MOBILE KEYSTROKE DYNAMICS: ASSESSMENT AND IMPLEMENTATION

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

By

Shea Allison Ryan

December 2014
The thesis of Shea Allison Ryan is approved:

________________________________________  ______________________________
John Noga, Ph.D                          Date

________________________________________  ______________________________
Adam Kaplan, Ph.D                          Date

________________________________________  ______________________________
Jeffrey Wiegley, Ph.D, Chair              Date

California State University, Northridge
Dedication

To my parents.
Acknowledgements

Thank you to my chair, Dr. Jeffrey Wiegley, for helping me at every step of my learning and for his invaluable life advice. Thank you Dr. John Noga for teaching me how to think like a programmer, and Dr. Adam Kaplan for filling in the gaps. Tim Martin, you were the inspiration for this paper. Thank you to Toby Messinger for all the competition. You beat me here. And finally, I will never be able to repay Guillaume Goussard for all the help and encouragement he has given me.
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature page</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Listings</td>
<td>ix</td>
</tr>
<tr>
<td>Abstract</td>
<td>x</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Objectives</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Exploration of Keystroke Dynamics on Mobile Devices</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Mobile Application</td>
<td>3</td>
</tr>
<tr>
<td>3 Target Audience</td>
<td>4</td>
</tr>
<tr>
<td>4 Keystroke Dynamics Algorithms</td>
<td>5</td>
</tr>
<tr>
<td>4.1 A Survey of Classification Algorithms</td>
<td>6</td>
</tr>
<tr>
<td>4.1.1 Euclidean Distance</td>
<td>6</td>
</tr>
<tr>
<td>4.1.2 Manhattan Distance</td>
<td>7</td>
</tr>
<tr>
<td>4.1.3 Mahalanobis Distance</td>
<td>8</td>
</tr>
<tr>
<td>4.1.4 Nearest Neighbor Mahalanobis Distance</td>
<td>9</td>
</tr>
<tr>
<td>4.1.5 Disorder Classifier</td>
<td>11</td>
</tr>
<tr>
<td>4.1.6 Neural Network</td>
<td>12</td>
</tr>
<tr>
<td>5 Some statistics</td>
<td>15</td>
</tr>
<tr>
<td>5.1 Variance</td>
<td>15</td>
</tr>
<tr>
<td>5.2 Covariance</td>
<td>15</td>
</tr>
<tr>
<td>5.3 Mahalanobis Distance</td>
<td>16</td>
</tr>
<tr>
<td>6 Platform Selection</td>
<td>19</td>
</tr>
<tr>
<td>7 Data Storage</td>
<td>20</td>
</tr>
<tr>
<td>7.1 What to store</td>
<td>20</td>
</tr>
<tr>
<td>7.2 Android storage capacity</td>
<td>20</td>
</tr>
<tr>
<td>8 Database Design</td>
<td>22</td>
</tr>
<tr>
<td>9 Requirements</td>
<td>24</td>
</tr>
<tr>
<td>9.1 Functional Requirements</td>
<td>24</td>
</tr>
<tr>
<td>9.2 Non-functional Requirements</td>
<td>25</td>
</tr>
<tr>
<td>10 Research and Development</td>
<td>26</td>
</tr>
<tr>
<td>10.1 Web-based Prototype</td>
<td>26</td>
</tr>
<tr>
<td>10.1.1 Technological choices</td>
<td>26</td>
</tr>
<tr>
<td>10.1.2 High level design</td>
<td>26</td>
</tr>
<tr>
<td>10.1.3 Timing vector computation</td>
<td>28</td>
</tr>
<tr>
<td>10.1.4 Results of the prototyping phase</td>
<td>28</td>
</tr>
<tr>
<td>11 Clacker Android Implementation</td>
<td>33</td>
</tr>
<tr>
<td>11.1 Software keyboard</td>
<td>33</td>
</tr>
<tr>
<td>11.2 Android Implementation</td>
<td>34</td>
</tr>
<tr>
<td>11.2.1 Hacker’s Keyboard</td>
<td>34</td>
</tr>
</tbody>
</table>
11.2.2 Libraries and Frameworks ........................................ 35
11.2.3 Implementation ................................................... 35
11.3 Results .................................................................... 36
12  Areas for Future Research  ........................................... 38
   12.1 Swype dynamics .................................................... 38
   12.2 Fingertip pressure .................................................. 38
   12.3 Fingertip shape and orientation .............................. 38
13  Conclusions ............................................................... 39
References .................................................................... 41
List of Tables

4.1 The average equal-error rates from the evaluation of the 14 detectors are ranked from best to worst (with standard deviations in parentheses) [1, Table 2]. .............................................. 12

4.2 The average zero-miss false-alarm rates from the evaluation of the 14 detectors are ranked from best to worst (with standard deviations in parentheses) [1, Table 2]. .................................................. 13

4.3 Experimental Results in User Authentication for Different Sizes of Users’ Profiles [2, p. 333] ................................................................. 13

5.1 Sample of student grades across three subjects. ............................... 15

10.1 The equal-error rate and zero-miss false-alarm rate of Nearest Neighbor (Mahalanobis) for Desktop and Mobile using different algorithms ...... 31
List of Figures

4.1 The blue line of the graph represents the FRR, the green line represents the FAR and the intersection of the two line is the EER. 6
4.2 Manhattan Distance vs Euclidean Distance. The green path represents the Euclidean distance between the two points, while the red, blue, and yellow paths are all representations of the Manhattan distance between the points. Note that the three Manhattan distance paths all have the same total length [3]. 8
4.3 The point squared in red does not fall within the true distribution of the data. A standard univariate tests like the Euclidean distance or the Manhattan distance might accept this point since, ignoring direction, it is within the “sphere” of data points surrounding the mean vector. The Mahalanobis classifier would correctly reject this point because it does not lie within the ellipsoid of set members [4]. 9
4.4 Finding the three nearest neighbors for an unclassified sample. The colored dots represent known objects of each class. A circle of increasing radius is drawn around point p until three already-classified points fall within the area of the circle around p. The circle’s radius represents the distance to the third-nearest neighbor. This point is quite close to its neighbors. 10
4.5 This point is farther from its neighbors than the point in 4.4, but still classifiable using kNN. 10
5.1 Covariance Matrix from the sample of student grades 5.1 17
5.2 Covariance Inverse Matrix from the sample of student grades 5.1 17
5.3 Sample of student grades across three subjects. 17
5.4 Difference between Lisa’s grades and mean grades as a column vector 18
5.5 Mahalanobis distance between Lisa’s grade and the distribution of existing grades 18
8.1 SQLite Database Schema 22
10.1 Perceived changes in distance between the Euclidean, Manhattan, and Mahalanobis distance-based classifiers. The spikes on the right arose when an impostor attempted to type the word. 29
10.2 Visualizing the difference between a mean timing vector and an unclassified sample. The yellow line is from an impostor. 30
11.1 The life cycle of an IME (Input Method Editor) 33
List of Listings

10.1 Keystroke event capture ................................................. 27
10.2 Mahalanobis distance computation ................................... 27
11.1 IME Service declaration .................................................. 34
ABSTRACT

MOBILE KEYSTROKE DYNAMICS: ASSESSMENT AND IMPLEMENTATION

By

Shea Allison Ryan

Master of Science in Computer Science

The majority of Americans now own a smartphone. We keep our most personal data in pockets and purses, seldom prepared for the pain and distress that loss of our phones can cause. For those with high-security data on their phone, theft of the device may prove catastrophic. Although lock screens can help prevent unauthorized access, they cannot detect unauthorized device usage. Additionally, an attacker who has learned the passcode/pattern can use the device at her discretion.

This paper explores the feasibility of increasing mobile security through the application of keystroke dynamics. Keystroke dynamics is a biometric based on typing. Typists develop individualized rhythms and patterns that can be used to distinguish the authentic user from an impostor. Traditionally, keystroke dynamics has been applied in high-security applications where users are typing on a full-sized physical keyboard. With the increasing prevalence of smartphones, the application of keystroke dynamics to the mobile domain could prove a powerful weapon against mobile data theft. To this note, I have explored the mobile application and accuracy of several well-known keystroke dynamics classifiers and developed an Android Input Method that implements typing pattern recognition using the best of these, the Nearest Neighbor Mahalanobis Distance classifier.
Keystroke dynamics, also called keystroke biometrics or typing dynamics, is a biometric based on typing style. Typists have individualized typing patterns that can be analyzed to verify the authenticity of the user. Like other biometrics, keystroke dynamics is not error-proof.

The origins of keystroke dynamics date back over 150 years to the invention of the telegraph. Telegraphists learned to identify one another by the “fist” or rhythm of Morse code. Since telegraph keystrokes are sent as they are entered, the receiving party can hear the rhythm with which the message was keyed. Experienced telegraphists could learn to identify the sender of a message by the rhythm of dots and dashes that were heard. This was a useful ability especially for military applications, where telegraph operators could spy on enemy transmissions. Even without being able to decipher the message, experienced telegraphists could identify the author. In this way, troop and ship movements could be tracked.

However, text today is not typically sent in key-by-key like the Morse code in a telegraph would be, so it is not typically possible to biometrically determine the sender of a message. It is difficult to take keystroke timing data unless it is collected at the time of key entry. Therefore, the system where keystrokes are being entered must be responsible for data collection/interception.

Today, keystroke dynamics is most often applied in situations where the authenticity of a user must be ascertained with extreme confidence. For instance, it may be used as an extra layer of protection for a password-protected application. If a user’s password is compromised, and the keystroke dynamics of the real user are known, the application may be able to reject the impostor despite having received valid credentials. However, a sufficiently knowledgeable attacker may be able to emulate the rhythm of the authentic user and still gain entry.

Despite its potential for increasing security, the real-life application of keystroke dynamics is generally restricted to high-security applications. Although significant research has been done into keystroke dynamics, there are a number of factors that have prevented widespread adoption.

While several algorithms are known for classifying keystroke dynamics, they all share common features and pitfalls. All existing classification algorithms require that the user undergoes a training period, during which the algorithm learns about the way that a given person types. For each word or pattern that will be in the set of analyzable patterns, the user must train the system by typing the pattern upwards of 100 times [1]. Furthermore, it is beneficial to break up the training session into multiple smaller sessions, spaced somewhat apart, to allow for and simulate the natural variances in rhythm and accuracy that occur.
with changes in the user’s emotional state, stress level, time of day, medication schedule, etc. [5]. For a system that can recognize multiple patterns for a single user, the user must train the system for each pattern. After the training phase has concluded, most algorithms are unable to adapt to changes that may arise in the keystroke dynamics of a user when, for instance, the user becomes more familiar with the pattern. While the dynamics of a commonly-typed pattern (such as the) may not vary much over time, users who must enter complex passwords will likely become more adept at typing it and could potentially trigger frequent rejections from the system. At this point in time, the training process would have to be repeated.

In addition to the inconvenience of having to train the system, even highly regular typists may key in the pattern in an abnormal manner, resulting in a false rejection. A meta-analysis of the best-known classifier algorithms for this domain found that even the best performing algorithm still gave false-alarms and failed to detect impostors with an average equal-error rate of 0.096 [1]. Clearly, the level of user effort and potential for frustration makes this a difficult subject to apply to real life for all but the most important of applications.

Nonetheless, the promise of keystroke dynamics is alluring. Being able to authenticate or perhaps even identify a user strictly by his typing patterns, requiring no extra equipment, would be lucrative not just to high-security industries but to us as users living in a connected world. Password breaches are not uncommon and can impact millions of users, many of which could be using the same password across multiple applications. Two-thirds of Americans own a smart phone, and we trust to the flimsy security of a password or pin to keep us safe in the event that our device is stolen [6]. When the value of the data on mobile devices becomes high enough, preventing unauthorized access may be sufficiently prioritized to necessitate the implementation of keystroke dynamics. It is with this in mind that Clacker was developed. Clacker is an open-source input method available for Android devices that provides keystroke dynamics.
Chapter 2

Objectives

The objectives of this project are to explore the accuracy and application of keystroke dynamics in the mobile domain, and to implement a mobile application that provides improved security through mobile keystroke dynamics.

2.1 Exploration of Keystroke Dynamics on Mobile Devices

To achieve the first goal, I created a JavaScript-based web application to collect and analyze keystroke data using a variety of algorithms. This application can be used on a mobile web browser.

2.2 Mobile Application

To achieve the second, I developed Clacker, an open-source Android input device with built-in keystroke dynamics. Clacker unobtrusively provides an extra security layer by continuously monitoring the input device for patterns of unusual typing.
Chapter 3

Target Audience

The primary target audience for Clacker is people whose need for security outweighs the inconvenience of getting false alarms when their own typing patterns fluctuate. Although the security-minded individual is the most likely candidate for installation, another interesting use case is for preventing texting while in unsafe conditions such as driving a vehicle, while intoxicated, or any other situation that might create a danger to use a smartphone.

With sufficient privilege/permissions to lock the typist out, Clacker can effectively make it impossible to text when the user is not in good condition to use the device. This could help the many people who pledge not to text while driving keep themselves honest and safe.
Chapter 4
Keystroke Dynamics Algorithms

There are several algorithms that are useful for Keystroke Dynamics. All of the algorithms require an enrollment period during which samples are taken. These samples are all from the valid user. After the enrollment period is completed, new samples can be classified as “accepted” or “rejected”. An accepted sample is one that is similar to the enrollment samples, while a rejected sample is dissimilar to it. Similarity and dissimilarity are determined by checking if some property is above or below a threshold value.

There are four important criteria to consider when evaluating a biometric classifier:

1. False Rejection Rate (FRR): The probability that the authentic user is rejected. Confusingly, also referred to as False Alarm Rate.
2. False Acceptance Rate (FAR): The probability that an impostor is accepted. Also referred to as Miss Rate.
3. Equal Error Rate (EER): The rate at which the FRR is the same as the FAR. The lower this value is, the more accurate the classifier algorithm is.
4. Zero-Miss False-Alarm Rate (ZMFAR or ZMFRR): The rate of false rejections when no impostors are accepted. The lower this value is, the more accurate the classifier algorithm is.

It is most desirable that the FRR and the FAR both be as low as possible, resulting in a low EER. That is, it is best when the authentic user never triggers the alarm and an impostor always triggers the alarm. Unfortunately, it is difficult to satisfy this criteria. By lowering the sensitivity of the classifier, False Rejection Rates decrease. However, this causes the False Acceptance Rate to increase. Conversely, increasing the sensitivity of the classifier results in an increased False Rejection Rate. An algorithm with a low EER but a high Zero-Miss False Alarm rate will reject the authentic user more frequently than an algorithm with the same EER and a lower Zero-Miss False Alarm rate. Therefore, algorithms with low EER and low Zero-Miss False Alarm rates are ideal candidates. The relationship of the false rejection rate and the false acceptance rate with respect to the equal error rate is demonstrated in Figure 4.1.
4.1 A Survey of Classification Algorithms

4.1.1 Euclidean Distance

For this classic classifier, a cloud of points is constructed using the training vectors. If a timing vector has \( n \) features, then the graph will have \( n \) dimensions. The center of this cloud represents the mean vector. The squared Euclidean distance between the test vector and the mean vector is used for classification.

The Euclidean Distance is simple to compute, but has two major drawbacks. First, it is sensitive to feature and scale variations. Second, it cannot handle correlation between features [7]. The squared Euclidean distance is defined by the following formula, where \( p \) is the test vector and \( q \) is the mean vector.

\[
d^2(p, q) = (p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2
\]
Algorithm 1 Euclidean distance algorithm

1: procedure \textsc{EuclideanDistance}(a, b) \triangleright The Euclidean distance between 2 vectors a and b
2: \hspace{1em} squaredDistance $\leftarrow$ 0
3: \hspace{1em} dim $\leftarrow$ \textsc{Length}(a)
4: \hspace{1em} for $i \leftarrow 1, \text{dim}$ do
5: \hspace{2em} squaredDistance $\leftarrow$ squaredDistance + \textsc{POW}((a[i] - b[i]), 2)
6: \hspace{1em} end for
7: \hspace{1em} return squaredDistance
8: end procedure

4.1.2 Manhattan Distance

Another classic classifier, the Manhattan distance classifier is similar to the Euclidean distance classifier. Like with the Euclidean distance, the training vectors are assembled into a cloud of points. However, the Manhattan distance between the test vector and the mean vector is used for classification. The advantage of the Manhattan distance over the Euclidean distance is that outliers in the data are less influential [7].

\[ d_1(p, q) = \|p - q\|_1 = \sum_{i=1}^{n} |p_i - q_i| \]

Algorithm 2 Manhattan distance algorithm

1: procedure \textsc{ManhattanDistance}(a, b) \triangleright The Manhattan distance between 2 vectors a and b
2: \hspace{1em} manhattanDistance $\leftarrow$ 0
3: \hspace{1em} dim $\leftarrow$ \textsc{Length}(a)
4: \hspace{1em} for $i \leftarrow 1, \text{dim}$ do
5: \hspace{2em} manhattanDistance $\leftarrow$ manhattanDistance + \textsc{ABS}((a[i] - b[i]))
6: \hspace{1em} end for
7: \hspace{1em} return manhattanDistance
8: end procedure

To visualize the difference between the Euclidean distance and the Manhattan distance in two dimensions, consider 4.2.
4.1.3 Mahalanobis Distance

Another distance classifier, the Mahalanobis distance classifier resolves an important issue that the Euclidean distance and Manhattan distance classifiers encounter: neither can account for the direction of the test vector with regard to the mean vector. If all of the training vectors lie in a randomly-distributed cloud around the mean vector, then these two classifiers will successfully classify a test vector regardless of its direction relative to the mean (vectors in any direction are equally likely to be in the set). But what if the shape of the cloud is an ellipsoid? A test vector that is near to the border (yet still inside) of the “far” side of the ellipsoid may be rejected since the radius of the sphere modeled by these two algorithms will be shorter than the length from the mean vector to the far edge of the ellipsoid. Similarly, a test vector that falls just outside of the “short” side of the ellipsoid may be accepted since the radius of the sphere is greater than the distance between the mean vector and the test vector.

\[
((\bar{x} - \bar{y})'C^{-1}(\bar{x} - \bar{y}))^{\frac{1}{2}}
\]

This principle is illustrated in figure 4.3.
Figure 4.3: The point squared in red does not fall within the true distribution of the data. A standard univariate tests like the Euclidean distance or the Manhattan distance might accept this point since, ignoring direction, it is within the “sphere” of data points surrounding the mean vector. The Mahalanobis classifier would correctly reject this point because it does not lie within the ellipsoid of set members [4].

4.1.4 Nearest Neighbor Mahalanobis Distance

The Nearest Neighbor method is a form of supervised machine learning. It is used to deduce what class an unknown-class sample belongs to. Known members of a class are fed to the system as training data; for keystroke dynamics, this means that the training timing samples from the authentic user are used. In the traditional kNN problem, for a given point \( p \) we must find the nearest \( k \) neighbors. If the majority of the \( k \) neighbors are of one particular class, then \( p \) is considered a member of that class (an odd value of \( k \) can prevent a split decision). For keystroke dynamics, the problem is slightly different—we are not concerned with deciding which class a sample belongs to, but rather deciding if it belongs to the authentic user or not. Since only authentic samples are typically available, there is only one known class. When a mystery sample is received, we compute the distances between it and all the training vectors. The smallest of these distances is used as an anomaly score. If the anomaly score is above a threshold value, the sample is rejected; otherwise, it is accepted. To visualize the increase in distances in two dimensional space, consider Figures 4.4 and 4.5.
Figure 4.4: Finding the three nearest neighbors for an unclassified sample. The colored dots represent known objects of each class. A circle of increasing radius is drawn around point p until three already-classified points fall within the area of the circle around p. The circle’s radius represents the distance to the third-nearest neighbor. This point is quite close to its neighbors.

Figure 4.5: This point is farther from its neighbors than the point in 4.4, but still classifiable using kNN.
Unlike the Mahalanobis or Euclidean classifiers, for the Nearest Neighbor method it is crucial to normalize the timing data (when using Euclidean distance). I normalized the data onto a -1 to 1 scale when computing the difference. To normalize to a -1 to 1 scale, first find the range of values for the particular metric across all training samples for that word. Then divide the difference between the sample’s value for that metric and the training vector’s value by the range. For instance, for the 5-length timing vector for “the”, we may have a minimum “t” hold time of 60ms and a maximum of 90ms, so the range is 30ms. If our sample has a “t” hold time of 87ms and we’re computing the distance against the sample with a 60ms hold time, the normalized difference is \( \frac{87 - 60}{30} = 0.9 \). This is repeated for the remaining four features. Since all timing features are considered equally important, they are all normalized to the same -1 to 1 scale. However, when normalizing data where one feature is more important than the others, that feature should be appropriately weighted, for instance on a -10 to 10 scale. It is clear why the data must thus be normalized—timing features with a greater average value than other timing features would incorrectly skew the importance of these features. The need for normalization applies only when using Euclidean distance for the Nearest Neighbor calculation, not when using Mahalanobis distance.

I found that on a desktop keyboard, the anomaly rejection threshold was lower than on a mobile. This is almost certainly due to less regular typing patterns on mobiles causing higher variability in authentic samples. To set the threshold, I computed the Mahalanobis distance between all points in the set to each other. The maximum of these values is a baseline threshold.

4.1.5 Disorder Classifier

The disorder classifier is interesting because of its usefulness on free text and its dissimilarity to the statistical classifiers discussed so far.

The disorder classifier works on \( n \)-graphs (i.e. digraphs, trigraphs) rather than discrete words. Samples of free or transcribed text are used for training. From the samples, the most common \( n \)-graphs are obtained. Typically, there will be few \( n \)-graphs longer than 4 characters [5]. The \( n \)-graphs are sorted by timing features. For instance, if the digraphs “en”, “th”, and “er” are all well-represented in the sample, and we take the inter-key flight time as the metric, we might find that the “th” has the shortest flight time, followed by “en” and then “er” ([“th”, “en”, “er”]). As the user types, the \( n \)-graphs in the sample are sorted under the same principle. Using \( n \)-graphs that occur in both the training text and the sample, the “disorder” of the sample is computed. Essentially, we tally up the total distance of “swaps” that would be required to put all of a sample’s \( n \)-graphs into their rightful places in the sorted training array. If our sample was ordered [“th”, “er”, “en”], we would have to make two swaps (“th” moves 0, “er” moves 1, “en” moves 1). If the total number of moves required is sufficiently below the total number of possible moves, then the sample is accepted. This method can also work on single key hold times under the same principle [5].
4.1.6 Neural Network

The Neural network approach uses machine learning to perform pattern recognition (classification). Neural networks are based on a natural biological phenomenon—they emulate the neurons in the brain. Connections between neurons in the brain grow stronger or weaker over time as the brain connects information. In a neural network, there are many interconnected nodes with weights. Similarly to the brain, neural networks adapt to new information by changing weights between nodes in the system. For keystroke dynamics, the objective of the neural network is to classify a test vector. This is achieved by feeding the training data to the system and adapting the weights between the nodes in order to minimize error rates. Neural networks have shown great promise for keystroke dynamics, but the data is conflicting. Killourhy and Maxion found that Neural networks were outperformed by simpler classifiers (including Euclidean distance), while Crawford believes that Neural networks perform better than the statistical classifiers [1, 8].

<table>
<thead>
<tr>
<th>Detector</th>
<th>equal-error rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Manhattan (scaled)</td>
<td>0.096 (0.069)</td>
</tr>
<tr>
<td>2 Nearest Neighbor (Mahalanobis)</td>
<td>0.100 (0.064)</td>
</tr>
<tr>
<td>3 Outlier Count (z-score)</td>
<td>0.102 (0.077)</td>
</tr>
<tr>
<td>4 SVM (one-class)</td>
<td>0.102 (0.065)</td>
</tr>
<tr>
<td>5 Mahalanobis</td>
<td>0.110 (0.065)</td>
</tr>
<tr>
<td>6 Mahalanobis (normed)</td>
<td>0.110 (0.065)</td>
</tr>
<tr>
<td>7 Manhattan (filter)</td>
<td>0.136 (0.083)</td>
</tr>
<tr>
<td>8 Manhattan</td>
<td>0.153 (0.092)</td>
</tr>
<tr>
<td>9 Neural Network (auto-assoc)</td>
<td>0.161 (0.080)</td>
</tr>
<tr>
<td>10 Euclidean</td>
<td>0.171 (0.095)</td>
</tr>
<tr>
<td>11 Euclidean (normed)</td>
<td>0.215 (0.119)</td>
</tr>
<tr>
<td>12 Fuzzy Logic</td>
<td>0.221 (0.105)</td>
</tr>
<tr>
<td>13 k Means</td>
<td>0.372 (0.139)</td>
</tr>
<tr>
<td>14 Neural Network (standard)</td>
<td>0.828 (0.148)</td>
</tr>
</tbody>
</table>

Table 4.1: The average equal-error rates from the evaluation of the 14 detectors are ranked from best to worst (with standard deviations in parentheses) [1, Table 2].
Table 4.2: The average zero-miss false-alarm rates from the evaluation of the 14 detectors are ranked from best to worst (with standard deviations in parentheses) [1, Table 2].

<table>
<thead>
<tr>
<th>Detector</th>
<th>zero-miss false-alarm rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Nearest Neighbor (Mahalanobis)</td>
<td>0.468 (0.272)</td>
</tr>
<tr>
<td>2 Mahalanobis</td>
<td>0.482 (0.273)</td>
</tr>
<tr>
<td>3 Mahalanobis (normed)</td>
<td>0.482 (0.273)</td>
</tr>
<tr>
<td>4 SVM (one-class)</td>
<td>0.504 (0.316)</td>
</tr>
<tr>
<td>5 Manhattan (scaled)</td>
<td>0.601 (0.337)</td>
</tr>
<tr>
<td>6 Manhattan (filter)</td>
<td>0.757 (0.282)</td>
</tr>
<tr>
<td>7 Outlier Count (z-score)</td>
<td>0.782 (0.306)</td>
</tr>
<tr>
<td>8 Manhattan</td>
<td>0.843 (0.242)</td>
</tr>
<tr>
<td>9 Neural Network (auto-assoc)</td>
<td>0.859 (0.220)</td>
</tr>
<tr>
<td>10 Euclidean</td>
<td>0.875 (0.200)</td>
</tr>
<tr>
<td>11 Euclidean (normed)</td>
<td>0.911 (0.148)</td>
</tr>
<tr>
<td>12 Fuzzy Logic</td>
<td>0.935 (0.108)</td>
</tr>
<tr>
<td>13 k Means</td>
<td>0.989 (0.040)</td>
</tr>
<tr>
<td>14 Neural Network (standard)</td>
<td>1.000 (0.000)</td>
</tr>
</tbody>
</table>

Up to a limit, the more samples collected during the training phase, the better the classifier algorithm will perform. Anomalous features in a single sample in the training set are rounded out by the abundance of other samples. 4.3 demonstrates this effect for a Disorder Classifier on free text. Every doubling of the number of samples reduces the FAR and FRR rates by approximately 40%.

Table 4.3: Experimental Results in User Authentication for Different Sizes of Users’ Profiles [2, p. 333]

<table>
<thead>
<tr>
<th>Nbr of samples in user’s profile</th>
<th>d = R_{2,3,4} + A_2</th>
<th>d = R_{2,3,4} + A_{2,(3,4)}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FAR (%)</td>
<td>FRR (%)</td>
</tr>
<tr>
<td>2</td>
<td>0.2622</td>
<td>20.0788</td>
</tr>
<tr>
<td>4</td>
<td>0.09479</td>
<td>10.0119</td>
</tr>
<tr>
<td>6</td>
<td>0.05822</td>
<td>6.8703</td>
</tr>
<tr>
<td>8</td>
<td>0.04303</td>
<td>5.2985</td>
</tr>
<tr>
<td>10</td>
<td>0.03477</td>
<td>4.4097</td>
</tr>
<tr>
<td>12</td>
<td>0.029</td>
<td>3.8498</td>
</tr>
<tr>
<td>14</td>
<td>0.0276</td>
<td>3.1667</td>
</tr>
</tbody>
</table>

A point of interest is that the majority of studies on Keystroke Dynamics required test subjects to copy text instead of write freely. The reason for this is that many test subjects encounter a sort of “writer’s block” when asked to write whatever comes to mind, and they simply cannot find any words to write. Fortunately, another study found that, for at least two classification algorithms, there were minimal overall differences in the ability to
classify timing data for free vs. transcribed text [5]. Although the authors point out that this is no guarantee that all classification algorithms will be immune to the effects of free vs. transcribed text, it provides a high level of assurance that findings in other papers on this subject will be valid for the purposes of this application.
Chapter 5

Some statistics

5.1 Variance

Variance is defined as how spread out values are in a set. If all the values are the same, the variance will be zero. If values all fall closely near a mean, the variance will be small. If values are widely distributed, the variance will be high. Variance is never negative since it is only a measure of spread not the values themselves.

\[ \sigma^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{X})^2}{n} \]

5.2 Covariance

Covariance measures how two random variables change together. As an example, consider again Figure 5.1. It appears that physics grades are correlated to math grades, while it’s more difficult to judge how Art History grades relate to Math grades. To better understand how grades in these different subjects relate to one another as a whole, the covariance can be used. A positive covariance will indicate that as one grade increases, so does another, while a negative covariance will indicate that as one grade increases, the other grade decreases. A covariance near zero could indicate that grades are unrelated between subjects. However, as will be shown, it is not necessarily true that two variables with low covariance are independent.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Student Grades</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tony</td>
</tr>
<tr>
<td>Physics</td>
<td>82</td>
</tr>
<tr>
<td>Math</td>
<td>90</td>
</tr>
<tr>
<td>Art History</td>
<td>90</td>
</tr>
</tbody>
</table>

Table 5.1: Sample of student grades across three subjects.

\[ \text{cov}(X, Y) = \frac{\sum_{i=1}^{n} (x_i - \bar{X})(y_i - \bar{Y})}{n} \]

First, find the Expected value (mean grade) for each subject.

\[ E[Physics] = \frac{82 + 50 + 90 + 40 + 85}{5} = 69.4 \]
\[ E[\text{Math}] = \frac{90 + 60 + 85 + 35 + 85}{5} = 71 \]

\[ E[\text{ArtHistory}] = \frac{90 + 95 + 70 + 80 + 40}{5} = 75 \]

Now, find the covariance between Physics and Math and between Physics and Art History.

\[ \text{cov}(P, M) = \sigma(P, M) = \]

\[ \frac{(82-69.4)(90-71)+(50-69.4)(60-71)+(90-69.4)(85-71)+(40-69.4)(85-71)}{5} = 403.6 \]

\[ \text{cov}(P, A) = \sigma(P, A) = \]

\[ \frac{(82-69.4)(90-75)+(50-69.4)(95-75)+(90-69.4)(70-75)+(40-69.4)(80-75)+(85-69.4)(40-75)}{5} = -199 \]

These results support the previous assumption that Physics and Math scores are positively correlated, and indicate that Physics and Art History grades are generally inversely correlated. The higher absolute value of the Physics/Math Covariance could indicate that the relationship is stronger than that between Physics and Art History, but the covariance alone is not sufficient evidence to prove this conjecture.

### 5.3 Mahalanobis Distance

The Mahalanobis distance is a measure of how far a point is from a distribution, with consideration taken for direction from the mean of the distribution. The Mahalanobis accounts for differences in the variance between directions. The variance in Y may be lower than the variance in X, for instance, so points that are far from the mean with respect to Y will be further than points that are far from the mean with respect to X. Unlike the covariance, which operates on only two random variables, the Mahalanobis distance can operate over any number of random variables. The Mahalanobis distance can tell us, for instance, how similar a new student’s grades are to the existing students’ grades. If our new student has a high Physics grade, low Math grade, and high Art History grade, we can expect that the Mahalanobis distance will be greater than if the student had a high Physics grade, high Math grade, and low Art History grade. The lower the Mahalanobis distance is, the more similar the test scores are to the existing scores.

To compute the Mahalanobis distance, we must compute the Covariance Matrix. In order to compute the covariance matrix for \( n \) random variables, we require at minimum \( n \)
samples. For the grade data, we have three subjects (Math, Physics, Art History), so we require at least three students’ grades in each subject. Since we have five students’ grades, we will use all five. The more samples we have, the more clear the relationship between the grades becomes.

\[
\text{cov}(X) = \begin{bmatrix}
\text{var}(X_1) & \text{cov}(X_1, X_2) & \cdots & \text{cov}(X_1, X_n) \\
\text{cov}(X_2, X_1) & \text{var}(X_2) & \cdots & \text{cov}(X_2, X_n) \\
\vdots & \vdots & \ddots & \vdots \\
\text{cov}(X_n, X_1) & \text{cov}(X_n, X_2) & \cdots & \text{var}(X_n)
\end{bmatrix}
\]

For brevity’s sake, only the resultant matrix is shown. Numbers have been rounded to the nearest hundredth.

\[
\text{cov}(\text{Subject grades}) = \begin{bmatrix}
413.44 & 403.60 & -199.00 \\
403.60 & 434.00 & -135.00 \\
-199.00 & -135.00 & 380.00
\end{bmatrix}
\]

Figure 5.1: Covariance Matrix from the sample of student grades 5.1

Next, we take the inverse of this matrix.

\[
\text{cov}^{-1}(\text{Subject grades}) = \begin{bmatrix}
0.04515 & -0.03894 & 0.00981 \\
-0.03894 & 0.03617 & -0.00754 \\
0.00981 & -0.00754 & 0.00509
\end{bmatrix}
\]

Figure 5.2: Covariance Inverse Matrix from the sample of student grades 5.1

Now, we compute the Mahalanobis distance between a new student’s three grades and the other students’ grades.

First, the new student’s grades.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Student Grades</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lisa</td>
<td></td>
</tr>
<tr>
<td>Physics</td>
<td>95</td>
</tr>
<tr>
<td>Math</td>
<td>86</td>
</tr>
<tr>
<td>Art History</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 5.3: Sample of student grades across three subjects.
The mean grades from the other students were previously computed for each subject.

We compute the difference between Lisa’s grades and the mean grades as a columnar vector.

\[
differenceCol = \begin{bmatrix} 95 - 69.4 \\ 86 - 71 \\ 63 - 75 \end{bmatrix} = \begin{bmatrix} 25.6 \\ 15 \\ -12 \end{bmatrix}
\]

Figure 5.4: Difference between Lisa’s grades and mean grades as a column vector

The final step is to compute the distance. We multiply the difference vector by the covariance matrix inverse by the difference vector transpose. The square root of the sole item in the matrix is the Mahalanobis distance.

\[
\text{Mahalanobis Distance} = \sqrt{\begin{bmatrix} 25.6 \\ 15 \\ -12 \end{bmatrix} \begin{bmatrix} 0.04515 & -0.03894 & 0.00981 \\ -0.03894 & 0.03617 & -0.00754 \\ 0.00981 & -0.00754 & 0.00509 \end{bmatrix} \begin{bmatrix} 25.6 \\ 15 \\ -12 \end{bmatrix}}
\]

\[
\text{Mahalanobis Distance} = \sqrt{5.24433}
\]

Figure 5.5: Mahalanobis distance between Lisa’s grade and the distribution of existing grades

While this number alone tells essentially nothing about Lisa’s grades in particular, it has been an illustrative example of the computation effort required to find the Mahalanobis distance. To fully classify Lisa’s grades as “normal” or “abnormal”, this distance must be compared to a threshold value. If we use the critical chi-squared ($\chi^2$) value for three degrees of freedom at a critical alpha of 0.001, which is 16.27, then it is safe to say that Lisa’s grades are similar to the other students [9].
The Android platform was selected over competing mobile operating systems for several reasons. First, Android is open source and the developer community actively contributes to the development of features on Android. This makes it easier to find free resources to use and reference for development than if the platform were closed-source. With the availability of these resources, I was able to gain a solid understanding of how input devices interact with the event system and thereby capture timing data. Second, at the time of writing, only Android allows developers to implement their own keyboards outside of the native input device.

Other platform choices are iOS and Windows Phone. The combination of a closed-source OS and an inability to gain access to the input device make these both unlikely candidates for a development platform for this type of application. With Android, however, there are 100% open source keyboards available to download. This allowed me to use an existing keyboard, “Hacker’s Keyboard”, as a starting point for adding biometric features. Hacker’s Keyboard uses the popular Apache License 2.0, which is approved by the Open Source Initiative [10, 11, 12]. One disadvantage to using a third party input device, as opposed to the native input device, is that users must manually set the input method, which prompts them with a warning that the input method may log keystrokes. This presents a barrier to entry but is effectively the only route that can be taken.
Chapter 7
Data Storage

The Nearest Neighbor Mahalanobis Distance (NNMD) algorithm requires that training samples be saved. The sample data is aggregated during initial usage of the application and will grow until enough samples are collected to adequately compute the covariance matrix.

7.1 What to store

To be able to perform correctly, the program needs to store data in a persistent manner. Some data comes with the application itself, such as the dictionary of words. Other data is collected when the application runs.

Words
Every word that will be in the dictionary must be stored.

Training Timing Vectors
For each word, many training samples must be saved to discover how the user typically types that word.

Covariance Matrix Inverse
Once enough samples are collected for a particular word, we compute the covariance matrix inverse exactly once and store it for future use.

Threshold
The acceptance threshold for each word is unique. Once enough samples have been collected, the threshold for acceptance must be computed to minimize the number of false alarms and maximize the number of true alarms.

7.2 Android storage capacity

The Android system provides multiple methods for storing data

Shared Preferences
Store private primitive data in key-value pairs. This option does not match our requirement since the data should be used in relation with each other. This storage mechanism is intended to be used to store the user’s preferences for a given application.

Internal Storage
Store private data on the device memory. This allows the Android application to save any data in a file on the Android filesystem under the application’s exclusive usage. This choice would allow the application to define its own data structure.
External Storage
Store public data on the shared external storage. The use of the external device is recommended when the data to be stored are intended to be used other applications: for example, a downloader application will store the retrieved files with this mechanism so that a PDF file could be opened later using a PDF reader.

Network Connection
Store data on the web with your own network server. I cannot use this option because the Clacker application is intended to be standalone and not require a network connection per battery-preservation requirements.

SQLite Databases
Store structured data in a private database using the SQL language to retrieve the data into a usable format. Built-in to Android OS.

After analysis of the different options listed above, and pondering their advantages and constraints, the best choice for the persistent data storage appeared to be the native SQLite Database. In following the Android developer guidelines, the database Java class implements BaseColumns to enable the tables to have an “ID” column that works well with some database adapters [13].
Chapter 8  
Database Design

Since the dictionary and keystroke data must be preserved between phone reboots, this native database was the natural choice for data storage.

ER Diagram: keystroke_dynamics

<table>
<thead>
<tr>
<th>timing_vectors</th>
<th>dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: INT</td>
<td>id: INT</td>
</tr>
<tr>
<td>dictionary_id: INT</td>
<td>word: VARCHAR</td>
</tr>
<tr>
<td>timing_vector: VARCHAR</td>
<td>covariance_matrix_inverse: BLOB</td>
</tr>
<tr>
<td>mahalanobis_distance: FLOAT</td>
<td></td>
</tr>
<tr>
<td>create_time: TIMESTAMP</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.1: SQLite Database Schema

The database schema was designed to optimize code readability, minimize space requirements, and allow for simple queries. Although the dataset is small and the schema is simple, performance is still an important factor in the database design since anytime the keyboard is in use the database could potentially be queried.

SQLite will cache the results of database calls when it is possible, making repeated queries fast.

More storage is needed when using Nearest Neighbor Mahalanobis Distance than when using the Mahalanobis Distance because, with Nearest Neighbor, the Mahalanobis distance of each training vector must be saved, in comparison to the Mahalanobis classifier where only the timing vectors themselves must be saved. One optimization that could be made and
would improve both the memory footprint and computation speed would be to go through
the sample vectors after the Mahalanobis distance has been computed and discard vectors
that are essentially identical. This would reduce the set of data that must be evaluated as
potential neighbors when a new test vector is being classified.

For each word in the dictionary, the covariance matrix inverse is computed after enough
samples have been collected for that word. This matrix is computed exactly once per word
and stored in the same table as the word for ease of lookup. The covariance matrix inverse is
serialized and stored as a BLOB. This solution requires deserializing the data, but provides
a clean and efficient mechanism for storage and retrieval.

I chose to limit words in the dictionary to a maximum of 20 characters. The 100 most
common words in English are all under 20 characters (in fact, the longest of these words
are only 6 characters [14]). A user using the included dictionary will not need storage for
long words since all words are chosen out of these 100 most common words. The longest
commonly used word in English is “uncharacteristically”, which is 20 characters long
[15].
Chapter 9

Requirements

9.1 Functional Requirements

It shall use an algorithm with a false rejection rate below 10%
The authentic user should not have to be bothered by the alarm erroneously triggering. However, the unpredictability of mobile typing patterns makes it unlikely that any algorithm will perform as well on a mobile device as on a physical keyboard. Therefore the algorithm used should be proven to have a low false rejection rate on a physical keyboard.

It shall use an algorithm with a false acceptance rate below 10%
Impostors should be detected with a high accuracy or else the system is useless. However, the algorithm is expected perform worse on a mobile device than on a physical keyboard due to greater variability in typing on mobile devices. The algorithm should have a false acceptance rate of no more than 10% when applied to a physical keyboard.

It shall have a built-in dictionary of words to analyze
The analysis is performed on a set of key words that are recognized by the system. These words are selected from the list of the 100 most common English words to allow the user a fast start. The user should not have to build her own dictionary in order to use the application. The words shall be common American English variants.

It shall not allow default dictionary words that are substrings of each other
No word in the default dictionary can contain another word in full or else the larger of the two words will be undetectable when the shorter word is detected.

It shall run on Android versions ≥ 4.0 (Ice Cream Sandwich)
The Android versions 4.0 and above represent more than 50% of the current market share [16].

It shall lock the phone after three sequentials alarms have been triggered
The alarm may be triggered several times without apparent consequence. Once this threshold is reached, the phone should lock if possible. It is unlikely that the user will not have a lock screen since they are willing to install a program whose sole purpose is to detect unauthorized use of the device.
9.2 Non-functional Requirements

It shall be fast
The maximum acceptable latency between an action on the touchscreen and the result must be less than 72 ms [17]. This duration must take into account the system processing necessary to perform the required action.

It shall not reveal passwords
Because the system can intercept key strokes, it can passively capture passwords typed by the user. Unnecessary data shall not be logged in any way.

It shall take up minimal space
Design considerations should be made to minimize the memory footprint of the application.

It shall not drain the battery
Even if the keyboard is not active the service remains in background in order to display the keyboard when needed. During the background period the service must limit its activity to not use the battery.

It shall not create a memory leak
The keyboard activity can be opened at anytime and is running as a background service. The service shall not be a memory eater and shall its consumption at a reasonable level [18].

It shall not maliciously collect data
The application may not log keystrokes for any purpose other than keystroke dynamics. If keystrokes do not belong to a word in the dictionary, they must be discarded as soon as possible.
Chapter 10
Research and Development

10.1 Web-based Prototype

In order to prepare the development of the mobile Android application, a prototype was developed using JavaScript and capturing timing data in a web browser. The prototype consists of a simple web application that performs analysis of free text. This test application aims to test the algorithms and validate the flow of the application in an easier way than implementing an entire Java application that requires installation on a real device to be tested. Using an emulator is not helpful because the emulator keys are clicked with a mouse rather than touch events. Meanwhile, the browser can intercept keydown and keyup events which are generated even when using a soft keyboard.

10.1.1 Technological choices

The prototype was written using node.js. I chose node.js because it allows for rapid prototyping and development, and I am familiar with it. I used the Express framework to handle routing and controllers, and I used the Jade templating engine for creating views. All the code to perform keystroke dynamics runs in the front end for easy debugging and modification. Once the app was running, I simply opened a browser page and connected to my localhost to begin testing on my laptop.

Once I was satisfied with the implementation of the Mahalanobis distance classifier, I began testing on an actual mobile device. Using beta versions of Chrome for mobile (38.0.2125.102) and OSX (39.0.2171.36 beta), I was able to quickly connect my test device, a Samsung Galaxy SIII (SPH-L710 Build/KOT49H) running Android 4.4.2, to the node.js server running on my laptop. When the mobile device is connected via USB, Chrome can proxy requests from the phone’s browser to the server running on localhost, so there was no need to acquire real hosting or use a proxy. When connected via USB, it is also possible to debug the phone’s browser instance on the computer. Since client-side JavaScript can be updated in the node.js project without having to restart the node.js application, it is quick and easy to make adjustments to the code, especially in comparison to the build process required for native Android projects.

The usage of a web based application significantly accelerated the validation of the technical choices by imposing fewer constraints and limitations than if the implementation had been done directly on the targeted system. Done on Android natively, every change to the code would require a complete new build of the project and re-deploy to the target device.

10.1.2 High level design

The prototype works nearly exactly the same as the actual application. As the user types, timing metrics are obtained for all keystrokes by listening to the “keydown” and
“keyup” events available in the browser. As text is input, the working string is compared against all known words in the dictionary.

```javascript
$(
  '#input'
).on(
  'keydown',
  function (e) {
    keys.push(
      {
        letter: String.fromCharCode(e.keyCode),
        downtime: e.timeStamp,
        uptime: null
      }
    );
  });

$(
  '#input'
).on(
  'keyup',
  function (e) {
    var nearest = keys[findNearest(keys, String.fromCharCode(e.keyCode))];
  });
```

Listing 10.1: Keystroke event capture

If no words from the dictionary are a substring of the working string, then all metrics except for the the last n characters, where n is one less than the length of the longest word in the dictionary, are discarded. If the working string contains a member of the dictionary, the metrics for that word are extracted and a timing vector is generated. The timing vector is then used in one of two ways. If the covariance matrix for the word has not yet been established, the word is added into the sample set. If the covariance matrix has already been established, the Mahalanobis distance for the test vector is computed and the test vector is classified. HTML5 Local Storage as well as MongoDB were used to save the covariance matrix, timing samples, and training samples.

Once the training phase is over, when a new sample is received I compute the Euclidean, Manhattan, and Mahalanobis distance between the sample and the mean vector. In order to visualize the different classifier’s distance perceptions, I created real-time charts in the web application using the JavaScript graphing library “Flot”. With this I was able to track the perceived distances among all of the classic distance classifiers found for various typing samples 10.1, as well as compare a mean distance vector against a sample10.2.

```javascript
  covariance = JSON.parse(localStorage.getItem(theWord + "CovarianceMatrixInverse"));
  mean = JSON.parse(localStorage.getItem(theWord + "MeanVector"));
  threshold = JSON.parse(localStorage.getItem(theWord + "Threshold"));
  mahalanobisDistance = computeMahalanobisDistance(covariance, mean, timingVector);
  mDistanceArray = JSON.parse(localStorage.getItem(theWord + "Distances"));
  mDistanceArray.push(mahalanobisDistance);
  localStorage.setItem(theWord + "Distances", JSON.stringify(mDistanceArray));
  console.log("Mahalanobis distance", mahalanobisDistance);
```

Listing 10.2: Mahalanobis distance computation

This saves computation time by alleviating the need to re-compute the covariance matrix every time a new test vector is detected, and preserves the data when the page is refreshed or the server is restarted. A simple reset button clears all test data from local storage to allow for re-training.
10.1.3 Timing vector computation

While developing the prototype, a problem capturing the timing events in a classical keyboard context arose—users do not necessarily release keys in the order that they depress them. As an example, the sequence “t-h-e” may be keyed as t\_down, h\_down, t\_up, e\_down, e\_up, h\_up. Matching a keyup with the correct keydown required backtracking through the captured key events to find the most-recent “unpaired” keydown for the given keyup. This manifested itself as a particularly tricky problem when the last letter in the sequence was released before an earlier letter, leaving the key up timing data null for a previous letter and preventing computation of the covariance matrix or Mahalanobis distance. As a result, the algorithm for capturing the key events was adjusted to require that all expected keydown and keyup events had been captured before any calculations were made. This problem is much less likely to manifest in the mobile / soft keyboard context, as the majority of mobile phone users type with at most two fingers. While the problem did not occur in my testing whatsoever on the mobile, the algorithm for capturing timing data is more robust due to having been tested on a normal keyboard.

10.1.4 Results of the prototyping phase

From the data gathered with the eb application against classical and soft keyboard typing samples, it was apparent that a user’s typing characteristics change dramatically between environments. In a classical environment, users may begin typing a second or even third letter while one key is still being depressed, leading to a “negative” flight time (time between key release and key press) for a digraph. In the mobile setting, in particular for single-finger typists, this typing characteristic will never arise due to the physical limitation of typing with only one finger—only one key can be held at any given time.

Testing on a standard keyboard unfortunately provided too great of a degree of confidence in the algorithm. While the Mahalanobis distance was an acceptable classifier on standard keyboard, it proved essentially useless on a mobile device. Deliberate attempts to trigger the alarm often failed on the phone, while typing “normally” occasionally produced alarms. My deduction was that more training samples are required in the mobile environment than on a normal keyboard. Increasing the number of samples improved impostor detection, but detection capabilities still were not comparable to the results on a standard keyboard. It was at this point that I decided to try the Nearest Neighbor as my classifier.

I implemented several variants of Nearest Neighbor. To begin, I tried both the classical Euclidean distance-based Nearest Neighbor and Mahalanobis distance-based Nearest Neighbor. For the Euclidean classifier, I normalized the results onto a -1 to 1 scale. Results with the Nearest Neighbor Mahalanobis classifier were significantly improved over those with the Euclidean distance. Next, I experimented with normalizing the Mahalanobis distance. This proved unfruitful and in fact counter productive. However, I also tried reweighting the Euclidean distance by giving flight times 5x greater weight than hold times and vice versa. Weighting flight times provided a marked improvement over the standard Euclidean distance, while weighted hold times decreased performance. Weighting the normalized Mahalanobis distance in the same fashion did not provide an improvement in either
I implemented functionality to compare the EER and ZMFAR between two arbitrary users using each of the described Nearest Neighbor methods above.

In order to compute the error rates, I collected 100 training samples along with 100 authentic samples and 100 impostor samples for the word “there”. To collect the samples for this task, I set up a MongoDB database (‘clacker’) with a single collection (‘timing’). I selected MongoDB because it is quick to set up and stores data in a JSON-compatible format. The ability to upsert (update if a matching object exists, insert otherwise) makes adding and updating a single operation. Furthermore, since the entire web application is written in JavaScript and JSON was already the defacto format of the timing data, MongoDB made the job simple. The user can enter a name on the web application and, as timing metrics are recorded for a particular word, the database is filled with timing metrics for that user.

To compute the Equal Error Rate (ERR), I incremented the acceptance threshold from 0.01 by hundredths (0.01) and ran each of the Nearest Neighbor algorithms for each of the 100 authentic samples and each of the 100 impostor samples. When the threshold was at a point such that the number of rejected authentic samples matched the number of accepted impostor samples, the equal error rate could be found by dividing the number of rejected authentic samples by the total number of authentic samples.

For the Zero-Miss False Rejection Rate (ZMFRR), I employed a similar process. I used the same samples from the Equal Error Rate experiment. I decreased the threshold from 1000 by hundredths until none of the 100 impostor samples was accepted. The Zero-Miss False Rejection rate was thus the number of authentic samples that were rejected at this threshold divided by the total number of authentic samples.

Figure 10.1 shows how these rates varied between desktop and mobile usage.

![Figure 10.1: Perceived changes in distance between the Euclidean, Manhattan, and Mahalanobis distance-based classifiers. The spikes on the right arose when an impostor attempted to type the word.](image)
Figure 10.2: Visualizing the difference between a mean timing vector and an unclassified sample. The yellow line is from an impostor.
Table 10.1: The equal-error rate and zero-miss false-alarm rate of Nearest Neighbor (Mahalanobis) for Desktop and Mobile using different algorithms

<table>
<thead>
<tr>
<th>Detector</th>
<th>EER</th>
<th>ZMFRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desktop Nearest Neighbor (Normalized Euclidean)</td>
<td>42%</td>
<td>52%</td>
</tr>
<tr>
<td><strong>Desktop Nearest Neighbor (Mahalanobis)</strong></td>
<td>20%</td>
<td>43%</td>
</tr>
<tr>
<td>Desktop Nearest Neighbor (normalized Mahalanobis)</td>
<td>76%</td>
<td>87%</td>
</tr>
<tr>
<td>Desktop Nearest Neighbor (normalized hold time weighted Mahalanobis)</td>
<td>76%</td>
<td>93%</td>
</tr>
<tr>
<td>Desktop Nearest Neighbor (normalized flight time weighted Mahalanobis)</td>
<td>57%</td>
<td>60%</td>
</tr>
<tr>
<td>Mobile Nearest Neighbor (normalized Euclidean)</td>
<td>60%</td>
<td>97%</td>
</tr>
<tr>
<td>Mobile Nearest Neighbor (normalized hold time weighted Euclidean)</td>
<td>51%</td>
<td>99%</td>
</tr>
<tr>
<td>Mobile Nearest Neighbor (normalized flight time weighted Euclidean)</td>
<td>32%</td>
<td>97%</td>
</tr>
<tr>
<td><strong>Mobile Nearest Neighbor (Mahalanobis)</strong></td>
<td>22%</td>
<td>78%</td>
</tr>
<tr>
<td>Mobile Nearest Neighbor (normalized Mahalanobis)</td>
<td>76%</td>
<td>99%</td>
</tr>
<tr>
<td>Mobile Nearest Neighbor (normalized hold time weighted Mahalanobis)</td>
<td>52%</td>
<td>99%</td>
</tr>
<tr>
<td>Mobile Nearest Neighbor (normalized flight time weighted Mahalanobis)</td>
<td>52%</td>
<td>99%</td>
</tr>
<tr>
<td>Mobile vs desktop Euclidian</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>Mobile vs desktop (Mahalanobis)</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>Mobile vs desktop (normalized Mahalanobis)</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>Mobile vs desktop (normalized Mahalanobis) Weighted on hold time</td>
<td>1%</td>
<td>3%</td>
</tr>
<tr>
<td>Mobile vs desktop (normalized Mahalanobis) Weighted on flight time</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

This indicates that the Nearest Neighbor Mahalanobis classifier produces the most accurate result with an Equal Error Rate of 22%.

The normalization of the input data doesn’t show drastic changes in the output. However, if some dimensions are weighted, then the performance can improve. This is particularly true for the Euclidean distance. The way the weight is applied on the timing vectors modifies the results.

Euclidean classification is more accurate when the flight time is weighted more than
the hold time.

The success rate might be increased when a tolerance or ±1% of the Equal Error Rate is accepted.
11.1 Software keyboard

Since version 1.5 (Cupcake) of Android, the operating system has provided a class to create an Input Method Editor `android.inputmethodservice.InputMethodService` from which a final implementation can be derived to provide input capabilities.

![Diagram of the life cycle of an IME (Input Method Editor)](image)

Figure 11.1: The life cycle of an IME (Input Method Editor).

An input method editor (IME) is a user control that enables users to enter text [19]. An IME can be any source that can be used by the users to input data, such as on-screen
keyboard, speech-to-text service, or handwriting recognition software. The IME has to be selected by the user after being installed, then can be used across all applications that use a keyboard as source of unique input; only one IME can be enabled at a time.

The IME extends the android.inputmethodservice.InputMethodService class, and must be declared in the Android application AndroidManifest.xml file 11.1.

```xml
<!— Declares the input method service -->
<service android:name="LatinIME"
    android:label="@string/english_ime_name"
    android:permission="android.permission.
    BIND_INPUT_METHOD">
    <intent-filter>
        <action android:name="android.view.InputMethod" />
    </intent-filter>
    <meta-data android:name="android.view.im" android:resource="@xml/method" />
</service>
```

Listing 11.1: IME Service declaration

The Classes required to implement an IME can be found in the android.inputmethodservice and android.view.inputmethod, while the KeyEvent class from this last package is required to handle the keyboard characters.

The main part of the IME is the InputMethodService. The InputMethodService implements the service lifecycle, the callbacks to present the on-screen keyboard, and the reception and delivery of the user inputs to the application 11.1.

The other part of an IME is the graphic interface the user interacts with. On Android, an IME user interface has two sections:

**Input view**
The Input View is the key layout of the virtual keyboard (where the user types).

**Candidate View**
The Candidate View is the area where the IME can render corrections or suggestions.

### 11.2 Android Implementation

#### 11.2.1 Hacker’s Keyboard

In order to accelerate the development of Clacker, I used “Hacker’s Keyboard” [10] as a base to provide the graphic interface for the visual keyboard and also the internal wiring of the key stroke events.

“Hacker’s Keyboard” is an open source project that implements an alternative keyboard for Android. The application was released in December 2013 and has been installed more than 1,000,000 times. The APK (Android Application Package) size is 1.7Mbytes, and the
installed application uses less than 2.5Mbytes of the device storage. The minimum Android version that is required to use “Hacker’s Keyboard” is 2.2 (Froyo).

“Hacker’s Keyboard” meets all the requirements for the input method 9 and allowed me to focus on the core aspects of keystroke dynamics.

### 11.2.2 Libraries and Frameworks

#### 11.2.2.1 android native libraries

Android provides a rich application framework that allows developers to build innovative apps and games for mobile devices in a Java language environment. The Android API provides all the methods necessary to build the InputMethodService as well as the different views of the keyboard and the settings.

**android.inputmethodservice** These are base classes for writing input methods (such as software keyboards). These APIs are not for use by normal applications—they are a framework specifically for writing input method components. Implementations will typically be derivatives of InputMethodService.

**android.view.inputmethod** Framework classes for interaction between views and input methods (such as soft keyboards). In most cases, the main classes here are not needed for applications, since they are handled for the developer by TextView. When implementing a custom text editor. However, it is necessary to implement the InputConnection class to allow the current input method to interact with the view.

#### 11.2.2.2 org.apache.commons.math3

Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang [20].

The Math library from the Apache Software Foundation provides a set of well-tested and well-documented mathematical functions and objects, including Matrix classes and associated Matrix functions such as Inverse and Transpose, and statistical functions such as Variance and Covariance. This is quite handy for computing the Mahalanobis Distance.

The Apache Math library is fully supported by the Dalvik virtual machine.

### 11.2.3 Implementation

#### 11.2.3.1 Event capture

Each time a virtual key is pressed or released, the virtual keyboard fires an “onPressed” or “onReleased” event that is treated asynchronously to analyze the key and if any modifiers were active at the time of the action (“Shift”, “Alt”, etc.). This event also triggers
a capture on the Clacker class that will record the character, a timestamp of the date of the action (pressing/releasing). Each character is stored as an object with these properties. The input keyCode is converted into the corresponding character before being concatenated to a string with the previous characters of the word. The first character of a “word” is concatenated to an empty string. In parallel, the timecode in millisecond of the keyPress event is appended to an ArrayList of the previous timecode; the same happens at the “keyRelease” event. A word is terminated when a white space character is entered (“Space”, “Tab”, “Enter”, etc...). At this time, the accumulator contains the word and the timecodes for each of the “press” and “release” event.

The ArrayList structure is a resizable array implementation of the List interface that offers constant-time operations for the size, isEmpty, get, set, iterator, and listIterator operations. The add operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking).

11.2.3.2 Screen Lock

In order to prevent unauthorized usage of the device, the screen shall be locked after a certain number of rejections. Android allows an application to programmatically lock the screen. This operation requires serveral privileges and action from the user. To authorize this the application must have the BIND_DEVICE_ADMIN permission in the AndroidManifest.xml file. This permission grant the right for the application to interact with the Device Administration API. This API provides device administration features such as wiping the data after too many failing logins, disabling the camera, or locking the screen immediately.

The user must also allow the application to interact with the device in Settings/Security/Device Administrator. This screen displays the list of the applications that ask for the administrator rights to interact with the device. Selecting the application will display a brief description of the required permission and the reasons.

If the current user does not have the admin rights, the methods that require such permissions will throw an exception. The application has to check whether the user has set the correct privileges using the isAdminActive() method.

11.2.3.3 Database storage

As mentioned previously, Android offers full support of the SQLite Database, with direct Java interfaces to access the API. However the recommended method to create a new SQLite database is to create a subclass of SQLiteOpenHelper and override the onCreate() method, in which you can execute a SQLite command to create tables in the database.

The SQLiteOpenHelper class is a helper class to manage database creation and version management. Using this class as parent offers an easy way to create the database when the application is started for the first time and each time the database schema is updated.

11.3 Results

Results so far have been promising but far from perfect.
While the NNMD classifier on a physical keyboard has an equal zero-miss false-alarm rate of 0.09 in a research setting, in practice the value is significantly higher, implying decreased reliability on a mobile device.

I suspect that the high error rate on mobile can be attributed to a number of factors. First, it is likely that research participants in prior studies were undistracted by outside factors, allowing them to concentrate fully on typing. Meanwhile, the real-world testing scenarios I employed were subject to uncontrolled influences such as distraction from coworkers, frustration, and exhaustion. Second, consistent typing style on a phone is improbable. While I consistently type on a physical keyboard using a “homerow”-based approach, I noticed a tendency in myself to hold the phone in whatever manner I found best at the time, sometimes using my left hand to cradle the device while pecking at keys with my right index finger, and other times holding the phone with my right hand and typing with my right thumb. Although I attempted to control for this by only testing using the left cradle method, it raises a serious concern over how the typical user might hold her device. Third, and perhaps least controllable, I often found myself partially mistyping words and using delete to go back, preventing the algorithm from detecting that a key word had been typed.

Despite these shortcomings, the NNMD implementation was still able to detect and reject impostors with an EER of 0.22. The ability to vary the number of sequential alarms required to lock the phone prevented annoyance while I used the device and proved capable of still detecting true impostors. Assuming independence, after three sequential rejections there is only a 0.1 (1%) likelihood that the user is not the authentic user.
Chapter 12

Areas for Future Research

12.1 Swype dynamics

The Swype keyboard is an input method that takes into account the physical interface of a touchscreen. Instead of pressing and releasing individual virtual keys, the user can drag the tip of the finger on the screen from letter to letter to form a word. In this way, a path is drawn by the finger. Traditional timing data may still be relevant, since each key is still “typed” in a sequential order, but a more interesting question is whether the user can be identified by the particular shape of these paths. To answer this question requires the ability to classify arbitrary shapes, which is almost certainly the domain of machine learning.

12.2 Fingertip pressure

Some mobile devices can detect or infer fingertip pressure. It could be fruitful to analyze not only timing data but pressure data when constructing a dataset for analysis. The statistical classifiers discussed in this paper could be used with little to no modification. Unfortunately, fingertip pressure only works on resistive displays but the majority of smartphones use capacitive displays.

12.3 Fingertip shape and orientation

A further area of interest is analysis of fingertip shape and orientation. Depending on the most comfortable wrist angle and viewing distance, the orientation of the fingertip may be rotated left or right from a straight vertical alignment. For two-handed mobile typists, it could be possible to infer which hand typed a particular key based on orientation of the ellipsoid formed by the fingertip against the screen. Even for single-finger typists, the orientation of the ellipsoid could be expected to vary between letters depending on typing style (moving the entire arm vs moving just the finger). Additionally, the surface area contacted by the finger could be used to draw conclusions about the user.
Chapter 13

Conclusions

Traditional keystroke dynamics algorithms for physical keyboards can be used on mobile devices with little to no modifications. The unobtrusive nature of keystroke dynamics makes it an excellent solution for adding an extra layer of security to the mobile environment. However, due to the variability in typing habits of mobile users, it is difficult to obtain low error rates regardless of the algorithm used.

By requiring a threshold number of sequential errors to be triggered before resultant actions are taken, we maintain accuracy and shield the authentic user from the annoyance of false alarms that would surely occur without such precautions. In the meantime, impostors are detected and rejected with high accuracy.

The keystroke dynamics algorithms that were tested in a mobile environment performed relatively the same as they did on a traditional environment with regard to best-to-worst ordering. Pure Euclidean distance was the least accurate, while Nearest Neighbor Mahalanobis distance was the most accurate. The Nearest Neighbor Mahalanobis and Nearest Neighbor Euclidean with Flight-Time weighting were both clearly superior to other methods, with the Nearest Neighbor Mahalanobis at an average EER of 22% and Nearest Neighbor Euclidean (Flight-Time weighted) at 32%, while other methods clustered around 50%.
References


