,000,000	
Algorithm component	**Time (s)**	**%**	**Time (s)**	**%**	**Time (s)**	**%**
Loading the image	0.062	2.2	0.938	1.3	4.873	1.1
Building the image vector	0.015	0.5	0.160	0.2	0.819	0.2
Generating the keystream	2.712	97.2	73.922	98.5	435.28	98.7
Total	2.789	100.0	75.02	100.0	440.98	100.0

Table 5. Profiling results for encrypting 10KB of text on 2 threads using 128-bit keys.

In the tables 6 and 7 we performed the same benchmarks as those in tables 4 and 5, we used 1 thread for the benchmarks in table 6 and 2 threads for table 7, but this time we changed the key size to 512-bits. We can see again the effect of multithreading in these tables, which reduced the total times in table 7 by more than 40% than the total times in table 6.

Number of pixels	1,048,576		26,214,400		144,000,000	
Algorithm component	**Time (s)**	**%**	**Time (s)**	**%**	**Time (s)**	**%**
Loading the image	0.062	2.8	0.938	1.7	4.873	1.6
Building the image vector	0.015	0.7	0.416	0.7	3.558	1.2
Generating the keystream	2.153	96.5	54.409	97.6	300.74	97.3

Total	2.23	100.0	55.763	100.0	309.17	100.0

Table 6. Profiling results for encrypting 10KB of text on 1 thread using 512-bit keys.

Number of pixels	1,048,576		26,214,400		144,000,000	
Algorithm component	**Time (s)**	**%**	**Time (s)**	**%**	**Time (s)**	**%**
Loading the image	0.062	4.8	0.938	3.2	4.873	3.0
Building the image vector	0.015	1.2	0.160	0.5	0.819	0.5
Generating the keystream	1.226	94.1	28.507	96.3	155.97	96.5
Total	1.303	100.0	29.605	100.0	161.66	100.0

Table 7. Profiling results for encrypting 10KB of text on 2 threads using 512-bit keys.

We can also see the effect of the key size on the total time by comparing the results in tables 4 and 6, or tables 5 and 7. When we used a 512-bit key size in table 6, the total times were less than those in table 4 (which use a 128-bit key size) by more than 55%. The same applies on the total times in tables 5 and 7, when we used 512-bits as the key size in table 7, the total times were less than those in table 5 by more than 50%.

From the results in tables 3 through 7, we can conclude that generating the keystream is the component that has the most effect on the overall performance of the system. Hence, we will analyze this component more closely in the next section.

6.2 The Stream Cipher's Performance

We will start by analyzing the throughput of the stream cipher, but to do this, we needed first to modify the code a little by making the keystream generator generate new bytes inside the encryption loop instead of pre-generating the full keystream as before. This is to help us simulate the encryption of a continuous data, in which a new keystream byte is

generated as new data bytes come in. This has the implication that we cannot multithread this process, since we will be generating one key at a time.

As we can see in figure 16, the throughput of the stream cipher is highly related to the number of pixels in the image, in which a smaller number of pixels gives a higher throughput and vice versa. It is also affected by the key size used, since a key size of 512-bits is equal to four keys of size 128-bits, hence the total number of keys that will be generated will be a quarter of the number of keys in the case of a 128-bit key.

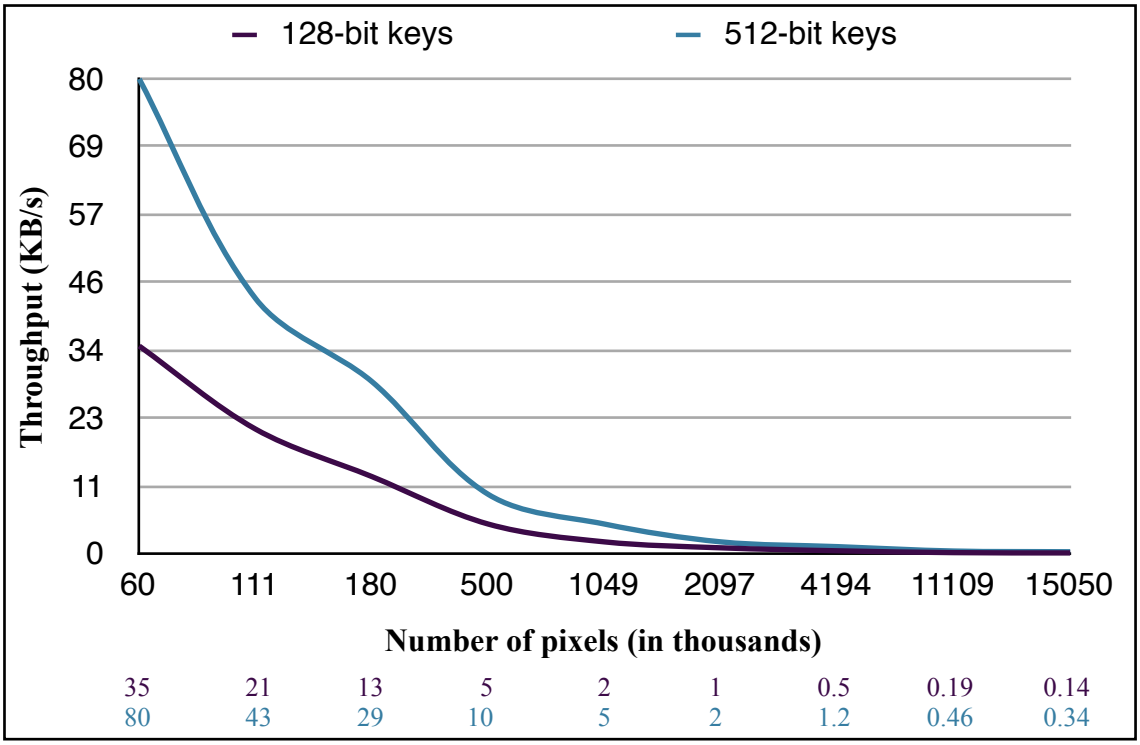

Figure 16. The stream cipher's throughput in kilobytes per second with 128- and 512-bit keys in relation to the number of pixels.

So where exactly does the stream cipher spend most of its time? as we can see in the profiling results in tables 8 and 9, it is the hash functions that take up most of the time, and then performing the exclusive-or operation between the hash and the image vector.

Class name	Method signature	Time (s)	%
HashGenerator	get128BitHash(byte[])	247.867	52.7%
KeystreamGenerator	XOR(byte[], byte[])	141.085	30.0%
KeystreamGenerator	shiftBitsRight(byte[], int)	81.539	17.3%
KeystreamGenerator	getNewByte()	0.073	0.02%
KeystreamGenerator	generateKey(byte[], int)	0.051	0.01%
HashGenerator	getHash(int, byte[])	0.013	0.00%
Total		**470.628**	**100.0%**

Table 8. Profiling results for the stream cipher for encrypting 1 MB of data using an image with about 1 million pixels and 128-bit keys.

Class name	Method signature	Time (s)	%
HashGenerator	get512BitHash(byte[])	167.089	77.7%
KeystreamGenerator	XOR(byte[], byte[])	27.422	12.8%
KeystreamGenerator	shiftBitsRight(byte[], int)	20.373	9.5%
KeystreamGenerator	getNewByte()	0.061	0.03%
KeystreamGenerator	generateKey(byte[], int)	0.019	0.01%
HashGenerator	getHash(int, byte[])	0.008	0.00%
Total		**214.972**	**100.00%**

Table 9. Profiling results for the stream cipher for encrypting 1 MB of data using an image with about 1 million pixels and 512-bit keys.

The difference in the percentages of the get128BitHash and the get512BitHash methods in tables 8 and 9 is related to the hash functions themselves, because as illustrated in figure 17, SHA-1 (which is used in get128BitHash) is approximately two times faster than SHA-512.

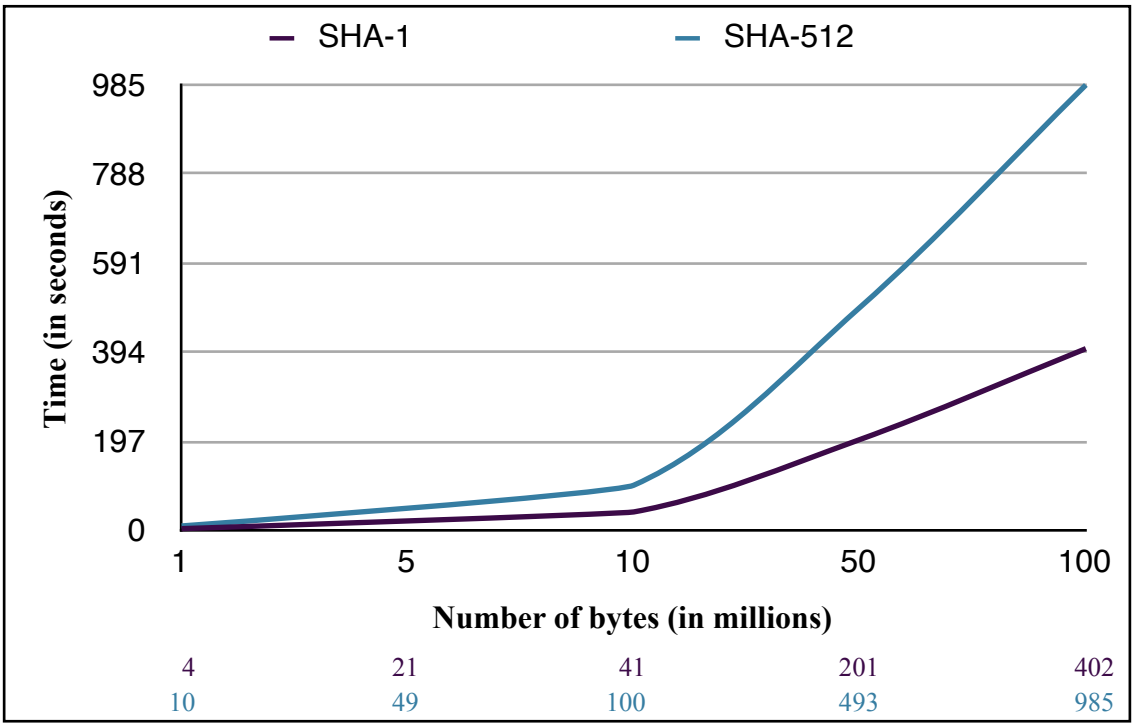

Figure 17. SHA-1 and SHA-512 performance in relation to the number of input bytes

In section 4.5, we mentioned that we chose Apple's implementation for the SHA-1 algorithm over Sun's and Bouncy Castle's implementations because it was faster, and as we can see in figure 18, Apple's implementation for the SHA-1 algorithm is more than 50% faster than Sun's implementation, and is about 66% faster than the implementation of Bouncy Castle.

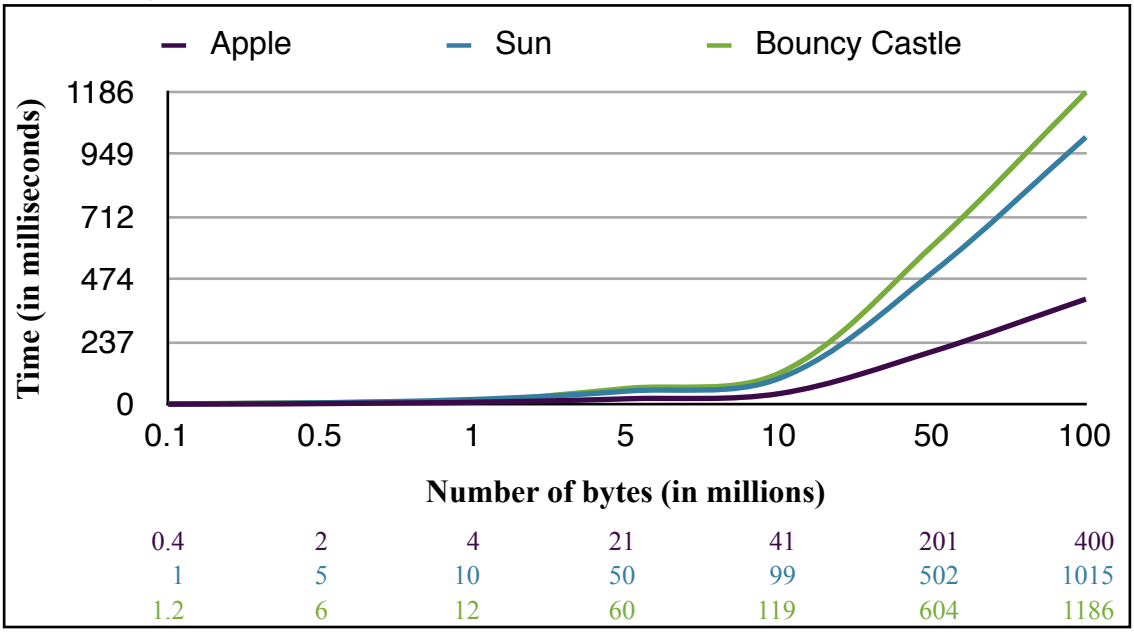

Figure 18. Performance comparison between Apple's, Sun's and Bouncy Castle's implementations of the SHA-1 algorithm.

We may be able to improve the performance by providing our own implementation of the hash functions and customizing them in two ways. The first, since object allocation is expensive[14], is to make them allocate as few objects as possible. The second is to make them deal with and return an array of long variables instead of byte, and also apply this customization on the image vector. The second customization means that we will have less array elements, since a long variable is 64-bits, one long variable can hold 8 bytes, this leads to less loops to perform (exactly eighth the loops currently performed), which would make the XOR and the shifting methods about eight times faster.

6.3 Multithreading

In section 4.8, we talked about how we used multithreading to speed up some parts of the system, we also said that multithreading may actually decrease the time for lightweight tasks. An example of a lightweight task in our system is the encryption / decryption step, and as we can see in figure 19, multithreading can be useful in this step only when the number of text bytes exceeds a threshold which makes the task a heavy one, which is approximately 20 million bytes.

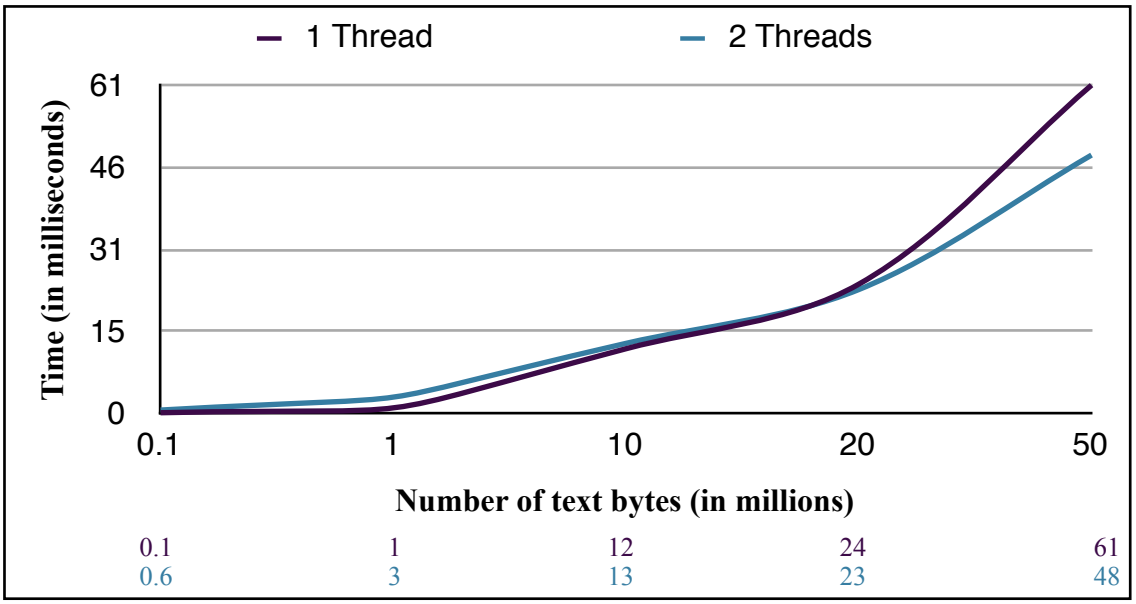

Figure 19. Performance comparison for running the encryption step using 1 thread versus 2 threads.

Actually, this applies too on the tasks that we applied multithreading to. Building the image vector step for example, may actually be performed faster on a single thread than on multiple threads in some cases as illustrated in figure 20. This happens when the number of pixels is relatively small and hence the task of processing them becomes a lightweight task. The same thing applies to generating the keystream, if we needed only two keys for example, a single thread may generate them faster than two threads, because the task is now small, and hence it became a lightweight task.

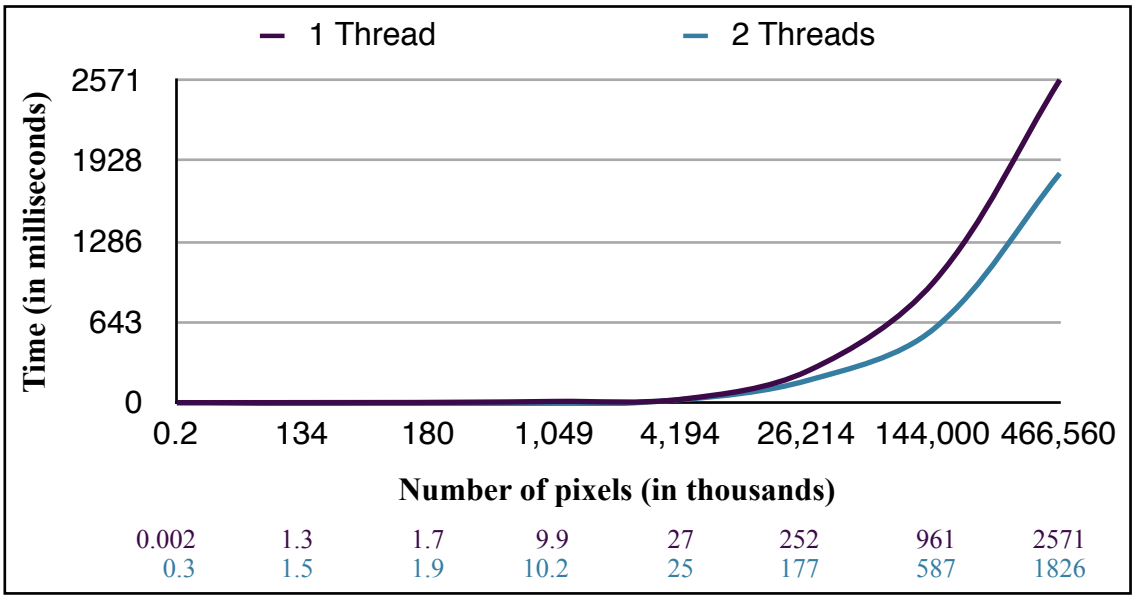

Figure 20. Performance comparison of building the image vector on 1 thread versus 2 threads.

Now, we want to find out the optimal number of threads, that is, the number of threads which gives the best performance. We will perform the tests on building the image vector step using an image with 11 million pixels to make the task a heavy one.

Of course, there is no such magic number that would give the best performance on all platforms, so we tried to find the optimal number on three different platforms and then observe the results. As we can see in figure 21, two threads seems to be always better than a single thread, this

may or may not be true for a machine that has a single processor with a single core.

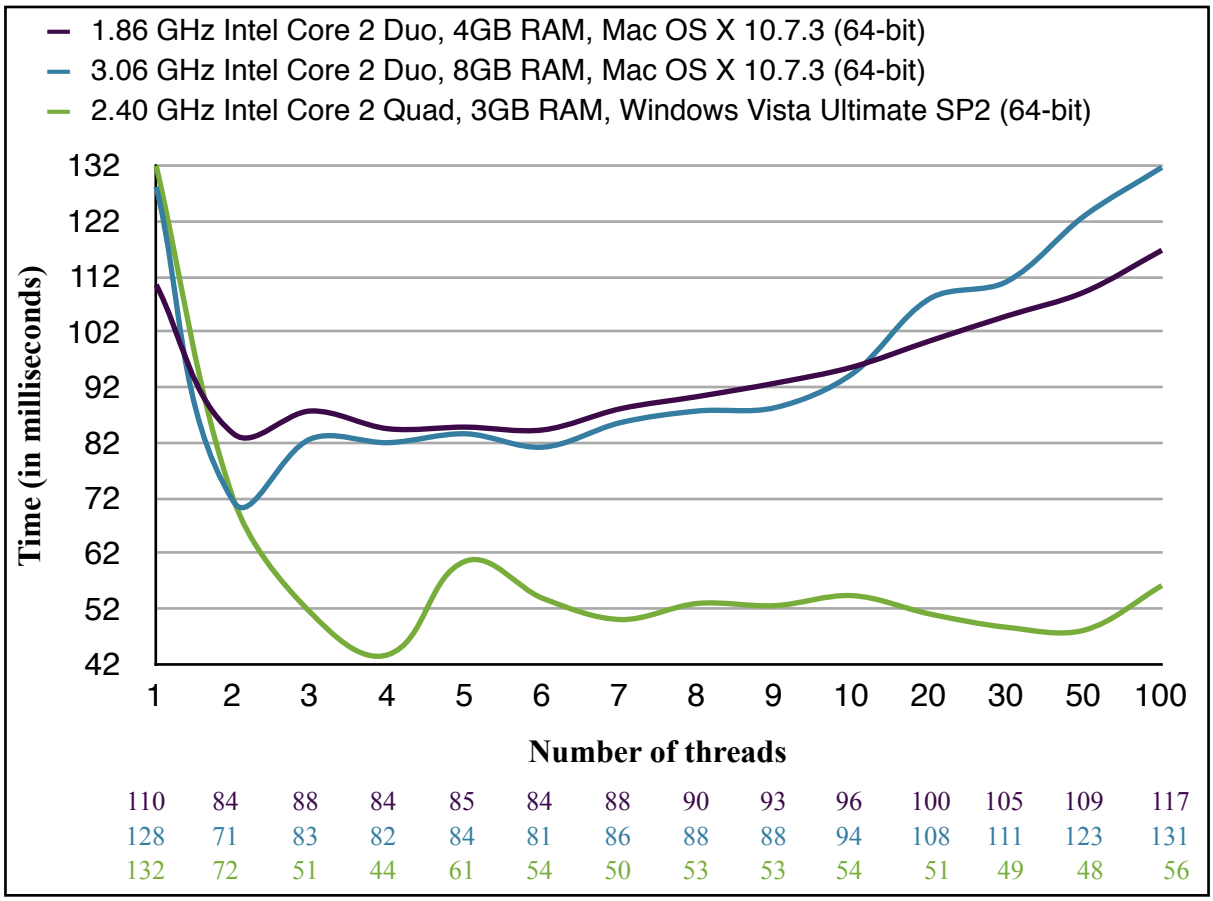

Figure 21. Comparing the time taken for building the image vector from an image with 11 million pixels against the number of threads on different machines.

For the machine with a 1.86 GHz Core 2 Duo processor, 2 threads gave the best performance, which was 83.5 milliseconds. And the one with a 3.06 GHz Core 2 Duo processor, 2 threads also gave the best performance, which was 71.3 milliseconds. Finally, for the machine that has a 2.4 GHz Core 2 Quad processor, 4 threads gave the best performance, which was 43.8 milliseconds.

For these machines, the optimal number of threads seems to be equal to the number of cores in the machine's CPU. So for the machines that has a core 2 duo processor, 2 threads seems to be the optimal number of

threads, and for the machine that has 4 cores, which is the case for a Core 2 Quad processor, 4 threads seems to be the optimal number.

Chapter 7: Closure

7.1 Conclusions

In this report, we have shown how we implemented an image-based encryption system. We showed that the performance of the system depends on the keystream generator, which in turn depends on the number of pixels in the image and the key size specified. The throughput of the stream cipher varies from 80 KB per second using a key size of 512-bits and an image with 60 thousand pixels, down to only 144 bytes per second using a key size of 128-bits and an image with 15 million pixels. We have also shown that the optimal number of threads equals the number of cores available in the CPU(s).

7.2 Recommendations

For the best performance, we recommend to always use 512-bits as the key size, and to use small images under the condition that it has a period that covers the number of text bytes.

For the best security, we also recommend using 512-bits as the key size, because it gives a greater keystream period, and it has a greater collusion resistance than the smaller key sizes. Also, one needs to make sure that the keystream period for a particular image is greater than the number of text bytes that need to be encrypted.

7.3 Future Work

We would re-implement the algorithm so that the arrays become of type long, instead of byte, in order to improve the system's throughput. This requires providing our own implementation of the SHA algorithms so that they return arrays of long variables, this is required to eliminate the need for conversion between byte and long arrays.

We would also take the image's dimensions (height and width) into account when building the image vector to decrease the chance of collusion between images.

References

1. RSA Security Inc. Password-Based Cryptography Standard, PKCS #5 version 2.1, 2006.
2. M.J.B. Robshaw, Stream Ciphers Technical Report TR-701, version 2.0, RSA Laboratories, 1995.
3. RSA Laboratories. What is a block cipher? accessed on May 8th, 2012. http://www.rsa.com/rsalabs/node.asp?id=2168
4. RSA Laboratories. What is Electronic Code Book Mode? accessed on May 8th, 2012. http://www.rsa.com/rsalabs/node.asp?id=2170
5. RSA Laboratories. What is Cipher Block Chaining Mode? accessed on May 8th, 2012. http://www.rsa.com/rsalabs/node.asp?id=2171
6. RSA Laboratories. What is a stream cipher? accessed on May 8th, 2012. http://www.rsa.com/rsalabs/node.asp?id=2174
7. RSA Laboratories. What is RC4? accessed on May 8th, 2012. http://www.rsa.com/rsalabs/node.asp?id=2250
8. RSA Laboratories. RSA Security Response to Weaknesses in Key Scheduling Algorithm of RC4. accessed on May 8th, 2012. http://www.rsa.com/rsalabs/node.asp?id=2009
9. GSM Association. GSMA Statement on Media Reports Relating to the Breaking of GSM Encryption. accessed on May 8th, 2012. http://www.gsma.com/newsroom/gsma-statement-on-media-reports-relating-to-the-breaking-of-gsm-encryption
10. T. Xydis, S. Blake-Wilson, “Security Comparison: Bluetooth™ Communications vs. 802.11”, 2001.
11. Microsoft Developer Network, 7.5.12 “The checked and unchecked operators (C#)”. accessed on May 14th, 2012. http://msdn.microsoft.com/en-us/library/aa691349(v=vs.71).aspx
12. The code was taken from a post by Dave L. on StackOverflow.com. accessed on May 8th, 2012. http://stackoverflow.com/a/140861/408286
13. The documentation for the BufferedImage class. accessed on May 8th, 2012. http://docs.oracle.com/javase/6/docs/api/java/awt/image/BufferedImage.html
14. Shirazi, J. Java Performance Tuning, chapter 4. O’Reilly, 2000.