Project 1 Report (CS6238)

Lei Shao, Ruiting Qu

1. Specification

1.1 Goal

The goal of this project is to implement the strengthened authentication scheme described in the password hardening paper. And it is some kind of simplified in implementation. What we mainly focused on is the part of using keystroke features to authenticate a given user.

1.2 Functional Requirement & Clarification

The “client” securely sends the password and keystroke feature values of a certain user to the “server” where the actual authentication is done. In our simplified program, “client” and “server” are all locally simulated. The communication between these two are done by read/write of file on disk, as well as the operation to variables/data structures which keeped by the program. Log-in request will be sent by loading a test file which contains a set of log-in attempts from different users. For a time period of log-in authentication, the “server” will only focus on one certain user’s attempts. And that user should be the one who registered for itself by inputing name and password in the procedure of initialization, which means the program will not keep personal informations for every user, especially for the data that stored in program’s variable/data structures without saving to disk(username, password, etc. History file will be saved on disk for any user).

2. Design

The general design of this program is strictly followed the instruction document for this project, and will be described in details with following paragraphs. The program is designed to be consist of several modules with different function. Main modules includes: *Initialization, Log-in Authentication, Update & Training.*

2.1 Initialization(Sign-up)

In this module, the key function is guiding a user register for itself to the authentication system and store the related authentication files that will be useful for future log-in authentication. The initialization behaves as following steps:

1) An initial configuration for all constants is done at the begining. Most of them are fixed due to the project requirement(also the paper), but they’re easy to re-configure in the part of Initialization() code.
    a) Number of feature fixed to 15;(say, m)
    b) Size of history file fixed to 8; (say, h)
2) Select a 160 bits prime value q.

3) Choose a random *Hpwd* with value less than q.
4) Produce a random polynomial.

$$f(x) = hpwd + a_1x + a_2x^2 \ldots. + a_{n-1}x^{n-1}$$

   a) Highest exponent equals to number of feature minus 1. (say, exp - 1)
   b) For every "x^exponent" in the polynomial, choose a random number belongs to Zq and make it to be the correspoding coefficients. In total exp-1 random 160 bits numbers(or less).
   c) Make the *Hpwd* the f(0) of this polynomial.
   d) Pick 2m points from the previously created polynomial.
      i) For x value, when 1<= i <= m, calculate every 2i and 2i + 1's permutation which refects to a number belongs to Zq. Make these 2m value the X values of 2m points.
      ii) Calculate 2m corresponding Y values for the 2m X values by putting X values into created polynomial and get the result, then we have complete 2m points, extracted from the random polynomial.
   e) Create an instruction table for the user.
      i) Configure parameters, make the instruction talbe size of num_feature * 2
      ii) User the formulation mentioned in the paper to calculate α and β

$$\alpha_{ai} = y^0_{ai} + G_{r,\text{pwd}_a}(2i) \bmod q$$
$$\beta_{ai} = y^1_{ai} + G_{r,\text{pwd}_a}(2i+1) \bmod q$$

      In detail, the G function is designed to be a keyed hash function, we chose HMAC with SHA1. And inputs are user password, and 2i/2i+1. All operations in the program need to be mod the big prime q.
   f) The final step for initialization is creating the history file for the user. The file size should be h * m (history file length * number of features). It's to store the successful log-in feature vectors.
      i) History file is padded at the begining to keep a constant size. The padding number is -1, distinguishing with possible feature values.
      ii) Finally, the system encrypts the history file with *Hpwd* and save it to disk ,the filename contains the name of user. And at this point, the system can save whatever user's history initial file.

2.2 Log-in Authentication

In this module, the program gets one log-in attempt(from testfile, that belongs to another module, will describe later), with username, password, keystroke feature vector(size of 15).

1) Configuration:
   a) Read values of parameterss from former modules of the program, such as number of feature.

    b) create variables for internal data, such as extracted α or β, recovered points, standard lagrange coefficients, etc.

2) Compare the every feature in the feature vector to the threshold, to extract related α or β from instruction table. Criteria should follows the paper:
    a) Select αi if feature(i) < threshold(i)
    b) Select βi if feature(i) >= threshold(i)
3) Compute and recover m points from the selected m , using formulation in paper:

$$(x_i, y_i) = \begin{cases} (P_r(2i),\ \alpha_{ai} - G_{r,\mathsf{pwd}'}(2i) \bmod q) \\ \qquad\qquad\qquad\qquad \text{if } \phi_i(a,l) < t_i \\ (P_r(2i+1),\ \beta_{ai} - G_{r,\mathsf{pwd}'}(2i+1) \bmod q) \\ \qquad\qquad\qquad\qquad \text{if } \phi_i(a,l) \geq t_i \end{cases}$$

   The details should be similar with the procedure we talked about in Initialization, it's a reverse computation. The location for mod q is one thing need to be concerned.
4) Compute the standard Lagrange coefficient for interporlation using the formulation in paper:

$$\lambda_i = \prod_{1 \leq j \leq m, j \neq i} \frac{x_j}{x_j - x_i}$$

5) Compute the recovered *Hpwd'*,

$$\mathsf{hpwd}' = \sum_{i=1}^{m} y_i \cdot \lambda_i \bmod q$$

6) Using the *Hpwd'* to decrypt the previously encrypted content of user's history file, if a properly formed table with size 8 * 15 is produced, log-in is successful. Otherwise, (maybe a wrong content, or wrong format), log-in is failed.

   Notes: In this step, we designed to use hashcode of content to examine the integrity of decrypted content, comparing with the previous correct one. However, if the *Hpwd'* is extremely incorrect, the decryption algorithm will not allow it to produce a well-formed table, since the crypto schemes always have strict requirement for its inputs. Then the crypt function will throw out a exception. Our solution is to catch such exception and tell parent functions that this is a wrong *Hpwd* and demonstrate that this log-in attempt is failed.

2.3 Update & Training

The update & training module mainly concentrate on the behavior after a successful log-in attempt. The log-in authentication module will return a log-in result, and determine whether the update&training module should be triggered.

1) When a log-in attempt succeeds, the system firsly insert the current log-in feature vector into the user's history file. If the history table is full, delete the oldest one to keep this file's length constantly equals to 8.(8 latest records). If there are still padding lines, just replace one padded line with the current feature vector.
2) If history file is not full. Every time after a successful log-in, we do not make so much extra works, but just generate a new *Hpwd,* use it to encrypt the updated history file, and create new random polynomial, new instruction tables. All of this is done the same way as initialization.
3) If history file is full, which means no padded lines are found in the file. We need to make some more complex trick to train the data. To provide a more accurate authentication based on keystroke features in the future.
    a) Firstly, we computes the mean and standard deviation for the 15 features in current 8 feature vectors.
    b) We user the same threshold described in log-in authentication part. These thresholds are selected depending on experiments references and also a random chozen vector when do the testing.
    c) Create a new *Hpwd*
    d) Create a new polynomial, random one, but produce 2m points followed the method mentioned in paper,

       When , $|\mu_{ai} - t_i| > k\sigma_{ai}$

       $$\mu_{ai} < t_i \Rightarrow f_a(P_{r'}(2i)) = y^0_{ai} \wedge f_a(P_{r'}(2i+1)) \neq y^1_{ai}$$
       $$\mu_{ai} \geq t_i \Rightarrow f_a(P_{r'}(2i)) \neq y^0_{ai} \wedge f_a(P_{r'}(2i+1)) = y^1_{ai}$$

       When, $|\mu_{ai} - t_i| \leq k\sigma_{ai}$

       The features in history file seems not trained well to be distinguishing, so the new 2m points should just be picked from the new random polynomial.
    e) Create a new instruction table, with same methods as before, based on the new created polynomial.

2.4 Testing

This module is much simpler, and is mainly focus on reading the test file into the program.

1) The test file is formatted as:
   <seq number for user> <username> <f_i's (hopefully 15 of them with space between each)>
   And it contains several log-in attempts from different users.
2) After initialization, the program ask the one who want to log in for username and password. For speeding up, the username and password will be asked for only once for one set of test query.
3) Test file will be inputed and ran automatically by the program. Results will shown after a few seconds or less.

4) *Notes:* In this program, after the system asks for your username&password, and you input it. The later result for all log-in attempts will based on this certain user and the inputed password.
   If the username is not the one registered in initialization module, all attempts will fail. If the test file contains other usernames, they will be ignored and shown as “Wrong username” when testing. If the password input is not the one input in initialization module, all attempts even from the right person will not pass.

3.Implementation

Detailed design for our program has already been described in the former section. Now we are going to introduce some specific implementation for our program. We will not go through everything, since the source code is also one deliverable. We will just pick some key parts to show how we implement our design. This part will be organized by objects, not the real order during program running.

3.1 Dependencies

The dependencies for this program is important to state since it will influent whether you can successfully run it on your own machine.

1) **System Environment**
   The final version of program works fine on OS X 10.9.2
2) **Programming Language**
   Java version "1.6.0_65"
   Java(TM) SE Runtime Environment (build 1.6.0_65-b14-462-11M4609)
   Java HotSpot(TM) 64-Bit Server VM (build 20.65-b04-462, mixed mode)
3) **Programming Platform/Tools**
   Eclipse Standard/SDK
   Version: Kepler Service Release 2
4) **Imported Libraries**
   No external libraries imported, just JRE System Library[JavaSE - 1.6]

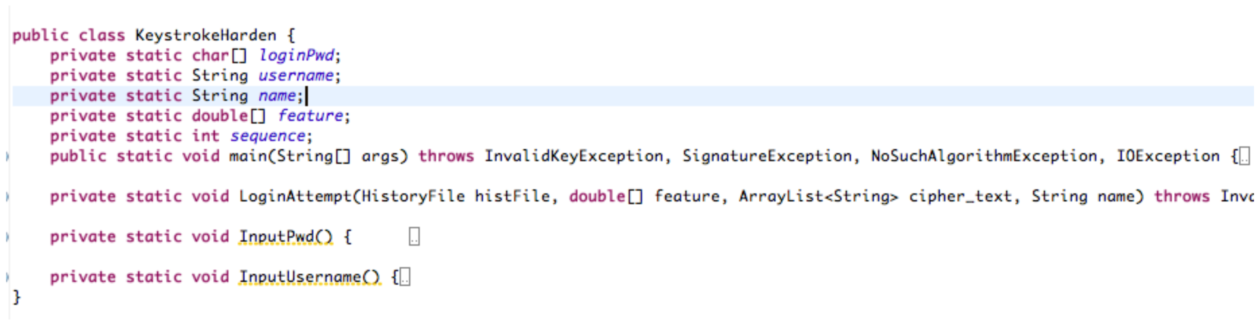
3.2 Class & Data Structure

Classes and some important inner data structures will be introduced in this section.

1) **KeystrokeHarden.class**

This class holds the main entry point of the program. It triggers different modules to work orderly and transfer parameters between them. The whole testing module is adhered with this class. In the main function. The class provide two functions “InputPwd()” and “InputUsername()” for getting user authentication inforamtion when making log-in attempts, before running the test file.

The general view of class is shown as below:



```
public class KeystrokeHarden {
    private static char[] loginPwd;
    private static String username;
    private static String name;
    private static double[] feature;
    private static int sequence;
    public static void main(String[] args) throws InvalidKeyException, SignatureException, NoSuchAlgorithmException, IOException {

    private static void LoginAttempt(HistoryFile histFile, double[] feature, ArrayList<String> cipher_text, String name) throws Invo

    private static void InputPwd() {

    private static void InputUsername() {
}
```

The password and username are got from console typing. And in this way, the input information will not be shown as plaintext in console when typing. Digit check is added in it, so the input length of password can be controled to fix at 8. Partial code shown below:

```
...
    Console console = System.console();
    if (console == null) {
        System.out.println("Couldn't get Console instance");
        System.exit(0);
    }
    loginPwd = console.readPassword("Please enter your secret password(8 digits): ");
    while(loginPwd.length != 8) {
            loginPwd = console.readPassword("Password should be 8-digits: ");
    }
}
```

When loading the test file, the program read the text file by line, saves lines into String, and split each string by spaces, then extracts certain elements following the pattern of test queries. Basically,

```
sequence = Double.valueOf(string[0]).intValue();
name = string[1];
for(int i = 0; i < 15; i++) {
        feature[i] = Double.valueOf(string[i+2]);
}
```

Another important implementation related to this class is that, it connects and organizes the Log-in authentication and update&training in a "LoginAttempt" function.
This function makes a previous created historyfile(histFile), all informations contained in one login attempt query(except for the sequence number) as input. And make following steps:

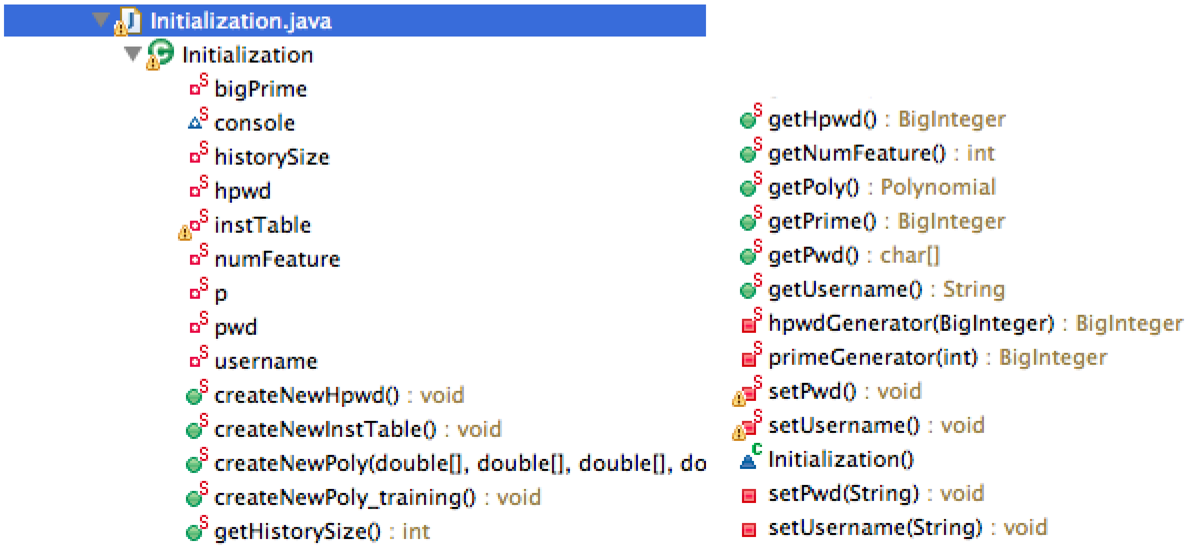
1. Call LoginAuth() to create a login instance, mainly do the Log-in Authentication function we talked in design section. It's to recover a *Hpwd'* given related elements.
2. A decryption to previous created history file, using the newly recovered *Hpwd'* is excuted. Return a boolean value(boolean success) showing whether the log-in is success.
3. Update history table and compute the means/standard deviations, depending on whether the history file is full.

```
//**********Update & Training***************
if(success) {
    //Update if success
    histFile.updateHFile(feature);
    System.out.println("Updated history file.");
    //histFile.printHistory();
    if(HistoryFile.isFull()) {
        //System.out.println("Now Full, Start calculation!");
        histFile.Update();
        histFile.encryptTable(Initialization.getHpwd());
    }
    else {
        System.out.println("History file is not full, system still need training.");
        histFile.Update_training();
        histFile.encryptTable(Initialization.getHpwd());
    }
}
else {
    System.out.println("Loggin failed");
}
```

Notes: Followed by the project instruction document, no matter the history file is full. A successful log-in makes it insert the current feature vector, (maybe then training), and encrypt the history file with a new *Hpwd*. Here, we make this specific function in histFile.Updata() function, also the histFile.Update_training() function. It generate a new *Hpwd* then give it to the inner parameter keeped by Initialization(). So here, when we call the histFile.encryptTable(..), we get the current stored *Hpwd* in Initialization() to acheive our goal.

2) Initialization.class

The Initialization.class is the most important class in this program. Though its main function is just the initialization(sign up for new user), it keeps many parameters and data structures that are very useful for any other classes. You can see many interfaces for different parameters are provided here for other classes or functions. The components of this class is shown below:



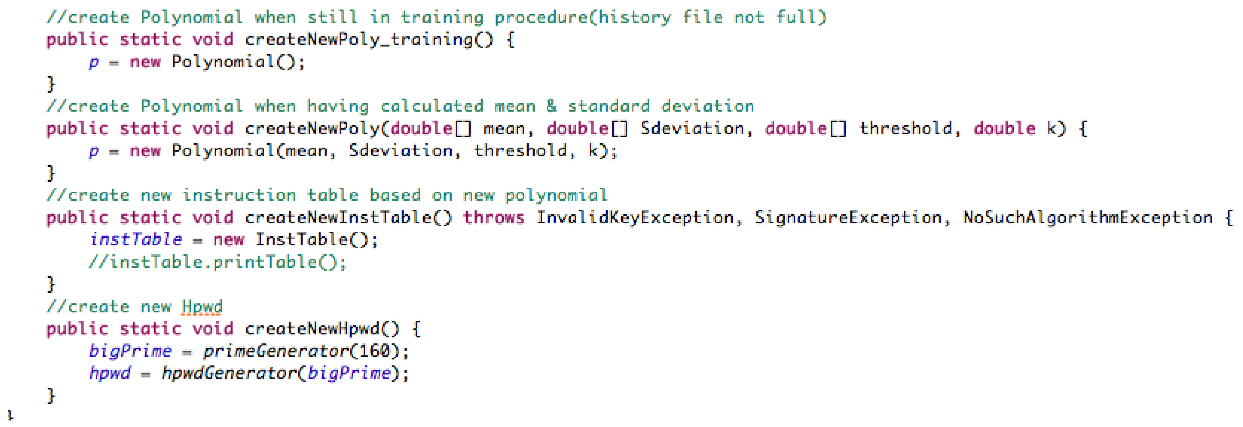
```
Initialization.java
  Initialization
    bigPrime
    console
    historySize
    hpwd
    instTable
    numFeature
    p
    pwd
    username
    createNewHpwd() : void
    createNewInstTable() : void
    createNewPoly(double[], double[], double[], do
    createNewPoly_training() : void
    getHistorySize() : int
    getHpwd() : BigInteger
    getNumFeature() : int
    getPoly() : Polynomial
    getPrime() : BigInteger
    getPwd() : char[]
    getUsername() : String
    hpwdGenerator(BigInteger) : BigInteger
    primeGenerator(int) : BigInteger
    setPwd() : void
    setUsername() : void
    Initialization()
    setPwd(String) : void
    setUsername(String) : void
```

One important data type that need to be emphasized here is the BigInteger. Since the requirement for this strengthen authentication scheme is that generating a 160 bits prime big number q then take a random number as *Hpwd* that belongs to Zq. Also many other later computations are involved in this long big integer. The common Int, long cannnot store that long. They will lose digits/informations of the number if you forcely use them. Also an array will be inconvinient during computation. Java provides BigInteger as a very good solution to this problem. And it also contains some built-in methods, supporting basic mathmatical operations as well as some mod operations. It's really helpful for our whole program.

For generating a big prime, we use:
primeNumber = BigInteger.probablePrime(length, new Random());
which has very high probabiliy to come up with such a number.

The same methods for getting user input from console occurs here for user sign-up.

These functions is very important for the updating section, they distinguish training or not training situation, and helps to call functions from instTable.class and Polynomial.class as a behavior of history file, then make updates. *Notes:* so we can see that, actually the whole process of our program is followed with the behavior of a certain user's history file.



```
    //create Polynomial when still in training procedure(history file not full)
    public static void createNewPoly_training() {
        p = new Polynomial();
    }
    //create Polynomial when having calculated mean & standard deviation
    public static void createNewPoly(double[] mean, double[] Sdeviation, double[] threshold, double k) {
        p = new Polynomial(mean, Sdeviation, threshold, k);
    }
    //create new instruction table based on new polynomial
    public static void createNewInstTable() throws InvalidKeyException, SignatureException, NoSuchAlgorithmException {
        instTable = new InstTable();
        //instTable.printTable();
    }
    //create new Hpwd
    public static void createNewHpwd() {
        bigPrime = primeGenerator(160);
        hpwd = hpwdGenerator(bigPrime);
    }
}
```

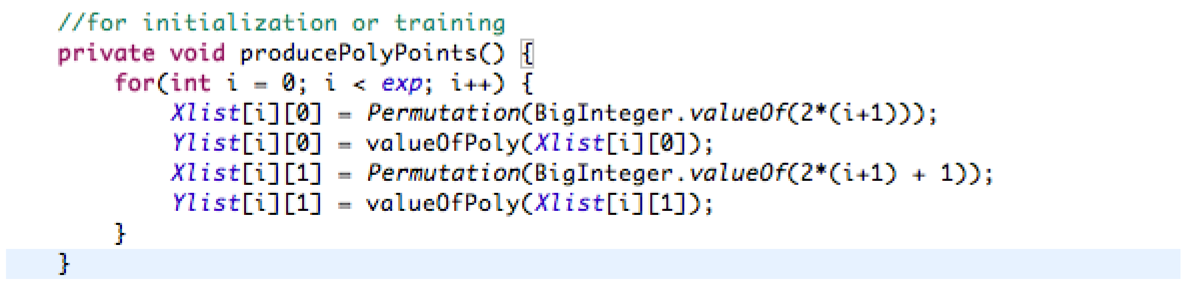
3) Polynomial.class

In this class, the main function is to generate a random polynomial and produce 2m points for later use(creating the instruction table). The 2m points' X, Y values are stored in two 2-dimension arrays. That is for easy getting the value and corresponds to the same position in instruction table. In sum, that's good for most of computations.

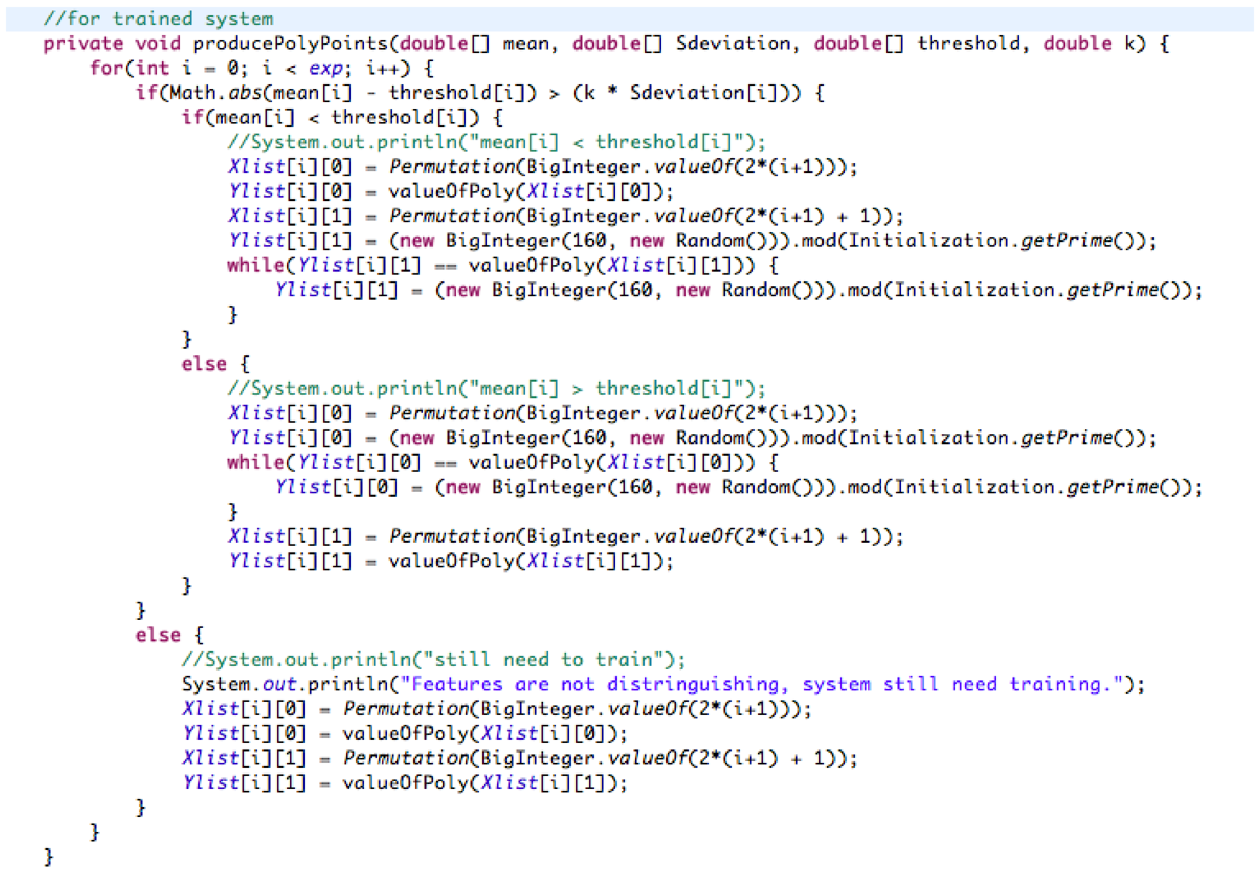
The function for creating random constants for the polynomial and computing the permutation for 2i/2i+1 to produce the X value of points are also contained in this class.

While a special situation is that, if the table is full, and need to be further trained, the Y values of 2m points will not all produced from the random polynomial. A preference of whether use

random number or ones taking security information of *Hpwd* is determinded by the comparision between threshold, current feature's mean, standard deviation, and given constant k. As we talked about in the design section.
Then we came up with different variant function to produce the points:



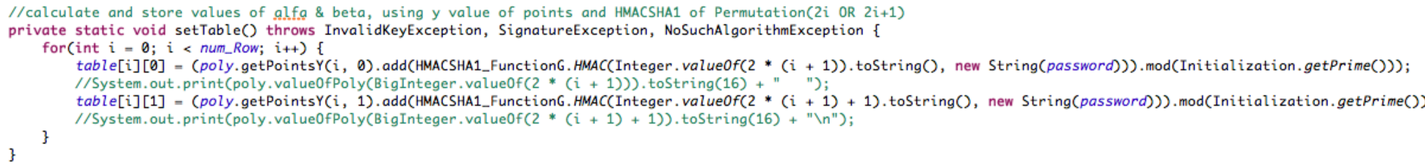
```
//for initialization or training
private void producePolyPoints() {
    for(int i = 0; i < exp; i++) {
        Xlist[i][0] = Permutation(BigInteger.valueOf(2*(i+1)));
        Ylist[i][0] = valueOfPoly(Xlist[i][0]);
        Xlist[i][1] = Permutation(BigInteger.valueOf(2*(i+1) + 1));
        Ylist[i][1] = valueOfPoly(Xlist[i][1]);
    }
}
```



```
//for trained system
private void producePolyPoints(double[] mean, double[] Sdeviation, double[] threshold, double k) {
    for(int i = 0; i < exp; i++) {
        if(Math.abs(mean[i] - threshold[i]) > (k * Sdeviation[i])) {
            if(mean[i] < threshold[i]) {
                //System.out.println("mean[i] < threshold[i]");
                Xlist[i][0] = Permutation(BigInteger.valueOf(2*(i+1)));
                Ylist[i][0] = valueOfPoly(Xlist[i][0]);
                Xlist[i][1] = Permutation(BigInteger.valueOf(2*(i+1) + 1));
                Ylist[i][1] = (new BigInteger(160, new Random())).mod(Initialization.getPrime());
                while(Ylist[i][1] == valueOfPoly(Xlist[i][1])) {
                    Ylist[i][1] = (new BigInteger(160, new Random())).mod(Initialization.getPrime());
                }
            }
            else {
                //System.out.println("mean[i] > threshold[i]");
                Xlist[i][0] = Permutation(BigInteger.valueOf(2*(i+1)));
                Ylist[i][0] = (new BigInteger(160, new Random())).mod(Initialization.getPrime());
                while(Ylist[i][0] == valueOfPoly(Xlist[i][0])) {
                    Ylist[i][0] = (new BigInteger(160, new Random())).mod(Initialization.getPrime());
                }
                Xlist[i][1] = Permutation(BigInteger.valueOf(2*(i+1) + 1));
                Ylist[i][1] = valueOfPoly(Xlist[i][1]);
            }
        }
        else {
            //System.out.println("still need to train");
            System.out.println("Features are not distringuishing, system still need training.");
            Xlist[i][0] = Permutation(BigInteger.valueOf(2*(i+1)));
            Ylist[i][0] = valueOfPoly(Xlist[i][0]);
            Xlist[i][1] = Permutation(BigInteger.valueOf(2*(i+1) + 1));
            Ylist[i][1] = valueOfPoly(Xlist[i][1]);
        }
    }
}
```

4) InstTable.class

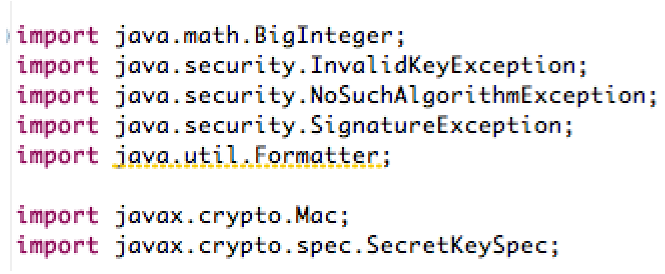
This class is pretty simple, its main function is to create a m * 2 instruction table by computing corresponding values followed with the formulation in paper:



```
//calculate and store values of alfa & beta, using y value of points and HMACSHA1 of Permutation(2i OR 2i+1)
private static void setTable() throws InvalidKeyException, SignatureException, NoSuchAlgorithmException {
    for(int i = 0; i < num_Row; i++) {
        table[i][0] = (poly.getPointsY(i, 0).add(HMACSHA1_FunctionG.HMAC(Integer.valueOf(2 * (i + 1)).toString(), new String(password))).mod(Initialization.getPrime()));
        //System.out.print(poly.valueOfPoly(BigInteger.valueOf(2 * (i + 1))).toString(16) + "    ");
        table[i][1] = (poly.getPointsY(i, 1).add(HMACSHA1_FunctionG.HMAC(Integer.valueOf(2 * (i + 1) + 1).toString(), new String(password))).mod(Initialization.getPrime()
        //System.out.print(poly.valueOfPoly(BigInteger.valueOf(2 * (i + 1) + 1)).toString(16) + "\n");
    }
}
```

5) HMACSHA1_FunctionG.class

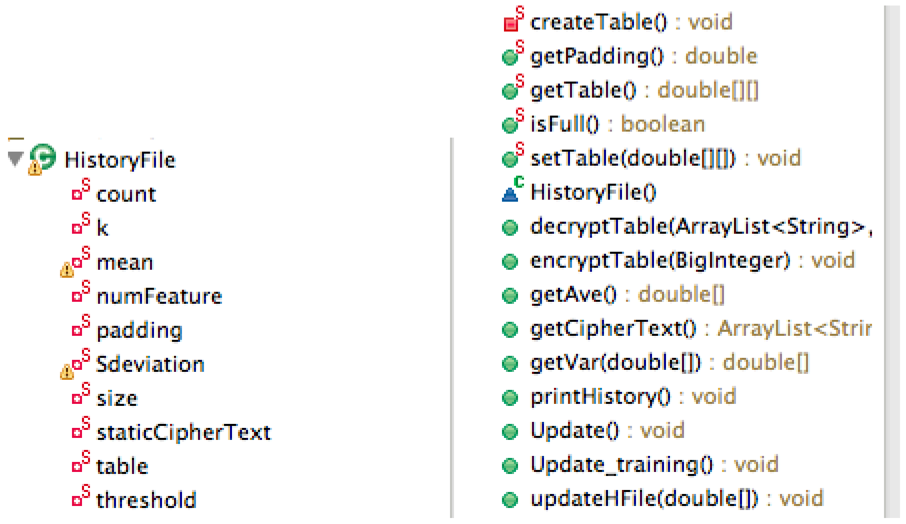
This class is basically used for providing a keyed hash function. It imports some security related libraries provided by JRE System Library,



```
import java.math.BigInteger;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.SignatureException;
import java.util.Formatter;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
```

6) HistoryFile.class

This class is also very important. As we talked about earlier, the whole process for this system is actually mainly based on the instance of HistoryFile.class. Here's the components:



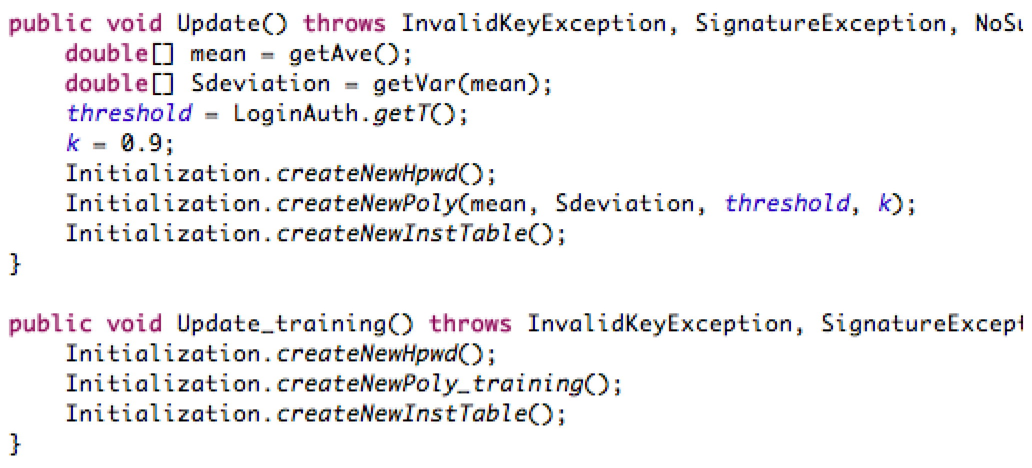
```
HistoryFile
    count
    k
    mean
    numFeature
    padding
    Sdeviation
    size
    staticCipherText
    table
    threshold
```

```
createTable() : void
getPadding() : double
getTable() : double[][]
isFull() : boolean
setTable(double[][]) : void
HistoryFile()
decryptTable(ArrayList<String>,
encryptTable(BigInteger) : void
getAve() : double[]
getCipherText() : ArrayList<Strir
getVar(double[]) : double[]
printHistory() : void
Update() : void
Update_training() : void
updateHFile(double[]) : void
```

The history file is a two dimension array 'double[ ][ ] table' and the measurements for all features over 8 successful logins are stored inside. Since the history file need to keep its constant size, padding is needed before 8 successful logins. In order to make it easy to tell from 'true' feature measurements, we choose a padding value '(double) -1.0' for all paddings.

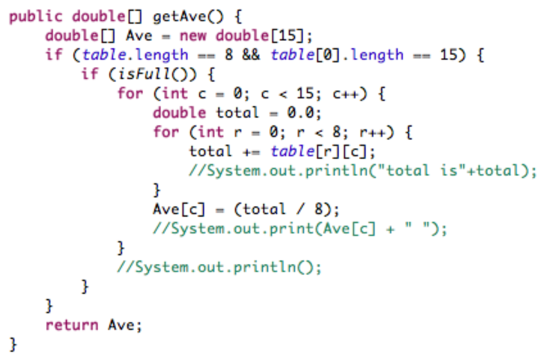
For other parameters, it holds ones related to updating and crypto.Besides basic behavior to the history table, such as create, set, get, print(for test), it also includes decryptTable(), encryptTable() that connected to Crypt.class. Which can be used when do the log-in authentication(extracting m points, then recover the *Hpwd'*).

Also it includes Update()/Update_training()/UpdateHFile() for editions to history file after one successful log-in attempt. The first two differentiate with each other by the use in different situations, non-full talbe & non-trained table, or full & trained table.

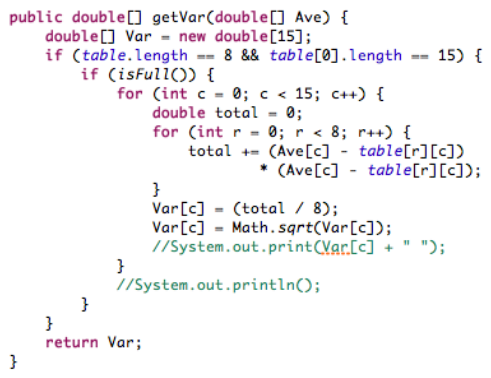


```
public void Update() throws InvalidKeyException, SignatureException, NoS
    double[] mean = getAve();
    double[] Sdeviation = getVar(mean);
    threshold = LoginAuth.getT();
    k = 0.9;
    Initialization.createNewHpwd();
    Initialization.createNewPoly(mean, Sdeviation, threshold, k);
    Initialization.createNewInstTable();
}

public void Update_training() throws InvalidKeyException, SignatureExcep
    Initialization.createNewHpwd();
    Initialization.createNewPoly_training();
    Initialization.createNewInstTable();
}
```

getAve() and getVar() is to compute the mean and standard deviation for the feature vectors stored in a full history file.

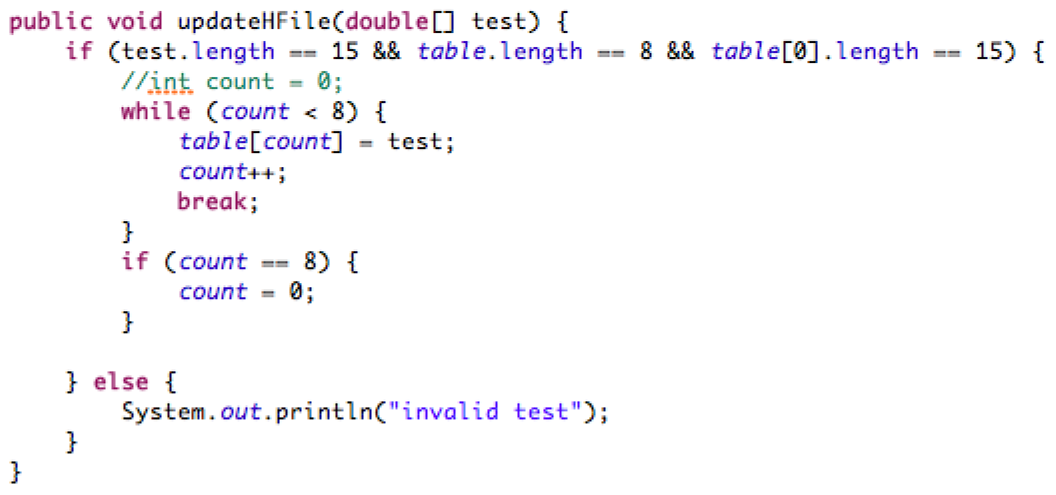


```
public double[] getAve() {
    double[] Ave = new double[15];
    if (table.length == 8 && table[0].length == 15) {
        if (isFull()) {
            for (int c = 0; c < 15; c++) {
                double total = 0.0;
                for (int r = 0; r < 8; r++) {
                    total += table[r][c];
                    //System.out.println("total is"+total);
                }
                Ave[c] = (total / 8);
                //System.out.print(Ave[c] + " ");
            }
            //System.out.println();
        }
    }
    return Ave;
}
```



```
public double[] getVar(double[] Ave) {
    double[] Var = new double[15];
    if (table.length == 8 && table[0].length == 15) {
        if (isFull()) {
            for (int c = 0; c < 15; c++) {
                double total = 0;
                for (int r = 0; r < 8; r++) {
                    total += (Ave[c] - table[r][c])
                            * (Ave[c] - table[r][c]);
                }
                Var[c] = (total / 8);
                Var[c] = Math.sqrt(Var[c]);
                //System.out.print(Var[c] + " ");
            }
            //System.out.println();
        }
    }
    return Var;
}
```

After initialization, during the login process using *hpwd*, the pointer count is used to indicate the right place to replace every time. We do the replacement by circle, so this history file will always contains the measurements of latest 8 logins.

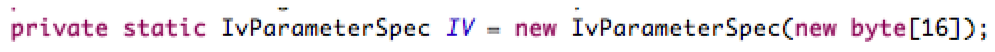


```
public void updateHFile(double[] test) {
    if (test.length == 15 && table.length == 8 && table[0].length == 15) {
        //int count = 0;
        while (count < 8) {
            table[count] = test;
            count++;
            break;
        }
        if (count == 8) {
            count = 0;
        }

    } else {
        System.out.println("invalid test");
    }
}
```

Once the login is granted, histFile.updateHFile(double[] test, double[][] table) to update the history file. When initialization, first, traverse the first column of the history file table, once (table[i][0]==-1.0), replace table[i] with test, which is the measurement of this successful login. After the initialization, history file table will contain 8 records.

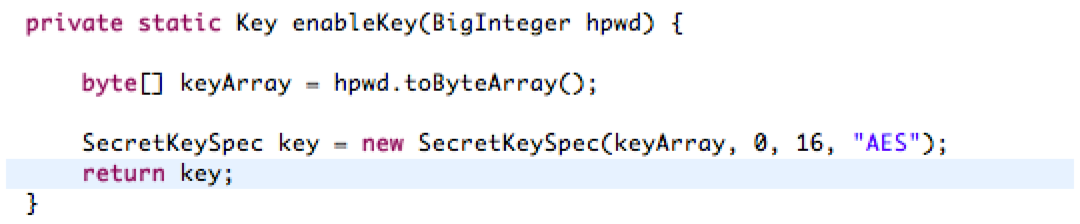
7) Crypt.class

Our project uses "AES/CBC/PKCS5Padding" as our encrypt scheme to encrypt the history file. The IV is randomly generated 16 byte IvParameterSpec.



```java
private static IvParameterSpec IV = new IvParameterSpec(new byte[16]);
```

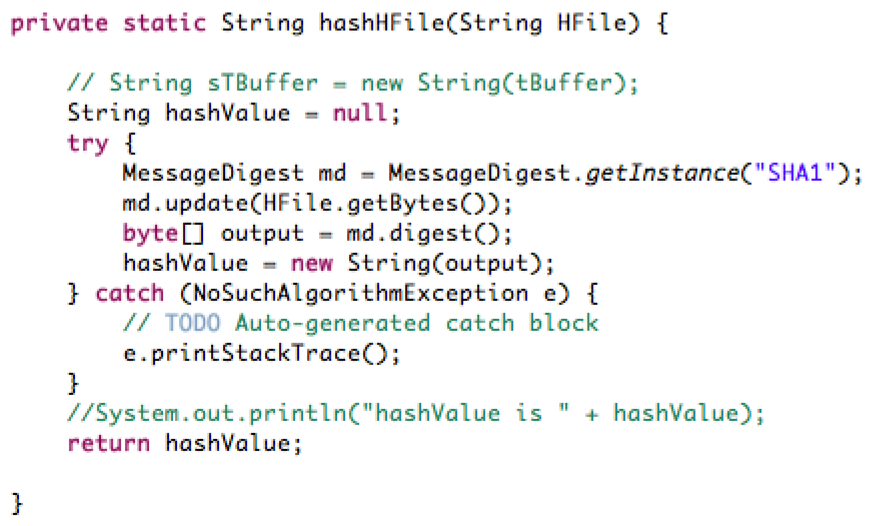
The key used in this scheme is generated from the harden password *hpwd'* computed after each successful login. The function enableKey(BigInteger hpwd) is used for key generation.



```java
private static Key enableKey(BigInteger hpwd) {

    byte[] keyArray = hpwd.toByteArray();

    SecretKeySpec key = new SecretKeySpec(keyArray, 0, 16, "AES");
    return key;
}
```

Basically, it take the hpwd' as the input and output a SecretKeySpec key. Since the *hpwd* is a 160 bit BigInteger as above mentioned, we generate a 16 byte AES key from the first 16 byte from hpwd'. Since the encrypt scheme we chose is a PRP and the encrypt key may change every time(with great likelihood), our implementation can well meet the requirements to safely store the encrypted history file.

The function hashHFile(String HFile) is used for hashing the history file before encrypt.



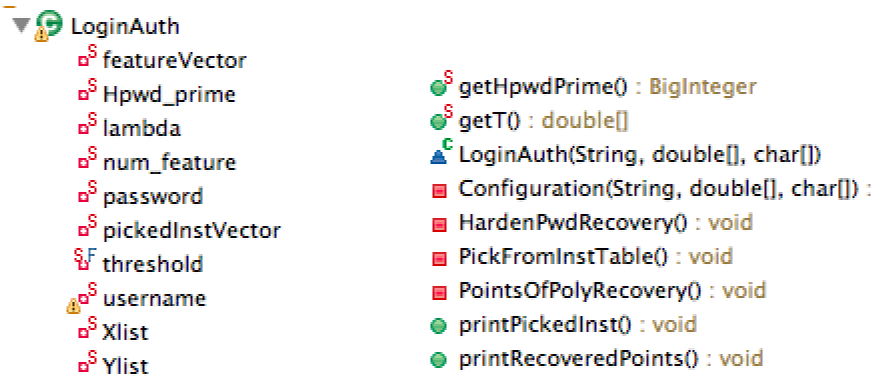
```java
private static String hashHFile(String HFile) {

    // String sTBuffer = new String(tBuffer);
    String hashValue = null;
    try {
        MessageDigest md = MessageDigest.getInstance("SHA1");
        md.update(HFile.getBytes());
        byte[] output = md.digest();
        hashValue = new String(output);
    } catch (NoSuchAlgorithmException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    //System.out.println("hashValue is " + hashValue);
    return hashValue;

}
```

Basically, we created a StringBuffer as the input to this function. All the feature values in the history file are stored in this StringBuffer separated by ','. We choose the Message Digest method and used 'SHA1' to hash 'historyfile' String—hashString. A static String iniHashValue is used to store the hash value of history file each time before encrypt.

The function encHFile(double[][] table, BigInteger hpwd) is used to hash and encrypt the feature values of each successful login using the above mentioned scheme. After encrypting the history file, PrintWriter writer is used to generate encrypted history file "Encrypt_History.txt", "UTF-8"

Since we chose to use symmetric encryption, so the decrypt scheme and related parameters are same as the encrypt ones. Function decHFile(ArrayList<String> cipherText, BigInteger hpwd) is used to decrypt the encrypted history file. Here the input is the encrypted history file, ArrayList<String> cipherText, this function decrypted the content in the ArrayList one by one and use a string String dHString = new String(dHashBuffer) to store all the decrypted measurements. The next step is to call the hashHFile(dHString) to compute the hash then check it against the String iniHashValue. The function checkHash(String newHashValue) is used to do so.If the return value of checkHash(hashHFile(dHString)) is true, the new computed hash value is the same as iniHashValue, the access is thus granted. Since SHA1 has a great property of collision resistance, once the hash value is same, we regard the history file encrypted by hpwd' is same as the original one.

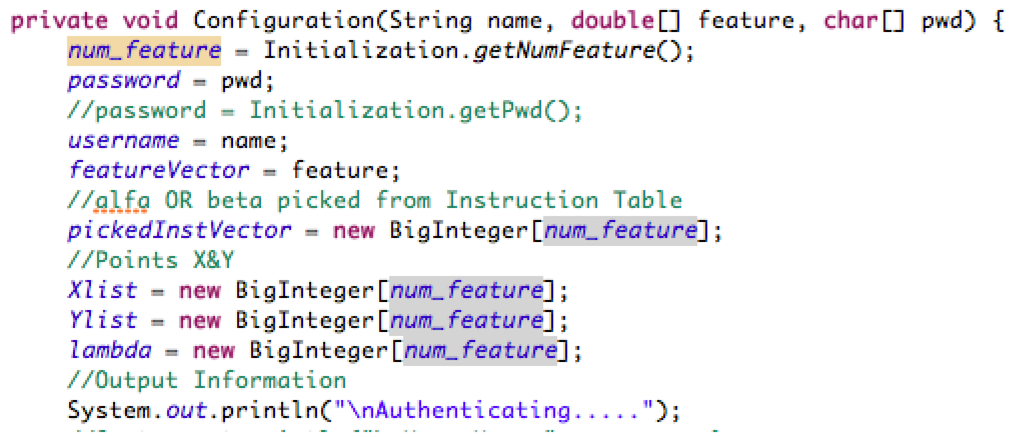
8) LoginAuth.class



```
LoginAuth
    featureVector                getHpwdPrime() : BigInteger
    Hpwd_prime                   getT() : double[]
    lambda                       LoginAuth(String, double[], char[])
    num_feature                  Configuration(String, double[], char[]) :
    password                     HardenPwdRecovery() : void
    pickedInstVector             PickFromInstTable() : void
    threshold                    PointsOfPolyRecovery() : void
    username                     printPickedInst() : void
    Xlist                        printRecoveredPoints() : void
    Ylist
```

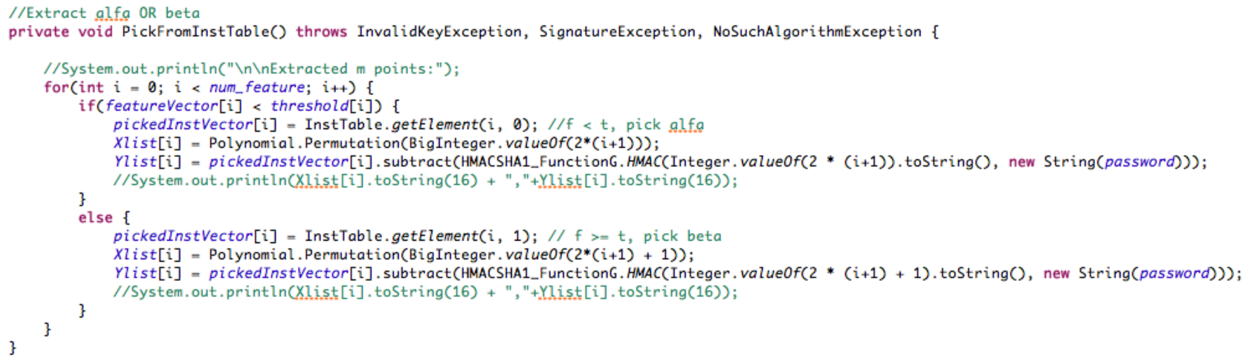
This class holds essential responsibility for login authentication. The main function of it with steps is as follow:

Configuration(): setting all related data structures.



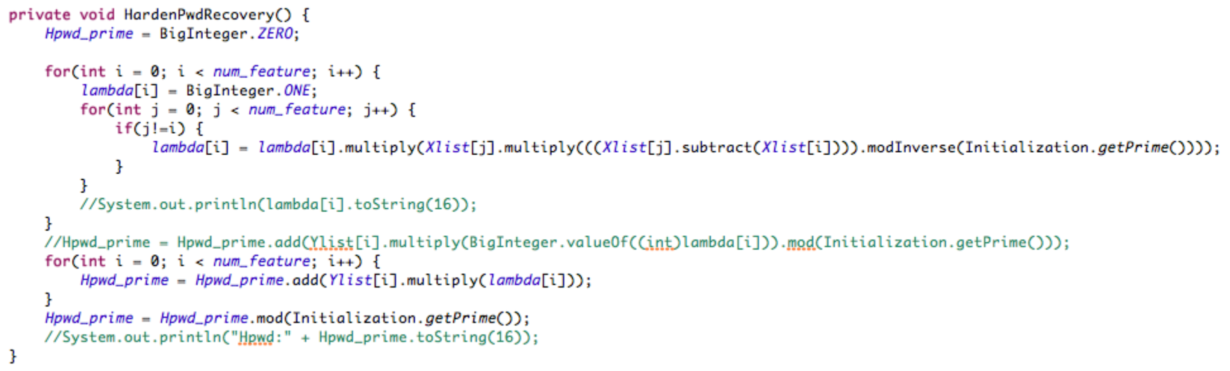
```
private void Configuration(String name, double[] feature, char[] pwd) {
    num_feature = Initialization.getNumFeature();
    password = pwd;
    //password = Initialization.getPwd();
    username = name;
    featureVector = feature;
    //alfa OR beta picked from Instruction Table
    pickedInstVector = new BigInteger[num_feature];
    //Points X&Y
    Xlist = new BigInteger[num_feature];
    Ylist = new BigInteger[num_feature];
    lambda = new BigInteger[num_feature];
    //Output Information
    System.out.println("\nAuthenticating.....");
```

PointsOfPolyRecovery():It also uses BigInteger array just similar as instTable() (well, it uses BigInteger[][] actually) to hold the extracted m points. Another array holds the α or β value selected from instruction table.



```
//Extract alfa OR beta
private void PickFromInstTable() throws InvalidKeyException, SignatureException, NoSuchAlgorithmException {

    //System.out.println("\n\nExtracted m points:");
    for(int i = 0; i < num_feature; i++) {
        if(featureVector[i] < threshold[i]) {
            pickedInstVector[i] = InstTable.getElement(i, 0); //f < t, pick alfa
            Xlist[i] = Polynomial.Permutation(BigInteger.valueOf(2*(i+1)));
            Ylist[i] = pickedInstVector[i].subtract(HMACSHA1_FunctionG.HMAC(Integer.valueOf(2 * (i+1)).toString(), new String(password)));
            //System.out.println(Xlist[i].toString(16) + ","+Ylist[i].toString(16));
        }
        else {
            pickedInstVector[i] = InstTable.getElement(i, 1); // f >= t, pick beta
            Xlist[i] = Polynomial.Permutation(BigInteger.valueOf(2*(i+1) + 1));
            Ylist[i] = pickedInstVector[i].subtract(HMACSHA1_FunctionG.HMAC(Integer.valueOf(2 * (i+1) + 1).toString(), new String(password)));
            //System.out.println(Xlist[i].toString(16) + ","+Ylist[i].toString(16));
        }
    }
}
```

HardenPwdRecovery(): is used to compute the recovered *Hpwd’* which will be used to decrypt the content of encrypted history file, then get result that whether the user is authenticated.



```
private void HardenPwdRecovery() {
    Hpwd_prime = BigInteger.ZERO;

    for(int i = 0; i < num_feature; i++) {
        lambda[i] = BigInteger.ONE;
        for(int j = 0; j < num_feature; j++) {
            if(j!=i) {
                lambda[i] = lambda[i].multiply(Xlist[j].multiply(((Xlist[j].subtract(Xlist[i]))).modInverse(Initialization.getPrime())));
            }
        }
        //System.out.println(lambda[i].toString(16));
    }
    //Hpwd_prime = Hpwd_prime.add(Ylist[i].multiply(BigInteger.valueOf((int)lambda[i])).mod(Initialization.getPrime()));
    for(int i = 0; i < num_feature; i++) {
        Hpwd_prime = Hpwd_prime.add(Ylist[i].multiply(lambda[i]));
    }
    Hpwd_prime = Hpwd_prime.mod(Initialization.getPrime());
    //System.out.println("Hpwd:" + Hpwd_prime.toString(16));
}
```

An important points here is that, when computing the Lagrange coefficient, a tricky method that using Multiply + modInverse instead of using divide can perfectly avoid any precision lose for the computation.

3.3 Choosing Constants(threshold t and constant k)

The results from many keystroke dynamic researches showed that, the distribution of the keystroke measurements follow the normal distribution[1][2] N~(μ,σ2). So the PDF(probability density function) is:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Some research showed that the distribution of keystroke dynamics are quite intense for particular users.[2]

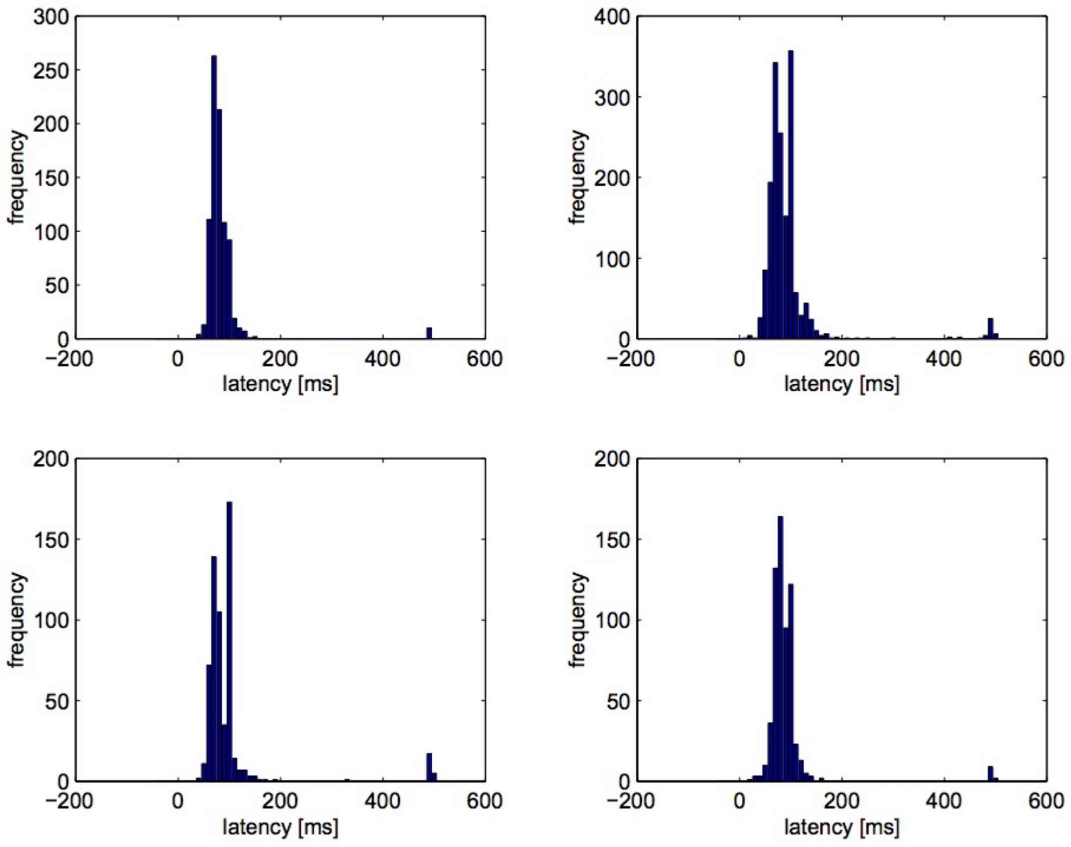


Figure 1: Keystroke Latency Distribution

We figured out that the equation in the paper we discussed at class is like computing the confidence interval of these distributions.

$$|\mu_{ai} - t_i| \leq \quad k\sigma_{ai}$$

The PDF for standard normal distribution N~(0,1) is:

$$f(x) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{x^2}{2}\right)$$

As shown in Figure 2, the x axis represents the value k, and the precent represents the corresponding space covered. As for the value k, when the keystroke dynamics obey standard normal distribution and k is 1, 68.3% of all measurements can fall in the interval [t - σ, t + σ], when k is 2, 99.5% of all measurements can fall in the interval [t - 2σ, t + 2σ].

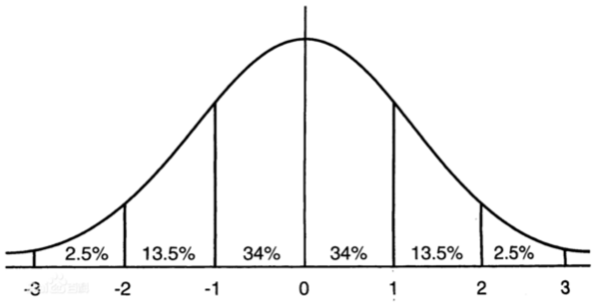


Figure 2 : Standard Normal Distribution

When comparing Figure 1 with Figure 2, we could get a intuitive conclusion that if we want to get appropriate precent of distinguishing features the k value should be smaller than 1.

The research result from the paper we discussed at class [3]showed relationship between k values and the number of distinguishing features

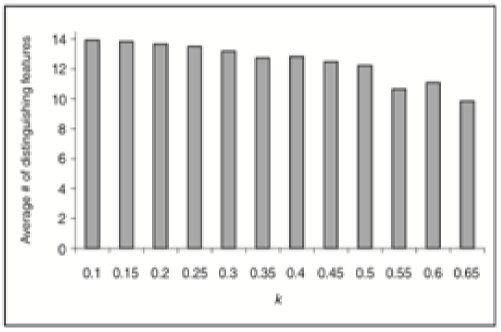


Fig. 3. Average number of distinguishing features

Another consideration is the rate of False Negatives since it is an important evaluation of usability[3]. (Since the False Positives are low under experiments, the priority consideration here is the rate of False Negatives).

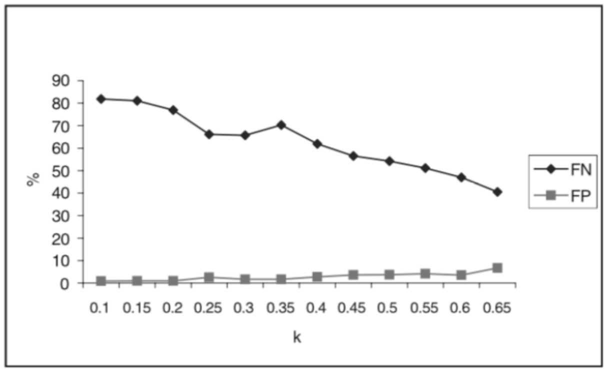


Figure 4: Average number of False Negatives(FN) and False Positives(FP)

5. Evaluation

5.1 Test File Characteristics

Four kinds of test queries are contained in our test file “testfile.txt”:
1-11: one feature value been changed, slightly or drastically.
12-25: three feature values been changed, slightly or drastically.
26-40: five feature values been changed, slightly or drastically.
41-50 records for different user.

Another “testfile2.txt” we provided contains more mixed requests. But make sure the test file you want to run named as “testfile.txt” and locates at the same directory as Keystroke.JAR file.

5.2 Test Result

The test result seems to be reasonable.

The authentication for all log-in attempts will done in several seconds or less.
The sequence number will be shown for each request,

- If you input wrong username, which is not exist in the test file, all request will show “Wrong username”.
- If you input wrong password, the ones for your username will not successfully login.

If you do all things correctly,

- You can see, for request from other username, it will show “Wrong username”
- for right username, the first 8 requests will always be correct, since the history file is not full. After 8 successful requests, If the new request has proper feature vector, it will show “Login success!!”, and update history file , compute the mean / standard deviation,

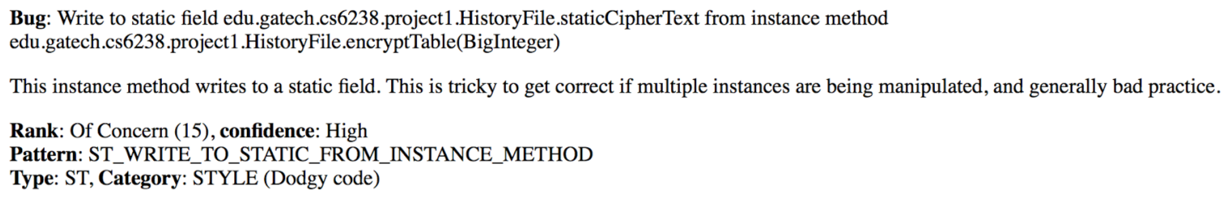
encrypt the history file with new Hpwd, update polynomial, instruction table, etc…for detail you can read the report. If feature vector is not proper, it will show “Loggin failed”

Warning: since the login authentication is combined with Initialization, to have expected result, you need to keep them consistent. Which means, only if you login as the user/password you registered for can our program give you a proper authentication. That seems make sense.

More specificly, for certain K value we set, we can fully control the granularity and accuracy. And the program is sensitive to the requests that even changes a little but go out of proper range, the program can easily pick out such requests and deny it’s authentication.

5.3 Security Analysis

We used the static analysis tool Findbugs to check the source code, fortunately, we have not find any severity security flaws or bugs in our program. However, there may exists some security concerns, for example we used several static variables in our code, like counters. This get the blow warning.



Bug: Write to static field edu.gatech.cs6238.project1.HistoryFile.staticCipherText from instance method edu.gatech.cs6238.project1.HistoryFile.encryptTable(BigInteger)

This instance method writes to a static field. This is tricky to get correct if multiple instances are being manipulated, and generally bad practice.

Rank: Of Concern (15), **confidence**: High
Pattern: ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD
Type: ST, **Category**: STYLE (Dodgy code)

Another kind of warning coming from that when passing a parameter to a self defined fuction, our code failed to check whether or not the parameter is null. Since these fuctions did not accept user input and if the parameter is null or not well formed, we will be informed from other part of the code.

Bug: Null passed for nonnull parameter of new String(byte[]) in edu.gatech.cs6238.project1.Crypt.encrypt(String, BigInteger)
This method call passes a null value for a nonnull method parameter. Either the parameter is annotated as a parameter that should always be nonnull, or analysis has shown that it will always be dereferenced.
Rank: Scary (8), **confidence**: Normal
Pattern: NP_NULL_PARAM_DEREF
Type: NP, **Category**: CORRECTNESS (Correctness)

We fixed this bug successfully, by usig the following codes:

```
String cipherText = null;
                    if(encrypted!=null){
                     cipherText = new String(encrypted);
                    }
```

Now, this bug no longer exists.
Another kind of bug comes from the deal local storage.
Bug: Dead store to login in edu.gatech.cs6238.project1.KeystokeHarden.LoginAttempt(HistoryFile, double[], ArrayList, String)
This instruction assigns a value to a local variable, but the value is not read or used in any subsequent instruction. Often, this indicates an error, because the value computed is never used.
Note that Sun's javac compiler often generates dead stores for final local variables. Because FindBugs is a bytecode-based tool, there is no easy way to eliminate these false positives.
Rank: Of Concern (15), **confidence**: High
Pattern: DLS_DEAD_LOCAL_STORE

Type: DLS, **Category**: STYLE (Dodgy code)
We have reviewed the code and find that the warning is incorrect.

We have fixed the bugs if possible and now the only existing ones are writing to static variables. Since we used this method to pass paremeters within class and it is carefully checked, we believe that they could not cause security problems

Reference

[1]Monrose F, Rubin A D. Keystroke dynamics as a biometric for authentication[J]. Future Generation computer systems, 2000, 16(4): 351-359.

[2]Song D, Venable P, Perrig A. User recognition by keystroke latency pattern analysis[J]. Retrieved on, 1997, 19.

[3]Monrose F, Reiter M K, Wetzel S. Password hardening based on keystroke dynamics[J]. International Journal of Information Security, 2002, 1(2): 69-83.