Using Keystroke Dynamics to Authenticate a User Based on their Typing

Jack Francis

April 12, 2022

Abstract

Hello

Contents

Chapter 1

Introduction

Chapter 2

Survey of Literature

Chapter 3

Design and Implementation

The following procedure from section 3.2 onwards is a rough implementation of the validation procedure by Ramin Toosi and Mohammad Ali Akhaee in their excellent paper 'Time–frequency analysis of keystroke dynamics for user authentication'. [1] The paper is theoretical in nature and describes an approach for performing validation on one word and then comparing them. It is in essence a one-time system whilst mine is a continuous system that aims to keep the user safe. In my project I've modified and implemented their validation approach whilst adding data gathering, word forming, word selection, word storage and update function.

3.1 Data gathering and Forming

My program relies on capturing the users keystrokes and then processing them and then comparing them using a similarity measure. In order to do this, I decided to use the Keyboard Library REFERENCE HERE as it is a lightweight, secure and modern library that makes capturing keystrokes easy. In my project, I make use of the hook function of the library which is used to "hook" onto a users keyboard and record all of the users actions in and create keyboard events for each action. The record function which makes use of this hook function is shown in figure 3.1. The code snippet is very simple, first the program will 'hook' onto the keyboard using the Keyboard Libary mentioned above, record all keystrokes until the interval has passed and then stop recording. The start time of the interval and an array of Keyboard Events are then returned to the main body of the program. The start time of the interval is recorded and returned as it used further on in order to be able to place keyboard events on a time line in the context of the interval.

A keyboard event is generated every time the user does something on the keyboard, whether that be pressing or releasing a key. Further information such as the type of the action (whether it was an a press or a release), which key is this action happening on and a a highly accurate time stamp of when the event occurred. Figure 3.2, shows an example of a keyboard event produced by the function when the user presses down the 'h' key.

The first element in the array is the action, this can be either 'up' or 'down' which are both self-explanatory. The next field is the scan-code which is a field

```
def record(interval):
    recorded = []
    startTime = time.time()
    keyBoardHook = keyboard.hook(recorded.append)
    time.sleep(interval)
    keyboard.unhook(keyBoardHook)
    return recorded, startTime
```

Figure 3.1: Record Function

```
['down', 35,  'h', 1649693924, None, None, 'N']
```

Figure 3.2: The keyboard representation of a user pressing "h"

I don't use but is useful for identifying keys easily. After this, is the name of the key which in this case is 'h' as we pressed the 'h' key down. The next field is the time since the epoch in seconds which is useful as it is this precise time-stamp that is used to do the rest of the calculations. The other three fields are device, modifiers and whether or not the user used a keypad. None of these I use in my program and as such are discarded almost immediately.

A small amount of pre-processing is then done on this data before it is paired up. The first step is to remove the scan code, keypad, modifier and device from each keyboard event and convert them into something lighter and more usable. The next step is to take the start time that is returned by the record function and subtract this from the time stamp in each keyboard event to get the time that the action occurred in the interval. Figure 3.3 shows what the data in 3.2 looks like after going through this.

```
['down', 'h', 0.1]
```

Figure 3.3: Keyboard representation of a user pressing 'h' after pre-processing

The data collected at this point is stored as a 2D array with each sub array corresponding to an action. An example sub array is shown in 3.3.

In this form the data is unable to be used for anything, as it currently takes the form of a number of individual actions seem to have no relation to one another. Therefore, the next step is to form pairs from the data. A pair is formed of when 'down' action and one 'up' action where the key field matches and the 'down' is before the 'up'. The reason this is done is that it allows the program to work half as much data which reduces the number of unnecessary data points and allow the program to be able to form words using these pairs.

There are two main rules to follow when pairing the data. Due to the nature of how the data is collected, it is currently stored in chronological order which is very useful. In many cases, the user will press and release a

```
data = [
        ['down', 'h', 0.1]
        ['up', 'h', 0.2]]

pairs = [['h', 0.1, 0.2]]
```

Figure 3.4: Example of simple pairing

key in quick succession without pressing any other keys. Due to the chronological nature of the data, pairing theses types of presses is easy. All that is needs to be done is to iterate through the pairs and when we come to a 'down' action then simply select the next value in the array if it is an 'up' action and the key matches. Figure 3.4 shows an example of this type or pairing. However, this type of nice easy matching is not always the case.

In some cases, a user may press more than one key down at once. This might occur when the user is capitalising words using 'shift' or when the user is typing fast so they may be already pressing down the next key before releasing the previous. An example of what the data will look like when this is the case is shown in Figure 3.5. Applying the previous method in which we pair up keys with matching key types and opposite actions which are next to each other would result in the output shown in the pairs array. As you can see, this is not correct and would only lead to one pair where there should be two.

In order to fix this, it is necessary to include another case in the code. If the current action doesn't have a matching key and opposite action next in the array, then the program will iterate through the rest of the array starting from the current point in it searching for the next entry with a matching key and an opposite action that is after the current. If it finds one it will then pair them up. The key thing we assume for this to work is that every action has an opposite action. In nearly all cases we assume this to be true as it is highly unlikely that a user will hold down a key for the entire interval. If in a rare case this occurs however, this is handled. If the method cannot find a pair, then it is still added to the pairs list with an end time of the length of the interval.

```
data = [
        ['down', 'h', 0.1]
        ['down', 'shift', 0.11]
        ['up', 'shift', 0.2]
        ['up', 'h', 0.23]]

pairs = [['shift', 0.11, 0.2]]
```

Figure 3.5: Example of simple pairing

The finished algorithm is shown in Figure 3.6. The reason for the error handling is that when coming to the end of the data, attempting to access the next element to check if it can be paired up results in an index error.

The resulting pairing algorithm shown in figure 3.6 has a time complexity of $O(n^2)$. In a program which is all about speed and minimal impact to the user, it is essential that the program has a the lowest time complexity as possible. Due to the complicated nature of how users type I believe this is the best time complexity for a problem of this nature.

3.1.1 Forming Words

After forming the pairs, the next step is to form words from these pairs. The words that the program forms are essential as it is this that the program uses to compare users. In English words take many forms, as such it is needed to account for many different possibilities in the word forming function. This function takes in the list of pairs and returns an array of word objects. The reason I decide to

```
def rawPairs(self):
    """
    Converts the array from the process function into key pairs

    Returns:
        2D array: Consisting of a pair of actions from the array
            above.
    """
    pairsArray = []
    for i in range(len(self.processed)):
        try:

            if (self.processed[i][2] == 'down' and
                self.processed[i+1][2] == 'up' and
                self.processed[i][0].lower() ==
                self.processed[i+1][0].lower()):
                # If the next value in the array is the up action
                pairsArray.append([self.processed[i][0],
                    self.processed[i][1], self.processed[i+1][1]])
            else:
                # Otherwise, search for the next opposing action and
                    pair them up
                for x in range(i, len(self.processed)):
                    if (self.processed[x][0].lower() ==
                        self.processed[i][0].lower() and
                        self.processed[x][2] == 'up' and
                        self.processed[i][2] == 'down'):
                        pairsArray.append([self.processed[i][0],
                            self.processed[i][1],
                            self.processed[x][1]])
                        break
        except IndexError:
            pass
    return pairsArray
```

Figure 3.6: Pair Forming Function

pivot to an object orientated approach at this point in time is that these words are heavily utilised and I would like to have methods attached to them. For example, it is far easier to generate the Key Down Signal mentioned in following sections on a word by word basis rather than having one function in the main body of the calculation class. This reduces the amount of lines written and makes code easier to read and understand.

A word is defined in my program as a sequence of pairs bounded by punctuation, white space or the use of modifiers such as 'shift'. In latter stages of this report, I refer to these as break pairs. The one notable exception to this rule is when an apostrophe or a hyphen is detected. If this occurs, then the program will check the previous pair and the pair afterwords and if both are letters and not numeric or punctuation, the the pair is added to the word.

The data at this stage takes the form of a 2D array. The program will iterate through the 2D array it is given and check the key that the pair matches. If

it is a letter or a number then it is added to another array which is used to store the current word being formed. If a break pair is found, it is not added to the current word, the current word is used to form a word object which is then saved to an output array and in some cases further action will be taken depending on what type the break pair takes. If the break pair is a white space pair then the pair is simply skipped.. However, if the break pair is a modifier such as 'shift' or 'ctrl' then the relevant entry in the semantics dictionary is updated for that user. This dictionary is used in the validation section of the project and is another indicator on how a user types. Backspace handling is done separately. If the user has pressed backspace, then the last letter added to the word is removed from it. The program can handle multiple backspaces even if they delete the entirety of the current word. If this occurs, the previous word object is popped off the array to be the current word and the last letter of this new current word is removed.

When the program gets to the last pair in the input array, if the pair is not a break pair then the pair is added to the current word and the current word forms a word object which is then saved to the output array.

The program will then return the output array which at this time is formed of word objects and the semantics dictionary. The output array is then saved to the wordsOut attribute of the Calculation class while the semantics is saved to the semantics attribute in the class.

The state of the data after this section is simple. The data is an array of word objects. Each word object is in essence an array of pairs with associated timings attached. This whole section could be defined as the pre-processing of the data to get it into a format that can be used in order to perform similarity calculations.

3.1.2 Data Selection

If the program was to go through and check every single word for similarities, the cost in terms of time would be excessive and would make the program unfit for use especially if the user typed quickly during the interval. For example, if the program checked every word and the user typed XXX words in a 60 second interval, the time taken would be XXX which while highly accurate and secure would render the program unusable as the time taken to process and perform all the similarity calculations would be in excess of the interval and as such would lead to a lower degree of accuracy and security. Furthermore, this would severely impact the performance of the users computer and as such go against one of the main aims of the project.

As such, it is necessary to use a sampling method to choose words from the list of words chosen by the word forming function. While this is less accurate than checking every word, the performance gain over checking every word is huge with on average time saving of XX per interval. Choosing how many words were selected was the next problem I endeavoured to fix.

I conducted a number of tests measuring how long the entire validation procedure took. Initially I started with 4 words chosen per interval with one chosen every quarter of the interval. I then increased the number of words chosen by two each time with the interval remaining the same. At each testing point, the interval remained the same with a word selected every $\frac{Interval}{Amount\ of\ Words}$

The test data consisted of a user typing a paragraph. The same test data was used for all of the tests in order to allow a fair test to be conducted.

Figure XX shows the results of such a test. XXMORE HERE

In the end I settled on 4 words chosen with a word chosen every quarter of the interval. This struck a good balance between performance and accuracy with neither being impacted too negatively.

The process to select 4 words from each interval is simple. Given the word list returned by the word former, the program performs the calculation shown in Figure 3.1 where w is the word list n is the number of words in the words lists returned by the word former and where k is the amount of words to be chosen.

$$EveryX(w) = \frac{n}{k} \tag{3.1}$$

This value returned by this equation is then used in the main body of the function. First of all the program chooses the first value in the words list and adds it to the output. It will select the value returned by figure 3.1. After doing this, it will add on the value returned by the equation again and select the word at that index again. This will keep happening until either the amount of words selected by the program is the number of words to be chosen. The resulting words are then returned by the function to be used in the rest of the program.

3.2 KD Signal

Once the raw keystroke data has been formed into words and the words chosen, the next step is to transform the data from a word object made up of keystroke pairs into numerical data that can be used by later algorithms such as Dynamic Time Warping (DTW) and the correlation coefficient. The best way to do this is to transform the data into a measure of how many keys are being held down at a particular point in time. The resulting output is known as a key down signal (KDS). [1]

To convert a word into a key down signal, the start and end times of the word being transformed are used. Assume that w is the array of times that key actions occur in a particular word. w_1 being the time of the first action and w_n being the time of the last action. This part will loop through all timestamps until it ends with the final time which is denoted by w_n. The accuracy of this step is paramount as it is the level of detail that is the base accuracy for the rest of the steps. A higher accuracy means that the program will check more data points within this range at the cost of reduced performance as the level of points being checked increases. The current level of this is set to 4 decimal points which seems to provide a good balance between accuracy and performance. However this is customisable.

$$KDS(w) = \sum_{i=w_1}^{w_n} K(w_i) \tag{3.2}$$

K is the next step of the algorithm and is heavily based on the KDS algorithm shown in [1]. n is the array of key presses that is used in the previous step. This step of the algorithm iterates through all the key presses and uses a modified Heaviside step function denoted by h which is run twice per pair with the time input from the previous denoted as t and the 'down' action denoted as n_i^1 and

the 'up' denoted as n_i^2. The value returned by the Heaviside step function with the 'up' action is subtracted from the value returned by the 'down' function.

$$K(t) = \sum_{x=1}^{n_k} h(t, n_i^1) - h(t, n_i^2) \tag{3.3}$$

The reason for this subtraction is that the purpose of this measure is to return the number of keys pressed down at the time input. Once a key has been released it is essential that the key is removed from the measure. For example, if a pair exists with down action time being at 1 second and up action time being 1.1 seconds. At time, 1.5 the equation will equal $1 - 1 = 0$. However, if the time put in is 1.05 then the equation will be $1 - 0 = 1$ which indicates that one key was being held down at this particular time.

For every time input, each pair is checked with the sum of all the results stored in a dictionary along with the time input as the key. It's this dictionary that forms the KD signal and is used in further steps.

3.2.1 Heaviside Step Function

This is the bottom layer of the KD signal algorithm. It is a modified version of the Heaviside Step function.

$$h(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 > x_2 \\ 0.5 & \text{if } x_1 == x_2 \\ 0 & \text{if } x_1 < x_2 \end{cases} \tag{3.4}$$

The modification done is very simple, the only change is the addition of a third case which tests if the two times are equal to one another. Due to the nature of the use case for my project, there is a relatively high chance that the two times are equal to one another. In this case this means that the user at this time is currently in the process of performing that action whether that be pressing or releasing the key. The

3.2.2 Output

The resulting signal can be shown easily in graph format. fig 1.1 is the KD signal produced by a genuine user whilst fig 1.2 is the signal produced by an imposter user typing the same word. IMAGE GOES HERE

3.3 Dynamic Time Warping

3.3.1 Path

3.3.2 Cost Matrix

3.4 Validation Measures

- Talk about selecting values
- What effect does semantics have?

- Weighting of Euclidean vs correlation
- Chosen auth method
- In all cases what happens?
  - If not seen word before
  - If all validated
  - If all bar one are validated
  - etc

3.4.1 Euclidean Distance

3.4.2 Correlation Coefficient

3.4.3 Semantics??

3.5 Training

- first x vs ded training

3.6 Update

- Update everything

3.7 Storage

- Compression - file sizes too large
- Keyboard storage info

3.8 Pausing

- Uses auth method
- why? Sensitive info
- Implementation processes??

Chapter 4

Results and Discussion

- Test Results
- Calc and use FP, FN, TP, TN - get a percentage
- Discuss in relation to validation measure
- Mention struggling with small words maybe???
- Speed, security??

Chapter 5

Critical Appraisal

1. Summary and crit analysis
    - System works very well - provide examples using test data??
    - System is lightweight and secure
    - Compared to og planned, system is more complicated
    - Struggles with smaller words - less data points
    - NEED TO COME BACK TO THIS, NOT DETAILED AT ALL
2. Impact
    - Benefits
        - better security in combo with other sec methods
        - Lightweight and users won't notice
        - Doesn't spy on people due to only storing KDS and can turn off when user is doing something sensitive
        - Can be adapted to be used in the real word easily
    - Risks
        - Greater surveillance
        - Could easily be adapted maliciously - key logger
        - NEED MORE
3. Personal Development
    - Maths, maths, maths
    - Further git knowledge???
    - Exp Project Development
    - MORE

Chapter 6

Conclusion

Bibliography

[1] Ramin Toosi and Mohammad Ali Akhaee. Time–frequency analysis of keystroke dynamics for user authentication. *Future generation computer systems*, 115:438–447, 2021.