Cyber Security Encryption and Decryption Algorithms

Final report

Jesse Perez, Keyur Patel, Joshua Mullis, Luke Nye, Dennis Stewart

Department of Computer Science and Information Technology
Armstrong State University

Date:
1st May 2017

Table of Contents

1	Executive summary ……	2
2	Project background……..	3
2.1	Needs statement ……..	3
2.2	Goal and objectives……...3	
2.3	Design constraints and feasibility……4-5	
2.4	Evaluation of alternative solution……	5
3	Final design……..	6
3.1	System description……....6	
3.2	Complete module-wise specifications……6	
4	Implementation notes……	7-20
5	Experimental results…….20	
6	User's Manuals……..	25-43
7	Appendix…….43	

1 Executive summary

The purpose of this project is to provide an educational demonstration regarding two different cryptographic algorithms, Knapsack and A5/1. EKG data points are used as the data source to be ciphered (EKG, or ECG stands for electrocardiography, the study of recording cardiac biorhythms). The data points imported from an excel file (.xlsx) into the data application, where the user then has the option of choosing and displaying encrypted and decrypted values using either algorithm.

The objective of the GUI (graphic user-interface) is to give users a better understanding of how Knapsack and A5/1 works. The final design gives the user either option, and provides a short summary of their backgrounds. It mentions key points such as what makes A5/1 a more complex (and thereby safe algorithm), and provides graphical displays the support these claims.

Once the user has chosen the encryption algorithm they wish to use, they select the *Raw Data* button to load the data into the quantizer so that the data can be put into an EKG display. In order to simplify how the application functions and make it compatible for either scheme, a quantizer function has been included which converts the raw data points into more tolerable integer numbers. Every time the data points are mapped to the integers (with every button click), the lookup table is randomized so that with each encryption instance, a different list of integers numbers is produced, providing an added layer of security.

Once the user clicks the Encrypt button, the encrypted data is displayed on a graph. This provides a visual aid as to how randomized and distorted the points have become, as they will bear little resemblance to the original EKG graph. The user can then click on the “Decrypt” button which returns the data back to its original form.

As for A5/1, the user has to click the A5/1 tab to select it as an algorithm (Knapsack is the default). There is a preset initiation vector set for this algorithm, or if the user wishes, they can create their own using a 16 character hexadecimal input. The the user will complete the same process as they did with the Knapsack algorithm. Along with the EKG graph data, the program also shows the data in numerical form in a table. The table shows the data in both its raw and quantized form.

The implementation was done through various classes. That makes it easier for us the programmers to find any errors and correct them more effectively. It also allows another programmer to see how we implemented the program. Each class has their own specific aspect the project. The code does exactly what it is supposed to do. There are very few, if any, errors within the code. The program is very user friendly. Project management was very well organized. Each person in the group had a specific task to perform. During our weekly meetings, we talked about and answered each other’s question about what each of us did. By doing this, we were able to finish the project ahead of schedule with little problems. There were a few problems initially. As with many group projects, scheduling conflicts was a huge issue. Group members were not always able to meet at the same time. Over the course of the project, the group members were able to overcome the conflicts and meet to discuss the project.

2 Project background

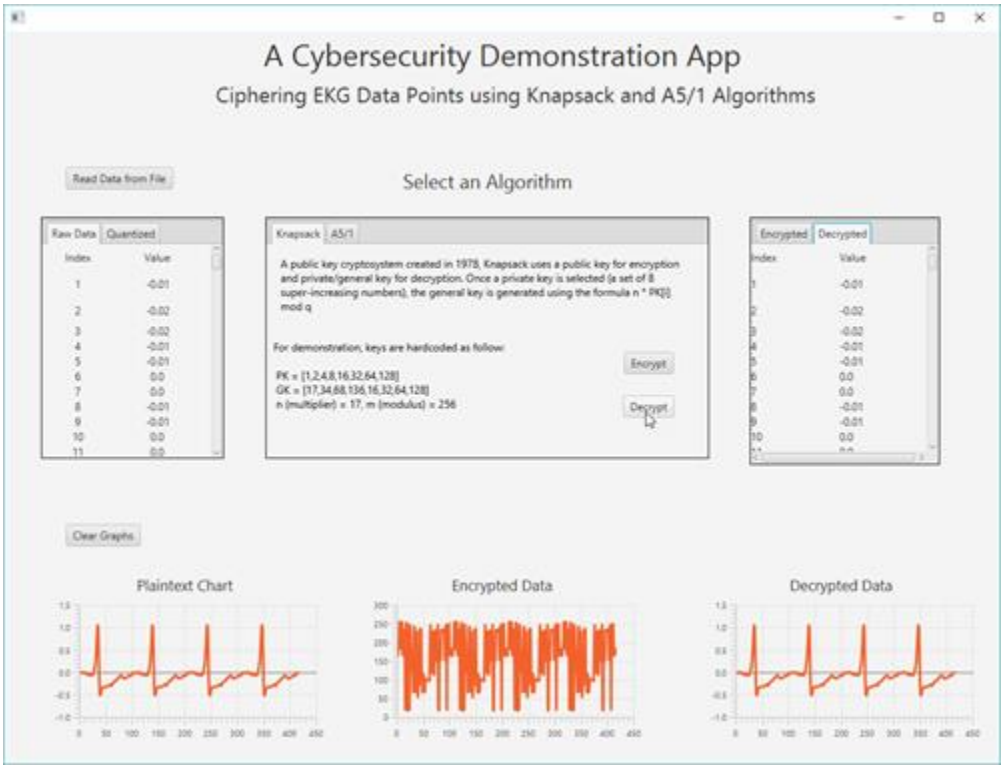
2.1 Needs statement

Hardware requirements for this project is 800 MHz Intel Pentium lll or equivalent, 512MB of RAM, 750 MB of free disk space. There are additional pieces of software needed beside the project file to make the application run with a new installation. The approach for this installation will be using NetBeans IDE 8.1 and newer versions. It simplifies the process of running all the different pieces of software and allows the user to see the code being used to better understand how it works. Before installing NetBeans, you must have the Java SE Development Kit(JDK) 8 installed. You can download JDK for free on http://www.oracle.com/technetwork/java/javase/download.

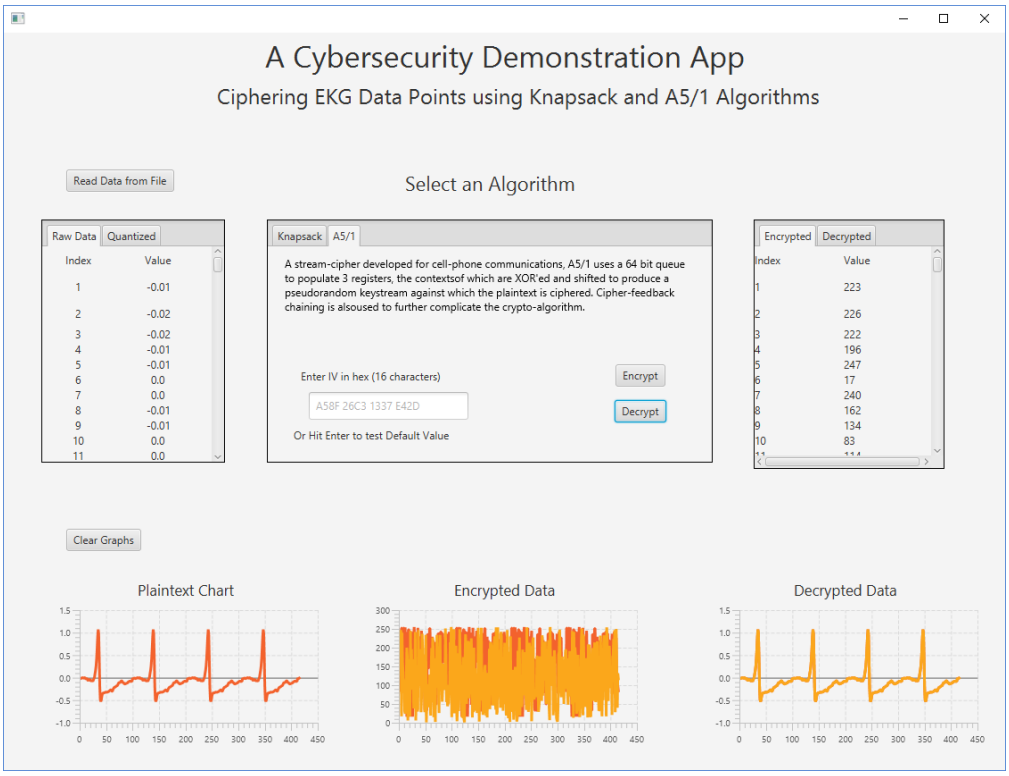
2.2 Goal and objectives

Our goal was to design and implement an encryption/decryption system that reads a file containing raw EKG data samples, quantize the EKG signal (convert the signals from analog to digital or vice versa using an 8-bit encoder/decoder), and then encrypt/decrypt the digital form of the EKG signal. The following images show the results.

Encryption/Screenshot of Knapsack encryption



Screenshot of A5/1 Encryption/Decryption



2.3 Design constraints and feasibility

The implementation of this program is done through various classes. This makes it easier for any programmer to find any errors and fix them. The program does what it is intended, but throughout its develop some complications were encountered.

One of the errors we had was with Quantizer class. This class is supposed to take the raw EKG data values and convert them into encryption-safe integers, but we were having trouble converting them to integers because we made one mistake in for loop but after trying couple different things we figured out. The second error we had was in the A5/1 class. In A5/1, class variables includes 3 registers (shifting arrays) and array lists that contain different elements of A5/1 (the queue, the keystream, the plaintext, and ciphertext). When encrypting, the plaintext is passed into the class constructor, along with the initiation vector which populates the queue and registers. Every cycle/pulse generates a keystream bit. After 8 pulses, the resulting 8-bit keystream is XOR'd with 8 bits of plaintext to create 8 bits of ciphertext. That 8-bit ciphertext is then copied into the rightmost 8 bits of the queue, and the rest of the queue suppose to be shifted left 8 times but the error in the loop caused the rest of the queue shifted right 8 times but, after trying couple things we figured out why it was shifting right 8 times.

2.4 Evaluation of alternative solutions

```
//below commented code reads data from excel file (ECG.xlsx) into an arraylist. this code being substituted with
//hardcoded data for deployment compatibility purposes (requires dependencies to run)
/*
try {

    FileInputStream excelFile = new FileInputStream(new File(filePath));
    Workbook workbook = new XSSFWorkbook(excelFile);
    Sheet firstSheet = workbook.getSheetAt(0);
    Iterator<Row> iterator = firstSheet.iterator();

    while (iterator.hasNext()) {
        Row nextRow = iterator.next();
        Iterator<Cell> cellIterator = nextRow.cellIterator();

        while (cellIterator.hasNext()) {
            Cell cell = cellIterator.next();
            dataArray.add(cell.getNumericCellValue());
        }
    }
    workbook.close();
    excelFile.close();

} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
*/
```

One of the intended features of this project was to read data directly from the given Excel file, but after realizing the complexities of having 3rd person libraries, the data was hard-coded in for demonstration purposes. The above picture shows commented code that reads data from an excel file (ECG.xlsx) into an arraylist. The hardcoded data used for deployment compatibility purposes (requires dependencies to run) is shown in the image below.

```
public static ArrayList scan(String filePath) {

    ArrayList<Double> dataArray = new ArrayList<>(Arrays.asList(-0.01, -0.02, -0.02, -0.01, -0.01, 0.0, 0.0, -0.01, -0.01, 0.0,
            0.0, -0.02, -0.03, -0.02, -0.02, -0.05, -0.07, -0.05, -0.04, -0.04, -0.05, -0.07, -0.07, -0.05, -0.07, -0.08, -0.05,
            -0.02, 0.01, 0.09, 0.16, 0.26, 0.42, 0.65, 0.91, 1.07, 0.98, 0.45, -0.19, -0.49, -0.53, -0.49, -0.41, -0.36, -0.34,
            -0.34, -0.34, -0.33, -0.33, -0.32, -0.32, -0.33, -0.32, -0.32, -0.31, -0.31, -0.29, -0.29, -0.27, -0.27, -0.29, -0.3,
            -0.3, -0.26, -0.24, -0.22, -0.21, -0.21, -0.2, -0.19, -0.18, -0.18, -0.16, -0.13, -0.11, -0.11, -0.11, -0.1, -0.08,
            -0.07, -0.07, -0.1, -0.11, -0.11, -0.13, -0.14, -0.14, -0.13, -0.13, -0.12, -0.12, -0.11, -0.09, -0.09, -0.08, -0.07,
            -0.08, -0.09, -0.08, -0.08, -0.01, -0.02, -0.02, -0.01, -0.01, -0.02, -0.02, -0.01, -0.01, 0.0, 0.0, -0.01, -0.01,
            0.0, 0.0, -0.02, -0.03, -0.02, -0.02, -0.05, -0.07, -0.05, -0.04, -0.04, -0.05, -0.07, -0.07, -0.05, -0.07, -0.08,
            -0.05, -0.02, 0.01, 0.09, 0.16, 0.26, 0.42, 0.65, 0.91, 1.07, 0.98, 0.45, -0.19, -0.49, -0.53, -0.49, -0.41, -0.36,
            -0.34, -0.34, -0.34, -0.33, -0.33, -0.32, -0.32, -0.33, -0.32, -0.32, -0.31, -0.31, -0.29, -0.29, -0.27, -0.27,
            -0.29, -0.3, -0.3, -0.26, -0.24, -0.22, -0.21, -0.21, -0.2, -0.19, -0.18, -0.18, -0.16, -0.13, -0.11, -0.11, -0.11,
            -0.1, -0.08, -0.07, -0.07, -0.1, -0.11, -0.11, -0.13, -0.14, -0.14, -0.13, -0.13, -0.12, -0.12, -0.11, -0.09, -0.09,
            -0.08, -0.07, -0.08, -0.09, -0.08, -0.08, -0.01, -0.02, -0.02, -0.01, -0.01, -0.02, -0.02, -0.01, -0.01, 0.0, 0.0,
            -0.01, -0.01, 0.0, 0.0, -0.02, -0.03, -0.02, -0.02, -0.05, -0.07, -0.05, -0.04, -0.04, -0.05, -0.07, -0.07, -0.05,
            -0.07, -0.08, -0.05, -0.02, 0.01, 0.09, 0.16, 0.26, 0.42, 0.65, 0.91, 1.07, 0.98, 0.45, -0.19, -0.49, -0.53, -0.49,
            -0.41, -0.36, -0.34, -0.34, -0.34, -0.33, -0.33, -0.32, -0.32, -0.33, -0.32, -0.32, -0.31, -0.31, -0.29, -0.29,
            -0.27, -0.27, -0.29, -0.3, -0.3, -0.26, -0.24, -0.22, -0.21, -0.21, -0.2, -0.19, -0.18, -0.18, -0.16, -0.13, -0.11,
            -0.11, -0.11, -0.1, -0.08, -0.07, -0.07, -0.1, -0.11, -0.11, -0.13, -0.14, -0.14, -0.13, -0.13, -0.12, -0.12, -0.11,
            -0.09, -0.09, -0.08, -0.07, -0.08, -0.09, -0.08, -0.08, -0.01, -0.02, -0.02, -0.01, -0.01, -0.02, -0.02, -0.01,
            -0.01, 0.0, 0.0, -0.01, -0.01, 0.0, 0.0, -0.02, -0.03, -0.02, -0.02, -0.05, -0.07, -0.05, -0.04, -0.04, -0.05, -0.07,
            -0.07, -0.05, -0.07, -0.08, -0.05, -0.02, 0.01, 0.09, 0.16, 0.26, 0.42, 0.65, 0.91, 1.07, 0.98, 0.45, -0.19, -0.49,
            -0.53, -0.49, -0.41, -0.36, -0.34, -0.34, -0.34, -0.33, -0.33, -0.32, -0.32, -0.33, -0.32, -0.32, -0.31, -0.31,
            -0.29, -0.29, -0.27, -0.27, -0.29, -0.3, -0.3, -0.26, -0.24, -0.22, -0.21, -0.21, -0.2, -0.19, -0.18, -0.18, -0.16,
            -0.13, -0.11, -0.11, -0.11, -0.1, -0.08, -0.07, -0.07, -0.1, -0.11, -0.11, -0.13, -0.14, -0.14, -0.13, -0.13, -0.12,
            -0.12, -0.11, -0.09, -0.09, -0.08, -0.07, -0.08, -0.09, -0.08, -0.08, -0.01, -0.02, -0.02, -0.01));

    //below commented code reads data from excel file (ECG.xlsx) into an arraylist. this code being substituted with
```

3 Final design

3.1 System description

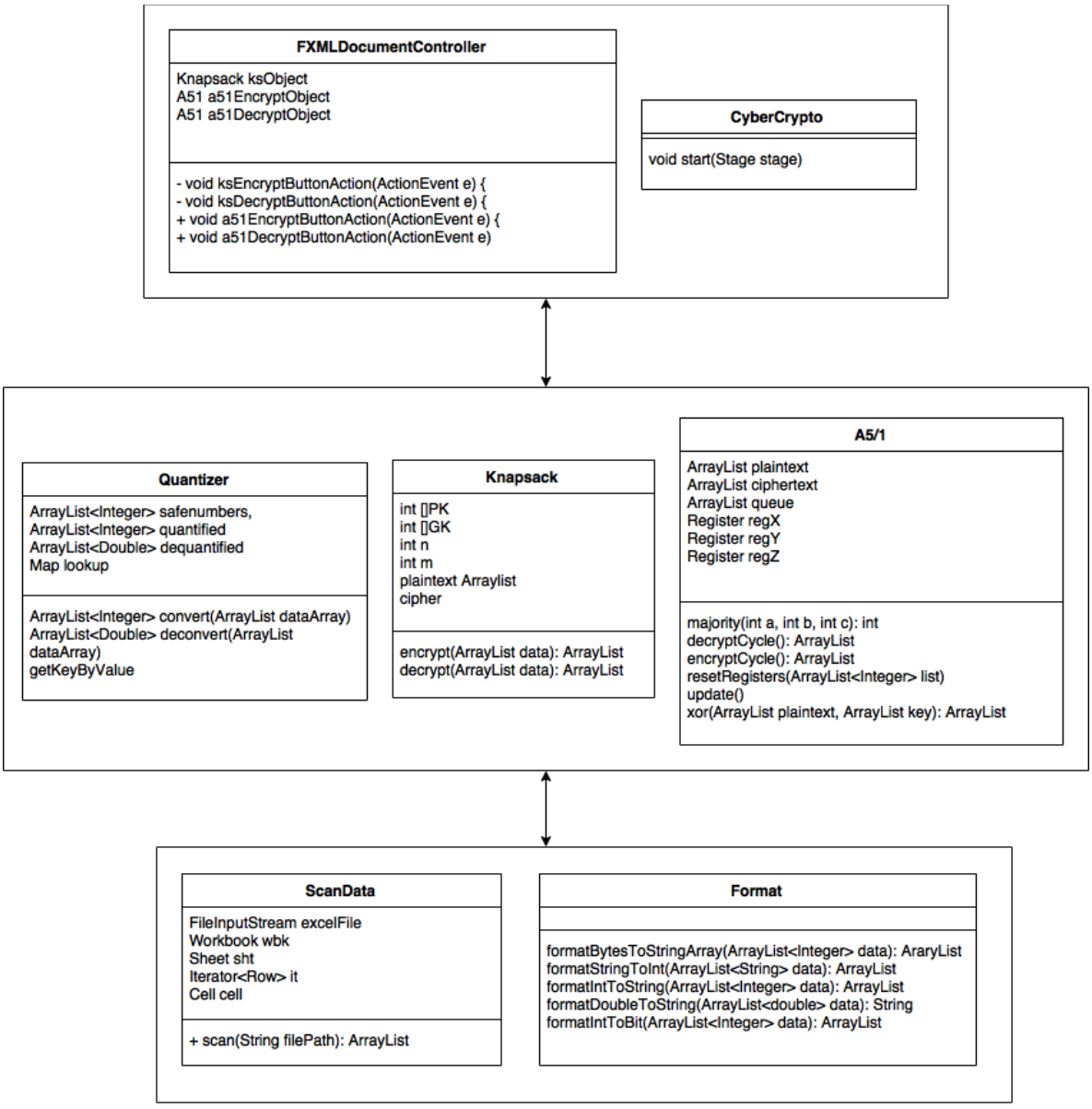
The CyberCrypto project runs utilizing both knapsack and A5/1 classes to cipher EKG data. The entire project uses seven classes which all have its own role in making the app run in unison. The CyperCrypto class launches when the program is started and tells the project to load the GUI from the FXML file and passes it off to the DocumentController file. The FXMLDocumentController class is the main class that streamlines all the encryption and auxiliary classes all while establishing GUI interaction. The ScanData class (not used in current version) is capable of reading data from excel files into the program. The FormatData class has a variety of extraneous methods used for formatting and converting the data from one form to another when required (from ArrayList<Integer> to integer[ ] to String to ArrayList<String>, for example). The quantizer class takes the raw EKG data and converts them into integers.

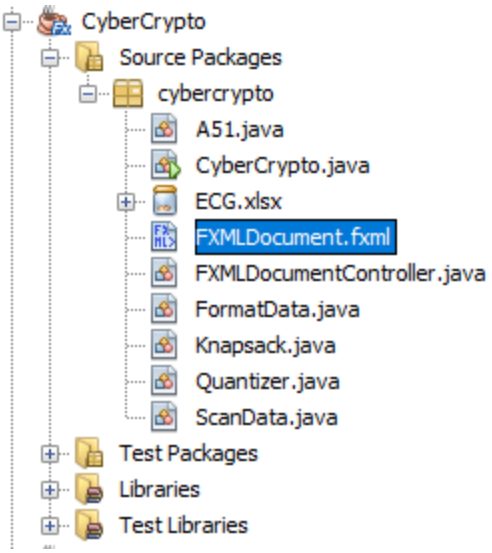
3.2 Approach for design validation

The testing for the program was initially trial and error. We wrote the code then ran the code in NetBeans to see if we obtained the desired outcome. While implementing the code, we saw some necessary corrections that were needed to be made. There was also some layout designs that needed to be made as well. One of the main corrections that was needed was instead of having to read the EKG data from an excel file, which is an external component, we decided to hard code the data into the program. The main reason for hard coding is so anyone could run this program provided that he or she has NetBeans. He or she would not need the external data file to run the program. Other design modifications was to make the Guided User Interface more user friendly and for the GUI layout to make more sense. We do not want anyone having any difficulties operating our program. After each modification, we tested our program to ensure the program ran more smoothly so that we can fix all of the bugs in the program before submission.

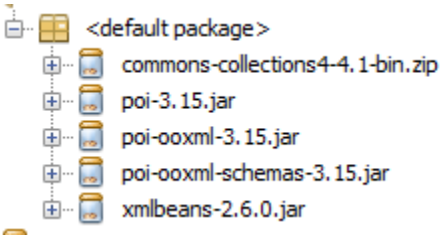
4 Implementation notes

The source package contains seven java classes with the main class being CyberCrypto.java. It also holds one FXML Document and excel spreadsheet provided for the project: “ECG.xlsx”.





There are five libraries used under the libraries directory. These are common libraries and descriptions of each can be found online in detail. For the purpose of this project none of the libraries were manipulated, they were used as is.



Source Package:

1. **CyberCrypto.java** – main class that launches, creating two ArrayLists, assigns the file "ECG.xlsx" to a string "fileName", and loads the GUI from the FXML file. The first ArrayList is a Double ArrayList named "rawData" to hold the values from the ECG.xlsx file and the second is an Integer ArrayList named "dataQuantified" to hold the values after the raw values have been passed through the Quantizer class. These quantized values are non-negative integers.

2. **FormatData.java** – is used to format data being used during the program using Integer and String ArrayLists. There are five methods used converting data to different formats:

    1. **ArrayList<String> formatDoubleToString** – is made of two loops. The first loop converts double values to integer values creating an integer array of the double values multiplied by 100 plus adding 128 and storing them in an "intArray". The second takes the values in the "intArray" and converts those to binary string values with padding to a length of 8 bits. It stores these values in an array named "stringArray" and returns the "stringArray". This method is not used in the current system, but was preserved as a potential resource in expanding/testing software.

2. **ArrayList<String> formatIntToString** – converts integer values to string binary values with padding to a length of 8 bits. It stores these values in an array named “stringArray” and returns the “stringArray”.

3. **ArrayList<Integer> formatIntToBit** – is made of two loops. The first loop converts integer values to binary strings with padding to 8 bits. The second loop converts each binary string character in “stringArray” to an integer value of 1 or 0 and stores them in “intArray”. The method then returns “intArray”.

4. **ArrayList<Integer> formatStringToInt** – converts string binary of 8 bits into equivalent integer decimal values returning the “intArray”.

5. **ArrayList<String> formatBytesToStringArray** – converts an integer ArrayList of individual bit values of 1 or 0 to a consolidated ArrayList of string binaries and returns the “stringArray”.

3. **Quantizer.java** – the class takes the raw EKG data values in and converts them to integers. The Knapsack algorithm involves adding numbers in the Public Key(GK) when encrypting, and those values exceed 256 which a byte of data can only store 256 numbers/permutations. The quantizer class will also check the numbers encrypted with the GK are within an acceptable range and put those “safe numbers” in a ArrayList to be mapped onto the raw data points.

1. **Quantizer()** – the constructor creates a HashMap named “lookup” and three ArrayLists: “safenumbers”, “quantified”, and “dequantified”. Next it will check for safe numbers, coinverting integers to binary strings, and pretending they are going through Knapsack encryption.

```
for (int i = 0; i < 256; i++) {
            String temp = Integer.toBinaryString(i);
            int sum = 0;
            while (temp.length() < 8) {
            temp = "0" + temp;
            }

            for (int j = 0; j < 8; j++) {
            if (temp.charAt(j) == '1') {
                        sum += gk[j];
      }
      }
```

If the sum is below 256 then lets add the number to our safe ArrayList.

```
if (sum < 256) {
            safenumbers.add(i);
}
```

2. **ArrayList<Integer> convert(ArrayList dataArray)** – with the safe numbers created this method will randomize the list to make the quantized numbers less predictable.

```
int arrayIndex = 0;
long seed = System.nanoTime();
Collections.shuffle(safenumbers, new Random(seed));
```

For each value in the array, check to see if it is in the "lookup" map. If it isn't, map it to a random "safenumber".

```
for (int i = 0; i < dataArray.size(); i++) {
             if (!lookup.containsKey((dataArray.get(i)))) {
 lookup.put(dataArray.get(i), safenumbers.get(arrayIndex));
                      arrayIndex++;
             }
}
```

After every raw data point is mapped to a "safenumber", run through the array one more time and dynamically create a new one with the corresponding mapped values. This will be the array that is encrypted and decrypted.

```
for (int i = 0; i < dataArray.size(); i++) {
       quantified.add((Integer) lookup.get(dataArray.get(i)));
}
return quantified;
```

3. **ArrayList<Double> deconvert(ArrayList inputArray)** – the method takes the array of safenumbers and remaps them to their original data points.

```
ArrayList<Double> temp = new ArrayList<>();
for (int i = 0; i < inputArray.size(); i++) {
          temp.add((Double) getKeyByValue(lookup, inputArray.get(i)));
}
return temp;
```

4. **<T, E> T getKeyByValue(Map<T,E> map, E value**) – the method allows searching of the map for a value, quantized safe number version, and returning the corresponding key (raw EKG data point).

4. **ScanData.java** – the class has the EKG data hard-corded to an ArrayList<Double> "dataArray" and returns "dataArray".and is also capable of reading in excel files.

```
public static ArrayList scan(String filePath) {

ArrayList<Double> dataArray = new ArrayList<>(Arrays.asList(-0.01,
            -0.02, -0.02, -0.01, -0.01, 0.0, 0.0, -0.01, -0.01, 0.0, etc.))
 return dataArray;
}
```

5. **FXMLDocument.fxml** – with JavaFX SceneBuilder 2.0 installed the file when selected open in NetBeans will open the SceneBuilder and allow manipulation of the Graphic User Interface as needed visually instead of coding manually.

```
<AnchorPane      id="AnchorPane"      prefHeight="825.0"      prefWidth="1123.0"
xmlns="http://javafx.com/javafx/8"          xmlns:fx="http://javafx.com/fxml/1"
fx:controller="cybercrypto.FXMLDocumentController">
    <children>
        <Button       fx:id="button"       layoutX="70.0"       layoutY="153.0"
onAction="#handleButtonAction" text="Read Data from File" />
        <Label    fx:id="label"    layoutX="126"    layoutY="120"    minHeight="16"
minWidth="69" />
      <LineChart     fx:id="chart1"     createSymbols="false"     layoutX="42.0"
layoutY="607.0" prefHeight="204.0" prefWidth="324.0" title="Plaintext Chart">
        <xAxis>
          <NumberAxis side="BOTTOM" fx:id="x1" />
        </xAxis>
        <yAxis>
          <NumberAxis fx:id="y1" side="LEFT" />
        </yAxis>
      </LineChart>
ETC.
</AnchorPane>
```

[JavaFX Scene Builder 2.0]

6. **FXMLDoumentController.java** – the class along with FXMLDocument.fxml establishes all the GUI objects and interactions; making calls to the other classes and methods based upon the button selections or text input. There are 12 objects declared immediately in the class allowing all methods to have access to the same variables. The first 5 are ArrayLists: “rawData”, “stringData”, “quantifiedData”, “encrypted”, and “decrypted”. The arrays hold the respective values as named. The string “input” object is a string variable used to capture the IV values for A5/1and the string “filePath” is the path to the ECG.xlsx file. The FormatData “formatObject”, Quantizer “QuantizerObject”, Knapsack “ksObject”, A51 “a51EncryptObject”, and the A51 “a51DecryptObject” variables are used to make calls to the respective classes and methods.

1. **clearGraphButtonHandler** – when selected clears the data from all three charts: “chart1”, “chart2”, and “chart3”.

2. **handleButtonAction** - the method populates the EKG data on the Raw Data tab, the Quantized tab, and Plaintext Chart.

3. **ksEncryptButtonAction** - the method quantizes the raw data, encrypts it using knapsack, and populates the Encrypted tab and chart. It creates a new Quantizer() object each call which will have the encrypted data vary as the Quantizer.lookup Map us instantiated based on the Random seed variable.

4. **ksDecryptButtonAction** - the method retrieves the last encrypted Knapsack arraylist, decrypts it, formats the decrypted binary string arraylist into an integer arraylist for dequantizing using the FormatData.formatStringToInt method, runs the quantized values through the “lookup” map to retrieve raw values, and populates the Decrypt tab and chart.

5. **A51EncryptButtonAction** - the method currently uses a basic input handler. Currently it does not check for wrong chars or out of bound inputs. A coded default value populates the IV. The method quantizes the raw data, converts the arraylist of integers to an arraylist of bits for encryption, creates an A51 object with the user input or default input and the prepared EKG data in bits, stores the encrypted data in “ciphertextCopy”, formats the data as 8 bit integer representation strings in “ciphertextStringArray”, converts the string arraylist into integers, and populates the Encrypt tab and chart.

6. **A51DecryptButtonAction** - the method creates an A51 object to decrypt the encrypted data. It is passed the same IV as the encrypted object and takes the ciphertext as a parameter instead of plaintext. It stores the decrypted data in an Integer arraylist “ciphertextCopy”, converts the “ciphertextCopy” arraylist to Bytes of string stored in an “ciphertextStringArray”, converts the string arraylist to integers in “tempArray”, runs the quantized values through the “lookup” map to retrieve raw values, and populates the Decrypt tab and chart.

7. **initialize** - the method creates objects that are common to the different methods and are instantiated upon the application being launched including the text that populate the GUI instructions and explanations:

```
FormatObject = new FormatData();
rawData = scan(filePath);
ksObject = new Knapsack();
```

```
                    Text knapsackText = new Text("A public key cryptosystem created
in 1978, Knapsack uses a public key for encryption and private/general key for
decryption. Once a"
                + " private key is selected (a set of 8 super-increasing numbers), the
general key is generated using the formula n * PK[i] mod q");

                    textflow1.getChildren().add(knapsackText);

                        Text knapsackText2 = new Text("For demonstration, keys are
hardcoded as follow: \n\n PK = [1,2,4,8,16,32,64,128] "
                + "\n GK = [17,34,68,136,16,32,64,128] \n n (multiplier) = 17, m
(modulus) = 256");

                    textflow2.getChildren().add(knapsackText2);

                        Text a51text = new Text("A stream-cipher developed for
cell-phone communications, A5/1 uses a 64 bit queue to populate 3 registers, the
contexts"
                + "of which are XOR'ed and shifted to produce a pseudorandom keystream
against which the plaintext is ciphered. Cipher-feedback chaining is also"
                + "used to further complicate the crypto-algorithm.");

                        a51textflow.getChildren().add(a51text);
```

8. **chartData** - the method plots the charts as called.

9. **populateGridPane** - the method populates the different tabs as called.

7. **A51.java** - the class variables include 3 registers (shifting arrays) and arraylists that contain different elements of a51 (the queue, the keystream, the plaintext, and the ciphertext). When encrypting, the plaintext is passed into the class constructor, along with the initiation vector which populates the queue and registers. Every cycle/pulse generates a keystream bit. After 8 pulses, the resulting 8 bit keystream is XOR'ed with 8 bits of plaintext to create 8 bits of ciphertext. That 8-bit ciphertext is then copied into the right-most 8 bits of the queue, and the rest of the queue is shifted left 8 times. Notice for decrypting, ciphertext is passed in as the plaintext parameter along with the same IV vector. However, after the keystream is XOR'ed with the ciphertext to create plaintext, notice that 8 bit portions of the ciphertext (corresponding with the length of the keystream) are then fed into the feedback mechanism, so its not an exact reverse of the encryption process.

```
public class A51 {

    Register regX;
    Register regY;
    Register regZ;

    ArrayList<Integer> keystream;
    ArrayList<Integer> queue;
    ArrayList<Integer> plaintext;
    ArrayList<Integer> ciphertext;

    //these values used for testing. they generate a default a51 object if no params
are passed
    int xSample = 349525;
    int ySample = 3355443;
    int zSample = 7401712;

    public A51() {
```

```
    //no arg constrcutor for testing
        regX = new Register(19, xSample);
        regY = new Register(22, ySample);
        regZ = new Register(23, zSample);

    }

    public A51(String stringInitVector, ArrayList<Integer> plain) {

        plaintext = plain;
        keystream = new ArrayList<Integer>();
        queue = new ArrayList<Integer>();
        ciphertext = new ArrayList<Integer>();

        //this method passes the IV into the queue
        queueStringInit(stringInitVector);

        int[] x = new int[19];
        int[] y = new int[22];
        int[] z = new int[23];

        //formatting to convert the passed String IV into 3 arrays to create the 3
registers
        String[] stringArray = stringInitVector.split("");
        for (int i = 0; i < 19; i++) {
            x[i] = Integer.parseInt(stringArray[i]);
        }
        for (int i = 0; i < 22; i++) {
            y[i] = Integer.parseInt(stringArray[i + 19]);
        }
        for (int i = 0; i < 23; i++) {
            z[i] = Integer.parseInt(stringArray[i + 41]);
        }

        //creating registers. see Register class below
        regX = new Register(19, x);
        regY = new Register(22, y);
        regZ = new Register(23, z);
    }

    //the majority method, used to discern which registers get shifted each encryption
pulse
    int majority(int x, int y, int z) {
        if ((x == 0 && y == 0) || (y == 0 && z == 0) || (x == 0 && z == 0)) {
            return 0;
        }
        return 1;
    }

    //based on majoriy values, approrpriate registers get shifted. the cipheroutput
bit is also produced and added to the keystream.
    int pulse() {

        int m = majority(regX.get(8), regY.get(10), regZ.get(10));
        //System.out.println("M: " + m);
        int cipherOutput = (regX.get(18) ^ regY.get(21) ^ regZ.get(22));

        if (m == regX.get(8)) {
            int t = regX.get(13) ^ regX.get(16) ^ regX.get(17) ^ regX.get(18);
            regX.push(t);
        }
        if (m == regY.get(10)) {
            int t = regY.get(20) ^ regY.get(21);
```

```
            regY.push(t);
        }

        if (m == regZ.get(10)) {
            int t = regZ.get(7) ^ regZ.get(20) ^ regZ.get(21) ^ regZ.get(22);
            regZ.push(t);
        }

        keystream.add(cipherOutput);
        return cipherOutput;
    }

    //XOR method. takes the last 8 bits of the keystream and XOR's with respective bits from plaintext
    ArrayList<Integer> xor(ArrayList<Integer> plaintext, ArrayList<Integer> keytext) {

        ArrayList<Integer> solution = new ArrayList<>();

        int keyIndex = keytext.size();

        for (int i = 8; i > 0; i--) {
            solution.add(plaintext.get(keyIndex - i) ^ keytext.get(keyIndex - i));
        }

        return solution;
    }

    //this method reflects the cipher-feedback process in our algorithm. after pulsing 8 times, this method is called to XOR the keystream with plaintext, and copy resulting last 8 bits of ciphertext into the queue while resetting registers
    void update() {

        //shifting queue left 8 units
        for (int i = 0; i < 56; i++) {
            queue.set(i, queue.get(i + 8));
        }

        //adding results of plaintext XOR keystream to ciphertext
        ciphertext.addAll(xor(plaintext, keystream));

        //takes last 8 bits of ciphertext and copies to right 8 bits in queue
        int ciphertextIndexOfLast = ciphertext.size();

        for (int i = ciphertextIndexOfLast - 8; i < ciphertextIndexOfLast; i++) {
            queue.set((i - ((ciphertextIndexOfLast - 8)) + 56), ciphertext.get(i));
        }

        //using the new queue, populates registers with new data
        resetRegisters((ArrayList<Integer>) queue);
    }

    //consolidates the amount of pulses/updates for the entire EKG file
    public ArrayList<Integer> encryptCycle() {

        for (int i = 0; i < 416; i++) {
            for (int j = 0; j < 8; j++) {
                pulse();
            }
            update();
        }
        return ciphertext;
    }
```

```
    //consolidates the total amount of pulses/cycles required for entire EKG file
    public ArrayList<Integer> decryptCycle() {

        for (int i = 0; i < 416; i++) {
            for (int j = 0; j < 8; j++) {
                pulse();
            }
            updateDecrypt();
        }
        return ciphertext;
    }

    //similar to the other update, but for decrypting. after every 8 pulses, this call
XORs the keystream with ciphertext to create plaintext, and then updates the queue
(and registers) by pushing the 8 corresponding bits of ciphertext to the queue
    void updateDecrypt() {

        for (int i = 0; i < 56; i++) {
            queue.set(i, queue.get(i + 8));
        }

        //adding results of plaintext XOR keystream to ciphertext
        ciphertext.addAll(xor(plaintext, keystream));

        //takes last 8 bits of ciphertext and copies to right 8 bits in queue
        int ciphertextIndexOfLast = ciphertext.size();

     //notice here the bits being pushed into queue are from the ciphertext generated
during encryption. although the information is stored in a variable named "plaintext,"
the encrypted ciphertext is what was actually passed into that plaintext variable for
decryption
        for (int i = ciphertextIndexOfLast - 8; i < ciphertextIndexOfLast; i++) {
            queue.set((i - ((ciphertextIndexOfLast - 8)) + 56), plaintext.get(i));
        }

        //using the new queue, populates registers with new data
        resetRegisters(queue);

    }

    //this method passes in new queue after an update and updates registers
    void resetRegisters(ArrayList<Integer> arraylist) {

        int[] x = new int[19];
        for (int i = 0; i < 19; i++) {
            x[i] = arraylist.get(i);
        }
        int[] y = new int[22];
        for (int i = 0; i < 22; i++) {
            y[i] = arraylist.get(i + 19);
        }
        int[] z = new int[23];
        for (int i = 0; i < 23; i++) {
            z[i] = arraylist.get(i + 41);
        }
        regX.set(x);
        regY.set(y);
        regZ.set(z);

    }

    //prints array contents
    static void intArrayPrint(Register reg) {
```

```
    int[] temp = reg.getArray();
    for (int i = 0; i < temp.length; i++) {
        System.out.print(temp[i]);
    }
    System.out.println("");
}

//used for testing, a method to simplify printing register contents
void registerPrint() {
    intArrayPrint(regX);
    intArrayPrint(regY);
    intArrayPrint(regZ);
}

//initiates IV into queue
void queueStringInit(String s) {
    String[] stringArray = s.split("");
    for (int i = 0; i < stringArray.length; i++) {
        queue.add(Integer.parseInt(stringArray[i]));
    }
}

//setter
void setPlaintext(ArrayList<Integer> plaintext) {
    this.plaintext = plaintext;
}

//class that defines behavior of registers
class Register {

    int[] registerArray;

    //register constructor for an array size and value in decimal
    public Register(int size, int decimalData) {

        registerArray = new int[size];
        String stringBinary = Integer.toBinaryString(decimalData);
        while (stringBinary.length() < size) {
            stringBinary = "0" + stringBinary;
        }
        String[] stringArray = stringBinary.split("");

        for (int i = 0; i < stringArray.length; i++) {
            registerArray[i] = Integer.parseInt(stringArray[i]);
        }
    }

    //alternative constructor using size and integer array
    public Register(int size, int[] intArrData) {

        registerArray = new int[size];

        for (int i = 0; i < intArrData.length; i++) {
            registerArray[i] = intArrData[i];
        }
    }

    //when registers pulsing, this method pushes in new value and shifts array
    void push(int t) {

        for (int i = registerArray.length - 1; i > 0; i--) {
            registerArray[i] = registerArray[i - 1];
```

```
            }
            registerArray[0] = t;
        }

        int[] getArray() {
            return registerArray;
        }

        int get(int i) {
            return registerArray[i];
        }

        void set(int[] intArray) {
            registerArray = intArray;
        }
    }
}
```

8. **Knapsack.java** -

```
public class Knapsack {

    int[] PK = new int[8];
    int n = 256; //modulus
    int m = 17; //multiplier
    int[] GK = new int[8];
    //bigInteger form of PK used for inverse mod calculations
    BigInteger[] bigPK = new BigInteger[8];
    ArrayList<String> lastEncryptedData;

    public Knapsack() {
        //pk = private key, gk = public key. though hardcoded in this project, gk is generally calculated via the pk and n and m  ( GK[i] = m * PK[i] mod (n) dynamic way to generate GK ->  for(int i = 0; i < 8; i++) { GK[i] = (PK[i] * m) * % n; //GK[i] = m.multiply(SIK[i]).mod(n); }

        PK = new int[]{1, 2, 4, 8, 16, 32, 64, 128};
        GK = new int[]{17, 34, 68, 136, 16, 32, 64, 128};

        //equivalent BigInteger array for PK, used for some Math methods
        for (int x = 0; x < 8; x++) {
            bigPK[x] = BigInteger.valueOf(PK[x]);
        }
    }

      //encryption method. takes in an arraylist of bytes in string-binary format (e.g. 01010101). based on which bits are true (1), corresponding number of that index in general key is added to a sum. once the sum is calculated it is then converted to a byte in string-binary notation
    ArrayList<String> encrypt(ArrayList<String> stringArray) {

        ArrayList<String> encryptedArray = new ArrayList<>();
        ArrayList<Integer> intArray = new ArrayList<>();

        //for each string of 8 bits in the arraylist
        for (int i = 0; i < stringArray.size(); i++) {

            int sum = 0;
            String byteString;
```

```
            byteString = stringArray.get(i);
            //checks every bit. if true (=='1'), its respective GK value added to sum
            for (int j = 0; j < 8; j++) {

                if (byteString.charAt(j) == '1') {
                    sum += GK[j];

                }
            }
            intArray.add(sum);
        }

        //this for loop converts the integer array calculated above into a string-
binary
        //representation
        for (int i = 0; i < intArray.size(); i++) {
            int x = intArray.get(i);
            String num = Integer.toBinaryString(x);

            //padding in case the binary representation isn't 8 bits long
            if (num.length() < 8) {

                while (num.length() < 8) {
                    num = "0" + num;
                }
            }
            //padding incase binary representation is over 8 bits, should never
happen, would cause data loss
            if (num.length() > 8) {
                num = num.substring(num.length() - 8, num.length());
            }
            encryptedArray.add(num);
        }
        //this random variable useful to make sure eventual decrypting occurs on the
most recently generated encrypted data. In the
        //GUI, users can generate multiple knapsack data sets, each with their own
lookup value (see quantizer and controller classes)
        lastEncryptedData = encryptedArray;
        return encryptedArray;

    }//end encrypt method

    ArrayList<String> decrypt(ArrayList<String> encryptedArray) {

        BigInteger bigN = BigInteger.valueOf(n);
        BigInteger bigM = BigInteger.valueOf(m);
        ArrayList<String> decryptedArray = new ArrayList<>();
        BigInteger invMod;

        //some math to decrypt
        for (int i = 0; i < encryptedArray.size(); i++) {

            int intValue = Integer.parseInt(encryptedArray.get(i), 2);
            BigInteger cipher = BigInteger.valueOf(intValue);
            invMod = (cipher.multiply(bigM.modInverse(bigN))).mod(bigN);

            int startIndex = PK.length - 1;
            String plain = "";

            //decomposing value over private key to get plaintext data back
            while (startIndex >= 0) {
```

```
                // In this case, DON'T add this number in the subset, so we get a
                // 0 for this bit of the plantext.
                if (invMod.compareTo(bigPK[startIndex]) < 0) {
                    plain = "0" + plain;
                } // Here we get a 1 for the plaintext bit
                else {

                    plain = "1" + plain;
                    invMod = invMod.subtract(bigPK[startIndex]);
                }

                // Go to figure out the next bit.
                startIndex--;
            }
            decryptedArray.add(plain);
        }
        return decryptedArray;
    }//end decrypt method

    public ArrayList<String> getLastEncrypted() {

        return lastEncryptedData;

    }

}
```

5 Experimental results

There is testing code used in A51.java, ScanData.java, and Quantizer.java to validate the functions. The coding is part of the source package and is grayed.

1. A51.java test code was used to verify the function the of A5/1 coding:

```
/*
    public static void main(String[] arg) {

        //a.registerPrint();
        //a.pulse();
        String xdata = "1010101010101010100";
        String ydata = "11001100110011001100110010";
        String zdata = "111000111000111000111100";

        String iv = xdata + ydata + zdata;
        System.out.println("iv is : " + iv);
        //String iv =
"11101011010000100111101110001110100011000101010111111110110101001";

        //System.out.println(stringArray.length);
        /*TestJavaStuff a = new TestJavaStuff(iv);
        a. registerPrint();
        for (int i = 0; i < 3; i++) {
            System.out.println(a.pulse());
            a.registerPrint();
        }
 */
```

```
//System.out.println(iv.length());
/* a.registerPrint();
        int pulseAmount = 8;

        for (int i = 0; i < pulseAmount; i++) {

            System.out.println("C : " + a.pulse());
            a.registerPrint();
        }

        //System.out.println(Arrays.toString(a.ciphertext.toArray()));
        //a.update();
        System.out.println(a.queue);
        System.out.println(a.ciphertext);
        a.update();
        System.out.println(a.queue);
        a.registerPrint();

        //intArrayPrint(a.regZ);
    }  */
```

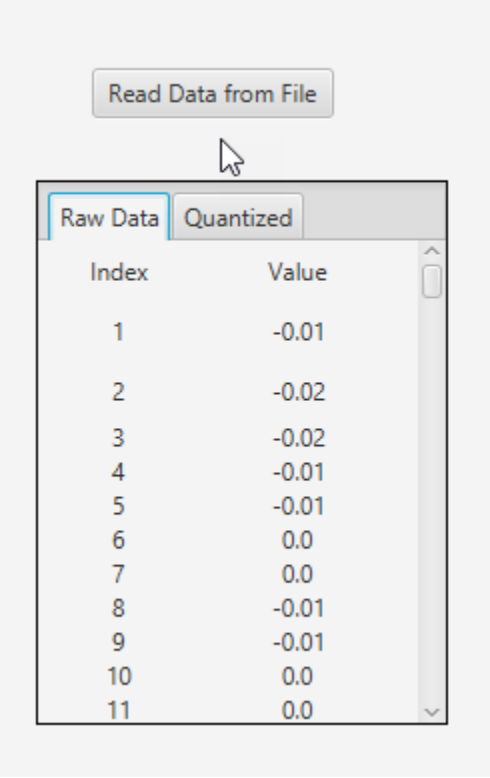
2. ScanData.java line 55 to 82 are coded to read the data from the provided Excel spreadsheet into ArrayList. This code was substituted for hard coded data for deployment compatibility purposes(it requires the five dependencies to run):

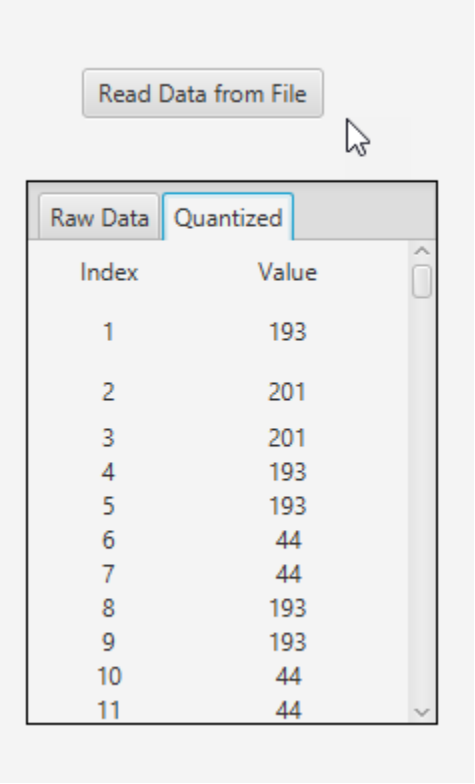
```
/*
         try {

             FileInputStream excelFile = new FileInputStream(new
File(filePath));
             Workbook workbook = new XSSFWorkbook(excelFile);
             Sheet firstSheet = workbook.getSheetAt(0);
             Iterator<Row> iterator = firstSheet.iterator();

             while (iterator.hasNext()) {
                 Row nextRow = iterator.next();
                 Iterator<Cell> cellIterator = nextRow.cellIterator();

                 while (cellIterator.hasNext()) {
                     Cell cell = cellIterator.next();
                     dataArray.add(cell.getNumericCellValue());
                 }
             }
             workbook.close();
             excelFile.close();

         } catch (FileNotFoundException e) {
             e.printStackTrace();
         } catch (IOException e) {
             e.printStackTrace();
         }
         */
```

3. Quantizer.java lines 63 to 65 to see the ArrayList created:

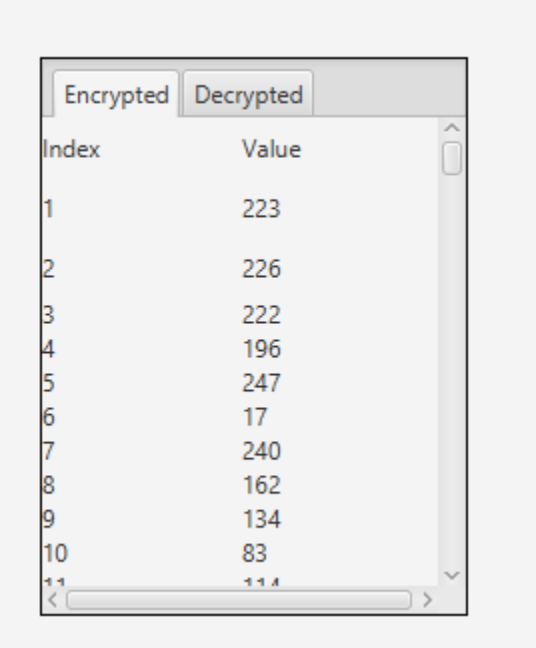
```
counter++; //for testing
      }//end of loop

      //System.out.print(arraylist);
```

Raw Data **Quantified Data**

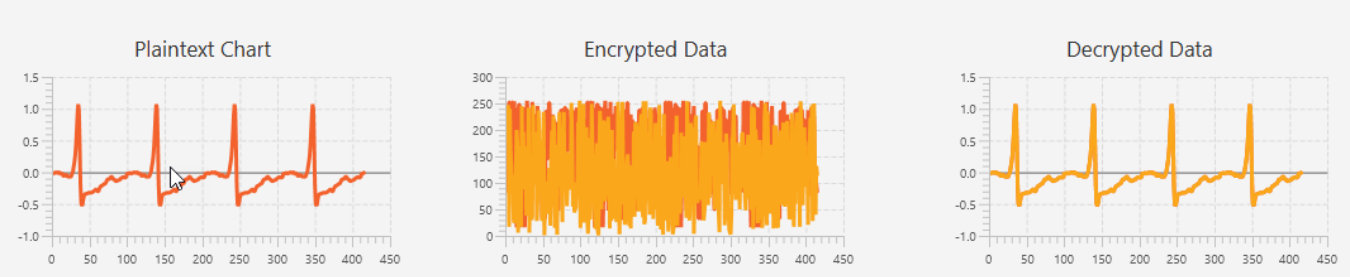




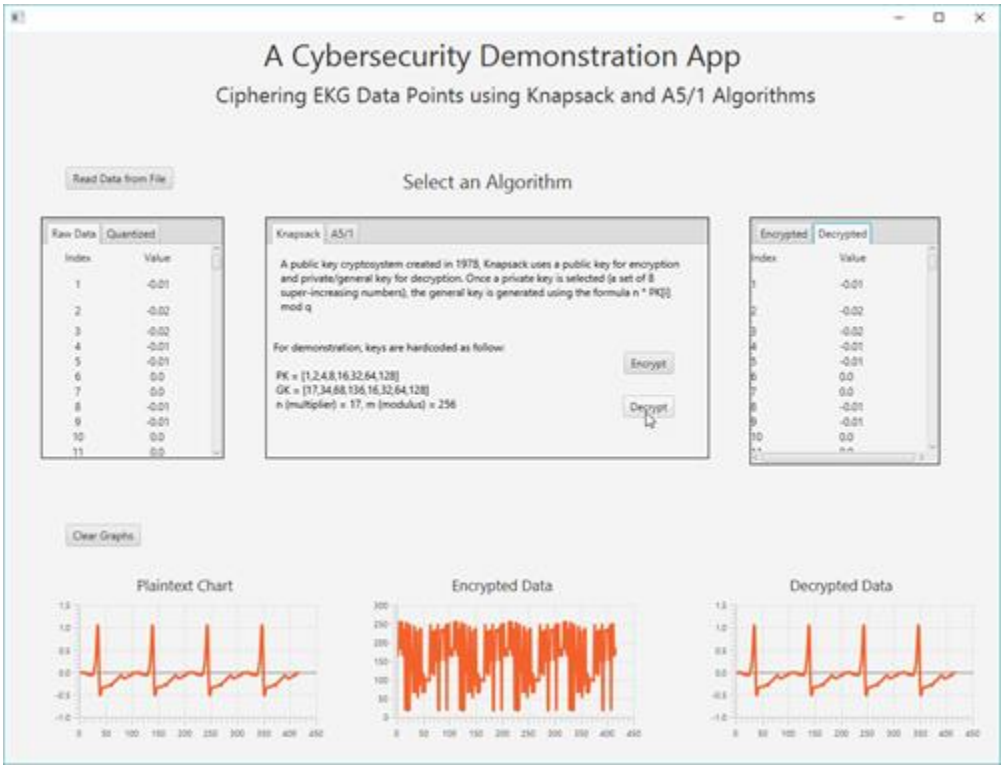
Encrypted Data **Decrypted Data**



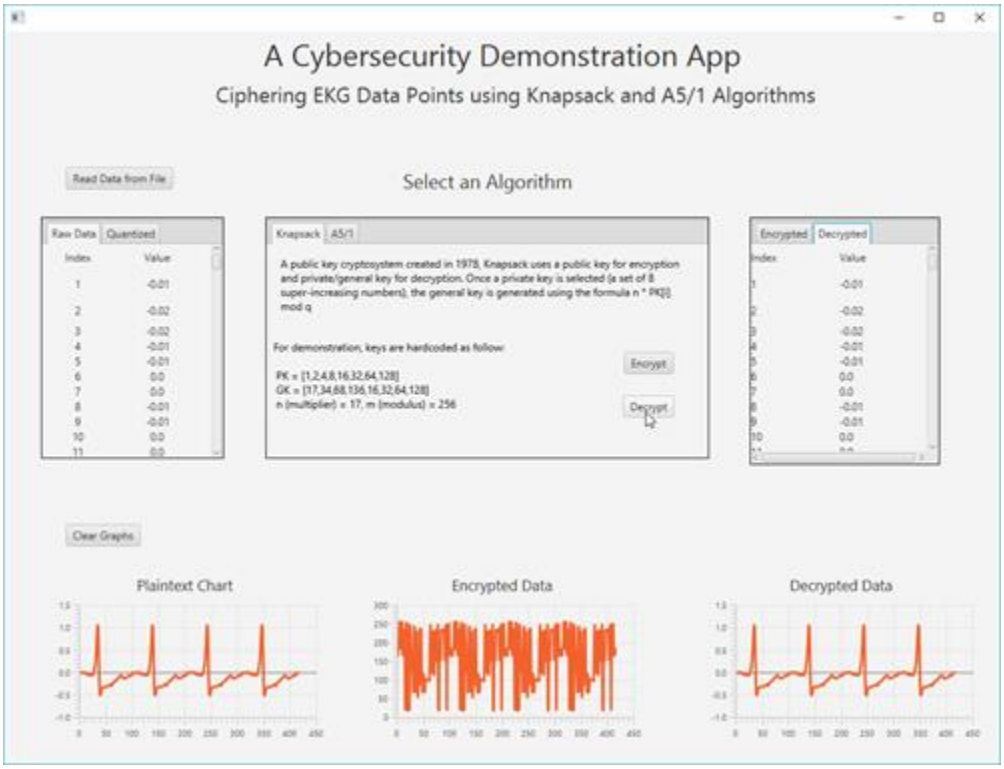
Graphs



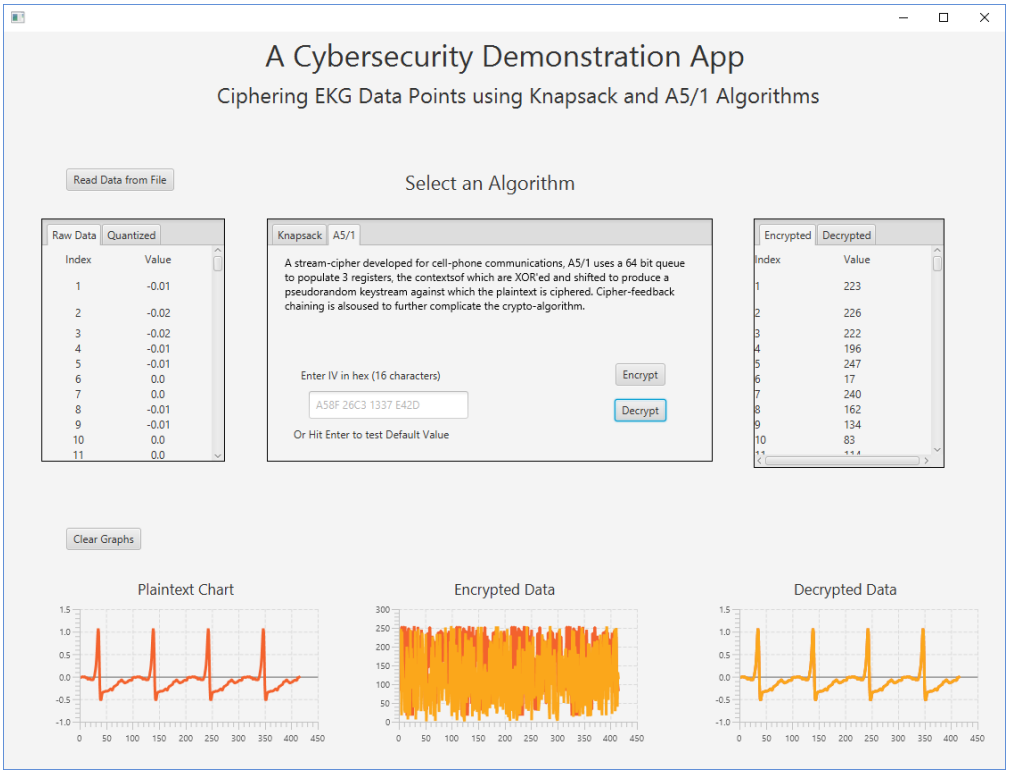
Knapsack Algorithm will raw data, quantified data, encrypted data and decrypted data with graphs.



A5/1 Algorithm will raw data, quantified data, encrypted data and decrypted data with graphs.



Knapsack and A5/1 Algorithm will raw data, quantified data, encrypted data and decrypted data with graphs.



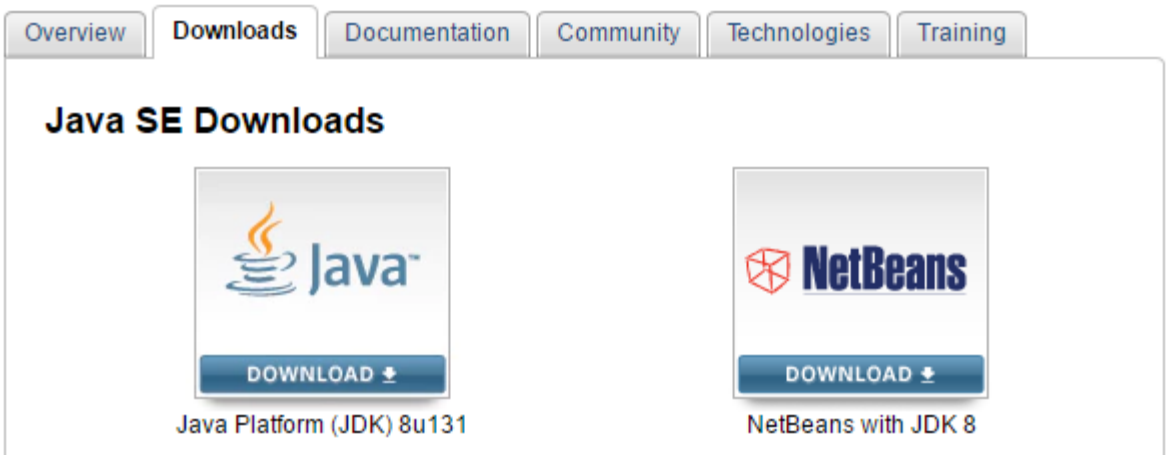
6 User's Manuals

Hardware Requirements:
Verify your system meets or exceeds the minimum hardware requirements:
800MHz Intel Pentium III or equivalent
512MB Ram
750MD of free disk space
Administrator privileges on your system
The instructions are designed to assist a user with knowledge using an Integrated Development environment(IDE) and Java programming; however, the installation and operation could be followed by an average user of a computer, familiar with downloading and installing software and see the project run successfully on a Windows based machine. Do not select Windows x64 downloads unless you know you are running a 64-bit windows operating system.

Software Installation
The project file the user should have on their desktop or "...\downloads" folder is:

There are additional pieces of software needed beside the project file to make the application run with a new installation. The approach for this installation will be using NetBeans IDE 8.2. It simplifies the process of running all the different pieces of software and allows the user to see the code being used to better understand how it works.
Before installing NetBeans, you must have the Java SE Development Kit(JDK) 8 installed. You can download the latest update of a the JDK at
http://www.oracle.com/technetwork/java/javase/downloads .

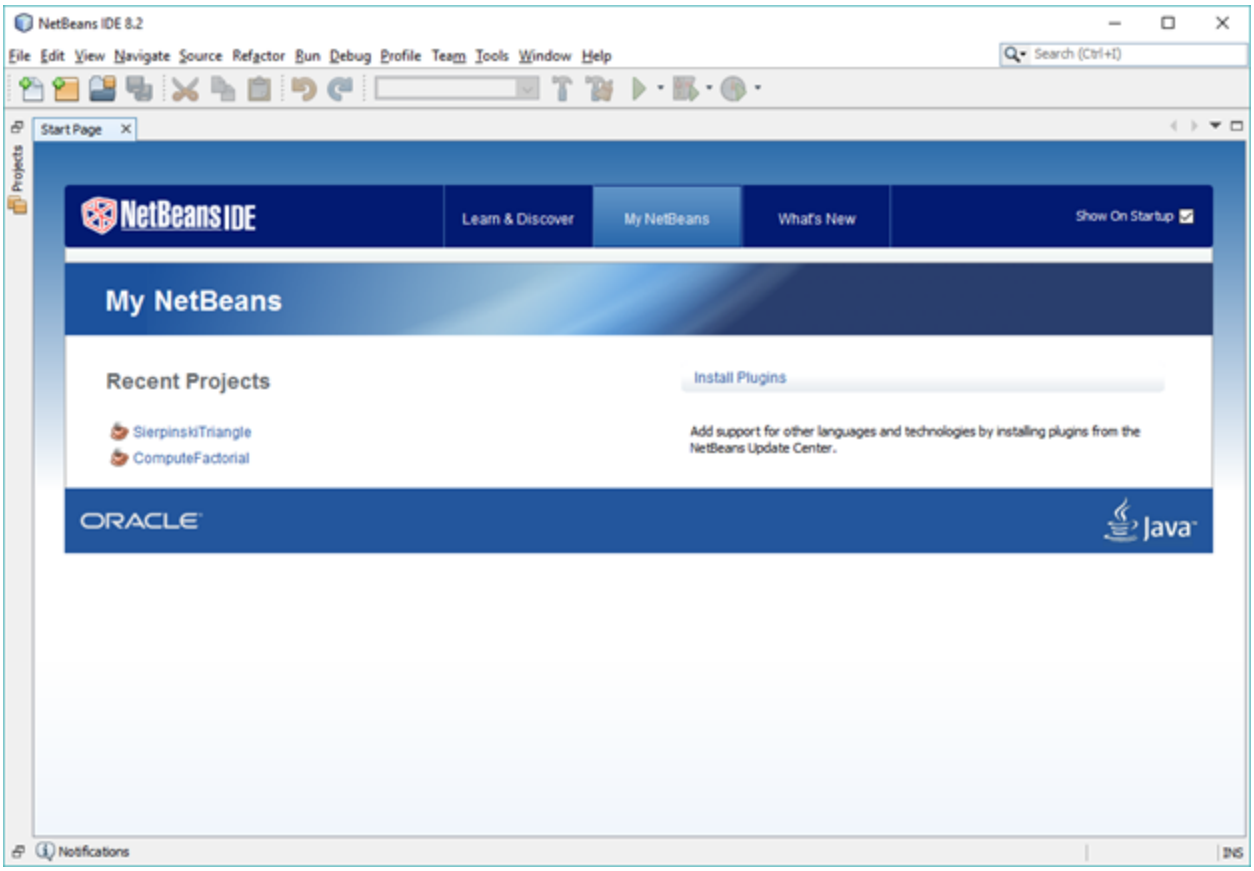
The link above gives two options:



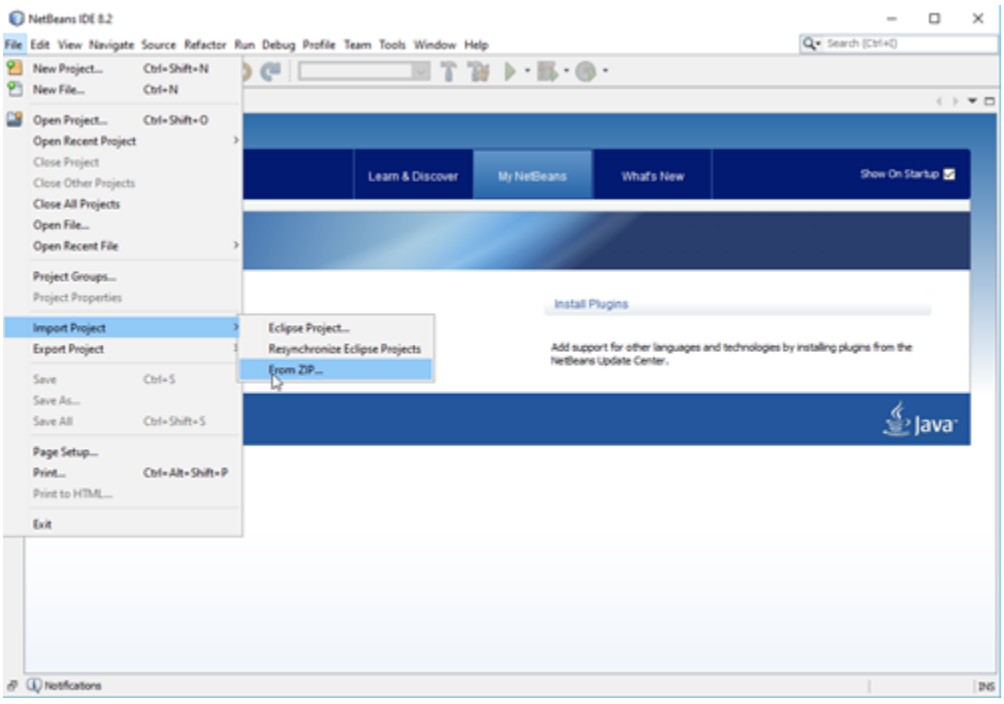
Please select the NetBeans download, accept the license agreement, and select Windows x86 or windows x64.
For specific installation instructions for JDK 8 and NetBeans IDE 8.2 software bundle:
http://www.oracle.com/technetwork/java/javase/downloads/install-jdk6-22nb691-177131.html
After the download completes, please install the software bundle following the setup instructions or refer to the installation instructions link above.

If you prefer to install them separate proceed to download NetBeans IDE 8.2 using the following link: https://netbeans.org/downloads/ after downloading and installing the Java Platform(JDK) 8 with the icon on the left above.
There are multiple download bundles available on NetBeans' website and they are all free. The Java SE, Java EE, and the All version will work for the project's application.
The Java SE version takes up minimal space if that is a concern.
Same as the JDK 8 download and installation, please download the version you prefer and follow the NetBeans' installation instructions during the setup process or select the link below:
https://netbeans.org/community/releases/82/install.html
Once both are installed you should now have a nice-looking NetBeans' icon on your desktop:

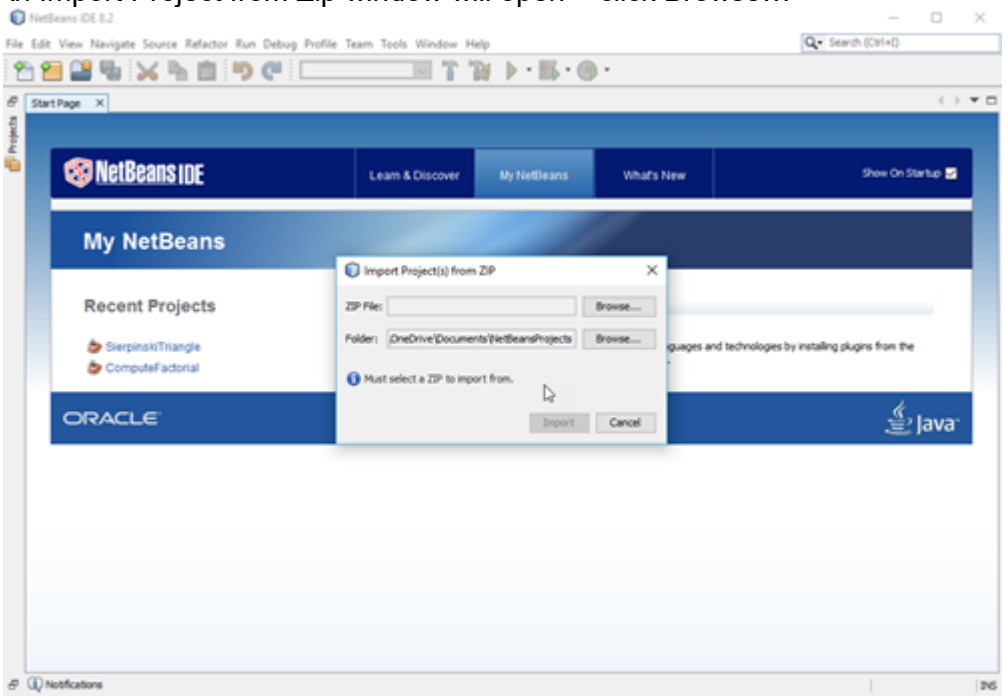
Clicking on the icon will give you a clean start screen once in NetBeans:



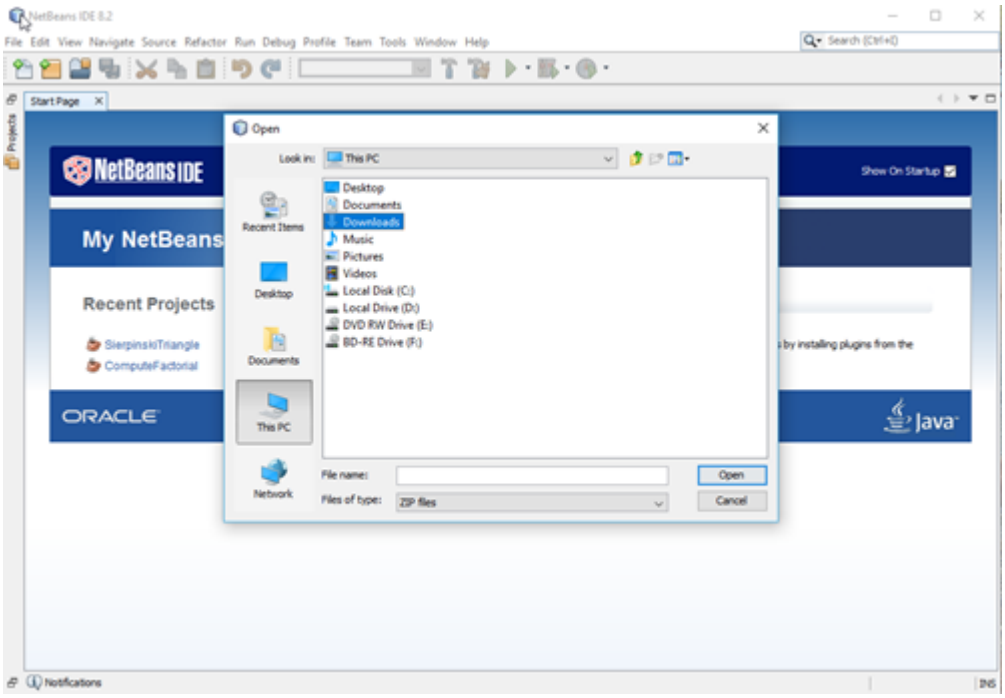
Now that NetBeans is running, you will need to import the Cyber-Crypto master.zip file. Click on File > Import Project > From Zip…



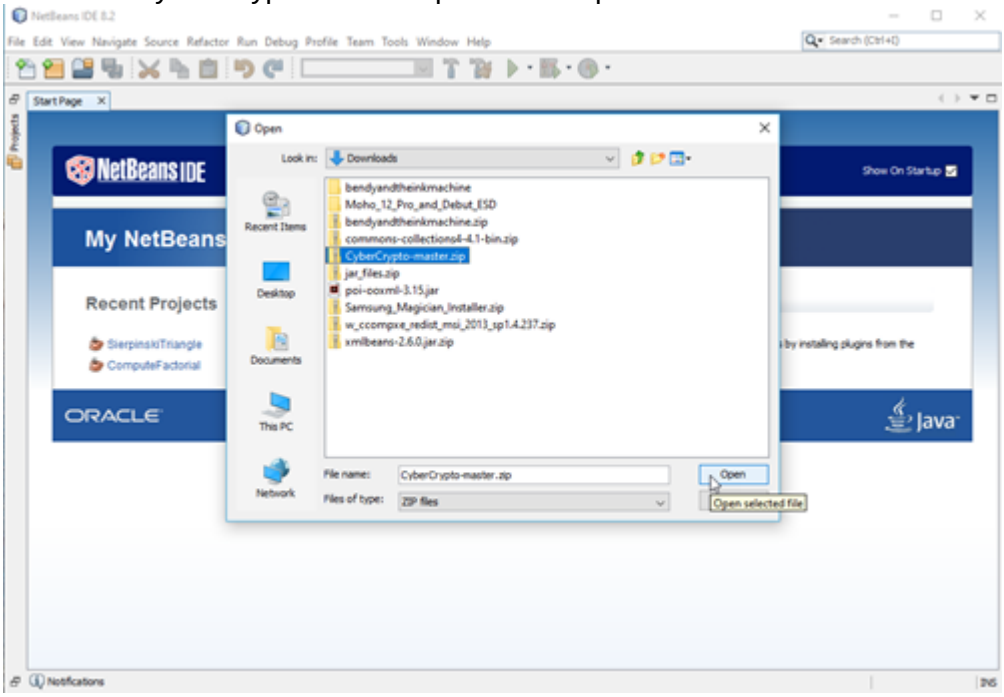
An Import Project from Zip window will open > click Browse…



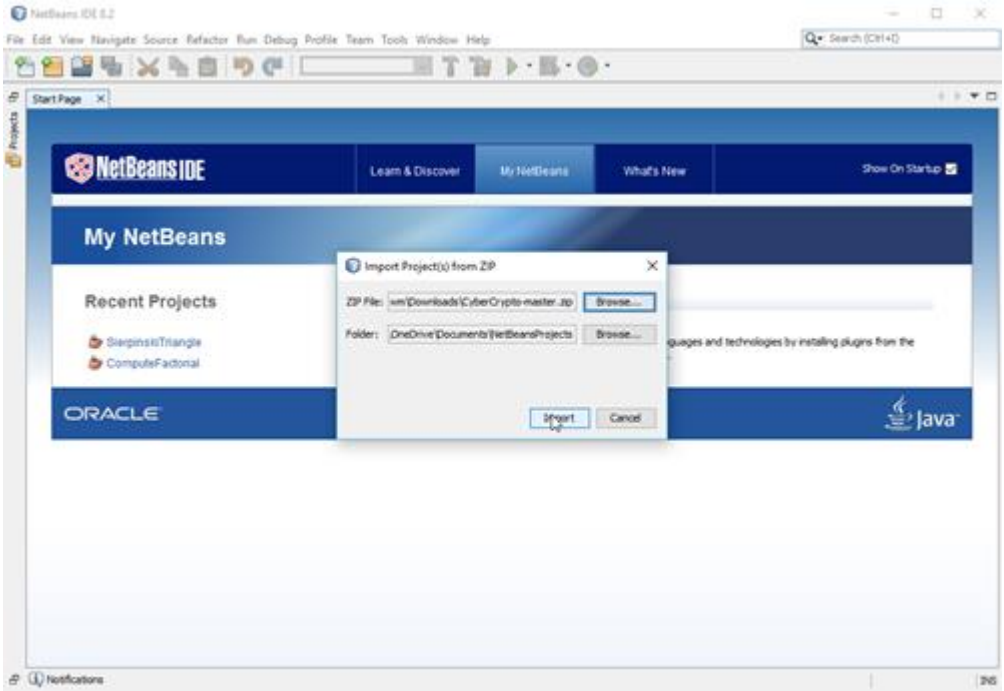
Find the location you saved the Cyber Crypto-master.zip file in. Typically it will be in your C:\Users\"User Name"\Downloads folder or you may have saved it to your desktop. \



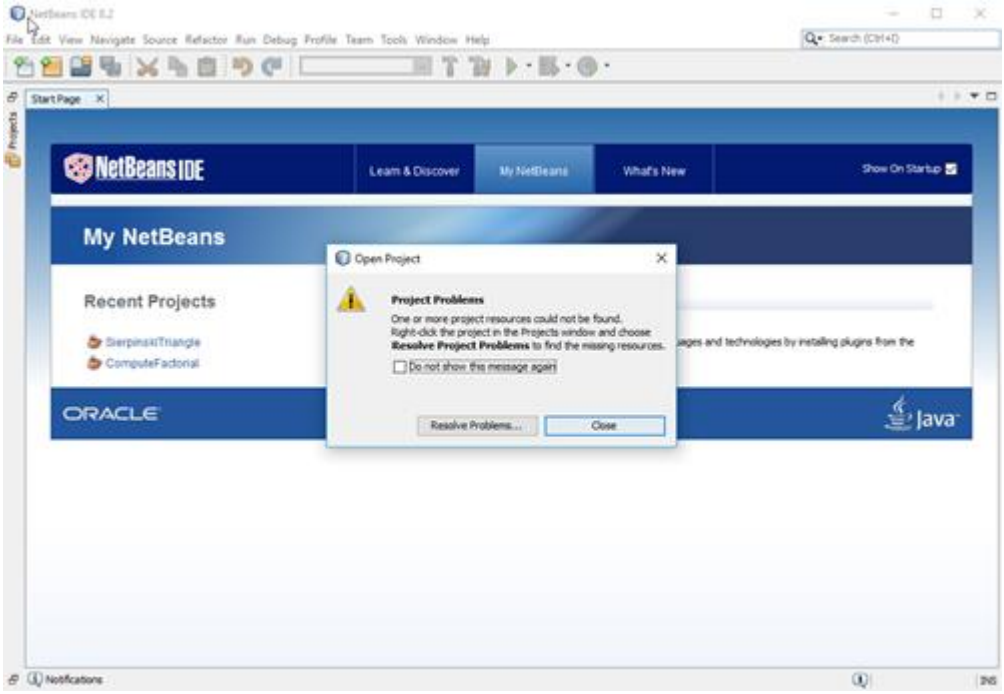
Select the Cyber Crypto-master.zip file click open.



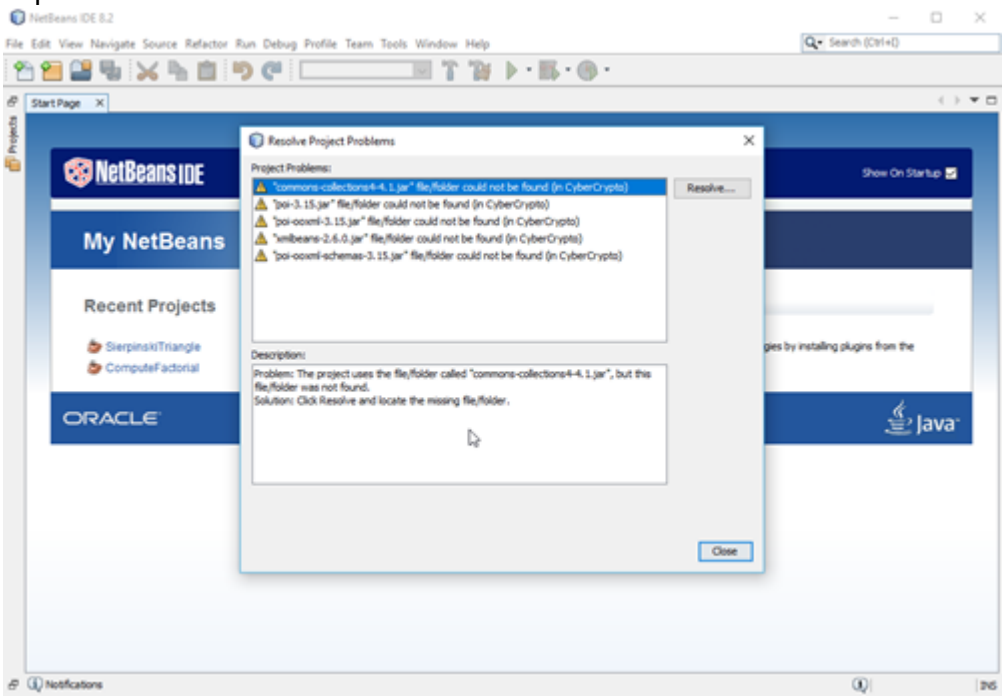
You should be now back to the Import Project from ZIP window click Import.



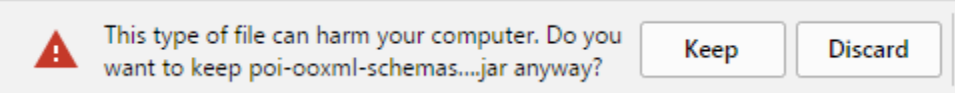
When NetBeans is opening the project, it will display a Project Problems window click resolve Problems.



The Resolve Project Problem window will open and you will have a list of five files the Cyber Crypto-master project is looking but can not find. For the project to run, you will have to resolve all five of the dependencies needed. Do not worry, this is normal and will be addressed step by step with links to the needed software.



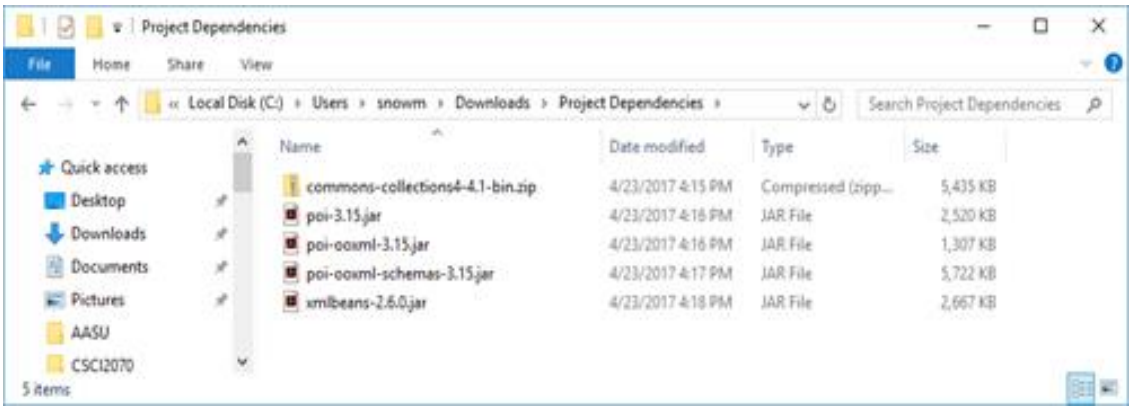
Leave the NetBeans IDE 8.2 alone for now and open a web browser. The files you need download to your desktop or download folder is the “commons-collections4-4.1.jar”, “poi-3.15.jar”, “poi-ooxml-3.15.jar”, “xmlbeans-2.6.0.jar”, and finally the “poi-ooxml-schemas-3.15.jar”.



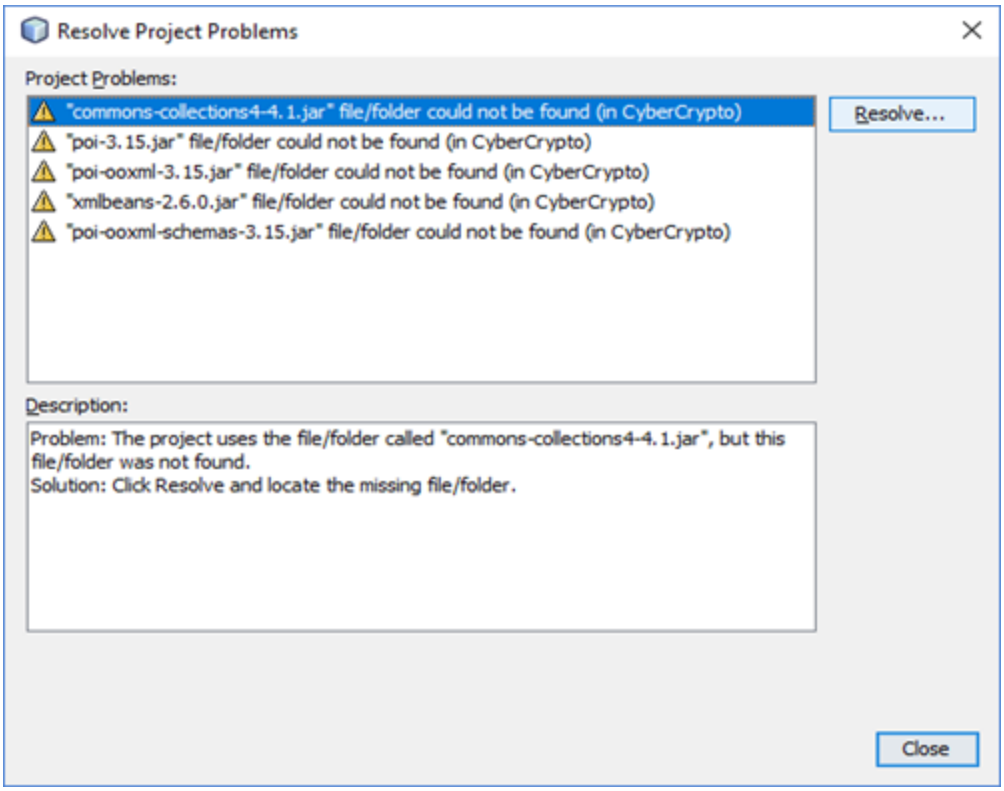
Your browser may show a warning like the one above, please click Keep when downloading.

1. commons-collections4-4.1.jar:
“ http://www-eu.apache.org/dist//commons/collections/binaries/commons-collections4-4.1-bin.zip ”
2. poi-3.15.jar : “ “ “ “ “
http://search.maven.org/remotecontent?filepath=org/apache/poi/poi/3.15/poi-3.15.jar ”
3. poi-ooxml-3.15.jar: “
http://search.maven.org/remotecontent?filepath=org/apache/poi/poi-ooxml/3.15/poi-ooxml-3.15.jar
4. xmlbeans-2.6.0.jar:
“http://central.maven.org/maven2/org/apache/xmlbeans/xmlbeans/2.6.0/xmlbeans-2.6.0.ja r”
5. poi-ooxml-schemas-3.15.jar:
“http://central.maven.org/maven2/org/apache/poi/poi-ooxml-schemas/3.15/poi-ooxml-schemas-3.15.jar ”

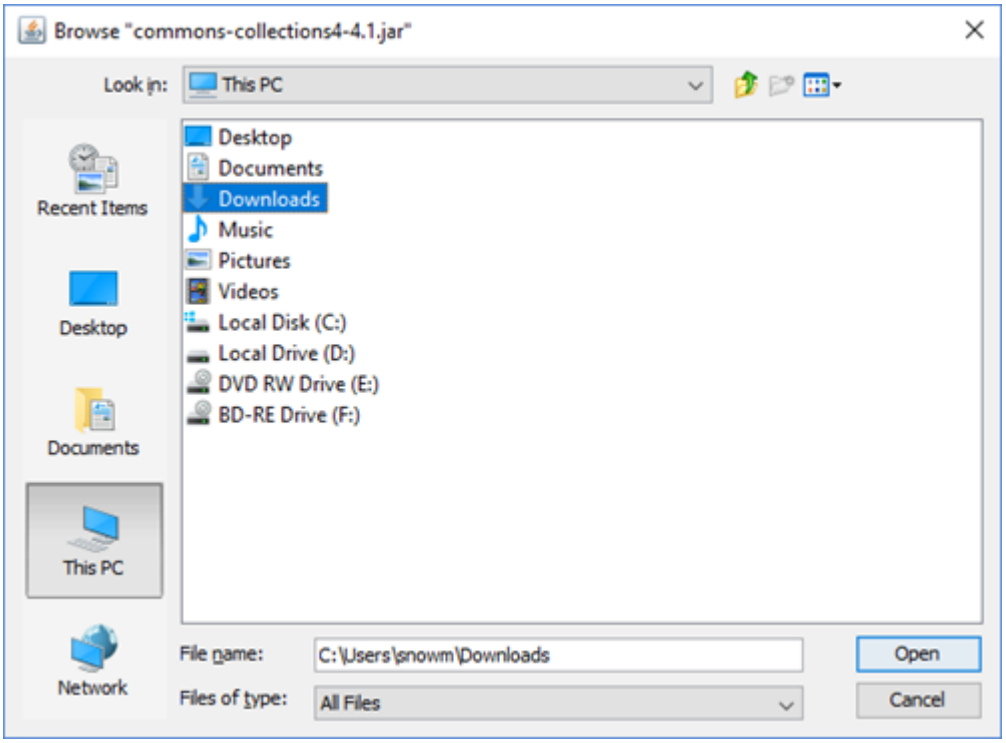
Now you should all five files in your downloads folder or desktop.



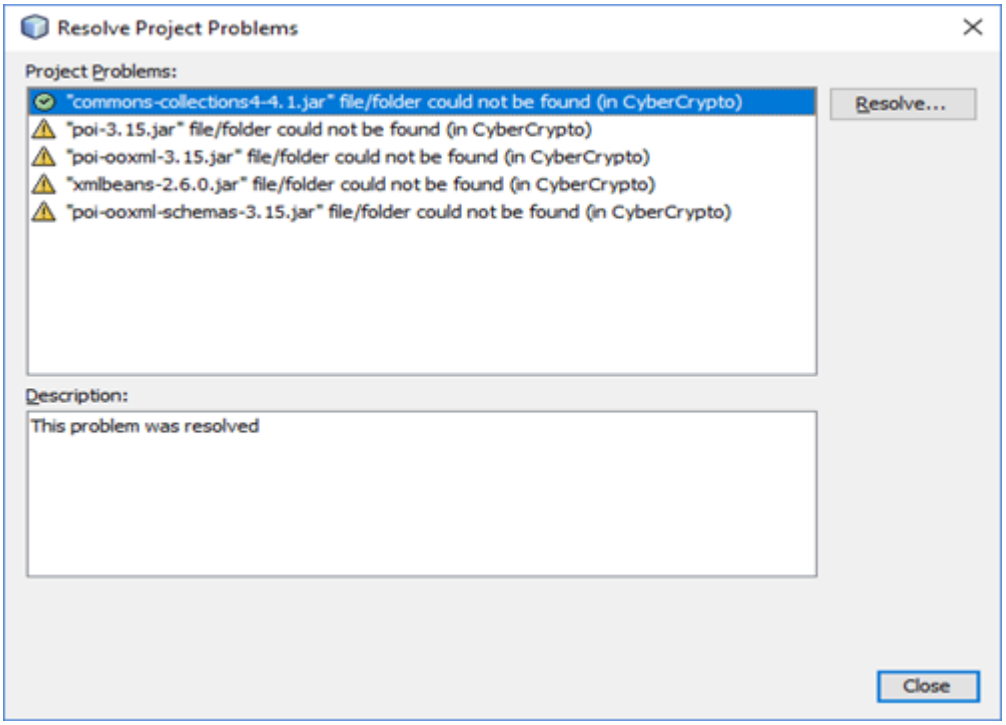
Go back to NetBeans IDE 8.2 and the Resolve Project Problems select the commons-collections4-4.1.jar click Resolve…



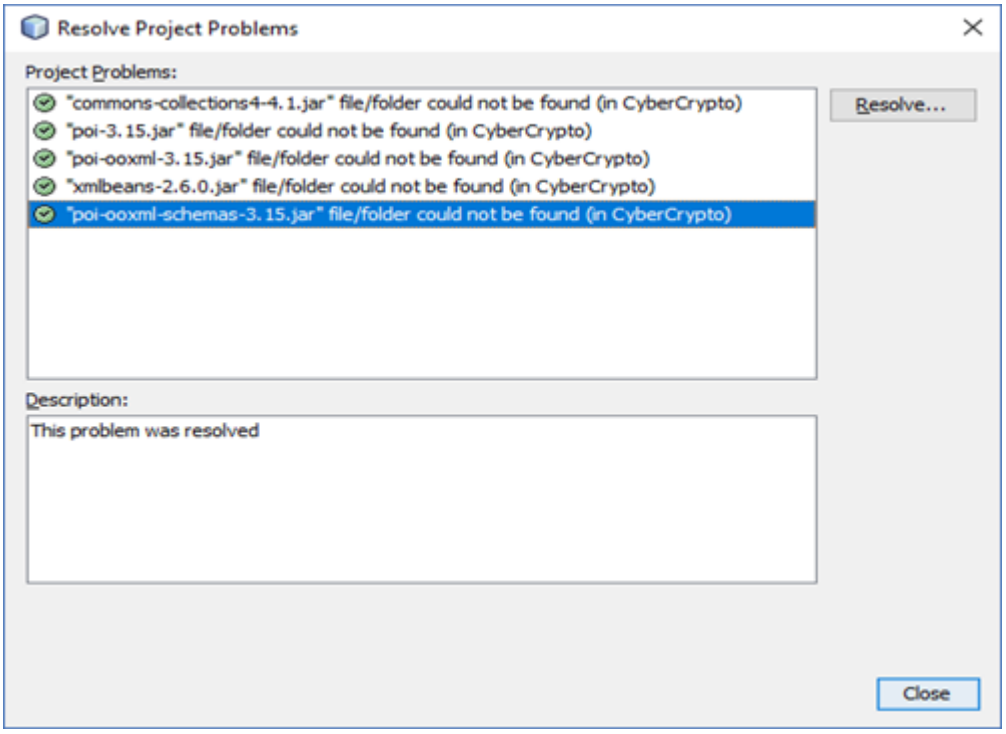
A Browse “commons-collections4-4.1.jar” window will appear Select This PC Select Downloads click Open or select Desktop click Open. You need to point the Browse window to the correct folder you downloaded the 5 files.



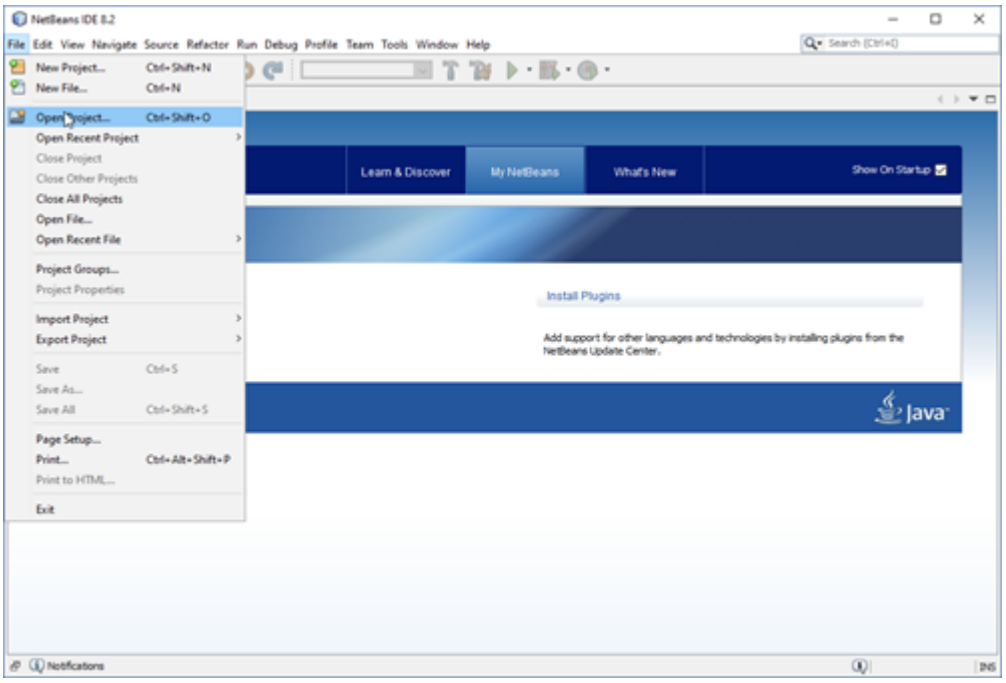
Your “commons-collections4-4.1.jar” problem should be resolved and have a green color circle with a check in the center.



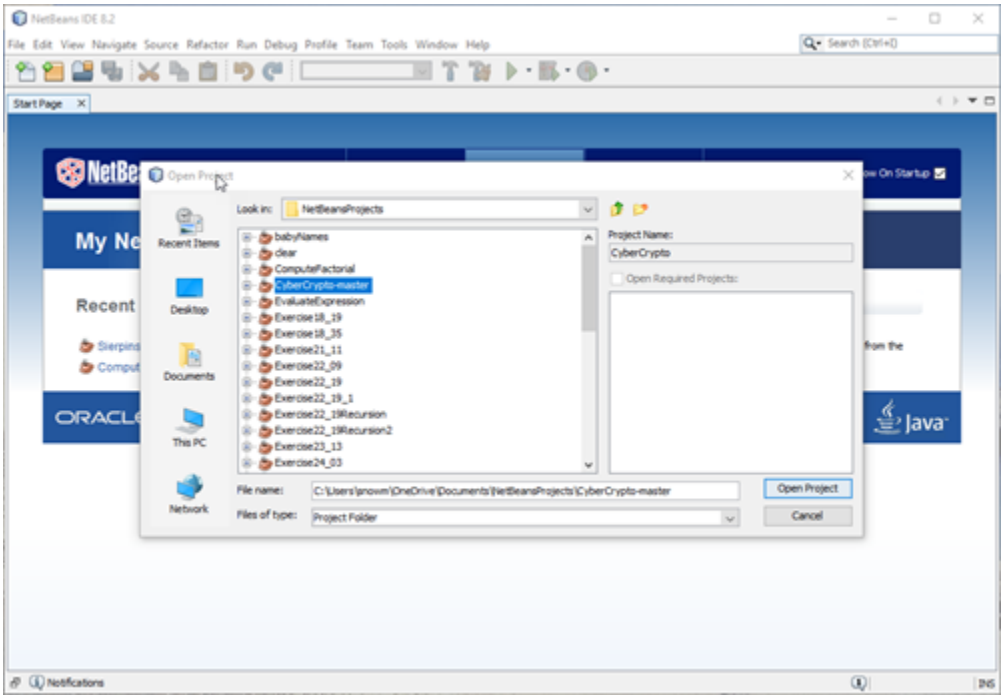
Work your way through the other four files following the exact same selections as above for “commons-collections4-4.1.jar” since all the files should have been download to the same folder.
Once you have resolved all the files your Resolve Project Problems will be fixed as below and select Close.



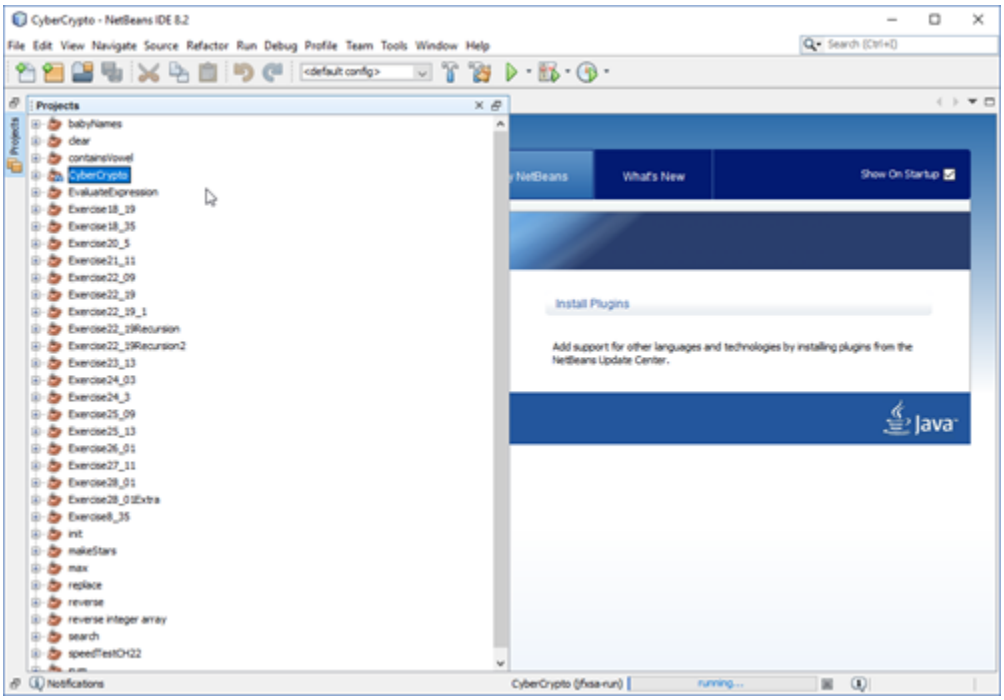
You will be back to the original NetBeans IDE Start Page. Select File > Open Project.



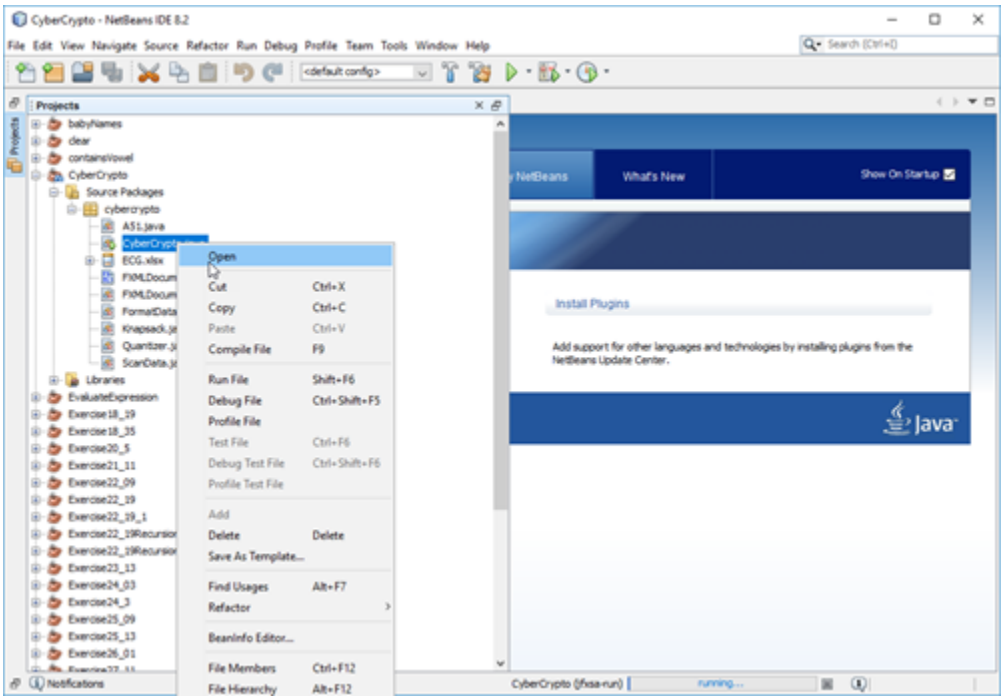
An Open Project window will appear Select CyberCrypto-master > Open Project.



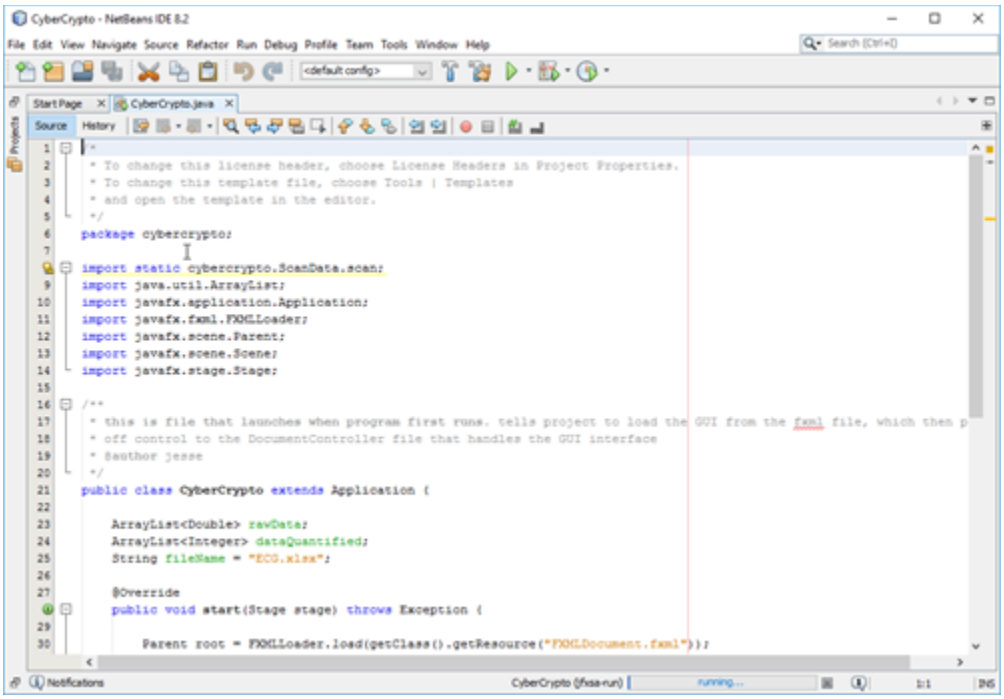
Select CyberCrypto.



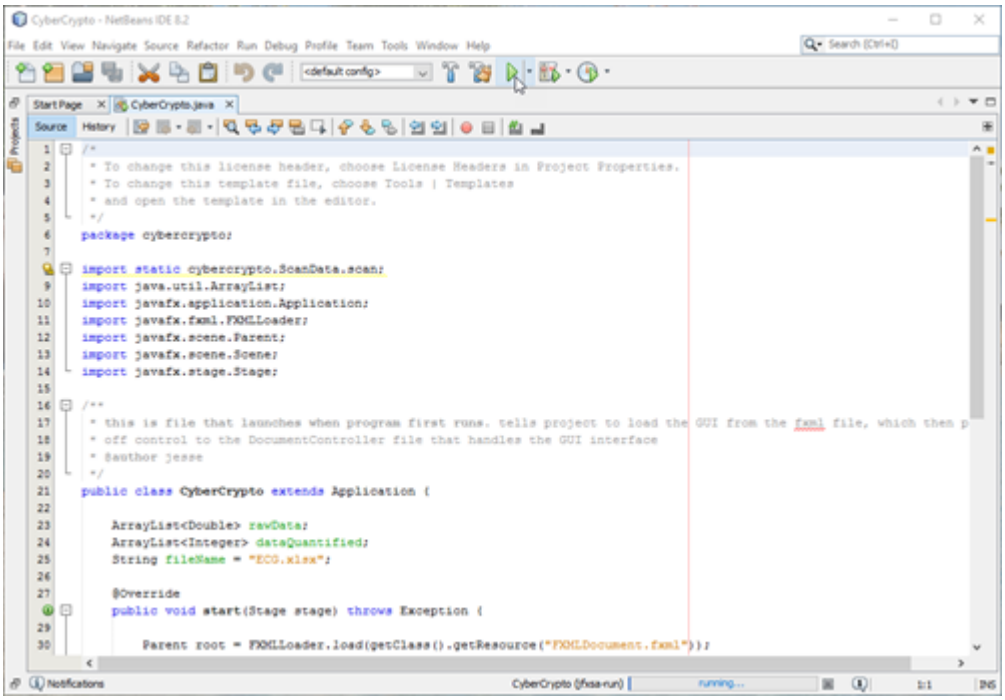
Select Source Packages cybercrypto Right click on CyberCrypto.java Select Open.



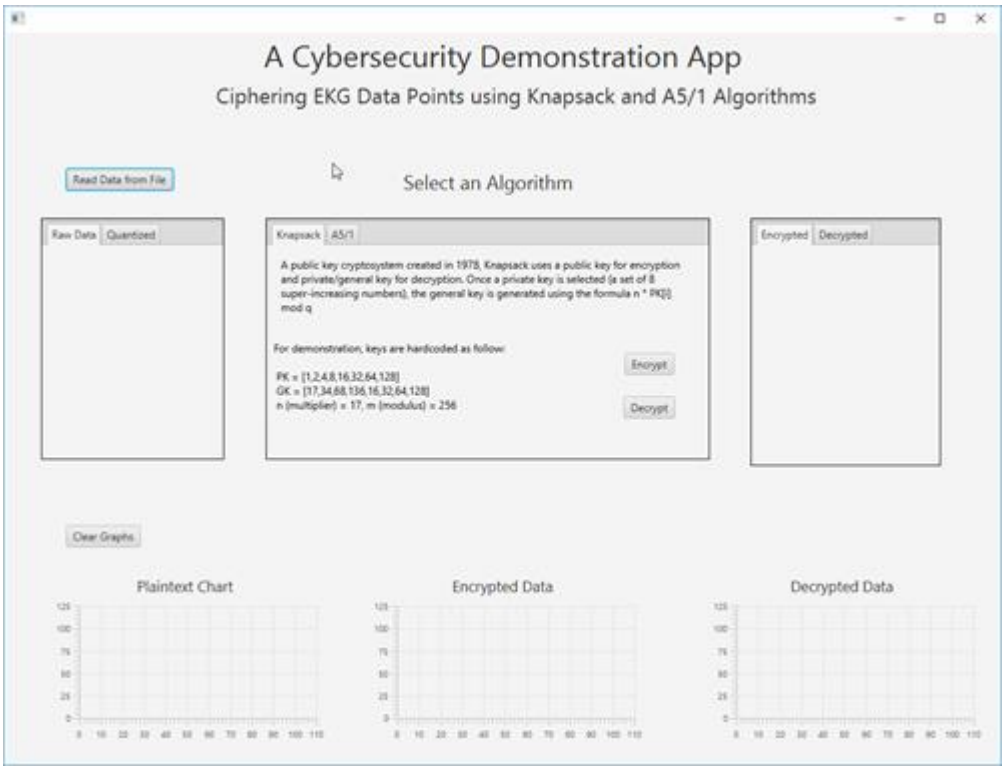
You are ready to run the Project CybeCrypto.java.



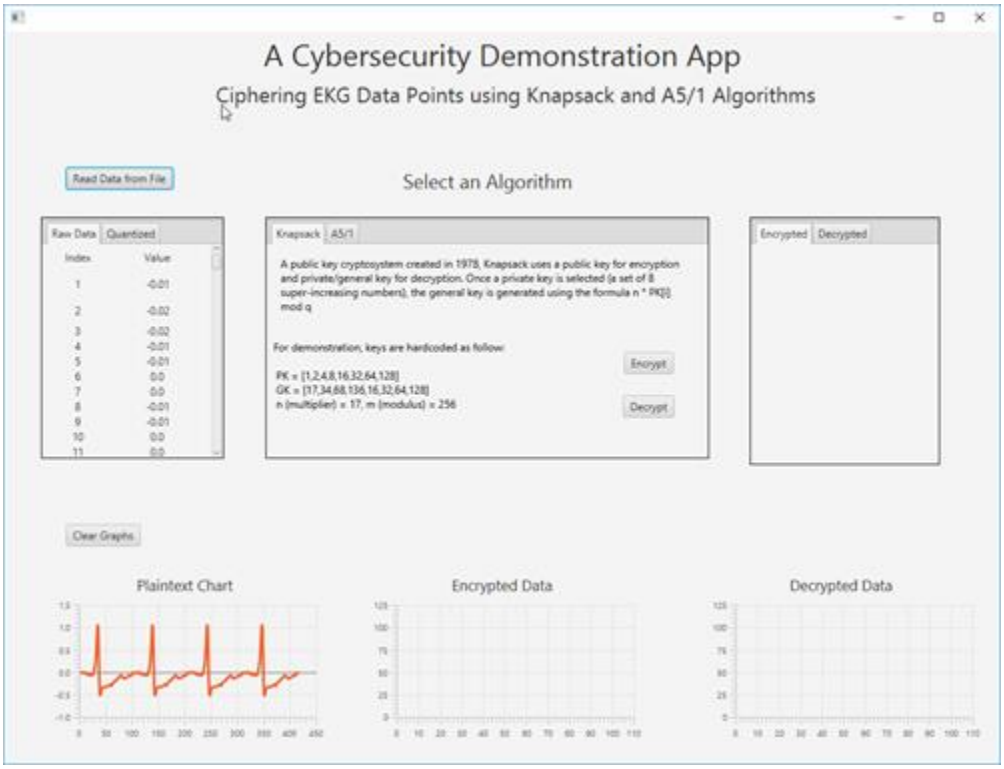
Select the Green Run button.



You are all set for the Operating instructions at this point.

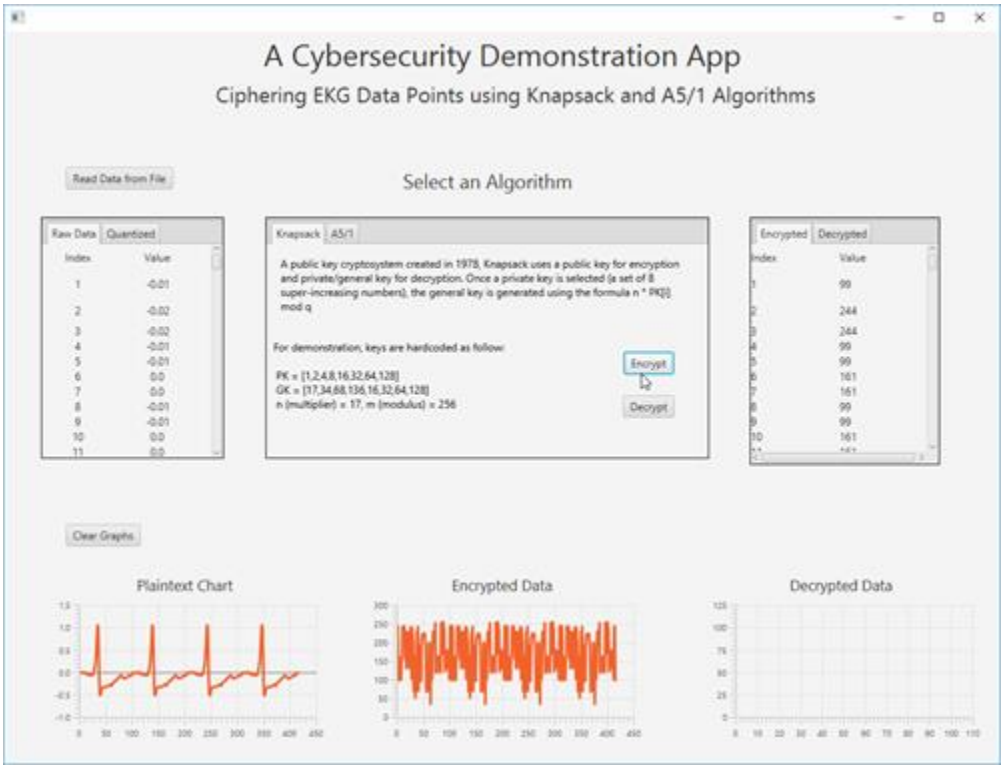


Operation Instructions:
Selecting “Read Data from File” – reads in the EKG Data Points from an excel spreadsheet that is part of the Cyber Cryto-master.ZIP that you imported into NetBeans. Data will be populated under Raw Data and Quantized tabs.
The “Raw Data” tab is the data from the Excel spreadsheet that you have read in and the “Quantized” tab shows the values the Quantizer.java class converted the raw data values to.
The “Plaintext Chart” shows the plotted graph for the Raw Data and can be cleared using the “Clear Graphs” button. After clearing the Plaintext Chart simply click “Read Data from File” to have the Plaintext Chart repopulate.

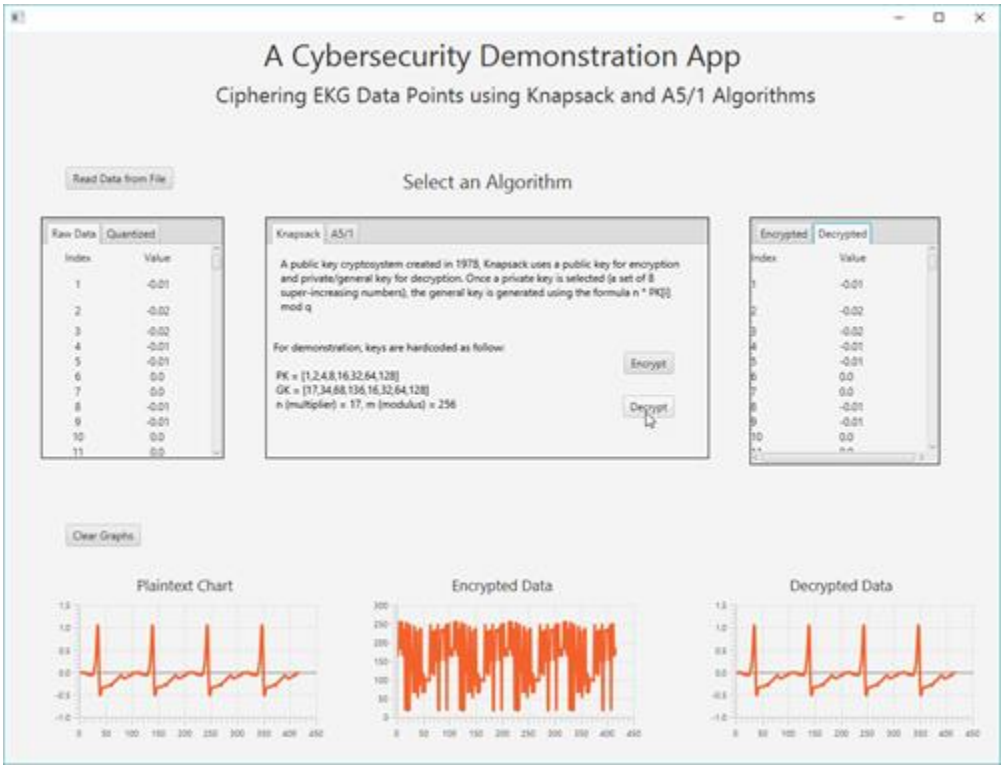


After reading in the file, there are two option for encryption: “Knapsack” or “A5/1”.

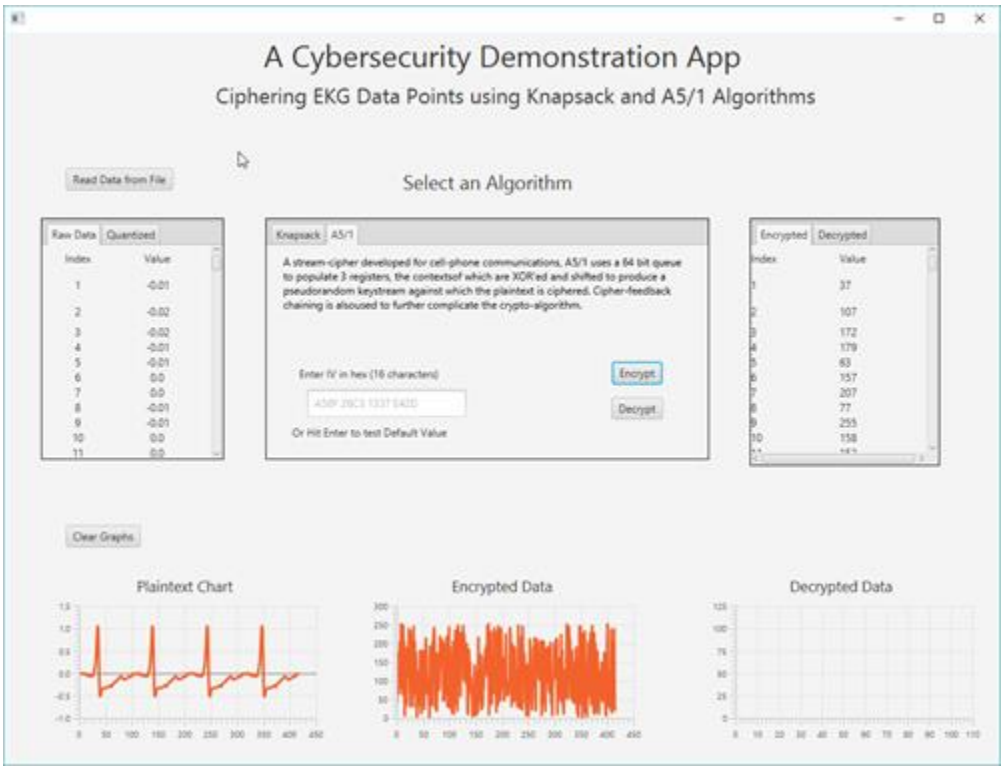
1. Knapsack has two selections: “Encrypt” and “Decrypt”.
Selecting “Encrypt” will encrypt the data using the Knapsack algorithm and show the encrypted data plotted in the “Encrypted Data” graph. The top right box’s “Encrypted” tab will show the encrypted values for each index.



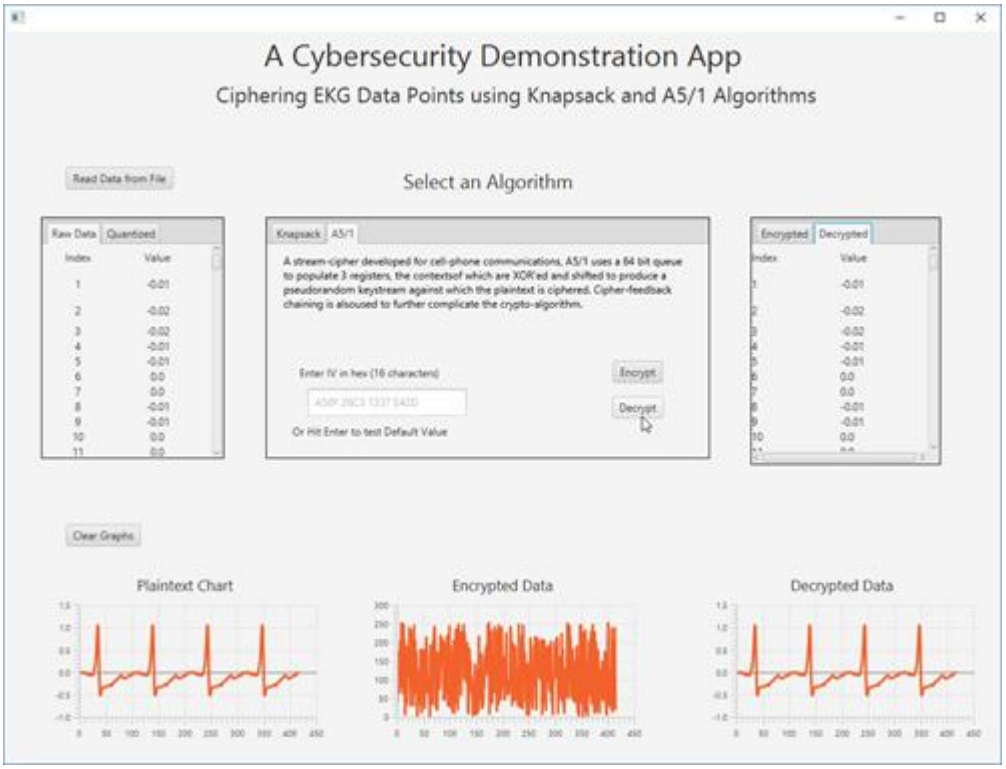
Selecting Decrypt will decrypt the Knapsack encrypted data and show the decrypted data plotted in the Decrypted Data graph. The top right box's "Decrypted" tab will show the decrypted values for each index and should match the original raw data indexes and values.



2. A5/1 has two selections “Encrypt” and “Decrypt” and a text box for the user to enter a IV in hexadecimal (16 characters).
The user may enter a custom IV for the A5/1 algorithm to encrypt the data. The IV must be a valid 16 characters hexadecimal entry.
Selecting “Encrypt” without entering a IV in the text box will use the Default Value IV encrypting the data using the A5/1algorithm and show the encrypted data plotted in the “Encrypted Data” graph. The top right box’s “Encrypted” tab will show the encrypted values for each index.



Selecting Decrypt will decrypt the A5/1 encrypted data and show the decrypted data plotted in the Decrypted Data graph. The top right box's "Decrypted" tab will show the decrypted values for each index and should match the original raw data indexes and values.



7 Appendix

All visuals and supplementary materials for this paper are included in their respective sections. There was no additional content to be supplied in an appendix.