Burn After Reading:

Secure Communication Via Secret Key Encryption

by

Michael M. Chandler, BSCS Candidate

Computer Science Senior Project
with Prof. Michael Zijlstra
Maharishi University of Management, Fairfield, IA

Fall 2014

The Problem

Recent history shows that many governments and powerful corporations have become intrusive into the private lives of everyday citizens. This comes primarily in the form of electronic surveillance for marketing, policing, and political purposes. The knowledge of being constantly surveilled and the veiled, combined with the lack of due process that attends modern "homeland security" protection measures, has a chilling effect on free speech and peaceful, legal dissent.

The Objective

My intention with this project was to both deepen my understanding of the practical application of encryption, and to produce something that could evolve into a useful tool to encourage the free exchange of ideas and information without fear of reprisal. To these ends, I have employed Java Standard Edition, a client-server architecture, and the Advanced Encryption Standard.

The Research

My original intention with this project was to create a Skype-like video chat application with encryption options. My research into the topic revealed that Java does not handle web cams well, making it advisable to import a media handling library called GStreamer-java. I began the project having never imported or built a library before, adding to the learning curve and increasing the time investment. GStreamer-java proved to be a troublesome library to implement; it was dependent on a now unsupported and largely unavailable version of the GStreamer library, which was originally designed to work with C. After much effort and the inability of knowledgeable advisers to find a solution, I came to the conclusion that, given the time allotted (and remaining) for the project, it would be more realistic to focus on a simple client-server text chat application with encryption, and to reserve the more advanced features for future iterations of the project.

Research: The Advanced Encryption Standard

The Advanced Encryption Standard (AES) is currently the best combination of security and speed, and is widely accepted by government agencies as a standard for safely hiding information. AES was created when the U.S. government needed an encryption algorithm that was at least equivalent in security to Triple DES, but with a faster execution. The chosen algorithm, Rijndael, was created by Joan Daemen and Vincent Rijmen and subsequently renamed AES.

AES accomplishes the necessary security and speed through a series of substitutions, shifting, and mixing of data bits, in block of 128 bits (16 bytes) at a time. It relies heavily on the XOR operation, which is inherently less processor-intensive, and hence fast in execution. Without getting too deeply into the Linear Algebra behind the Rijndael team's cipher, we will go into what, exactly, AES is doing to our data to keep it secret.

AES can be invoked with three different key length options: 128, 192, or 256 bit keys. For the sake of speed of execution, we will focus on the 128 bit keyed version. Longer keys' function is in extending the time required to employ a brute force attack on the encryption (setting up a program to guess all possibilities until the key is found). Where n is the key length, it requires an average of $2^n/2$ guesses before a cryptographic key can be found. To put this in perspective, according to an article in the EE Times:

> Faster supercomputer (as per Wikipedia): 10.51 Pentaflops = 10.51 x 1015 Flops [Flops = Floating point operations per second]
>
> No. of Flops required per combination check: 1000 (very optimistic but just assume for now)
>
> No. of combination checks per second = (10.51 x 1015) / 1000 = 10.51 x 1012

No. of seconds in one Year = 365 x 24 x 60 x 60 = 31536000

No. of Years to crack AES with 128-bit Key

= (3.4 x 1038) / [(10.51 x 1012) x 31536000]
= (0.323 x 1026)/31536000
= 1.02 x 1018

= **1 billion billion years** (emphasis mine) [1]

Although some researchers claim to have found a weakness in AES (Biclique Attack), the proposed attack merely reduces the time to key discovery to two billion years[2]. With this information in mind, it is easy to dismiss brute force as a method of attack on AES for the foreseeable future, even when using its smallest key.

The other primary method of attacking symmetric ciphers such as AES is cryptanalysis, attempting to discern the original text from the cipher text and various other bits of information one may have at hand. To foil these intelligent attackers, AES goes through a series of steps to obfuscate the meaning of the original text. Let us take one example, here adapted from CoastRD's excellent video on the subject [3]:

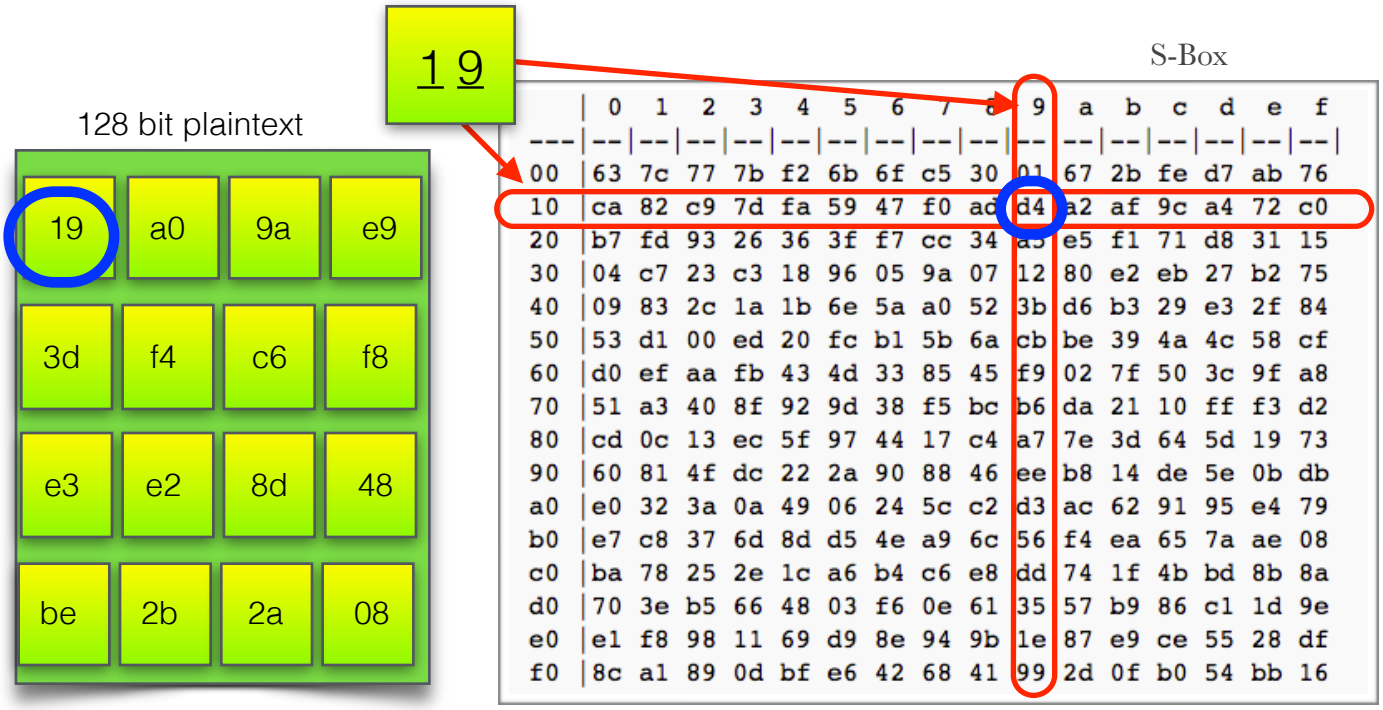

Figure 1: Plain Text Substitution Using S-Box

Figure 1 shows us the first step, substitution. Each byte of information is switched with its s-box equivalent, until we have something like Figure 2(next page). The rows are then shifted by index; that is, row 0 is shifted left 0 places; row 1's bytes are shifted left 1 place (wrapping the homeless byte back around to the right end), and so on.

This block is then multiplied by an invertible matrix, column by column. The data block is then transformed further by being XORed with a round key. The round key is unique to each round—each run through the substitution, shifting, and column mixing process. The 128 bit AES algorithm takes 10 rounds and thus 10 round keys. These round keys are derived from transformations on the 128 bit key itself. The output of round one becomes the input of round two, and so on through round ten. The output of round ten is one encrypted block of cipher text. Decryption is simply the encryption process run through "reverse rounds" using the

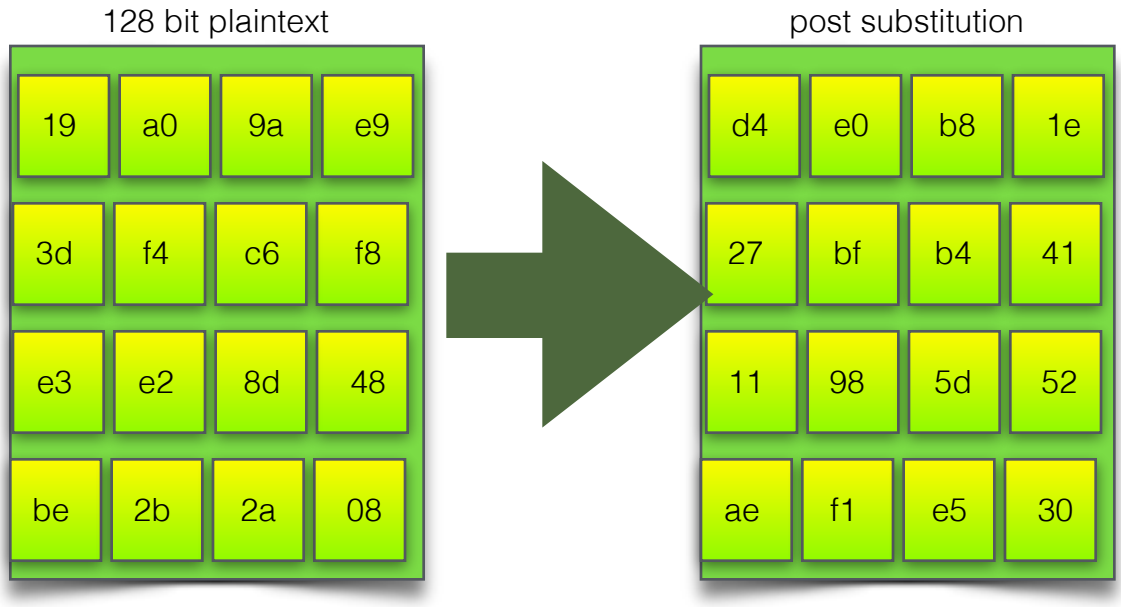

Figure 2: Plain Text Transformation

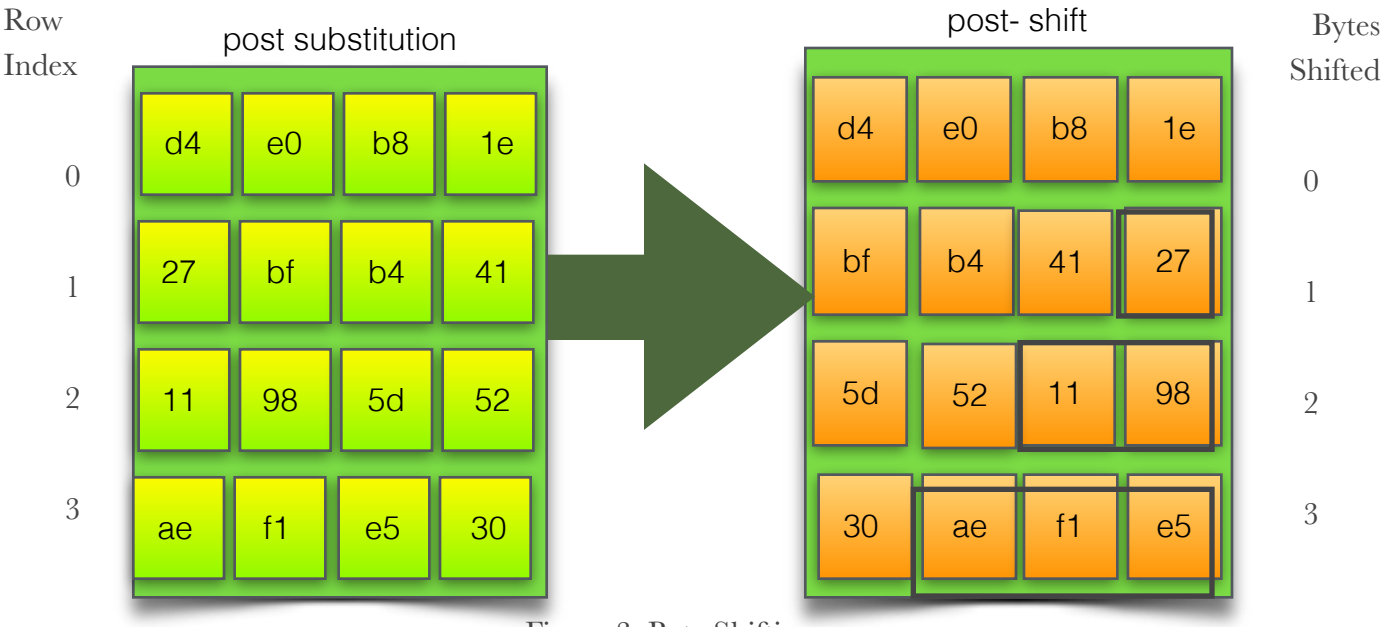

Figure 3: Byte Shifting

secret key and the inverse of the invertible matrix for the column mixing step.

This encryption process gives us discrete, encrypted blocks of data, but how do we keep them together as one contiguous whole, without compromising some of the valuable security we've established? Chaining these blocks together through a

method called Cipher Block Chaining (CBC) allows us to stitch together these discrete sets of encrypted data, keeping our entire set of data secret, yet whole. This is achieved by taking each encrypted block and XORing it with the next plaintext block before encryption. The initial block of plaintext has no encrypted block to XOR with it; for this, we take a random number called an initialization vector (IV) that is the same size as our block—128 bit IV for 128 bit plain text—and XOR it with the first block before encryption rounds begin. [4]

Research: Client-Server Chat

Client-server arrangements are a common form of networking, wherein a server—some computer—makes itself available as a conduit for the transfer of information to one or more "clients"—nodes on a network. Anyone who's used Google's computing power to search the internet has benefitted from this arrangement. The basic structure of a client-server network is shown in figure 4.

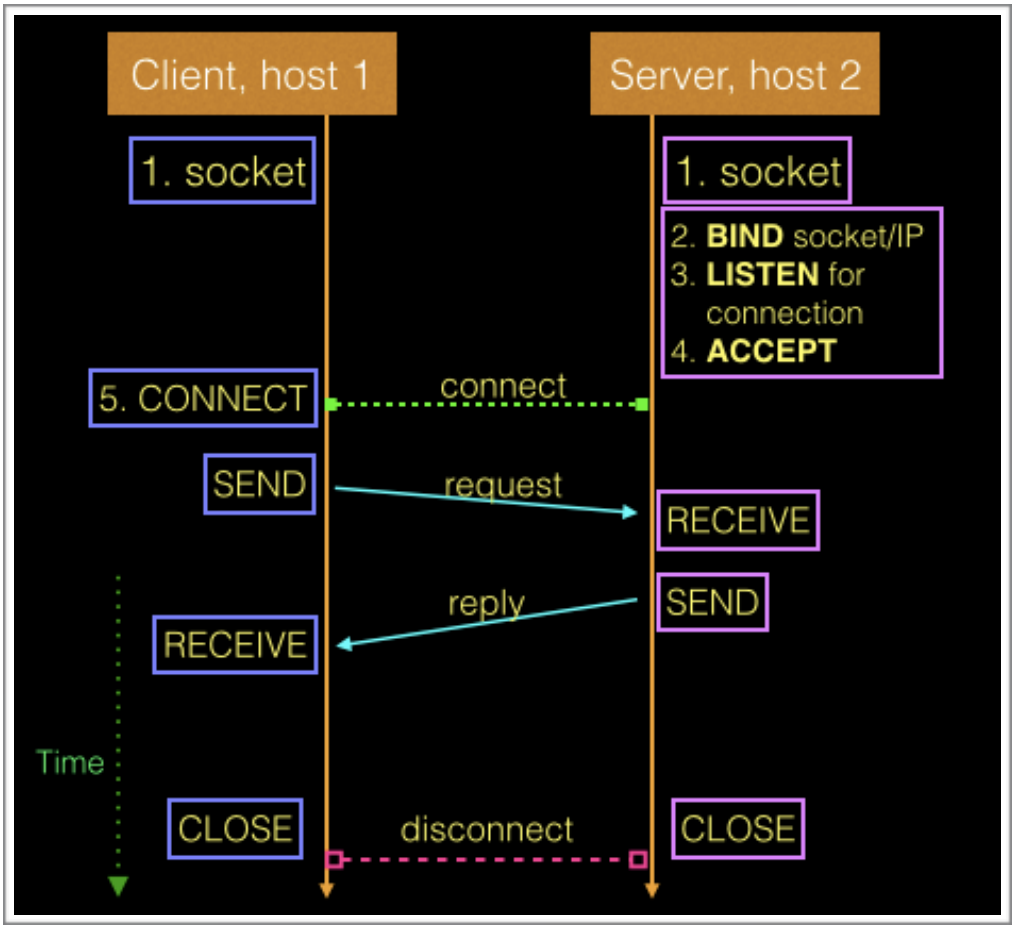

Figure 4, Client-Server Architecture.

As we can see in figure 4, both the server and the client invoke a "socket." A socket is simply a software object that allows network ports to communicate. A socket says, "I am using this network port for this purpose," which in our case is the exchange of encrypted text. The server socket accepts data from the client socket, and the client socket accepts data from the server socket. In Java, this is accomplished through the use of the DataInputStream and DataOutputStream objects. These objects can be set to read and write various types of data such as strings, ints, and bytes.

Implementation

The application was built in Eclipse, in the Java SE programming language. I began by researching and implementing a simple client-server chat. The chat application was designed to run one server and many clients, by listening on a specified port and opening a new thread for each incoming client. New connections to the server are first accepted, then assigned to a new port to free up the listening port for other incoming connections from clients. This portion of the app was adapted from Alex Bikfalvi's tutorial [5] on client-server chat, and has four classes:

- Server
- Server Thread
- Client
- Client Applet

The **Server class** works with the **Server Thread class** to open a socket, bind it to an IP address (the server's network address), and continuously listen for and accept incoming connections.

The **Client class** opens a socket for sending requests and data to the server, and for receiving information sent to the server by other clients.

The **Client Applet class** is the user interface where a user types in their name and the message they would like to contribute to the chat.

Pre-encryption, the chat program used a simple set of commands to exchange data. The client's DataOutputStream would call writeUTF(String), and the server's DataInputStream would call readUTF(String) at the appropriate points in the code. This works for unencrypted data, but to maintain secrecy some additional customization was required. For this, I created a **Crypt** class.

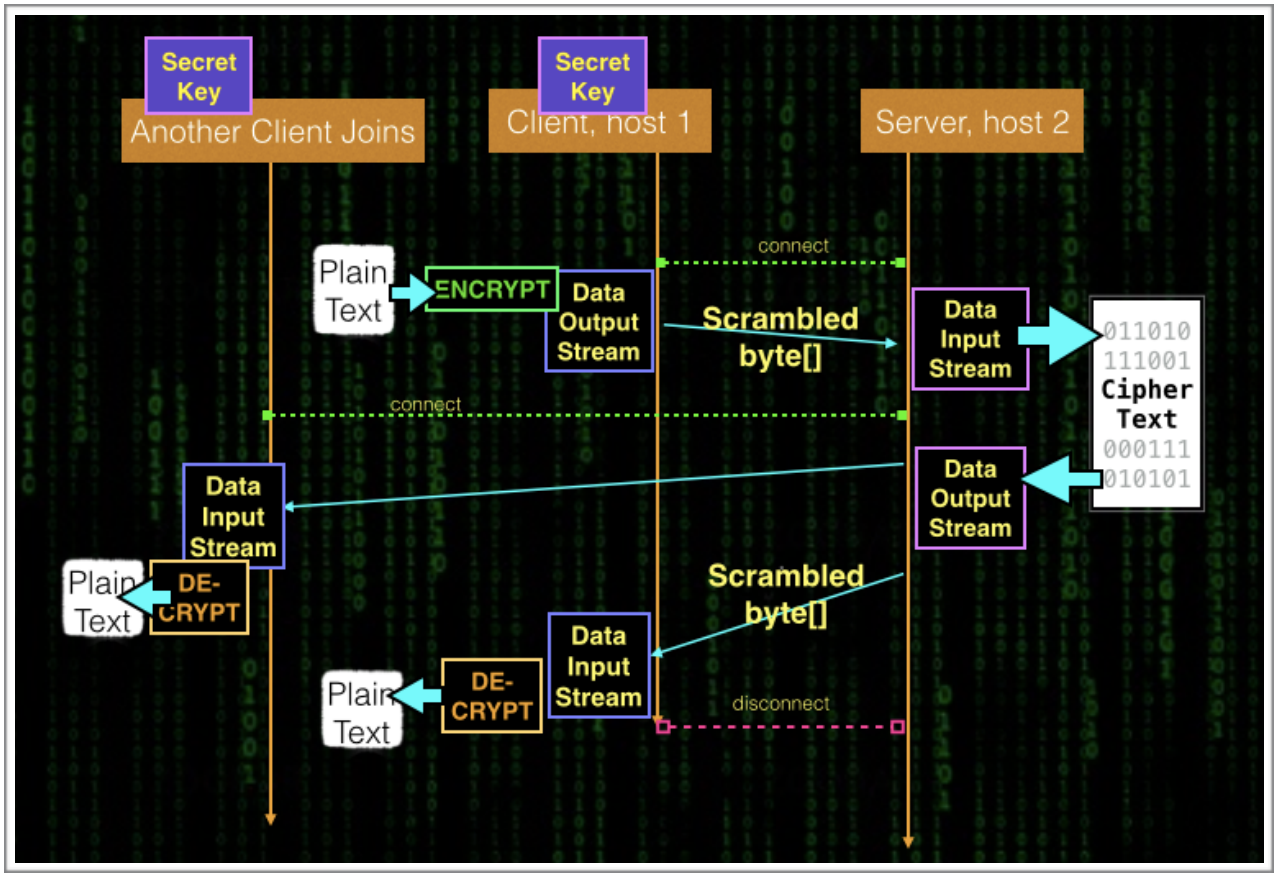

Figure 5, Encrypted Client-Server Chat

The Crypt class calls on the AES encryption method stored in the Sun library. Inside the Crypt class are two methods: encrypt, which takes in a string and returns a byte array (byte[]), and a decrypt method that takes in a byte[] and returns a string.

In order for the encrypted information to remain hidden, data must remain in the form of a byte[] the entire time it is traversing the network; that is, from before the client sends it to the server until it is sent back to the clients from the server, it must remain in byte[] form. To accomplish this, we must alter the behavior of the data streams.

We first call the Crypt.encrypt method on the string we intend to send, turning it into a byte array. We then write an integer, giving the length of the array we are sending, then immediately write the byte array to the DataOutputStream of the client. The server's DataInputStream then reads the integer length of the byte[], after which it invokes the readFully() method to capture the incoming byte[]. The server passes the byte[] to all clients' DataInputStreams, which then read the length of the array and the array itself before calling the Crypt.decrypt method. Because all clients share a common encryption key, all clients can decrypt the byte[], returning it to its original, string form. The server possesses no call to the methods of the Crypt class, thereby keeping the secret data safe from prying eyes.

Potential Weaknesses in the System

Although it is currently safe to assume that AES is secure, human nature and the forward march of technology demand our vigilance. Significant advances in computing—quantum computing, for example—could significantly reduce the time required to employ a brute force attack. Cryptologists must keep up with the technology of the times; just as a cryptanalyst could find all possibilities more quickly with quantum computing, so must a cryptologist use this technology to efficiently and sufficiently confuse and diffuse the plaintext data. Future encryption algorithms could use longer keys, or possibly employ randomly chosen keys as one of a set of possibilities. Quantum technology is still too young, at this point, to fully understand what it will be capable of.

The use of symmetric encryption without a secure key exchange method (such as PKI) assumes that all parties are already in possession of an un-compromised encryption key. There are many circumstances under which this is not an ideal arrangement, particularly if an unauthorized actor were to obtain the client

application. Coding the encryption key into the client application was a security compromise made in the interest of in time completion of the project, but is far from an ideal practice.

Future Development / Conclusions

Encryption of text hardly requires a proof of concept. This research was primarily a learning experience to lay the foundations for future endeavors. Going forward, I would like to research other encryption methods of operation, such PKI key exchange or public key (asymmetric) encryption for communication. Future iterations of the project could include audio chat, video chat, and perhaps VPN capabilities. The GUI could also use some polish to make it more pleasant to look at an more intuitive to use, as well as including the ability to register the product for a more personalized experience.

While I am satisfied with the level of knowledge I've gained from completing the encryption and text chat, I recognize that, to be successful, this project still requires a great deal of improvement. A basic understanding of networking and encryption implementation are enough of an accomplishment in the time given to motivate further work on this project, which I hope to bring up to (and beyond) the level of my original vision.

I see encryption as a mirror for the natural workings of levels of consciousness. When we meditate, we open ourselves to information that is unavailable to those without access to higher states of consciousness. Knowledge of correct technique is comparable to having the encryption key; correct practice of meditation opens available knowledge for our benefit. In this way, encryption keeps secret information hidden until and unless someone in possession of the secret key comes into contact with the hidden data.

Just as knowledge of the subtle workings of consciousness can be used for good or ill, knowledge of encryption methods can be used in support of humanity's greater good, or contribute to its downfall. With the correct balance of consciousness, morality, and technical expertise, I believe computer scientists sit in a unique

position of power to affect beneficial change. Encryption, applied judiciously and in service to humanity, is simply one more stepping stone on the path to human dignity and freedom.

Works Cited:

[1] M. Arora, " How secure is AES against brute force attacks?," EE Times, Vol. , no. , , 5/07/2012. http://www.eetimes.com/document.asp?doc_id=1279619 [Accessed 9/20/2014]

[2] D. Neal, "AES Encryption is Cracked: Researchers Find a Weakness in the System," The Inquirer, Vol. , no. , , 8/17/2011. http://www.theinquirer.net/inquirer/news/2102435/aes-encryption-cracked [Accessed 9/20/2014]

[3] CoastRD, "AES Rijndael Encryption Cipher Overview," CoastRD, Vol. , no. , , 1/20/2012.[online video] https://www.youtube.com/watch?v=H2LlHOw_ANg [Accessed 9/18/2014]

[4] M. Agarwal, A. Compoe, E. Pierce, Information Security and IT Risk Management, Edition of book, Hoboken, NJ: Wiley & Sons, 2014, p. 190.

[5] Bikfalvi, A. (2013, February 20). Developing a SIP Application in Java. http://alex.bikfalvi.com/teaching/upf/2013/architecture_and_signaling/lab/sip/ [Accessed December 13, 2014]

Borrowed/Adapted Image Sources:

1. Blue Earth Network: http://www.emperion.net/media/51874/shutterstock_109065071-cut.jpg
2. Locked File: http://www.technobol.com/wp-content/uploads/2013/11/How-to-lock-files-and-folder-with-password.jpg
3. S-box: http://en.wikipedia.org/wiki/Rijndael_S-box
4. PKI Chart: http://i.stack.imgur.com/Rcq1a.png
5. Matrix: http://static.zoonar.de/img/www_repository3/a7/07/af/10_ca4db8c007f26fc0d3f8c7080ac677b8.jpg

(All other images are the works of the author)

Appendix A, Server Class Code

```
/* Secure Client-Server Chat Program,
 * encrypted with AES 128 encryption
 * standard, using the core Java
 * libraries. Adapted from code found at:
 *
 * http://www.cn-java.com/download/data/book/socket_chat.pdf,
 * https://gist.github.com/bricef/2436364, and
 * http://stackoverflow.com/questions/10344132/read-byte-from-server,
 *
 * for a Senior Project at Maharishi University of Management,
 * Fairfield, Iowa, Fall 2014 Semester
 *
 * By Michael M. Chandler, BSCS Candidate
 */

package secure_Chat;

import java.io.*;
import java.net.*;
import java.util.*;


public class Server {


      private ServerSocket ss;

      // A mapping from sockets to DataOutputStreams. This will
      // help us avoid having to create a DataOutputStream each time
      // we want to write to a stream.
      @SuppressWarnings("rawtypes")
      private Hashtable outputStreams = new Hashtable();

      public Server (int port) throws IOException {
            listen (port);
      }

      @SuppressWarnings("unchecked")
      private void listen (int port) throws IOException {
            //create server socket:
```

```
            ss = new ServerSocket( port);
            System.out.println("Listening on " + ss);
            //Continue accepting connections forever:
            while (true) {

                  //next connection:
                  Socket s = ss.accept();
                  System.out.println("Connection from " + s);

                  //Data output stream for writing to the other side:
                  DataOutputStream dOut = new DataOutputStream
(s.getOutputStream());
                  //Save dOut for future use:
                  outputStreams.put (s, dOut);

                  //new thread for this connection:
                  new ServerThread (this, s);
            }
      }
      // Get an enumeration of all the OutputStreams, one for each
client
      // connected to us
      @SuppressWarnings("rawtypes")
      Enumeration getOutputStreams() {
      return outputStreams.elements();
      }
      // Send a message to all clients (utility routine)
      void sendToAll( byte[] bytes )  {
      // We synchronize on this because another thread might be
      // calling removeConnection() and this would screw us up
      // as we tried to walk through the list
      synchronized (outputStreams) {
      // For each client ...
      for (@SuppressWarnings("rawtypes")
      Enumeration e = getOutputStreams(); e.hasMoreElements(); ) {
            // ... get the output stream ...
            DataOutputStream dOut = (DataOutputStream)e.nextElement();
            // ... and send the message
            try {

                  //byte [] msg = Crypt.encrypt(bytes,
Crypt.getEncryptionKey());
                  dOut.writeInt (bytes.length);
```

```
                dOut.write (bytes);
                } catch( IOException ie ) { System.out.println( ie );
                } catch (Exception e1) {
                      // TODO Auto-generated catch block
                      e1.printStackTrace();
                }
          }
     }
}
          // Remove a socket, and it's corresponding output stream,
from our
          // list. This is usually called by a connection thread that
has
          // discovered that the connectin to the client is dead.
          void removeConnection( Socket s ) {
                // Synchronize so we don't mess up sendToAll() while
it walks
                // down the list of all output streamsa
                synchronized( outputStreams ) {
                      // Tell the world
                      System.out.println( "Removing connection to " + s
);
                      // Remove it from our hashtable/list
                      outputStreams.remove( s );
                      // Make sure it's closed
                      try {
                            s.close();
                      } catch( IOException ie ) {
                            System.out.println( "Error closing "+s );
                            ie.printStackTrace();
                      }
                }
          }

     //Main--use as java Server >port<
     public static void main (String[] args) throws Exception {

          //port # from command line:
          int port = Integer.parseInt (args[0]);

          //Server object that automatically accepts connections:
          new Server (port);
     }}
```

Appendix B, Server Thread Class Code

```
/* Secure Client-Server Chat Program,
 * encrypted with AES 128 encryption
 * standard, using the core Java
 * libraries. Adapted from code found at
 *
 * http://www.cn-java.com/download/data/book/socket_chat.pdf,
 * https://gist.github.com/bricef/2436364, and
 * http://stackoverflow.com/questions/10344132/read-byte-from-server,
 *
 * for a Senior Project at Maharishi University of Management,
 * Fairfield, Iowa, Fall 2014 Semester
 *
 * By Michael M. Chandler, BSCS Candidate
 */

package secure_Chat;
import java.io.*;
import java.net.*;

public class  ServerThread extends Thread {
     private Server server;
     private Socket socket;

     public ServerThread (Server server, Socket socket) {
          //save parameters
          this.server = server;
          this.socket = socket;

          start();
     }
     //Runs in a separate thread when start() is called in the
constructor:
     public void run() {

          try {
               //DataInputStream for communication; DataOutput Stream
sends to us
               DataInputStream dIn = new DataInputStream
(socket.getInputStream());
               //Continuously receive:
```

```
                while (true) {
                      //read next msg
                      int byteCount = dIn.readInt();
                      byte[] bytes = new byte[byteCount];
                      dIn.readFully(bytes);
                     //show encryption in console
                      System.out.println("Sending AES data: " +
bytes.toString());
                      //send to all clients
                      server.sendToAll(bytes);
                }
           } catch( EOFException ie ) {
           // This doesn't need an error message
           } catch( IOException ie ) {
           ie.printStackTrace();
           } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
           } finally {
           // The connection is closed,
           // so have the server deal with it
           server.removeConnection (socket);
           }
     }
}
```

Appendix C, Client Class Code

```
/* Secure Client-Server Chat Program,
 * encrypted with AES 128 encryption
 * standard, using the core Java
 * libraries. Adapted from code found at
 *
 * http://www.cn-java.com/download/data/book/socket_chat.pdf,
 * https://gist.github.com/bricef/2436364, and
 * http://stackoverflow.com/questions/10344132/read-byte-from-server,
 *
 * for a Senior Project at Maharishi University of Management,
 * Fairfield, Iowa, Fall 2014 Semester
 *
 * By Michael M. Chandler, BSCS Candidate
 */


package secure_Chat;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;


@SuppressWarnings("serial")
public class Client extends Panel implements Runnable {

     static TextField tf = new TextField("Type a message and hit
[return].");
     static TextField name = new TextField("Type your name before
chatting (no [return])");
     private TextArea ta = new TextArea();

     private Socket socket;

     private DataOutputStream dOut;
     private DataInputStream dIn;
```

```
//    static String encryptedString;
//    static String IV = "1234567890123456";
//    static String encryptionKey = "0987654321123456";



      public Client (String host, int port) {

            //set up window
            setLayout (new BorderLayout());
            //add tf2 name, modify "message" to include to include name

            add ("North", name);
            add ("South", tf);
            add ("Center", ta);
            //add (nLabel, BorderLayout.PAGE_START);
            //send and receive when someone types and hits return
            //using anonymous class as a callback
            tf.addActionListener ( new ActionListener() {
                  public void actionPerformed (ActionEvent e) {
                        processMessage (e.getActionCommand());
                  }
            });

            //connect to server
            try {
                  socket = new Socket (host, port);
                  System.out.println("Connected to " + socket);

                  //get streams and make them into data-in and -out
streams:
                  dIn = new DataInputStream(socket.getInputStream());
                  dOut = new DataOutputStream(socket.getOutputStream());

                  //new background thread for receiving
                  new Thread(this).start();

            }catch (IOException ie) {System.out.println(ie);}
      }
      //called when user types something:
      private void processMessage (String message) {
            try {
```

```
                byte[] sent = Crypt.encrypt ((name.getText() + ": " +
message), Crypt.getEncryptionKey());
                //send to server
                dOut.writeInt (sent.length);
                dOut. write (sent);
                //clear text input field
                tf.setText("");

          }catch (IOException ie) {System.out.println(ie);
          } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
          }
     }
     //background thread to show messages from other window
     public void run() {
          try {
                //continue receiving messages one by one
                while (true) {

                     //get next message
                     int byteCount = dIn.readInt();
                     byte[] msg = new byte[byteCount];
                     dIn.readFully(msg);
                     String message = Crypt.decrypt(msg,
Crypt.getEncryptionKey());
                     //...and print it
                     ta.append(message + "\n");
                     tf.setText("");
                }
          }catch (IOException ie) {System.out.println(ie);
          }catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
          }
     }
}
```

Appendix D, Client Applet Class Code

```
/* Secure Client-Server Chat Program,
 * encrypted with AES 128 encryption
 * standard, using the core Java
 * libraries. Adapted from code found at
 *
 * http://www.cn-java.com/download/data/book/socket_chat.pdf,
 * https://gist.github.com/bricef/2436364, and
 * http://stackoverflow.com/questions/10344132/read-byte-from-server,
 *
 * for a Senior Project at Maharishi University of Management,
 * Fairfield, Iowa, Fall 2014 Semester
 *
 * By Michael M. Chandler, BSCS Candidate
 */

package secure_Chat;

import java.applet.*;
import java.awt.*;

@SuppressWarnings("serial")
public class ClientApplet extends Applet {

      public void init() {
            String host = getParameter ("host");
            int port = Integer.parseInt(getParameter ("port"));
            setLayout (new BorderLayout());
            add ("Center", new Client (host, port));
      }
}
```

Appendix E, Crypt Class Code

```
/* Secure Client-Server Chat Program,
 * encrypted with AES 128 encryption
 * standard, using the core Java
 * libraries. Adapted from code found at
 *
 * http://www.cn-java.com/download/data/book/socket_chat.pdf,
 * https://gist.github.com/bricef/2436364, and
 * http://stackoverflow.com/questions/10344132/read-byte-from-server,
 *
 * for a Senior Project at Maharishi University of Management,
 * Fairfield, Iowa, Fall 2014 Semester
 *
 * By Michael M. Chandler, BSCS Candidate
 */

package secure_Chat;

import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.Cipher;

public class Crypt {
     static String IV = "1234567890123456";
     static String plaintext;
     static String encryptionKey = "0987654321123456";


          public static byte[] encrypt(String plainText, String
encryptionKey) throws Exception {
               plainText = (Client.name.getText() + ": " +
Client.tf.getText());
               Cipher cipher = Cipher.getInstance ("AES/CBC/
PKCS5Padding", "SunJCE");
               SecretKeySpec key = new SecretKeySpec
(encryptionKey.getBytes("UTF-8"), "AES");
               cipher.init (Cipher.ENCRYPT_MODE, key, new
IvParameterSpec (IV.getBytes ("UTF-8")));
               return cipher.doFinal (plainText.getBytes("UTF-8"));
          }
```

```
          public static String decrypt (byte[] cipherText, String
encryptionKey) throws Exception {
               Cipher cipher = Cipher.getInstance ("AES/CBC/
PKCS5Padding", "SunJCE");
               SecretKeySpec key = new SecretKeySpec
(encryptionKey.getBytes("UTF-8"), "AES");
               cipher.init (Cipher.DECRYPT_MODE, key, new
IvParameterSpec (IV.getBytes("UTF-8")));
               return new String (cipher.doFinal(cipherText),
"UTF-8");
          }
          public static String getEncryptionKey (){
               return encryptionKey;
          }

     }
```