## .About The Application :

### 2.1 Introduction

VIP Co.Auth is a project meant to monitor many important data points. The primary aim of VIP Co.Auth is to provide continuous authentication to users without them having to worry about any unauthorized usage of their devices. VIP Co.Auth's core design is very simple as it just stores data inside the user's device and later onto the cloud. This data is used for training, using one clustering algorithm among a few, which in turn establishes profiles for users. These profiles define how users typically use their device on a day to day basis. If, by any chance, an outsider gets access to this protected device, then their behaviour is monitored and compared, post log-in, against the original user's profile created during training. Therefore, if major anomalies are detected, the idea is to make the device inaccessible and it can only be further used if the user is able to establish themselves as the original user.

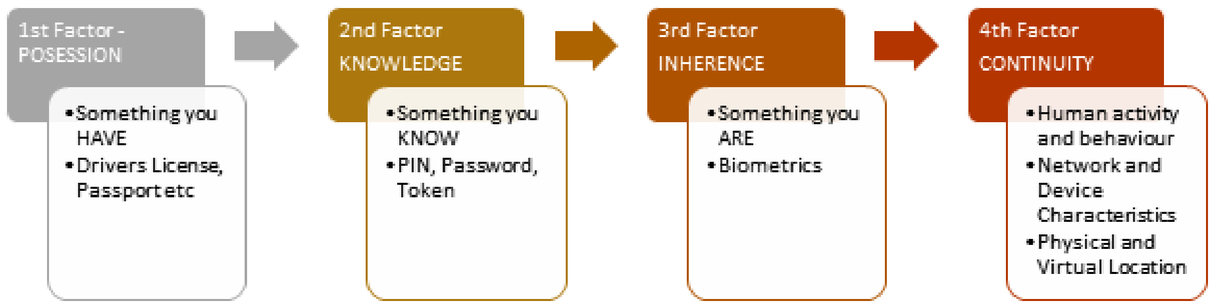

*Figure 1 Basics of Continuous Authentication*

### 2.2 Collected Data

Initially, the plan was to collect many data points which was believed to provide better results. In time, few of these data points proved to be redundant or not helpful. The aim of collecting data was to help in creating a profile for the individual user. So, the following were initially collected:

- Package Name: Every application has their own package name. This data is helpful as it is unique and helps in distinguishing between various apps.
- Start Time (of each app): By monitoring the opening time of each application, the user's behaviour can be understood.
- End Time (of each app): Monitoring the closing time of each app seemed to be redundant as the End time can be identified by noting the opening time of the next app. So, this data point was, later on, **removed**.

- Usage Time (of each app): The amount of time spent in each application is a very important data. User behaviour significantly varies in this aspect.
- Button Clicks: Within an app, monitoring the various buttons clicked is also an important factor as it details how an individual surfs through an app. Later on, this data point proved to be a privacy concern. Hence, it was **removed**.
- Tap count: The total taps made on the screen throughout an app was also monitored.
- Location: Location is an important data factor as it almost instantaneously helps in determining any anomalies.
- Screen co-ordinates: This data point proved to be too vague as not much could be derived from it. Therefore, it was also **removed**.

Once these data points were collected, the next plan was to collect data that was more device based than usage based. So, the next and final data points that were collected were as follows:

- WiFi/Network Name: Identifying the network name or the WiFi name helps in monitoring anomalies as which network the user generally connects to is noted.
- Total Number of Click Events: This data point is similar to total tap count where the only difference is that Total Number of Click Events is based on the number of button clicks and not random taps.
- Transmitted and Received Data (over the internet): Internet usage varies from person to person and so understanding the internet usage of a user greatly helps in creating their user profile.
- Checking Network Connection: This data point is to simply observe if the user is using WiFi or mobile data or isn’t connected at all.
- IP Address: Since IP addresses can vary based on the network that is used, this data point is crucial in detecting anomalies.
- Battery Status: This data point is used to know the status of the battery, i.e., if it is charging or discharging. This helps in understanding the behaviour of the user in detail.
- Battery Charging Source: It is also important to know which source the user generally uses to charge their phone (via USB or adapter).
- Battery Percent: A simple yet important data point is identifying the level (percentage) of the battery.

- Day of The Week: The last data point that was decided to be added was to know the day of the week. In this case, the week would start from a Sunday.
- Sensor Data: Sensors were put to use to identify the gyroscope, accelerometer and orientation values.

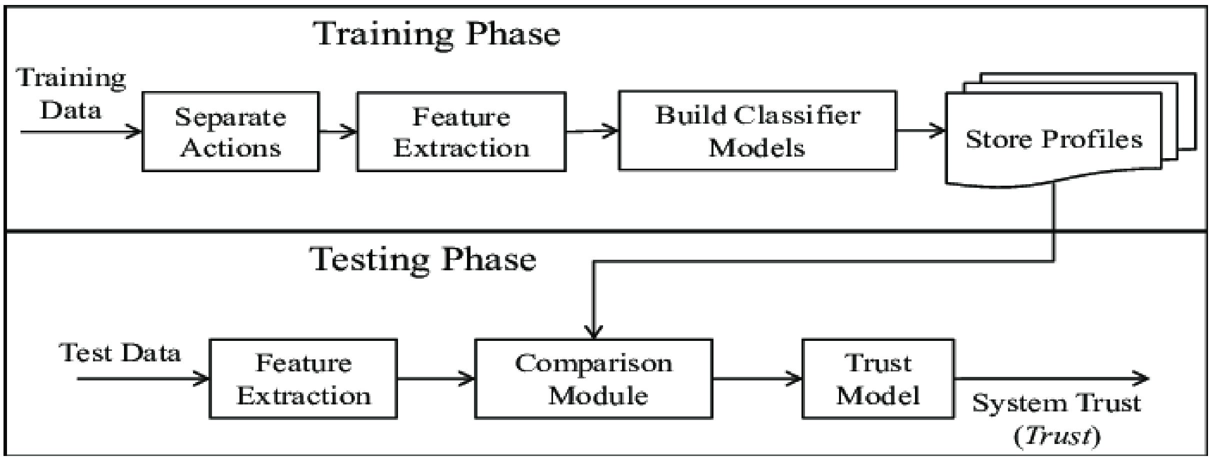

*Figure 2 The Training and Testing phase of CA*

## 2.3 How Data Were Collected

The most fundamental code in VIP Co.Auth is where it identifies the type of event that has occurred due to the actions of the user. These events can be anything - such as a click of a button or selection of any item or a view etc. To identify these events, Google provides services called Accessibility Services. VIP Co.Auth makes use of Accessibility Services to define two events (out of many):

1. TYPE_VIEW_CLICKED: This event is triggered whenever the user clicks a view such as a button.
2. TYPE_WINDOW_STATE_CHANGED: This event is triggered whenever the user switches between applications.

Calendar.getInstance() gets the current time, but it isn't formatted yet, and mdformat defines the format as to how the time should be shown (HH:mm:ss in this case). StartTime gets the time in the proper format.

Now that VIP Co.Auth is able to identify the type of events, it also needs to identify if the switch between apps is not between the same application. For eg: A user might open Facebook then go to the recents menu and then open Facebook again. The event TYPE_WINDOW_STATE_CHANGED will be called twice unnecessarily in this case. To prevent that, a condition check is performed. This is done by comparing the previous package name 'b' with the current package name. If the package name is the same then that indicates no switch occurred between apps, otherwise there's a switch.
Since there are apps which can keep occurring when a user switches between apps, there was a need felt to ignore these packages so as to reduce redundancy. Therefore, the following app packages

were ignored:

- The launcher app (home app)
- Settings
- Google search app (widget on the home screen)
- System UI
- Input method (this can occur within any app that needs any sort of an input from the user)

*Time2* is noted when one of these packages are opened. *Time2* becomes the opening time of one of the 'ignored' packages, which can be used to find the closing time of the previous app. This is done in this manner because the user may spend an indefinite amount of time, for eg: in the launcher, and then proceed to open a different app. If the opened app is none of the 'ignored' apps, then the code proceeds to the 'else' part inside the **onAccessibilityEvent()** method. The calendar object does the same thing as previously mentioned. *dayOfWeek* is one of the data points collected and it stores the day of the week in a number. Sunday is 1, Monday is 2 and so on.

Proceeding further, a JSON object is created. In this scenario, JSON objects are used to store data in to a file. JSON data is easy to parse and analyze and so its usage in VIP Co.Auth is justified. The method obj.put() stores any value that is passed into the object with a tag associated with it.

To read data from the JSON object, an array is initialized named *tempjsonArray*. The data that is read is henceforth simply put into the studio log to be viewed by the programmer. Now since this has been done, the next step is to store the data into a file.
The call for writing happens in both **average() getsensor()** methods. Once the file has some data in it, the else part of the code is always skipped. What's happening is this – The file's data (if any) gets stored in *tempjsonArray* and then it proceeds to add data from the JSON object back into *tempjsonArray*. This way the existing data is unharmed, and the new data is added after the previous entries.

Two more methods come into picture here. They are **store()** and **read().** These two methods are called inside **average()** and **getsensor(). store()**, for both methods, is used to create a permanent file into the internal storage of the device. Inside **store()** we find that the specified directory for the file to be stored is the DOWNLOADS folder in the internal storage of the device. Moreover, the name of the file to be stored is “MainFile”.

Data is written into the file as new entries keep coming. The object *fw* of the class FileWriter inside the method **write()** writes the data from the json object into the file. The method **read()** fetches this particular file from the DOWNLOADS folder and makes it possible to read the contents within the logs.

The next method in the code is called **onServiceConnected().** This method is called when there’s a connection to the service. There’s a communication channel (IBinder) here working in the background and that will be discussed later.

As we can see inside **onServiceConnected**(), *info* is an object of the class AccessiblityServiceInfo. *Info.eventTypes* tells the service what all events are supposed to be used. In this case, the events TYPE_WINDOW_STATE_CHANGED and TYPE_VIEW_CLICKED are used as mentioned previously.

WindowManager is an interface that applications use to communicate with it. WindowManager is responsible for managing which windows should be visible, how they are laid out on the screen etc. In the above code, WindowManager makes use of several constants such as FLAG_NOT_FOCUSABLE, FLAG_NOT_TOUCHABLE, TYPE_SYSTEM_ALERT and FLAG_WATCH_OUTSIDE_TOUCH. The first constant makes sure that the window won’t ever receive any input focus or button events. So whatever touch is made, that touch goes to window behind or under the current window. FLAG_NOT_TOUCHABLE makes sure that window won’t receive any touch events.

.Now we come to the method **getLocation()** which retrieves the location of the device. The first condition checks if the permission ACCESS_FINE_LOCATION is given in the manifest.

*LocationManager* makes use of constants such as LOCATION_SERVICE, NETWORK_PROVIDER and GPS_PROVIDER. These three constants help in retrieving the location of the device from the network provider or from the GPS if it is enabled in their device. So, if the location has some value other than null, the latitude and longitude is stored in *latti* and *longi* respectively. The pre-defined methods that make this work are **getLatitude()** and **getLongtitude()**.

Next, we come to the **onStart()** method. An intent is invoked every time the onStart() method is called. This method basically starts the service if it isn’t already running.

The way the name of WiFi or the mobile network is collected is by the help of a method named **checkNetworkConnectionStatus()**. This method makes use of the class ConnectivityManager and a constant neamed CONNECTIVITY_SERVICE. The method then checks if there is an active network established in the device. If an active network is detected, then it proceeds to verify if the network is a WiFi network or a mobile network. This is done with the help of two more constants named TYPE_WIFI and TYPE_MOBILE. Once the network is identified, it is simply stored into the JSON object. The method stores the value '1' for a WiFi connection, '0' for a mobile network and '-1' for no connection.

The data point *tapcounter* was found using the method **onTouchEvent()**. *tapcounter* increments itself every time **onTouchEvent()** is called and its value is passed to JSON object (in the onAccessiblityEvent Method).

To find out the transferred data over the internet, two user-defined methods are put to use. They are **networkStart()** and **networkEnd().** The former method gets the data transferred but it doesn't have a defined time frame. This method also checks if the device supports data monitoring over the internet.

The method **networkEnd()** defines the time frame. The time frame, in this case, is basically when any particular app opens and closes. The amount of data that takes place within this time is stored. Furthermore, the transfer of data is by default given in bytes. Within the application, it is converted to kiloBytes.

VIP Co.Auth makes use of the device's sensors to monitor the gyroscope, accelerometer and orientation movements. These movements can be used effectively to detect outliers if an unauthorized person has gained access. This data is collected every 5 seconds. So, the data that is collected every 5 seconds for a period of 5 minutes is sent to the method named **average()** where the average of those data is calculated.

The data collected through the sensors is stored in the Json Objects which are stored in a different file named 'SensorFile' for every 5 minutes, after which the data is sent to the file 'MainFile' and 'SensorFile' is refreshed.
The next step involved uploading data to the cloud so that the device doesn't face any issues with storage. The basic idea is to let the device keep storing data into the file for a certain amount of time

(let's say, 24 hours). When data is collected for this period, the data from the file (not the file itself) is uploaded to the cloud and the existing file in the device is deleted. This makes sure that no data is lost and everything is stored in the cloud without any redundant data.

The method **setAmazons3Client()** initializes the Amazon object with the required credentials. Once this is done, it calls the method named **sendDataToS3().** This method uploads the data stored inside the file. To do this, the MAC address of the device is passed as a parameter so as to differentiate between devices. The MAC address is found using the method **getMacAddr()** and once that is done, **sendDatatoS3()** calls the method **deleteAWS()**. This method deletes the file 'MainFile' from the local storage.

Once the data has been collected , it goes through the second phase , learning phase and detection while it learns ,Many applications require being able to decide whether a new observation belongs to the same distribution as existing observations (it is an *inlier*), or should be considered as different (it is an *outlier*). Often, this ability is used to clean real data sets.