UD-PUF API

Timothy Dee, Nicholas Montelibano

December 16, 2015

1 Public API

1.1 UserDevicePair

This class is the most useful object in the UD-PUF library. It is used to represent a user device combination. This class implements the authentication component of the library. This class holds a list of challenges for a given user. For each challenge there are presumably multiple responses. Authentication entails comparing a new response for a given challenge to the existing profile built for that challenge.

1.1.1 Enumerated Types

RatioType This enumerated type can take values which correspond to distance, pressure, and time. Several methods use the RatioType to select which of the respective values to set. For example, the method which sets the allowed number of standard deviations from the average for a given metric takes a RatioType and a double. This method uses the RatioType to determine which of distance, time, or pressure should have number of standard deviations set to the double value.
AuthenticationPredicate This enumerated type can take on values which indicate how the failed point ratios of each of pressure, distance, and time will be taken into consideration. For example, if AuthenticationPredicate.PRESSURE is chosen the authenticate method will consider only whether the failed point ratio for pressure is below its respective threshold. AuthenticationPredicate(s) exist for many combinations of distance, pressure, and time.

1.1.2 Constructors

Below are listed the constructors for the UserDevicePair class. The final constructor on line 5 provides the ability to create a UserDevicePair object by specifying all of the parameters. The other constructors provide default parameters for the parameters which are not specified.

Type	Parameter	Meaning	Default Value
int	userDeviceID	UserDeviceID can be used to keep track of the user	
List<Challenge>	challenges	A list of Challenge objects corresponding to challenges already constructed for the user, device	empty ArrayList<Challenge>
double	time_allowed_deviations	This is a parameter used during authentication. When testing for points which don't match the profile this value determines the number of standard deviations a point in a response may be away from the corresponding average for that point in the profile. Points which fall outside of this number of standard deviations will be considered not within the profile. We call these failed points.	1.0
double	distance_allowed_deviations	Same as time_allowed_deviations, but for distance	1.0
double	pressure_allowed_deviations	Same as time_allowed_deviations, but for pressure	1.0
double	time_authentication_threshold	The ratio of failed points to total points at which the user will be considered within the profile and pass authentication.	0.4
double	distance_authentication_threshold	The ratio of failed points to total points at which the user will be considered within the profile and pass authentication.	0.9
double	pressure_authentication_threshold	The ratio of failed points to total points at which the user will be considered within the profile and pass authentication.	0.7
double	new_response_confidence_interval	Contains the confidence interval for the response from the most resent authentication.	-1

```
public UserDevicePair(int userDeviceID){}

public UserDevicePair(int userDeviceID, List<Challenge> challenges){}

public UserDevicePair(int userDeviceID, List<Challenge> challenges, double
    allowed_deviations, double authentication_threshold){}
```

1.1.3 Public Methods

```
// Adds challenge to list of challenges correlating to this user/device pair
public void addChallenge(Challenge challenge){}

// gets the challenges for this user, device
public List<Challenge> getChallenges(){}

/**
 * true if the new_response_data has a certain percentage of points which
 * fall within the profile for the challenge indicated by challenge_id
 *
 * The testPressListVsDistrib() method in the Util file seems to be
 * performing the authentication
 */
```

```
public boolean authenticate(List<Point> new_response_data, int challenge_id){}

// return the userDeviceID
public int getUserDeviceId(){}

/**
 * return the number of failed points from the previous authentication.
 * Return -1 if there is not previous authentication.
 */
public double failedPointRatio(){}
```

1.2 Profile

The profile class is meant to contain all information about a user's behavior which is relevant for authentication. In our case this means the average and standard deviation for each of the points in the user's response. profile also contains methods to return a confidence interval. This confidence interval represents how much trust we put in the accuracy of this profile. In other words, a high confidence interval implies higher probability that authentications are correct compared to a lower confidence interval.

1.2.1 Constructors

Here, listed, are the constructors for the Profile class. A user of this library will not need to worry about creating a profile as this is completed by the Challenge class.

```
public Profile(List<Response> normalizedResponses, List<Double> time_lengths,
    List<Double> motion_event_counts) {}

public Profile() {}
```

1.2.2 Public Methods

The public methods of the profile class are, for the most part, used internally. the methods getPressureMuSigmaValues(), getPointDistanceMuSigmaValues(), getTimeDistanceMuSigmaValues(), getTimeLengthSigma(), getTimeLengthMu(), getMotionEventCountSigma(), getMotionEventCountMu(), and getNormalizedResponses() are all used in authentication. Other methods provided return the confidence interval for the profile.

```
public void addNormalizedResponses(List<Response> normalizedResponses) {}

public MuSigma getPressureMuSigmaValues() {}

public MuSigma getPointDistanceMuSigmaValues() {}

public MuSigma getTimeDistanceMuSigmaValues() {}

public double getTimeLengthSigma() {}

public double getTimeLengthMu() {}

public double getMotionEventCountSigma() {}

public double getMotionEventCountMu() {}

public ArrayList<Response> getNormalizedResponses() {}

public double getConfidence_interval() {}

```

```java
public double get_sd_pressure_contribution() {}

public double get_sd_time_contribution() {}

public double get_sd_distance_contribution() {}

public double get_num_motion_event_contribution() {}

public double get_sd_motion_event_contribution() {}
```

1.2.3 Confidence Intervals

Confidence interval is a ranking of how good a profile or authentication is. A profile will have a high confidence interval if all of the responses contained in the profile are close to one-another. Confidence intervals for profile are computed according to the equation 1.

There is also a notion of confidence interval for an authentication which is separate from the profile confidence interval. Authentications will have a high confidence interval if the response being used to authenticate is close to the responses in the profile. Confidence intervals for authentications are computed according to the equation 2.

Grades for pressure, time, and distance are created independently in both profile confidence interval and authentication confidence interval.

$$1 - \Sigma_{i=1}^{N}(\sigma_i/\mu_i) \tag{1}$$

$$1 - \Sigma_{i=1}^{N}(|p_i - \mu_i|/\mu_i) \tag{2}$$

1.3 Response

1.3.1 Constructors

Response constructor takes in a responsePattern. This responsePattern represents the Points the user traced in response to the provided challenge.

```java
public Response(List<Point> responsePattern){}
```

1.3.2 Public Methods

Normalize method preforms the following function. Normalizes points in response. The normalizingPoints are a list of points to normalize the response to. In other words the response will then contain exactly these point having some pressure determined by the original response.

```java
public List<Point> getResponse() {}

public void normalize(List<Point> normalizingPoints, boolean isChallengeHorizontal) {}
```

1.4 Challenge

1.4.1 Constructors

Takes a list of Points corresponding to the challenge points presented to the user.

```java
public Challenge(List<Point> challengePattern, int challengeID) {}
```

1.4.2 Public Methods

The challenge contains a number of responses. These responses correspond to the responses generated by the user when they are presented this challenge. A response is normalized when it is added to the Challenge.

```
// add a response to this challenge
// this method will normalize the response before adding it
public void addResponse(Response response) {}

// return the mu sigma profile for the responses to this challenge
public Profile getProfile() {}

// return the challenge points
public List<Point> getChallengePattern() {}

// return the ID of this challenge
public int getChallengeID() {}

// Determine if the challenge is more horizontal than vertical in oreantation
public boolean isHorizontal(){}
```

1.5 Point

1.5.1 Constructors

Constructor which take x, y, pressure represented x position, y position, and pressure of the point respectively.

```
public Point(double x, double y, double pressure) {}

public Point(Point p) {}
```

1.5.2 Public Methods

```
public double getX() {}

public double getY() {}

public double getPressure() {}

// compares each of x, y, pressure for equality
public boolean equals(Object p) {}
```

2 Examples

2.1 Creating a UserDevicePair

```
 Challenge challenge;
 Response response;
 List<Point> response_points;

 // create a userDeficePair
 ud_pair = new UserDevicePair(0);

```

```java
// create a list of challenge points
List<Point> challenge_points = new ArrayList<Point>();

// sample points for testing
challenge_points.add(new Point(100, 100, 0));
challenge_points.add(new Point(200, 100, 0));
challenge_points.add(new Point(300, 100, 0));
challenge_points.add(new Point(400, 100, 0));

// add the challenge to it which I want to authenticate against
// create 3 responses to add to this challenge
challenge = new Challenge(challenge_points, 0);

for (int i = 0; i < 3; i++) {
   response_points = new ArrayList<Point>();

   // create the response
   for (int j = 0; j < 32; j++) {
 response_points.add(new Point((300 / 32) * j + 100, 100, i));
   }

   response = new Response(response_points);
   challenge.addResponse(response);
}

// the mu sigma for the responses should be
// mu : 1
// sigma : sqrt(2/3)
ud_pair.addChallenge(challenge);
```

2.2 Creating a Response Object

```java
// create the response object
List<Point> response_points = new ArrayList<Point>();

// populate the response_points list with 10 points
for (int i = 0; i < 9; i++) {
   response_points.add(new Point(i, i, .1 * i));
}

response = new Response(response_points);
```

2.3 Creating a Challenge Object

```java
// construct some test data
List<Challenge> challenges = new ArrayList<Challenge>();
List<Point> challenge_points = new ArrayList<Point>();

// sample points
challenge_points.add(new Point(100, 100, 0));
challenge_points.add(new Point(200, 200, 0));
challenge_points.add(new Point(300, 300, 0));
challenge_points.add(new Point(400, 400, 0));

challenges.add(new Challenge(challenge_points, 0));
```