CSM # 1

Secure Communication

Field Session 2008

CLIENT:
Dr. Roman Tankelevich

WRITTEN BY:
Kevin Kaiser
Michael Keesey
Phillip Woods

Submitted
June 19, 2008

Executive Summary

Information can be passed discretely between parties on the internet using a steganographic process. Information can be embedded within an image, without altering its appearance. This report contains a description of a prototype that we implemented which uses this idea. The software uses a chaotic sequence to map control information within a bitmap image. The control information contains information about the sender, intended receiver and a message ID. The image can be sent via email to others, where the data is then extracted. A client/server pair was implemented to use this method. The server contains a database which logs all transactions and associates user messages with message IDs. With this design, a user can securely send a message to another user.

The prototype supports different approaches to hiding control information within an image. The software is setup to handle encryption on both the client and server. Two different chaotic maps are also available for the mapping process. Benchmarking results indicate that client side encryption is more efficient. This method is less secure, however, because an attacker can reverse engineer the algorithm. There is also a difference in complexity between the two chaotic algorithms. We hope that this prototype meets the client's needs and provides a basis for future development of a steganographic encryption environment as a communication tool.

Table of Contents

Abstract

There exists a need to securely pass electronic documents through insecure media such as the Internet. We implemented a prototype system in order to securely send messages through email by encrypting messages and associated information within images. The client manages user-owned stamps and encrypts the messages within the stamps. The server also has encryption capabilities and tracks stamp transactions between users.

A primary goal of this project was to create a series of benchmarks to ensure that the overall system is scalable and efficient. The prototype was implemented for the Windows platform, but with an emphasis on creating platform independent code.

1. Introduction to Project

Our goal for this project was to develop a secure internet communication tool for sending important documents through email. We developed a server and client side process that can encrypt a message inside an image by using a chaotic equation to help secure the sender and receiver's identity and personal information from outside third parties. The final deliverable for this project is a prototype that can be used to test the efficiency and usability of our developed methodology.

The scope of this project is the creation of a client-server architecture that allows for the transmission of messages between two clients using images. The focus of this project is the development of the underlying algorithms and technologies to make this possible.

2. Requirements and Specifications

2.1 Functional Requirements

2.1.1 Client Interface

The client interface allows the user to select a preferred stamp to send the message, select the desired recipient of the message, and specify the message they wish to send. When a user has received an image with encrypted contents, the client software will display the message for the user. The client interface takes the form of a command line interface.

2.1.2 Server Interface

The administrative interface for the server uses the command line. The overall purpose of the server is to track stamp ownership and transactions and provide a repository for passed messages for receiver retrieval.

2.2 Nonfunctional Requirements

Our goal was to have a working prototype by the end of the six week field session. Per the client's request, this software is coded in C++ within the Microsoft Visual Studio .NET environment. The software is functional within a windows environment. However, code is written to be cross platform whenever possible.

3. Design Approach

3.1 Overview

The design of the server/client pair is based on the combination of secure information transmission and trusted communications. All communications are intended to be encrypted, by encrypting the information within the image. Ideally, the sockets should be secured when clients are communicating with the server. This prevents an attacker from utilizing a man-in-the-middle attack at any point of the message transmission.

Trusted communications, based on the cryptographic signing of messages to other parties, will ensure that an attacker cannot misrepresent themselves as someone else. Even if an attacker somehow knows the control information within the image, they would be unable to get the message from the server unless they possessed the receiver's private key because the messages are stored on the server. Likewise, an attacker cannot impersonate another sender without possessing that person's private key. Because the prototype does not implement asymmetric keys for benchmarking purposes, this feature is not implemented in the current software.

Usable images will be stored on both the server and on clients. The server will serve as the overall repository for images so that users can download the images whenever they purchase or otherwise acquire them. Clients will store all of their stamps so that users can view them in an image gallery (not implemented). These images are currently in bitmap format (as opposed to a vector graphics format) like PNG and will be fairly small in size, limiting the impact of storing them in multiple locations.

3.2 Chaotic Encryption

Chaotic equations exhibit properties that make them ideal for data encryption. Solutions are often random, with the results similar to a white noise signal. Chaotic equations are extremely sensitive to initial values. A slight change in initial conditions will cause a completely distinct solution. These solutions are deterministic though, if given the same initial conditions and parameters. Solutions are also bounded. This makes chaotic equations ideal for our purposes, because the solution can easily be scaled to the size of the image to be mapped. For our purposes, a simple logistic map was used.

3.3 Steganography

Steganography is a way to transfer "secret" data without arousing suspicion that the data is present. We use bitmap stamps to carry the message across to the receiver. By implementing an encryption algorithm, it allows the user to insert classified information and encrypt it inside the stamp without ever being detected by attackers or third parties. Once the image reaches its destination, the receiver will decrypt the information from the stamp [1].

There are several approaches to hiding the message inside the image, but for now we are using the method that is most commonly seen in steganography and that is the substitution system, which substitutes redundant parts of an image with bits of the secret

message [1]. This will make it difficult for the attacker to notice any valuable information when the message is camouflaged inside the image. To add even more complexity for the attacker, parts of the message will be placed seemingly randomly throughout the whole image instead of being embedded in the same order. For this process we use the chaotic equation to randomly search and map the message inside the image. Now the information will be kept safe, as the only one who will be able to gain access to the secret message will be the receiver.

We were using a PNG file but due to some difficulties with the PNG library, we have decided to use a bmp file for the image. The idea of trying to encrypt the code into the stamp involves extracting the pixels from the image and encrypting one bit from the message into a color's least significant bit. A bmp pixel contains 24 bits. There are 3 color channels for every pixel: red, green, and blue. Each color channel gets 8 of the 24 bits [1], meaning that we can overwrite 3 bits per pixel. This indicates that the maximum length of information that can be encrypted is 3*(# of pixels) bits long. For our prototype, we assume that the images we are using are large enough to store the encrypted information.

3.4 Client and Server

The client and server communicate through sockets to transmit messages and other related information. Encryption and decryption can either be linked to the client or the server as indicated in Figure 1 below.

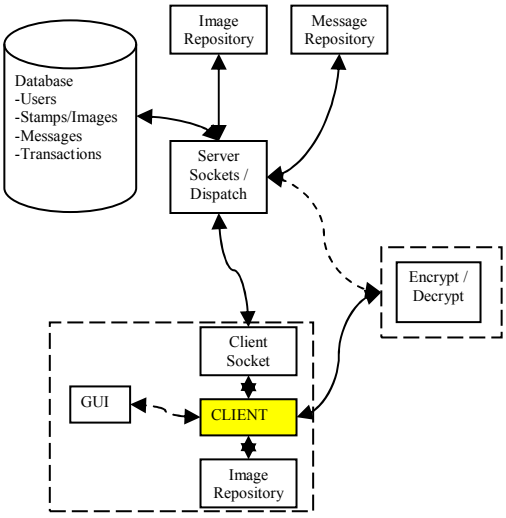

Figure 1

The server is primarily concerned with storing the messages, storing all stamps, and tracking all transactions carried out with the stamps. The client is intended to allow the user to choose the stamp they wish to use, what message to send, and to choose the receiver of the message. It then coordinates with the server to make the final image, which is then sent by the user's email client to the receiver. The receiving client then coordinates with the server to receive the message sent.

3.5 Timeline

In week 1, we met with our client and discussed project requirements. Team members conducted individual research during this week. In week 2 we gave our client an abstract on the design approach (methods, software, equations, etc…) and possible routes that we might take to get the project done. During week 3, we picked a route which satisfied the requirements and timeline for the client's needs. We immediately began working on the project by assigning subtasks to each member. Kevin was responsible for learning how to code in PHP and create the benchmarking software to test the quality of the server, the client and the different chaotic equations. Mike was responsible for getting the sockets and database access to work and Phil was responsible for finding and implementing a chaotic equation and embedding the control information inside the image. Figure 2 shows an estimate of the time required for each subtask.

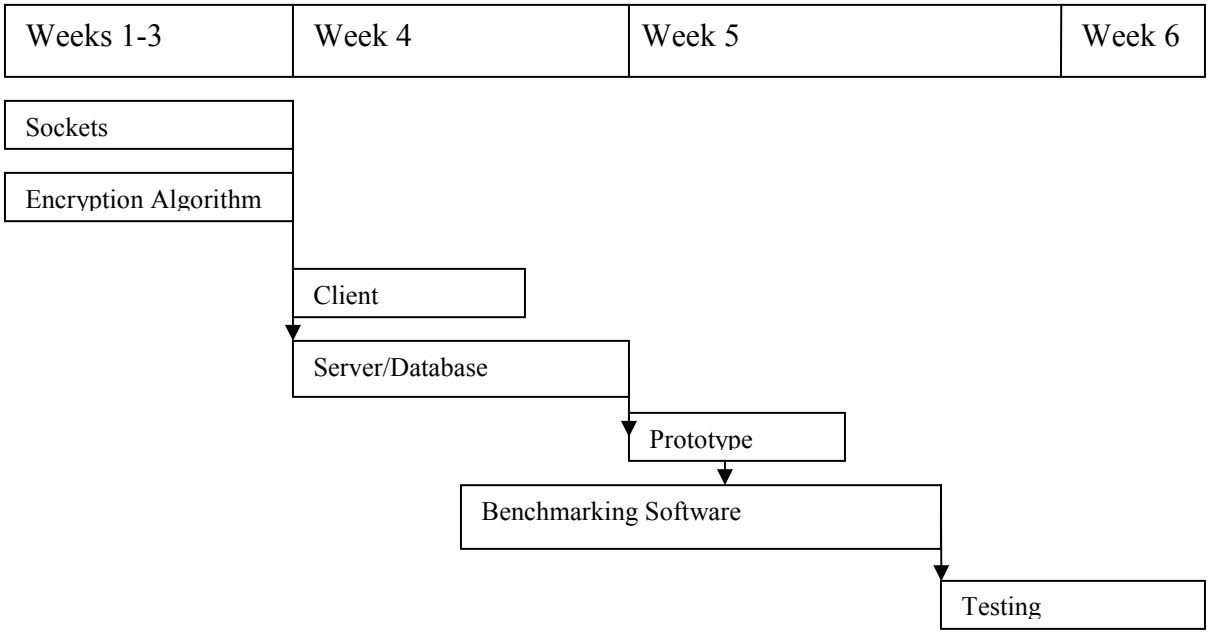

Figure 2

Phil was able to embed the control information inside the image using the Logistic equation as the chaotic equation by week 4 and later on with the Lorenz equation in week 5. Phil also implemented the client in week 4. Kevin was able to get the PHP benchmarking to work and display runtimes for the client/server with the Logistic and Lorenz sequences by the end week 5. Mike finished working on the sockets and database by the start of week 6. Because he was the only Computer Science major, Mike also provided implementation help to Kevin and Phil throughout the session. In week 6, we focused on the final paper, deliverables and presentation.

4. Implementation and Results

4.1 Server Implementation

Currently, the server is implemented with two executables for use with the different encryption/decryption methods. With the current implementation, encryption and decryption cannot take place in different sides of the client/server pair (e.g. the server cannot decrypt messages that a client has encrypted).

The first executable is designed for use with client side encryption. Sending (or encrypting) clients use the server to store their messages and to generate a Message ID to embed within the stamps. Receiving clients use the server to retrieve the messages stored on the server.

The second executable is designed for server side encryption. When a client wishes to send a message, he or she uploads their own ID, the proposed receiver's ID, and the message they wish to send. The server sends the bytes that make up the bitmap image with the information embedded within to the user. Similarly, an image receiver uploads the image they received to the server, which extracts the relevant information and returns to the user the appropriate message.

The following table outlines the size limits of various aspects of the server:

Fundamental Limits

Unique Stamps	2^{64}-1
Unique Users	2^{64}-1
Unique Messages	2^{64}-1
Message Size	2^{32}-2 bytes*

**This is due to needing room in char arrays for the '\0'*

Both server executables utilize unsigned 64 bit integers for Stamp IDs, User IDs, and Message IDs. The sizes of messages are stored in unsigned 32 bit integers.

Libraries used include the newest Boost revision that includes the Asio library for cross-platform socket communication and the Database_Access class.

4.2 Sockets Implementation

All sockets are implemented with the Asio library included with Boost 1.35.0 and above. Server sockets are implemented with asynchronous reads and writes which allows for multiple clients to be connected at a time. Asio's implementation should scale to several processors without needing to worry about thread safety, as all operations are single threaded after the reads and writes.

Client side sockets are implemented with a series of four single-threaded functions, with two functions for client-side encryption and two for server-side encryption. Sender clients can upload a message to the server and download the generated MessageID to embed

within the image. Additionally, they can upload the RecipientID and the message and receive the image with embedded information inside it already in binary form.

4.3 Database Implementation

The Database_Access class currently utilizes the MySQL C API for communicating with the MySQL server. It reads the configuration file "database.cfg" for the connection parameters.

The Users table associates 64bit UserIDs with the associated first, middle, and last names of the user. In future implementations, a field which includes the user's public key would be added to allow clients (or an encrypting server) to retrieve this key when necessary for asymmetric key cryptography.

UserID (Primary Key)
FName
MName
LName
Public Key

The Messages table associates the 64 bit MessageIDs with the base folder where the messages are stored. For example, assume message 18 is stored in *messages/18.txt*. In the associated row for message 18, the location is listed as *messages/*, with the 18.txt being implied based on the message's ID.

MID (Primary Key)
Message_Location

The Stamps table, like the messages table, associates the 64 bit StampID with the base folder where the messages are stored.

SID (Primary Key)
Stamp_Location

The Transactions table indicates when messages are read by recipients. It ties together the associated SenderID, ReceiverID, MessageID, and the date and time read.

SenderID
ReceiverID
MessageID
Date

4.4 Client Implementation

The client has four unique methods depending on whether it is a sending or receiving client and also whether encryption occurs on the client-side or server-side. The client relies on a client struct that contains all parameters necessary for client functionality and communication with the sockets. This functionality is determined through user I/O, reading a configuration file, and function calls to the client socket. Each of the four methods calls the appropriate socket function. The client includes the sockets library and the encrypt library.

4.4.1 Sending Client and Server Side Encryption

The client first reads the senderID and key from a config file. It then prompts the user for a stamp ID, receiving client ID, and a message. This information is passed to the server, where encryption occurs, through the socket. The socket returns the encrypted image,

which is saved to the user's machine. The user can then send the encrypted image to a recipient via email.

4.4.2 Receiving Client and Server Side Decryption

The client prompts the user for the location of an encrypted image on his or her disk. The client reads the image, and sends this along with the control information to the server, via the socket, where decryption occurs. The server returns the message associated with the encrypted control information, which is displayed by the client.

4.4.3 Sending Client and Client Side Encryption

The client first reads the senderID and key from a config file. It then prompts the user for a stamp ID, receiving client ID, a message, and a location to store the encrypted image. The client reads an image to disk, determined by the stamp ID, and calls the encryption function. This function returns the encrypted image which is stored to disk at a location specified by the user. The user can then send the encrypted image to a recipient via email.

4.4.4 Receiving Client and Client Side Decryption

The client prompts the user for the location of an encrypted image on his or her disk. The client reads the image, and calls the decryption function. This function decrypts the image and returns the embedded control information. This information is sent to the server via the sockets. The server returns the message associated with the control information, which is printed to screen by the client.

4.5 Encryption/Decryption Implementation

This library contains a function for encryption and a function for decryption. Both functions rely on a symmetric algorithm. This means that the same key is required at both ends for the decryption method to return valid information. Both functions rely on a struct that contains the control information to be encrypted/decrypted. This method includes a library called chaos which is used to create an array of hash values.

The encrypt function takes as parameters the location of the unencrypted image, a location to store the decrypted image, a struct of control information, and an encryption key. The function first reads the unencrypted image from disk. It then takes all the control information and converts it into a binary string. The function then calls the chaotic method, which returns an array of hash values dependent on the key. Each of these hash values correspond to a bit in the binary stream. The function overwrites the least significant bit of a color in the image. The location of each byte in the image is determined by the hash values returned by the chaotic method. The resulting image is then saved to disk. The encrypted function returns true if encryption occurred successfully and false otherwise.

The decrypt function takes as parameters the location of an image to be decrypted, and a decryption key. The function reads the encrypted image from disk. It then calls the chaotic method, which returns an array of hash values dependent on the key. The decrypt

function extracts the least significant bits from the image at locations specified by the hash values. The resulting binary stream is converted to a control information struct, which is returned by the function.

4.6 Benchmark Implementation

The benchmarking file has two functions:

1. Set up the program to run by initializing values to the sender's ID, receiver's ID and the key and put it in the client's configuration file.

2. After the program is done compiling, compute runtimes (in milliseconds) of the server and client and display the overall performance on the next page.

The benchmark is written in PHP and consists of two files called start.php and benchmark.php. Figure 3 shows the benchmark design. Start.php displays a page for the user to enter values for the senderID, receiverID and key. The user then sends the information to benchmark.php. There it creates a text file called config.txt inside the directory containing the program. During runtime, a text file is created called benchmarking.txt which includes time measurements for the clients. Different runtimes are reported depending on whether the client is sending or receiving and whether encryption is occurring on the server side or the client side. The benchmark.php uploads the text file created from the program and puts it inside an array. Then, it categorizes the clients by sender/receiver and client-side/server-side encryption and calculates the average for each one. The results of the benchmark are then displayed in table form.

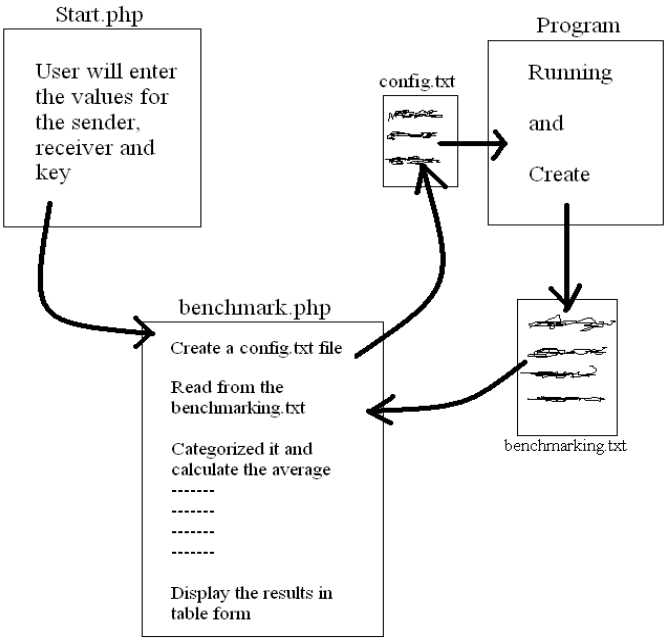

Figure 3

4.6.1 Benchmark Results

The following benchmarks were carried out on a 1.5Ghz Pentium M Processor with 50 trials for each type of encryption. Socket connections were made with the loopback interface on one machine instead of over a true network.

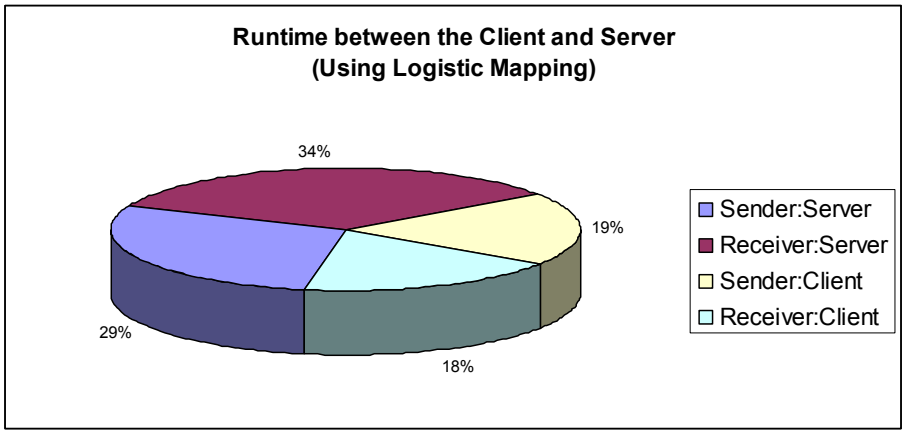

Figure 4

Figure 4 above relates the relative runtimes between the four different situations. As expected, the client side encryption and decryption methods required less time than the server side methods because there is less socket I/O with those methods. The times between senders and receivers are roughly equal. This is intuitive because the operations are roughly symmetrical.

Figure 5 shows the actual runtimes for each of the trials and shows the difference in runtimes between the Logistic and Lorenz maps. The Lorenz trials are slower than the Logistic trials (reflecting the higher complexity of the Lorenz map). However, this degree of slowdown is very small compared to the overall runtime. The more significant factor is whether the encryption is taking place on the server or the client due to the higher socket transfer overhead associated with server side encryption.

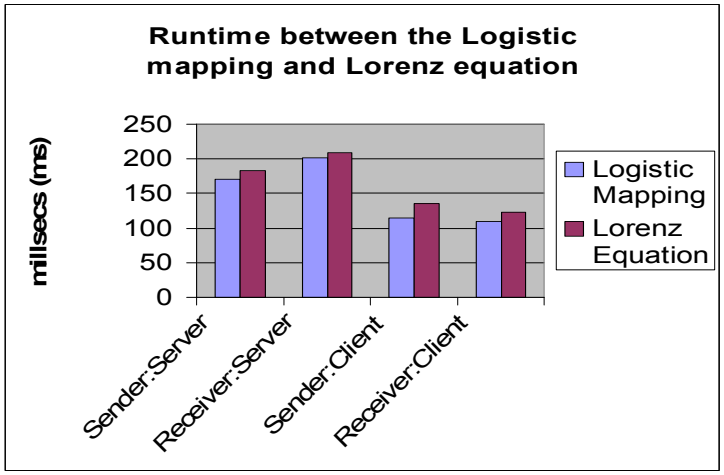

Figure 5

5. Conclusions and Future Directions

5.1 Conclusions

We feel our prototype successfully meets our client's needs. It takes control information and maps it to a bitmap image using a chaotic sequence as a hash function. The image can then be sent to a recipient, where the embedded data is encrypted. We implemented a client/server pair that accommodates this process. The prototype allows for two separate chaotic encryption methods, which can occur on the client or the server. We are pleased with this level of flexibility in the final software prototype.

Throughout the session, we found that we had to reduce the scope of our project to ensure that a final working prototype was completed on time. The implementation process proved to be more time consuming than our original estimate. Thus, the scope and functionality of the final deliverables had to be reduced. We are confident, however, that the prototype will be helpful to the client and serve as a springboard for further investigation into a steganographic encryption method.

5.2 Future Directions

The program should be modified to work with PNG images instead of bitmap images. This will greatly reduce storage and bandwidth usage.

Additionally the program could include a graphical user interface and stamp browser. Figure 6 shows a simple Windows Form GUI that we created but never implemented.

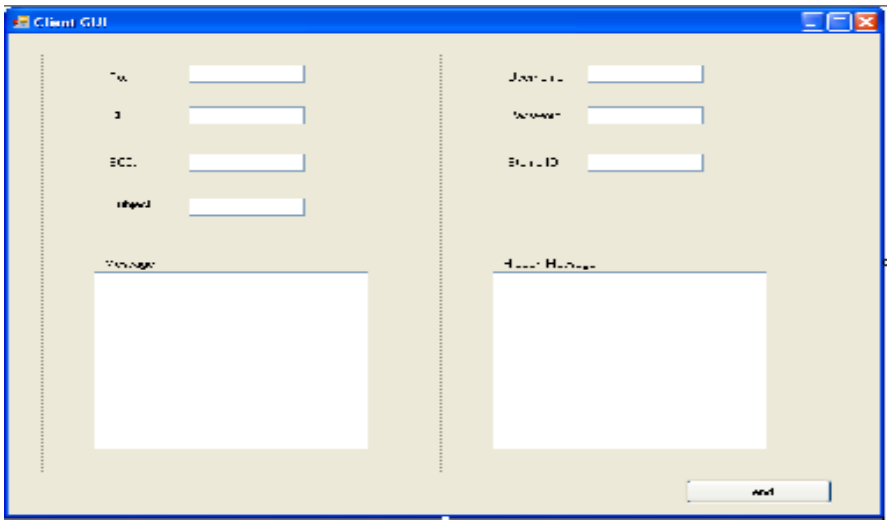

Figure 6

The current sockets should be modified to implement SSL transfers to ensure end-to-end encryption of all communication over the internet. The easiest approach would be to use the OpenSSL library and refer to its use within Asio at the boost::asio reference site.

Currently, the program uses a symmetric algorithm for encryption. Each client contains the same key, which is passed to the encryption/decryption methods. A better approach would be to use an asymmetric algorithm which uses public and private keys. The keys need to be interchangeable so that if key A encrypts a message, then key B is the only one able to decrypt it and vice versa. Additionally, much larger keys should be used. The current software uses 64 bit doubles.

The program can be modified to include message signing. Along with the typical control information, a sender should also embed a cryptographic signature within the image before sending to the receiver with their private key. Upon receiving the image, the receiver can validate the signature using the sender's public key to ensure that it came from the indicated person. This prevents an attacker from forging a send address.

In the case of client side encryption, the receiving client signs their request to the server with their private key. The server then validates the signature with the receiver's public key to ensure that the receiver is authentic. This prevents attackers from using stolen control information or a brute force attempt to retrieve a secret message from the server.

For server side encryption, signatures are only sent to and from the server because the server is the trusted party (which can be validated through SSL). In this method, the sending client signs its transmissions with the server so that the server (instead of the receiving client) does the validation of the sender. Because of this, the signature of the sender is not embedded within the image. All other aspects remain the same as with the client side encryption method.

References

[1] Katzenbeisser, Stefan and Fabien A.P. Petitcolas. Information Hiding Techniques for Steganography and Digital Watermarking. Artech House, 2000.