HONOUR SCHOOL OF COMPUTER SCIENCE
PART C

MAGDALEN COLLEGE
UNIVERSITY OF OXFORD

Key Generation from Behavioural Patterns using Smartphones

Supervisors:
Stefano Gogioso
Bob Coecke

Submitted By :
Sonal Vedi

Abstract

This project aims to create an Android application to generate cryptographic keys using user-specific behavioural patterns. The behavioural patterns are recorded through sensors on the Android device, and the keys are generated when required by processing the recorded data from recent periods. This is done by applying the k-means algorithm to classify the data, after which it can be discretised into voxels on which we will perform statistical analysis. This naturally leads to the fact that the key used for encryption and the one used for decryption will match if and only if the user maintains the same behavioural patterns. Therefore, it will be difficult for a malicious entity to generate the keys as they would be required to perform the same real-world activities as the original user for an extended period of time. For the application to record user data, users input activities they perform regularly and data for these is recorded when the activity starts. All data is stored in an SQLite database. Graphs and charts are used to display the data for testing but most of these would not be shown to the user.

Contents

1 Introduction

Common authentication methods today use passwords to verify the identity of a user. However, there is scope for cryptography and authentication without passwords - instead of fixed information, the user is asked to provide data which only they can generate on demand. We choose the route of using user-specific behavioural patterns to provide a string of bits whose randomness is unique to each individual.

Smartphones play a vital role in many people's daily lives, making them a natural choice for collecting user-specific data. This project aims to implement a mobile application to generate symmetric cryptographic keys from user-specific behavioural patterns. Randomness will be provided through the use of built-in sensors available on the majority of modern phones.

As the cryptographic keys will be generated locally and will never be stored or transmitted, we can guarantee privacy of the data used for generating the key. Data decryption will only be possible if the user's daily habits are stable enough. If stability decreases over time, this will provide a natural degradation of encrypted information over time; a measure of this will be given by error detection. Statistical analysis methods will provide robustness.

I have chosen to do this project because there is a possibility that this method of providing security can become commonplace considering the way mobile devices are used today. This would be done by providing an API to generate keys in the way this application does; these keys could then be used by anyone with an Android phone to encrypt their data. Developers are constantly trying to automate tasks so users can perform actions seamlessly with minimal interaction. Following this reasoning, this project can contribute to the way the average person uses their phone on a daily basis. In the future, it would be possible to replace all passwords which are currently strings of ASCII characters with cryptographic keys generated from behavioural patterns only when needed. This would have the obvious impact of relieving people of the need to create, remember, and type long passwords, and would have the further effects of providing security without putting the pressure of doing this on individual users.

In this project, I have achieved my aim of creating a proof of concept for generating security keys using user-specific behavioural patterns. I have successfully created an Android application which records data through sensors while the user performs their regular activities and have used this recorded data to generate cryptographic keys. This application shows this method of key generation is feasible and that further work could be carried out in the future to extend the project.

1.1 Roadmap of the Report

This report will begin by giving some brief information relevant to the project in Section 2, which the average reader may be unfamiliar with before moving on to laying out the requirements of the final product in Section 3 Section. 4 will show my designs of the application and how they were developed, which will lead on to Section 6: the implementation of the presented designs. After this I will test the application and provide results in Section 7 to show the project has been successful, as well as evaluate the performance of the product. Finally, I will conclude with Section 8, in which I will include an appraisal of not only the application, but also of my own methodologies and personal development.

2 Background

Most Android devices are built with some basic sensors, including an accelerometer, a magnetic field sensor, a pressure sensor, and many others. Depending on how one uses their device, these sensors will obviously record different data. As no two people are expected to have exactly the same daily lifestyle, there will be natural variations in the recorded data for different people, which provides the security this project aims to achieve.

Android applications are programmed in Java, but many common Java libraries are unavailable in the Android environment. For example, Java's Swing library cannot be used, as all user interfaces are defined using XML files. This restricts what can be done statically without the use of libraries, as the default XML tags available limit developers to a very small set of components. Basic components such as buttons and lists are provided, but anything more graphically-inclined, such as a scatter chart, is impossible with the standard Android XML tags. Therefore, instead of implementing various types of graphs myself, I chose not to reinvent the wheel and to instead use an Android library called MPAndroidChart available under the Apache Licence, Version 2.0 (PhilJay, n.d.). The programmatic parts of the application are all written in Java.

2.1 Technical Terms

This report makes use of common Android and mobile terms, as well as some field-specific jargon. This section provides an introduction to the terminology used, which should make this report easier to follow.

- **Status bar**: The bar at the top of all Android screens which houses notifications as well as persistent information, such as battery percentage and connectivity status.
- **Action bar**: A toolbar used by many Android applications which is specific to only the application itself. Contains the application-specific menu button, optional text, and extra menu icons which are seen as important enough to be displayed separately to the regular menu.
- **Toast**: A floating message which disappears automatically after a fixed period of time. Used to provide context-specific information (Figure 1).
- **Trit**: A generalisation of a bit from the values {0, 1} to the values {0, 1, 2}.
- **Gray Codes**: A type of binary encoding of integers where consecutive numbers differ in exactly one bit.

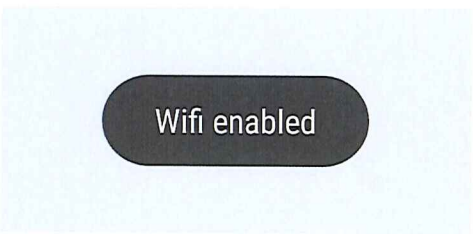


Figure 1: An example Toast from this application, shown when unit testing sensors.

3 Requirements

Below are the requirements the applications must satisfy. These will be used to gauge whether the application performs well enough, both in terms of usability for an end user, and the stability of cryptographic keys produced.

1. The user must be able to generate cryptographic keys via statistical analysis of sensor data recorded during regular user-specified activities.
2. Any generated key must not be stored, but instead regenerated when needed based on recent sensor data.
3. The statistical methods employed must be stable enough to allow for the key to be reconstructed successfully conditional upon sufficiently regular behaviour, such as could be expected in daily and weekly activities.
4. The user interface of the application should be straightforward, giving the user clear instructions where needed.
5. The statistical analysis must provide some guarantee that the generated keys are random.

4 Design

4.1 Using the Application

Below are four flowcharts displaying how to use this application to generate security keys. Flowcharts (a) to (c) are to be used by an average user, but Flowchart (d) is only included for the sake of developing this application. If this were to be released to the general public, this would not be available but it is included here to demonstrate the workings of the application as it was developed.

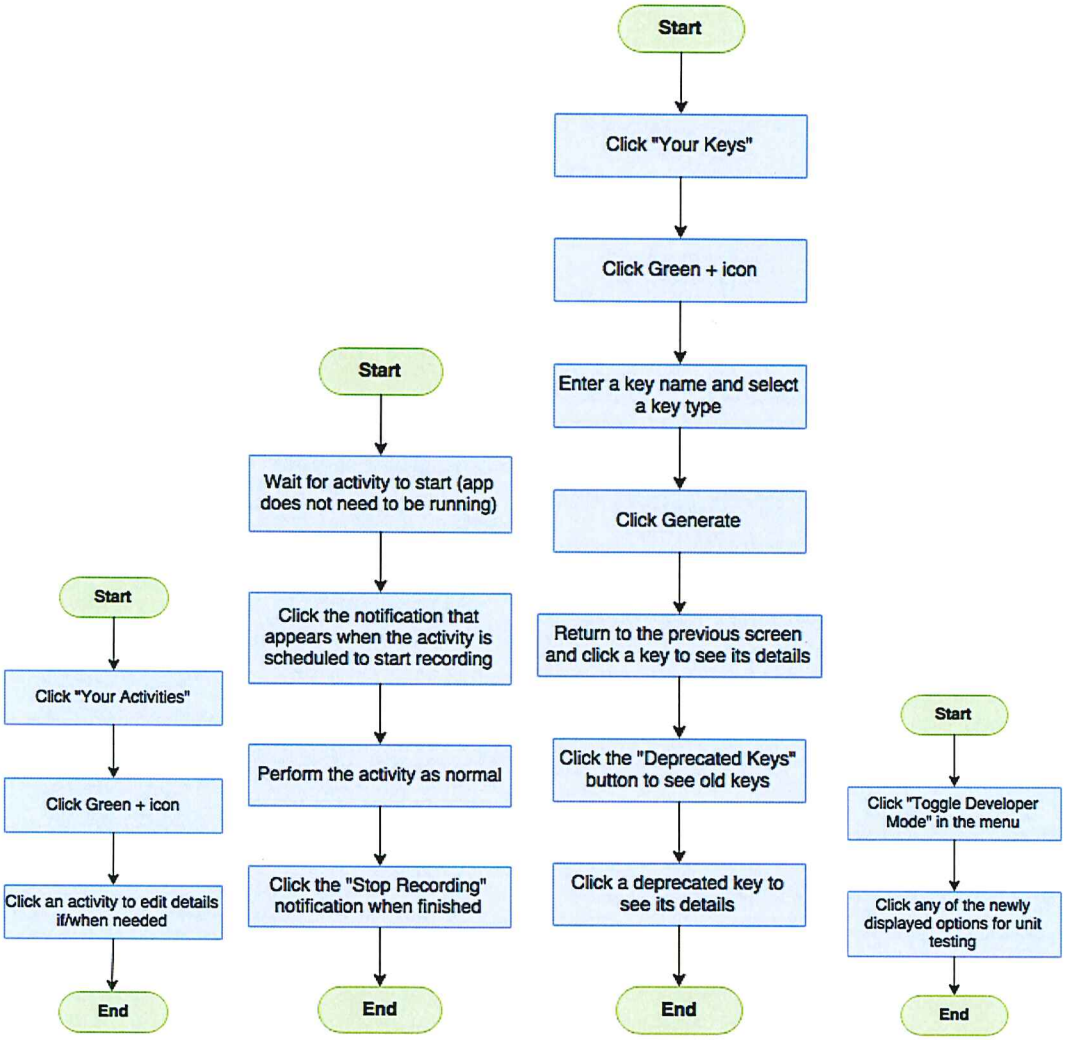


Figure 2: (a) How to specify a new activity the user performs. (b) How to record data for a pre-scheduled activity. (c) How to generate and view keys, including deprecated keys. (d) How to access developer mode (used only when developing).

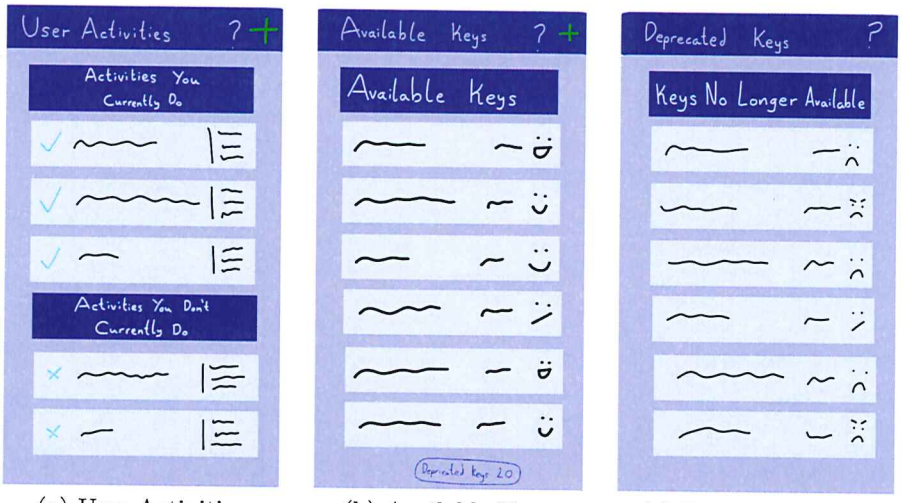
4.2 User Interface Designs

The following designs show the planned user interface for the application. These designs only focus on what a real user would see, and ignore the developer mode shown in the flowchart in Figure 2d. Note that all images used in the application will either be from Android libraries, or from Openclipart, a website hosting free images with all rights released (Openclipart, n.d.). As shown in these drawings, every screen has a question mark in the action bar, which the user can click to display a dialog box containing help and instructions, which addresses the fourth requirement in Section 3.

Figure 3 shows the first screen displayed to the user when the application is launched. The layout will dynamically readjust when the phone orientation is changed.

(a) Portrait (b) Landscape

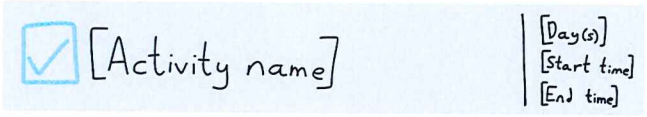
Figure 3: The home screen in the two different orientations.



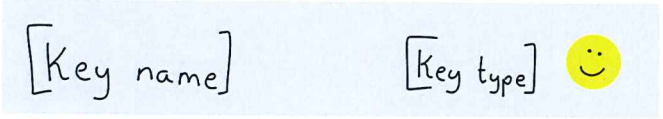
(a) User Activities (b) Available Keys (c) Deprecated Keys

Figure 4: (a) Clicking the calendar on the home screen shows a screen with a list of activities the user performs. (b) Clicking the keys shows their available keys. (c) Clicking the deprecated keys button on the Available Keys page shows a list of deprecated keys.

Figure 4 shows the remaining three screens that an end user will see. I have chosen to make all of these similar by making them all lists with a similar format and colour scheme. However, as the lists present different data, the layout of each list item will be different (Figure 5). The User Activities list items will display the activity name, a checkbox for whether or not the activity is being performed regularly, and a time and day(s) on which the activity is performed. In contrast, the Available Keys and Deprecated Keys list items will only show the key name, its type (e.g. 64 bit), and a key degradation indicator.



(a) User Activities list item



(b) Available Keys and Deprecated Keys list item

Figure 5: The two types of list items

When viewing the User Activities screen, the user can click the green + icon to add a new activity. This will bring up the input form in Figure 6. Also, clicking an existing activity in the list will show the same input form, but this time it will be pre-populated with the data for the clicked activity.

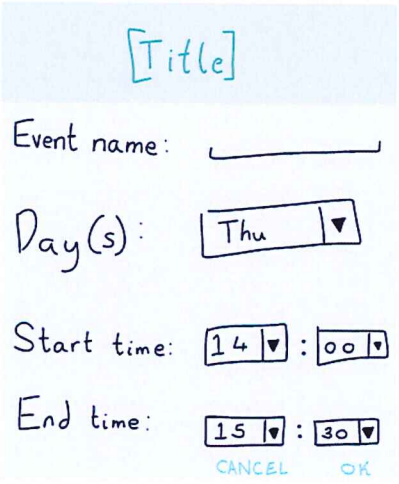


Figure 6: The input form to add or edit an activity

When the + icon on the Available Keys screen is clicked, the user will be shown a screen (Figure 7) where they can enter details for the new key and see a graph showing the number of bit available from some of the activities they perform, so they have an idea of where the bits may be selected from.

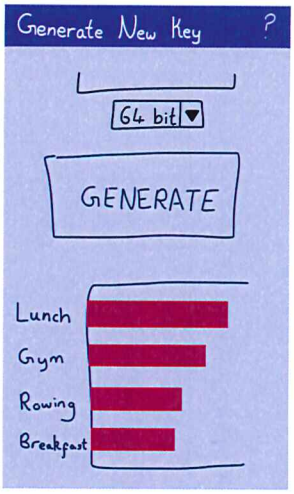
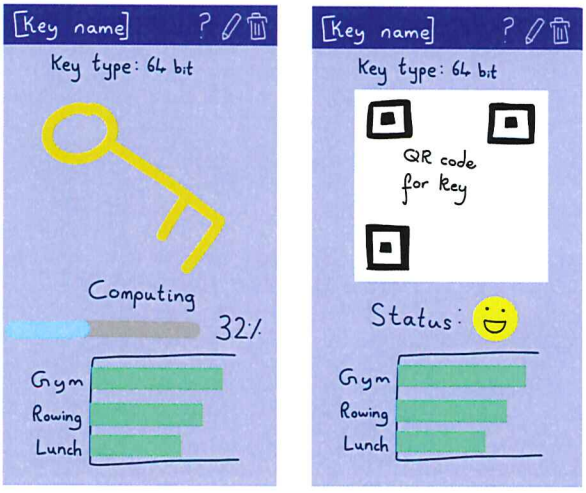


Figure 7: The screen shown when creating a new key

When a key is clicked from the Available Keys list, the user is shown a screen (Figure 8a) with a barchart showing the activities from which the bits for this key were selected. There is also a placeholder image where the key will be displayed, and a progress bar showing how much of the key has been recomputed. Once the key regeneration is done, the user is shown (Figure 8b) a QR code encoding the key, and a status indicator below it.



(a) Recomputing key (b) Displaying key

Figure 8: (a) The key must be recomputed before (b) it is displayed.

This concludes the designs of the application. I have shown designs for the flow of the application and the user interface. The statistical analysis is more involved and is therefore shown in Section 5 to aid the flow of this report.

5 Key Generation

5.1 Preparation for Key Generation

The main task of this application is to generate cryptographic keys. To do this, user-specific data is recorded as described in Section 6.2. The recorded data is nothing more than some 1-, 2- or 3-dimensional coordinates. To give these coordinates more meaning, we first perform the k-means clustering algorithm on them. Once clustering is complete, we need only perform statistical analysis to manipulate the data to obtain the keys.

5.1.1 K-means Clustering

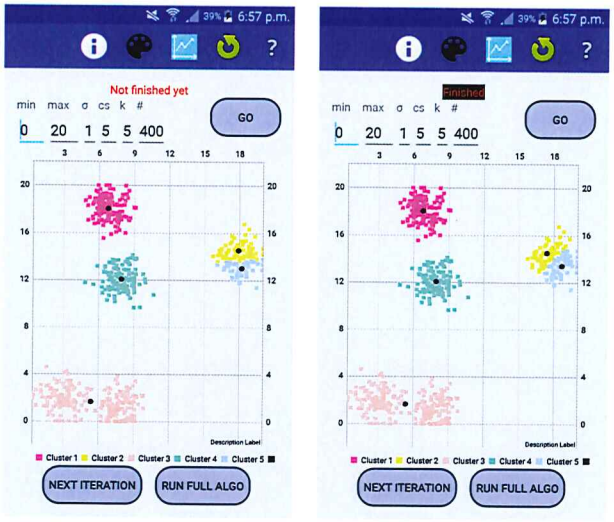
K-means clustering is an unsupervised Machine Learning algorithm, which means it is given input data without any corresponding output variables, and is given the task to find an underlying structure in the data (Brownlee, 2016). Specifically, the k-means algorithm takes an integer k and a set of datapoints S as an input. It partitions the points in S into k disjoint subsets $S_1, \ldots, S_k$ such that the function $W = \Sigma_{i=1}^{k} \Sigma_{p \in S_j} d(\mu_j, p)$ is minimised, where d is a distance measure between two points and μ_j is the centroid of subset S_j. In other words, the algorithm splits the original dataset into k clusters. The algorithm is straightforward and is explained in the pseudocode in Listing 1.

Listing 1: K-means Clustering

```
1 Select k points uniformly at random from the dataset to be the centroids
2 Repeat until centroids do not change:
3 For each point p in the dataset:
4 Assign p to the cluster for its closest centroid
5 Recompute the centroids of each cluster
```

There are many possible distance measure, such as Manhattan distance, Euclidean distance, or supremum distance. I have chosen to use the Euclidean distance as is conventional for this algorithm.

The problem with this algorithm is that it takes k as a parameter, but this will need to be computed dynamically for each dataset as it will not be the case that every run of every activity performed by every user will result in data with the same number of clusters. The methods used to calculate the optimal k are discussed in Section 5.1.3 and 5.1.4. The task of finding the optimal k and then performing the k-means clustering algorithm was too slow to be performed on the main UI thread, so these are both done in a background task (see Section 6.6). However, this was not the main problem with the k-means algorithm. The issue was that the starting centroids are chosen uniformly at random, so it is possible for the algorithm to converge to an incorrect clustering. For example, in Figure 9a, two of the initial centroids are on the same cluster. When this algorithm terminates (Figure 9b), the clusters are clearly incorrectly classified, even though the algorithm began with the correct value of k. Therefore, the application uses a slight variation of the k-means algorithm known as k-means++.



(a) Starting clusters (b) Final clusters

Figure 9: (a) The starting centroids are selected randomly, so the blue and yellow clusters start too close to each other, while the pink cluster spans across two of the real clusters. (b) The final configuration is clearly incorrect.

5.1.2 K-means++

Line 1 of Listing 1 states the starting centroids are chosen at random. This provides no guarantees that the algorithm will converge to the correct clustering. However, Arthur and Vassilvitskii (2007) show that by using a heuristic to assign the initial centroids, we "obtain an algorithm that is $O(\log k)$—competitive with the optimal clustering". In simpler terms, if a clustering C has a cost COST(C), then the optimal clustering, C_{OPT}, has cost COST(C_{OPT}) $\leq O(\log k)\cdot$COST(C_{km++}) $+ O(1)$. In comparison, there are no approximation guarantees for k-means (Arthur and Vassilvitskii, 2007).

To select initial centroids with the k-means++ algorithm, the first centroid is selected uniformly at random from the dataset S. For the following initial centroids, choose point p to be the centroid with probability $\frac{D(p)^2}{\sum_{q\in S} D(q)^2}$, where $D(x)$ is the shortest distance from point x to any centroid already chosen. By doing this we increase the probability of selecting points furthest from current centroids to be added to the list of centroids, giving a reduced values for our objective function W right from the initialisation step. The rest of the algorithm runs as k-means, and selecting centroids in this way does not add any noticeable time on to the algorithm. However, as with regular k-means clustering, to apply this algorithm k must first be known. The method used to select the optimal k is discussed in the following sections.

5.1.3 Gap Statistic

Tibshirani *et al.* (2001) proposed the "gap statistic" as a method to provide an estimate for the k in k-means clustering. The method is based on the well-known "elbow method" heuristic. The first step of the elbow method is to draw a graph as in Listing 2.

The graph will have a similar shape to the graph in Figure 10. As k increases, the total error decreases. However, there is a point where an increase in k no longer reduces the total error by much, making the graph look like an elbow. In Figure 10 this happens

when k=3. This value of k is optimal.

Instead of looking at W_k directly, the gap statistic standardises the comparison between $\log W_k$ and a "null reference distribution of the data" (Tibshirani *et al.*, 2001) - i.e. a distribution where there is no clustering, such as the uniform distribution. Listing 3 shows how to find the optimal k using the gap statistic.

Listing 2: Elbow method

```
Choose K to be the maximum number of clusters we will allow
Initialise array W of size K
For int k <- [1..K]:
cs = clusters formed from running k-means clustering on dataset
For Cluster c <- cs:
mu_c = centroid of cluster c
For Point p <- c:
W[k] = W[k] + (d(p, mu_c))^2
Plot W[k] against k
```

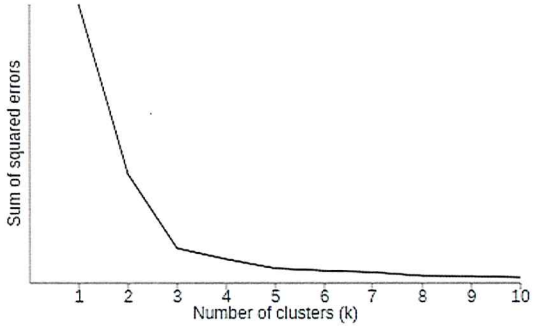


Figure 10: Plotting W_k against k gives approximately this shape. The "elbow" is judged by eye. (Gove, 2015)

Listing 3: Gap statistic

```
Choose K to be the maximum number of clusters we will allow
Initialise an array Gap of size K
Initialise an array S of size K
For int i <- [1..K]:

sum_log_W_kb = 0
Initialise an array Wkb of size B (selection of B is discussed below)

// Calculate W for null distribution with k clusters
// Use the average from B null clusters
For int b <- [1..B]:
Create a null dataset nd
Wkb[b] = W_k calculated for nd
sum_log_W_kb += log(Wkb[b])
avg_log_W_kb = sum_log_W_kb / B

```

```
// Use avg_log_W_kb to calculate the gap statistic
wk = W_k calculated for actual data
Gap[k] = avg_log_W_kb - wk

// Compute standard deviation and s_k for each k
s = 0
For int b <- [1..B]:
s += (log(Wkb[b]) - avg_log_W_kb)^2
sd_k = sqrt(s / B)
S[k] = sd_k * sqrt((B+1)/B)

// Choose the optimal k
Choose the smallest k such that Gap[k] >= Gap[k+1] - S[k+1]
```

Note that Listing 2 shows that every time W_k is calculated, the k-means algorithm needs to be run once. Listing 3 shows W_k is calculated $B+1$ times for each k, for a total of $k(B+1)$ times. K-means can be a very slow algorithm if run on a large dataset, or if unfortunate starting centroids are chosen. Therefore, when selecting B, although we want B to be large enough to provide an accurate estimate of `avg_log_W_kb` ($\log W_{kb}$ for the null reference distribution), we also want to keep it as small as possible to avoid keeping the user waiting while the algorithm runs. To avoid running the algorithm so many times, I then implemented another method of calculating k, explained below.

5.1.4 The f(k) Method

Pham *et al.* (2005) proposed a new method for finding the optimal k but left the method unnamed. For this method, we first choose a K and define W_k for each $k \in [1..K]$ as in Section 5.1.3 (Pham *et al.* (2005) and The Data Science Lab (2014) refer to this as S_k but we will use W_k for consistency with the previous section). Next we define a weight factor α_k for each $k \in [1..K]$ as follows:

$$\alpha_k = \begin{cases} 1 - \dfrac{3}{4N_d} & \text{if } k = 2 \text{ and } N_d > 1, \\ \alpha_{k-1} + \dfrac{1 - \alpha_{k-1}}{6} & \text{if } k > 2 \text{ and } N_d > 1 \end{cases}$$

where N_d is the number of dimensions in the data. Finally, $f(k)$ is calculated for each $k \in [1..K]$ as follows:

$$f(k) = \begin{cases} 1 & \text{if } k = 1 \text{ or } W_{k-1} = 0, \\ \dfrac{W_k}{\alpha_k W_{k-1}} & \text{otherwise} \end{cases}$$

Clearly W_k decreases as k increases as introducing more clusters means the sum of distances from points to their closest centroid decreases. The rate of decrease of W_k is what we are interested in, as shown in the explanation of the elbow method in Section 5.1.3. $\alpha_k W_{k-1}$ estimates W_k so the lower W_k is than the estimate of W_k, the further below 1 $f(k)$ will be. As we are interested in finding a k where the decrease in the rate of decrease (the second derivative) of W_k is the largest (the "elbow"), we want the value of $f(k)$ to be as low as possible. We can even use memoization to do this in one loop, storing nothing more than the current and last values of α_k and W_k (Listing 4).

Listing 4: f(k) method

```
1  Choose K to be the maximum number of clusters we will allow
2  best_fk = infinity
3  best_k = 1
4  k = 2 // alpha is undefined for k=1 so we start from 2
5  while k<=K:
6  Calculate W_k
7  Calculate alpha_k as explained above
8  Calculate f(k)
9  if (f(k) < best_fk):
10 best_fk = f(k)
11 best_k = k
12 prev_Wk = W_k
13 prev_alpha = alpha_k
14 return best_k
```

For this method, W_k is only computed once for each value of k, which means we only run the k-means++ algorithm $K - 1$ times, which is already more than $O(B)$ times faster than the gap statistic method. It is clear from this that the $f(k)$ method will be much faster than the gap statistic method, so it remains to be shown that the $f(k)$ method is still reliable for finding the optimal k. This was done through manual testing, and is shown in Section 7.

5.1.5 Using Cluster Data to Discretise Raw Datapoints

To generate a key, we must use the data obtained from clustering to discretise the space occupied by the datapoints. This is a very common Machine Learning technique which will allow us to calculate a probability distribution for the raw data by considering the probability of a newly recorded point falling in a given voxel.

For the following explanation, we consider some activity and sensor pair a for which enough data has been recorded to generate a key [1]. To discretise the data, we first define define $\min_d^{(a)}$ and $\max_d^{(a)}$ for $d \in \{x, y, z\}$ as the minimum and maximum values recorded by the sensor in dimension d, i.e.

$$\min_d^{(a)} = \min_d \{d \,|\, (x, y, z) \text{ was recorded for } a\}$$
$$\max_d^{(a)} = \max_d \{d \,|\, (x, y, z) \text{ was recorded for } a\}$$

This will be the range for our discretisation in each dimension. We then require a step size, i.e. the size of the voxels that the range will be broken up into.

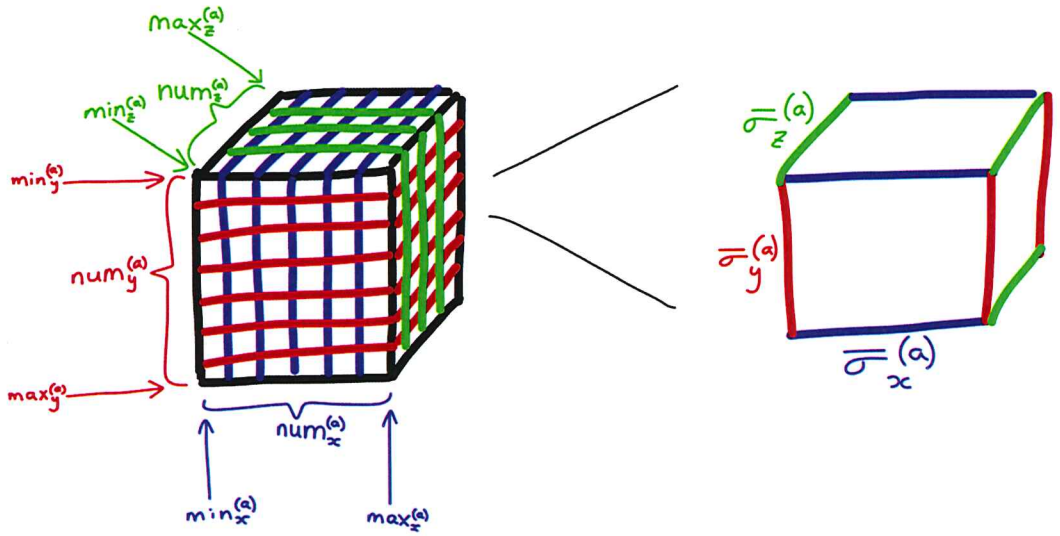
To calculate this step size, we fetch from the database a list of standard deviations for any cluster obtained by performing k-means clustering on the data recorded for a. Let $s_d^{(a)}$ be the median of the retrieved standard deviations in dimension d. This will be our step size. The reason for taking the median value is for robustness - no user can be expected to repeat their activities exactly as before without fail so taking the median allows us to ignore outliers.

[1] We use a as a superscript, e.g. $f^{(a)}$, to denote variables relating to the activity-sensor pair a; this should not be confused with exponentiation.

We now have the range and the step size for the discretisation, so we define $\text{num}_d^{(a)}$ to be the number of discrete cells in dimension d. This is calculated by:

$$\text{num}_d^{(a)} = \left\lceil \frac{\max_d^{(a)} - \min_d^{(a)}}{\bar{\sigma}_d^{(a)}} \right\rceil$$

We take the ceiling for the division in order to ensure the range includes $\min_d^{(a)}$ and $\max_d^{(a)}$. The diagram in Figure 11 visualises how these values come together to make the discretisation more intuitive. Figure 11a represents the entire discretised space, ranging from $\min_d^{(a)}$ to $\max_d^{(a)}$ in every dimension d. Figure 11b shows each voxel, of which there are $\text{num}_x^{(a)} \cdot \text{num}_y^{(a)} \cdot \text{num}_z^{(a)}$ in Figure 11a.



(a) (b)

Figure 11: A visualisation of the voxels in a three-dimensional space

5.1.6 Creating a Probability Mass Function $f^{(a)}$

We use the voxels created in Section 5.1.5 to define a probability mass function

$$f^{(a)} : \{1, \ldots, \text{num}_x\} \times \{1, \ldots, \text{num}_y\} \times \{1, \ldots, \text{num}_z\} \to [0, 1]$$

such that $f^{(a)}(x, y, z)$ is the probability of a newly recorded point falling in voxel (x, y, z). As $f^{(a)}$ is a random variable, we will calculate the sample mean and standard deviation:

$$\mu_{x,y,z}^{(a)} = \mathbb{E}[f^{(a)}(x, y, z)]$$

$$\sigma_{x,y,z}^{(a)} = \sqrt{\text{Var}(f^{(a)}(x, y, z))}$$

Let r be an integer such that we have recorded data for a for r runs of the activity. From here onwards, except in variable names and equations, the activity and the sensor will implicitly be assumed to be a. Clearly

$$\mu_{x,y,z}^{(a)} = \frac{1}{r} \sum_{i=1}^{r} f_i^{(a)}(x, y, z)$$

where $f_i^{(a)}(x,y,z)$ is the fraction of datapoints from run i which fall into voxel (x,y,z). This can be easily calculated by calculating the number of `DBSensorEventEntrys` for run i in the database where the coordinates are in the required range, and dividing by the total number of points recorded for run i. Both of these are done using a simple `SELECT` query with the `COUNT()` function in SQLite. Similarly, we calculate the sample variance to be

$$(\sigma_{x,y,z}^{(a)})^2 = \frac{1}{r-1}\sum_{i=1}^{r} |f_i^{(a)}(x,y,z) - \mu_{x,y,z}^{(a)}|^2$$

from which it is trivial to calculate the standard deviation as $\sigma_{x,y,z}^{(a)} = \sqrt{(\sigma_{x,y,z}^{(a)})^2}$.

This in fact is everything we need from our probability mass function so we are now able to move on to the final steps of preparation to generate a key.

5.1.7 Discretising the Range of $f^{(a)}$

In the way that we discretised the space over which the recorded datapoints fall, we now discretise the range of $f_{x,y,z}^{(a)}$ for each (x,y,z). To avoid confusion with the symbols in Section 5.1.5, we use the symbols m, M, and $\bar{\sigma}$ rather than min, max and s.

Define $m_{x,y,z}^{(a)} = \min_{1\le i\le r} f_i^{(a)}(x,y,z)$ to be the minimum calculated value of $f_i^{(a)}(x,y,z)$ over all runs i, and $M_{x,y,z}^{(a)} = \max_{1\le i\le r} f_i^{(a)}(x,y,z)$ to be the maximum such value. This already gives a range for the discretisation of the probability mass function; a step size remains to be found. The obvious choice, as in Section 5.1.5 would be to take the standard deviation calculated in Section 5.1.6, i.e. $\sigma_{x,y,z}^{(a)}$. However, it is possible for this to be 0 (e.g. if no datapoints fall in voxel (x,y,z)), and we cannot discretise a space with a step size of 0.

Instead, we first define $\varepsilon^{(a)} = \underset{x,y,z}{\text{median}}\{\sigma_{x,y,z}^{(a)} \mid \sigma_{x,y,z}^{(a)} > 0\}$. We can then define

$$\bar{\sigma}_{x,y,z}^{(a)} = \begin{cases} \sigma_{x,y,z}^{(a)} & \text{if} \sigma_{x,y,z}^{(a)} > 0 \\ \varepsilon^{(a)} & \text{otherwise} \end{cases}$$

We can take $\bar{\sigma}_{x,y,z}^{(a)}$ to be our step size with the guarantee that it will always be positive.

Discretising the range from $m_{x,y,z}^{(a)}$ to $M_{x,y,z}^{(a)}$ is not as easy as simply dividing by $\bar{\sigma}_{x,y,z}^{(a)}$, as we impose two more requirements on our discretisation. The first requirement is the the number of cells must be divisible by three. By enforcing this, we can discretise the range into small voxels of size $\bar{\sigma}_{x,y,z}^{(a)}$ as well as into large voxels of size $3 \cdot \bar{\sigma}_{x,y,z}^{(a)}$. This allows us to use the cells as trits. The second condition is that the mean value of $f_{x,y,z}^{(a)}$, i.e. $\mu_{x,y,z}^{(a)}$, must fall in the $(3k+2)^{\text{th}}$ cell, for some integer $k \ge 0$. This will mean a small variation of a point (one voxel to the left or right) in the small voxels will keep the point in the same large voxel. This is best explained through Figure 12. The numbered boxes represent the small voxels of step size $\bar{\sigma}_{x,y,z}^{(a)}$. The red numbers above the boxes show the trit represented by the voxel below it, and the blue numbers number the large voxels of step size $3 \cdot \bar{\sigma}_{x,y,z}^{(a)}$. We ensure $\mu_{x,y,z}^{(a)}$ occupies a voxel labelled as 1 so that if a future calculation of $\mu_{x,y,z}^{(a)}$ shifts it enough for the new value to occupy an adjacent box, although it will have a different red number, its blue number will remain the same. Counting the number of voxels (x,y,z) where $\mu_{x,y,z}^{(a)}$ falls in the wrong box will give us an estimation of the degradation of the key. This is expanded on in Section 5.2.1.

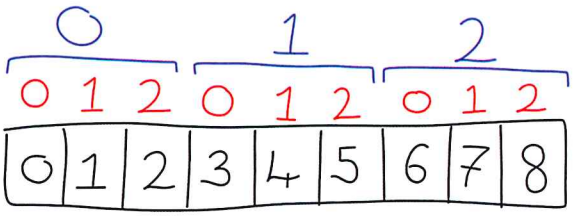


Figure 12: A visualisation of the discretisation of the range of $f^{(a)}(x, y, z)$

To meet these requirements, we define variables $l'^{(a)}_{x,y,z}$ and $u'^{(a)}_{x,y,z}$, respectively denoting the number of small voxels below ($l'^{(a)}_{x,y,z}$ for lower) and above ($u'^{(a)}_{x,y,z}$ for upper) the small voxel containing $\mu^{(a)}_{x,y,z}$.

$$u'^{(a)}_{x,y,z} = \left\lceil \frac{M^{(a)}_{x,y,z} - \mu^{(a)}_{x,y,z}}{\bar{\sigma}^{(a)}_{x,y,z}} \right\rceil \quad , \quad l'^{(a)}_{x,y,z} = \left\lceil \frac{\mu^{(a)}_{x,y,z} - m^{(a)}_{x,y,z}}{\bar{\sigma}^{(a)}_{x,y,z}} \right\rceil$$

We then tweak these variables to meet our requirements. Figure 13 shows the possibilities that can occur if we construct our discrete cells such that $\mu^{(a)}_{x,y,z}$ falls in the $(l'^{(a)}_{x,y,z} + 1)^{\text{th}}$ cell. From this it is easy to see that we should extend the range over which we discretise to meet the second requirement. We make $\mu^{(a)}_{x,y,z}$ fall in the $(l^{(a)}_{x,y,z} + 1)^{\text{th}}$ cell, where

$$l^{(a)}_{x,y,z} = \begin{cases} l'^{(a)}_{x,y,z} + 1 & \text{if } l'^{(a)}_{x,y,z} = 0 \mod 3 \\ l'^{(a)}_{x,y,z} & \text{if } l'^{(a)}_{x,y,z} = 1 \mod 3 \\ l'^{(a)}_{x,y,z} + 2 & \text{if } l'^{(a)}_{x,y,z} = 2 \mod 3 \end{cases}$$

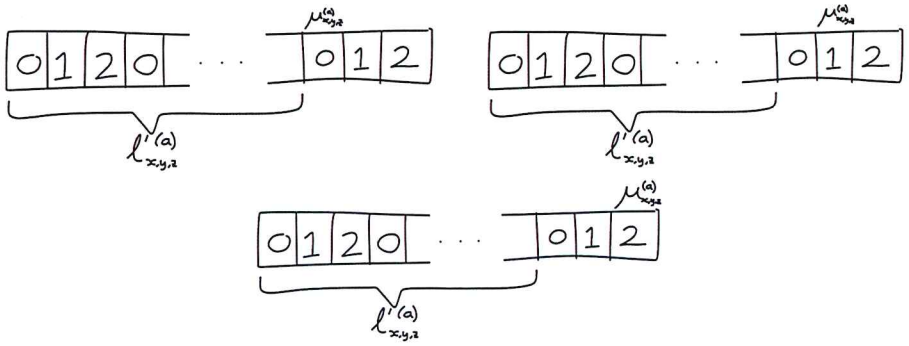


Figure 13: The three possibilities for the voxel into which $\mu^{(a)}_{x,y,z}$

Similarly, we can use Figure 14 to identify that in order to meet the first requirement, we should once more extend the range of the discretisation in order to discretise into $l^{(a)}_{x,y,z} + 1 + u^{(a)}_{x,y,z}$ cells, where

$$u^{(a)}_{x,y,z} = \begin{cases} u'^{(a)}_{x,y,z} & \text{if } l^{(a)}_{x,y,z} + 1 + u'^{(a)}_{x,y,z} = 0 \mod 3 \\ u'^{(a)}_{x,y,z} + 2 & \text{if } l^{(a)}_{x,y,z} + 1 + u'^{(a)}_{x,y,z} = 1 \mod 3 \\ u'^{(a)}_{x,y,z} + 1 & \text{if } l^{(a)}_{x,y,z} + 1 + u'^{(a)}_{x,y,z} = 2 \mod 3 \end{cases}$$

This gives us our final discretisation, as shown in Figure 15. We have $l^{(a)}_{x,y,z} + 1 + u^{(a)}_{x,y,z}$ small voxels, and define $N^{(a)}_{x,y,z} = \frac{l^{(a)}_{x,y,z} + 1 + u^{(a)}_{x,y,z}}{3}$ to be the number of large voxels. The step

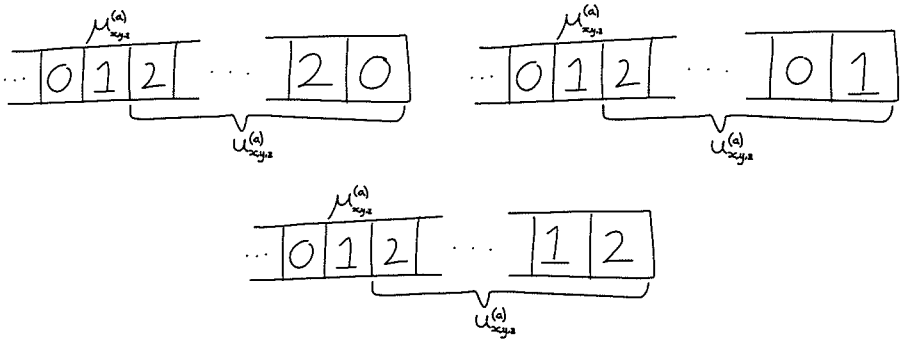


Figure 14: The three possibilities for the tri-arity of the number of voxels we have

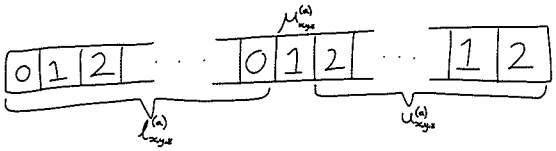


Figure 15: The final discretisation. Note the last voxel is labelled 2 so the number of voxels is a multiple of 3. Also note that $\mu^{(a)}_{x,y,z}$ falls in a voxel labelled 1. Thus both requirements are satisfied.

size was calculated near the start of this section to be $\bar{\sigma}^{(a)}_{x,y,z}$. However, we have extended our range by adding voxels on either side of the original $l'_{x,y,z}$ and $u'_{x,y,z}$, so we define a new minimum and maximum for the range:

$$\bar{m}^{(a)}_{x,y,z} = \mu^{(a)}_{x,y,z} - (l^{(a)}_{x,y,z} + \frac{1}{2}) \cdot \bar{\sigma}^{(a)}_{x,y,z}$$
$$\bar{M}^{(a)}_{x,y,z} = \mu^{(a)}_{x,y,z} + (u^{(a)}_{x,y,z} + \frac{1}{2}) \cdot \bar{\sigma}^{(a)}_{x,y,z}$$

At this point we store in the database the values $\bar{m}(a)_{x,y,z}$, $\bar{M}v_{x,y,z}$, $N(a)_{x,y,z}$, and $\bar{\sigma}^{(a)}_{x,y,z}$ with the composite primary key as the activity and sensor pair[2] a that produced the data which was manipulated to obtain these values. This concludes the preparation for generating a cryptographic key.

5.2 Key Generation

We now have everything we need to generate or regenerate a security key as well as to check robustness. To generate a key, we must first have performed the preparation in Section 5.1.5 to 5.1.7. This preparation is done when data is recorded (see Section 5.2.2). We recalculate $\mu^{(a)}_{x,y,z}$ as we did in Section 5.1.6. The reason for this is that it is possible the user has recorded more data for activity-sensor pair a since the preparation was first done, meaning the expected value of $f^{(a)}$ may have changed. We fetch $\bar{m}^{(a)}_{x,y,z}$ and $\bar{\sigma}^{(a)}_{x,y,z}$

[2]A composite primary key in an SQLite database is one that spans over multiple columns. In our case, each activity and sensor pair only appears in one record, but each activity or each sensor can appear in multiple records.

and use these to calculate

$$j^{(a)}_{x,y,z} = \left\lfloor \frac{\mu^{(a)}_{x,y,z} - \bar{m}^{(a)}_{x,y,z}}{3 \cdot \bar{\sigma}^{(a)}_{x,y,z}} \right\rfloor$$

which is the index of the large voxel into which the mean now falls, given any new data that may have been recorded. Fetch $N^{(a)}_{x,y,z}$ from the database and encode each $j^{(a)}_{x,y,z}$ in Gray codes using $\lceil \log_2(N^{(a)}_{x,y,z}) \rceil$ bits. Finally, we loop over all x, y, and z and concatenate the values of $j^{(a)}_{x,y,z}$ to form a large bit string. This gives us a cryptographic key and can be truncated to the required number of bits. The reason for using Gray code should now be clear: if $j^{(a)}_{x,y,z}$ differs by 1, the Gray code will only differ by 1 bit, by definition of Gray codes, so the generated key (a concatenation of the Gray codes) will only have a single incorrect bit. The statistical analysis performed on the key to generate it, ensures each bit is equally likely to be 0 or 1, satisfying requirement 5 of the project.

However, we do not take this as our final key. We instead create multiple such keys and take a small number of bits from each, concatenating them all to create one key of the desired length. This means many different activity-sensor pairs will be used in creating any key, so for any key to be forged, the malicious attacker would have to replicate various different activities to a high enough quality rather than just one activity. For each key, we store the ordered list of activity-sensor pairs used to create the key as well as the number of bits obtained from each pair.

5.2.1 Error Detection

The small voxels can be used to detect errors. Calculate

$$p^{(a)}_{x,y,z} = \left\lfloor \frac{\mu^{(a)}_{x,y,z} - \bar{m}^{(a)}_{x,y,z}}{\bar{\sigma}^{(a)}_{x,y,z}} \right\rfloor \mod 3$$

If $\mu^{(a)}_{x,y,z}$ is close enough to the original value computed when preparing to generate the key, $p^{(a)}_{x,y,z}$ will be 1. Otherwise it will vary slightly to 0 or 2. This is the reason we use trits rather than bits - using trits we can detect an error in either direction. We need not consider the case where $\mu^{(a)}_{x,y,z}$ varies enough for $p^{(a)}_{x,y,z}$ to be 1, as this would require $\mu^{(a)}_{x,y,z}$ to vary by at least three times the standard deviation. The probability of this is negligible given we expect the user to be fairly regular with their real-world activities. We can then measure the total error in the key as

$$E = \frac{\left(\frac{3}{2} \sum_{x,y,z} |p^{(a)}_{x,y,z} - 1| \right)}{v}$$

where v is the number of voxels used to create the key (i.e. if the key only takes 2 bits from an activity-sensor pair, it is possible it only used data from one voxel, so $v = 1$ in this case). E lies in $[0, 1.5]$ and is 0 if there are no errors in any of the used voxels. For a completely random key, we expect 1/3 of the $p_{x,y,z}$ values to be 1, so 2/3 of the voxels will cause an error, so E will be 1.

5.2.2 Putting Theory into Practice

There are some practical concerns with the above description of generating a key. Firstly, the preparation described in Section 5.1 needs to be done before the key can be made.

However, performing all of these calculations when generating a key would take too long, keeping the user waiting while a background thread works on the preparation and generation. Therefore, when a user performs an activity, k-means clustering is performed as soon as the recording is finished. This batch processing is unnoticeable to users as it happens in the background, and the user does not expect anything from the application at this point. Also, once the clustering is complete, if the number of runs of the performed activity is greater then 22 (a lower limit for the Central Limit Theorem to apply), the preparation is done for any sensor which is not already being used in a key. The reasoning behind this is that if a sensor is being used in a key, the values stored for $\bar{m}_{x,y,z}$, $\bar{M}_{x,y,z}$, $N_{x,y,z}$, and $\bar{\sigma}^{(a)}_{x,y,z}$ should not be changed, but if a sensor is not being used, we should keep these values up-to-date in case they are required for the next key to be generated.

Another concern is briefly touched upon in Section 5.2. A key is generated not by one activity-sensor pair, but by many pairs. The implementation for this takes a random number b of activity-sensor pairs, and each of these is used to generate a random number[3] of bits such that the concatenation of these bits is a key of the desired size. To do this, we need to be able to select activity-sensor pairs which will provide enough bits for the key. We do this by storing the number of bits provided by each activity-sensor pair in the database when we perform the key generation preparation, and select the key which provides enough bits by fetching the required activity-sensor pair from the database. Before generating a key, a chart is displayed showing the activities in the database which can provide the most bits.

The final concern is displaying the key. The length of the key will be a multiple of 8, which means it will be an integer number of bytes. We can use this fact to encode the key as an ISO-8559-1 string, where each byte represents one character. This is then passed to the ZXing library, which requires a string as an input, and will output the bitmap for a QR code. This QR code is displayed to the user along with a chart showing which activities were used to generate the key and in what proportions.

All of the above concerns deal with tasks that can take a long time and use up resources necessary for keeping the UI responsive. These are performed in a background thread, as discussed in Section 6.6.

[3]These random numbers are pairwise independent as knowing one does not tell us the other, but they cannot all be independent as they must sum to the key size.

6 Implementation

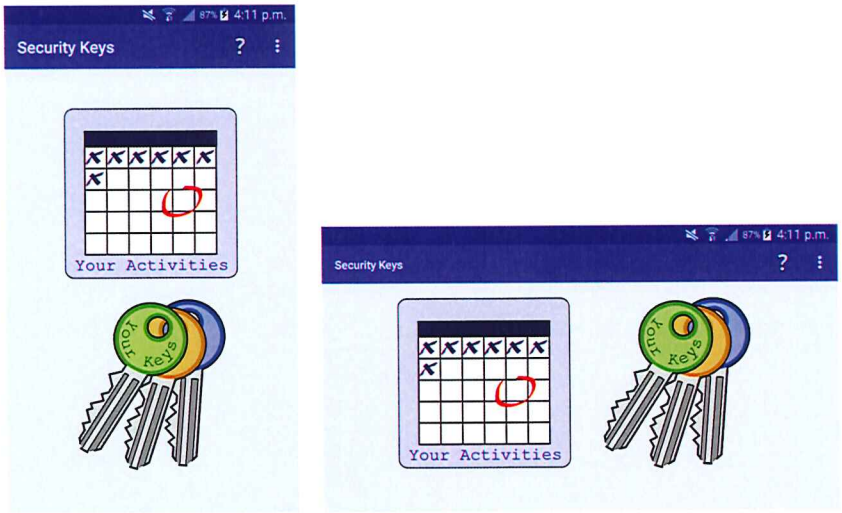
This section will discuss what was implemented, how it was implemented, and how it can be used. This should give a detailed insight into all of the features of the application and the important and interesting parts of the code without going into too much detail. Some screenshots in this report contain a small white circle, as in Figure 16. The white circle indicates where the screen was being touched while the screenshot was taken. These are not a part of this application.

(a) (b)

Figure 16: The difference between the icon when it (a) is not and (b) is being clicked. The white dot shows where the screen is touched, and will be seen in further screenshots below.

6.1 Home Screen

The home screen displays only two buttons (Figure 17). As per Android convention, long-pressing on either of these buttons (and any button or icon in the entire application) displays a toast with a description of what clicking the button does (Figure 18).

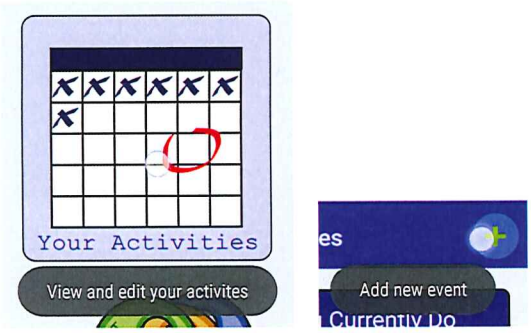


(a) Portrait mode (b) Landscape mode

Figure 17: The first screen the user sees when the application is opened. Only two buttons are visible. The view re-adjusts dynamically when the orientation of the device is changed.

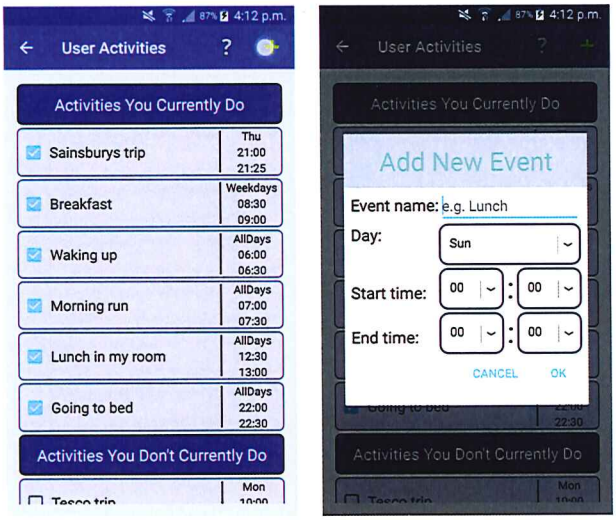
6.2 User Activities

Before using most applications, it is necessary for the user to perform some basic tasks to set up the application in a way that suits them. For this application, this is done by inputting data about activities the user performs via the User Activities screen which



(a) (b)

Figure 18: A descriptive toast appears if the user long-presses on (a) either of the buttons on the home screen or (b) any button in the application



(a) (b)

Figure 19: (a) The + button in the action bar being clicked. (b) The input dialog which appears, allowing the user to enter details of a new activity

can be reached by clicking the first button on the home screen. A user can create a new activity by clicking the + icon in the menu (Figure 19a). This brings up an input box into which the user can add their activity details (Figure 19b). On clicking "OK", this activity is saved in a database and displayed in a list along with all other activities entered. To edit an activity, the user can click on the item in the list corresponding to that activity to see another input box which is pre-populated with the details of the required activity. If a user changes their lifestyle and no longer performs an activity, they can click the checkbox on the list item corresponding to that activity to deactivate it. At this point, an Alert dialog pops up which notifies the user of the keys using data from the activity the user is trying to deactivate, and asks the user to confirm the deactivation.

If an activity is activated, the user is sent a notification exactly one minute before it is scheduled to start. If they click this, the sensors on the phone are activated and data recording begins. The reason for this is in case the user happens to not perform the activity just for one day (for example, skipping a morning run while still intending to continue a daily running regime). In this case no data should be recorded, so the user will simply be able to ignore the notification. Assuming the notification is clicked, the user is shown a new notification reminding them data is being recorded. This notification is persistent, so the user cannot remove it by swiping it away.

Finally, once the user has completed their activity, they can click this persistent

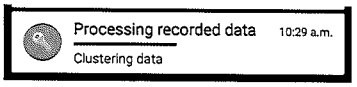


Figure 20: On Android devices, a notification containing a progress bar is the standard method of displaying progress for a task in a background thread.

notification to stop data recording. At this point the notification disappears and they are shown a third notification. This notification includes a progress bar (Figure 20) which shows the user how much of the recorded data has been processed and stored. This is persistent while the processing is taking place, but can be removed once it is finished. Thus data from activities the user performs can be collected with only two clicks from the user - one to start recording and one to stop it.

The first of these notifications plays a sound from the phone speakers when it appears. The next two notifications do not need to do this as they are merely state indicators rather than event notifiers. All of these notifications are programmed so that if the device has an LED light, it will flash orange when the notification is present in the status bar.

6.3 Recorded Data

Once data has been obtained, it is stored in an SQLite database. Data from new points which have not yet been processed are logically stored as `DBSensorEventEntry` objects; processed data only contains data from clusters after k-means clustering has been performed (see Section 5.1.1) so it is stored logically as a `DBClusterEntry` object. Once data is old enough, it is removed permanently from the database, ensuring the keys are always calculated using fresh data.

The `DBSensorEventEntry` class contains fields for x, y and z values which represent the datapoint recorded. The `DBClusterEntry` class contains a field for the centroid for the k-means cluster, the standard deviations in the three different dimensions for all points in that cluster, and the size of the cluster. In addition, both of these classes contain a field for the activity ID, a field for the sensor ID to record which sensor this data comes from, and a field indicating the run of the activity from which this object was obtained. This way if an activity consistently provides similar data, but one run has completely different data (e.g. if the user forgot to stop recording on time), the outlier run can be easily identified. Obviously the activity ID is needed as each activity will give unique data, so to recreate data for the cryptographic keys, we need to be able to identify where the data came from or, more specifically, which activity it came from.

Although this is the logical representation of the data in the Java code, the data stored in the table is merely stored as `Strings` and various numerical types, as SQLite does not allow storage of complex classes.

6.3.1 SQLite in Android

Interestingly, using SQLite in Android is quite different to the regular method of typing queries as `Strings` and executing them. Although queries take the same form, they are executed slightly differently in Android. Consider the dummy query "SELECT * FROM dummy_table" for the purposes of a short example (Listing 5). Line 2 returns a `Cursor` object from the `android.database` package. Counter-intuitively, calling `db.rawQuery()` does not have any visible effect on the database. The query is

only performed when `cursor.moveToFirst()` is performed. This caused a lot of bugs in my original implementation as the code for my deletion queries was being run without affecting the database.

Listing 5: Initialising the SQLite database

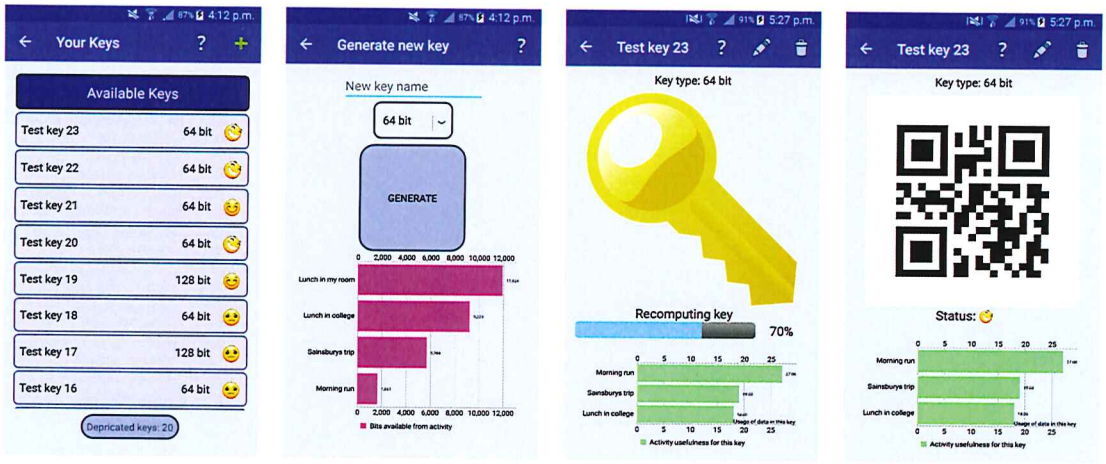
```
// Define the query
String selectQuery = "SELECT * FROM dummy table"
SQLiteDatabase db = this.getWritableDatabase();
Cursor cursor = db.rawQuery(selectQuery, null);

// Obtain and return the query results
ArrayList<DummyObject> dummyList = new ArrayList<>();
if (cursor.moveToFirst()) {
  do {
    DummyObject o = new DummyObject();
    o.setIntVariable(cursor.getInt(0));
    o.setStringVariable(cursor.getString(1))
    dummyList.add(o);
  } while (cursor.moveToNext());
}

// Avoid memory leaks
cursor.close();
```

6.4 Generated Keys

To view the generated keys, the user must click the second button on the home screen. This opens a new screen with a list of cryptographic keys the user has generated through the use of this application. Each element of the list displays the name of the key (decided by the user), and its strength (Figure 21a). As Emoji are commonly used by about 80% of people in the UK (The Telegraph, 2015), the strength of the key is displayed as an emoji which serves the added effect of giving the application a light, non-corporate feel.

To generate a key, the user clicks the green + icon in the action bar. This opens a new screen where the user can enter a name for the key, select the type of key and see a graph of some of the activities which are able to provide the most bits (Figure 21b). The user can enter a name for the new key and select a type of key, and then click generate. This will start the statistical analysis required to generate the bits of the key, and once this is completed, the user is automatically taken back to the list of keys, with the new key added.

If the user clicks on any one of the list items, they are taken to a page where the key is first regenerated (Figure 21c) and then displayed (Figure 21d). The user can edit the name of the key by clicking the pencil icon in the action bar. Although the key itself is not displayed as text, a QR code of the key is shown (see Section 6.4.1), with a barchart below it (see Section 6.4.2) showing the activities from which the bits for this key were derived. This way, if a user wants to stop an activity, they know which key(s) will no longer have access to fresh data. When a key deteriorates to the point that it becomes unusable the user will see a sad or angry emoji as its status (depending on the level of degradation), and can deprecate the key by clicking the bin icon in the action bar.



(a) (b) (c) (d)

Figure 21: (a) Existing cryptographic keys. The emoji on the right show the reliability of the key. The "Deprecated keys" text at the bottom is clickable and takes the user to a list of deprecated keys, similar to this page. (b) Clicking the green + icon in (a) opens a screen which allows the user to edit details of the key to be created. (c) When an existing key is selected from the list in (a), the corresponding key is regenerated, and (d) is displayed as a QR along with further details.

Deprecating a key frees the sensor data that was involved in its creation, so that the new data (which is clearly different enough from the original data) can be used to generate new keys. Clicking the "Deprecated keys" text below the list of keys shows a list of the deprecated keys. Clicking any one of these keys has the same effect as clicking an active key, but these cannot be edited.

6.4.1 Displaying the QR code

The QR code is displayed as a regular image, which is generated using the ZXing ("Zebra Crossing") library. To do this, the cryptographic key is converted to a `String`, and this is used to create a `BitMatrix` using the ZXing library. This `BitMatrix` is then easily converted into a Bitmap, which is the image displayed as the QR code. We perform the task of converting the key into a Bitmap image in a background thread to avoid an ANR dialog (see Section 6.6).

6.4.2 Displaying Charts

For this section, I will use a barchart as an example, but all other charts used in this application are displayed in a similar way as they are all from the MPAndroidChart library (PhilJay, n.d.).

Listing 6: Defining entries for a chart

```
1 ArrayList<BarEntry> entries = new ArrayList<>();
2 entries.add(new BarEntry(x, y));
3 BarData data = new BarData(entries);
4 barChart.setData(data);
```

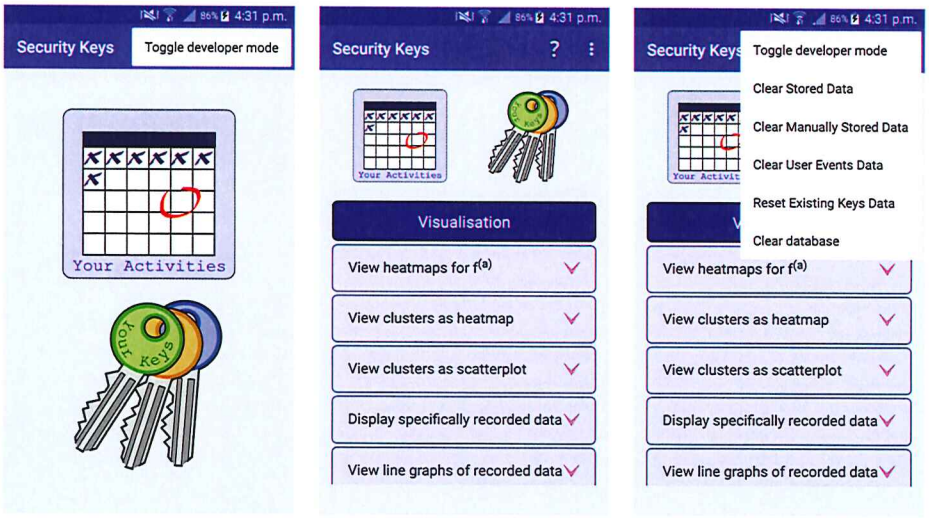
The first step is to include a barchart tag in the XML layout file for the screen on which to display the barchart. Next, Java code is added to create a list of entries and add these to the chart (Listing 6).

Here `x` and `y` are placeholders for `floats` representing the x and y values of the datapoints. This completes the minimum code required for a basic barchart. Extra options can be added, either for aesthetic purposes (e.g. changing the bar colour, size of points on a scatter chart, or adding animation) or for added detail (e.g. labelling the axes or specifying which gridlines to draw).

An issue with the library causes an error if datapoints are added to the chart in an unsuitable order. For some chart types, if the `entries` list above is not sorted in increasing order of x-value, a `java.lang.NegativeArraySizeException` is thrown. The documentation states this only happens for line charts, but actually the problem arises for scatter charts and bar charts too, with no helpful indication of how to solve it. This insufficient documentation resulted in this bug being a large time-sink even though it stemmed from a minor issue.

6.5 Developer mode

As shown in Figure 17, the user is only shown two buttons on the home screen. However, for the purposes of developing this application, I included more hidden buttons so I could perform unit testing throughout the development process. The home screen menu contains a "Toggle Developer Mode" option (Figure 22a). Upon clicking this, the user can scroll through 14 more testing options, including an option to test the k-means clustering algorithm implementation, an option to test the data recording done by the sensors, and an option to display recorded data as basic line charts (Figure 22b). The developer can click the arrow on the right of these options to view a description of the item, or click the item itself to be taken to a testing page. Developer Mode also changes the menu so database tables can be reset to a state required for testing (Figure 22c). For more detail on tests performed using these buttons, see Section 7.2.



(a) (b) (c)

Figure 22: (a) A "Toggle developer mode" option is included in the menu for debugging purposes. (b) Clicking the button shows various hidden options used for testing. (c) The menu is also expanded when in developer mode.

6.6 Threads

Many of the processes used in this application take too much time to be done on the UI thread (also called the main thread). When a task is being performed on the UI thread, the screen effectively freezes - the user cannot interact with the screen at all. Therefore, if a process on the UI thread takes longer than about a second, it should be performed on a background thread to avoid blocking the main thread. However, even if this is not done, it is impossible for the UI to stop responding entirely due to a failsafe in the Android operating system. This failsafe is the ANR dialog ("Application Not Responding") (Google, n.d.) (Figure 23) and allows the user to decide whether to allow the lengthy operation to continue or to force the application to close.

6.7 Design Patterns

When writing code in an Object Oriented language, it is always a good idea to follow common design patterns, as well as to adhere to the four pillars of Object Oriented Programming. One design pattern I have used is the Factory Pattern to instantiate classes for the statistical analysis on the recorded data. Another interesting pattern is the Null Pattern which I have used when a database lookup fails to return a relevant value, in which case I return a Null object rather than the `null` value. Moreover, I have carefully implemented inheritance to reduce the complexity of my code; I am able to call methods defined in an interface or an abstract class, which are performed by the implementation in a concrete class.

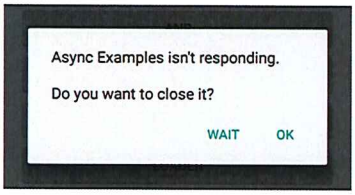


Figure 23: If an application is unresponsive for too long, an ANR dialog appears. (Google, n.d.)

As stated throughout this report, all lengthy tasks are performed on background threads. This ensures the application continues to be responsive and provides a smooth user experience.

7 Testing

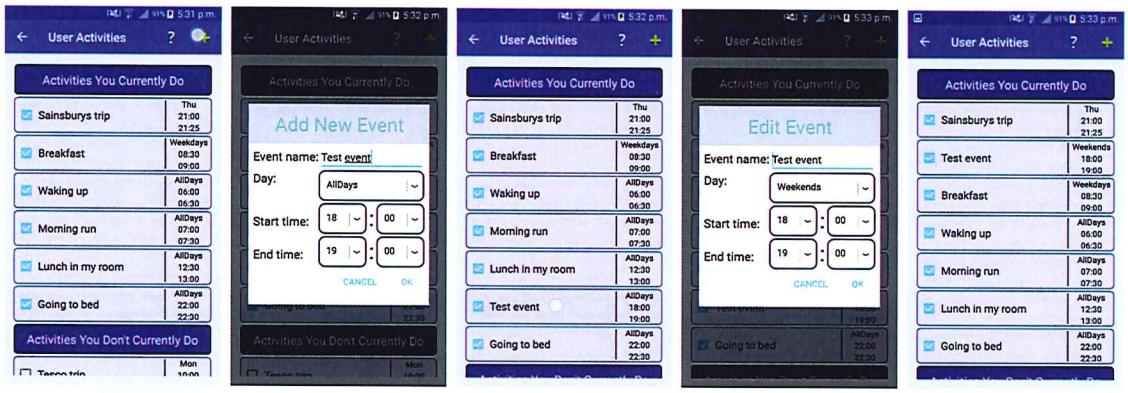
The following section details the testing strategy and shows the results of testing. All testing for this mobile application has been carried out on a Samsung Galaxy S5 running Android version 6.0.1. The white dot described in Section 6 appears in many of the following screenshots to show where the screen was being touched when the screenshot was taken.

7.1 Test Set 1

The following tests are all performed by simply using the application as an end user would. However, some tests, e.g. test 4, will need to be verified by checking the logs as a regular user would not be able to verify them.

Test 1: Do all buttons, list items, and menu options take the user to the correct screen or open the correct dialog? All buttons have been tested multiple times over the course of the development process and they all perform the expected function.

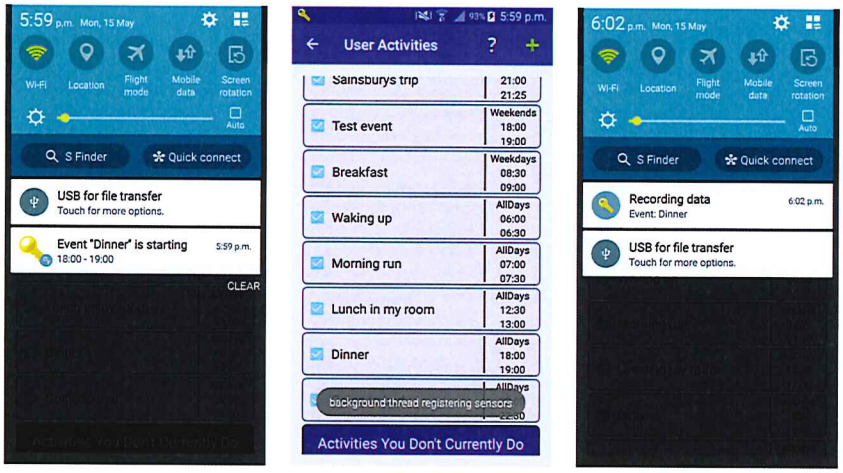
Test 2: Is the user able to add and edit activities they perform?



(a) (b) (c) (d) (e)

Figure 24: (a) Clicking the green + icon brings up (b) an input form where the user can enter details about the new activity. (c) The list of activities is updated dynamically to show the new activity, which can be clicked to show (d) an input form to edit the activity details. (e) Clicking OK saves the new data, which updates the list again.

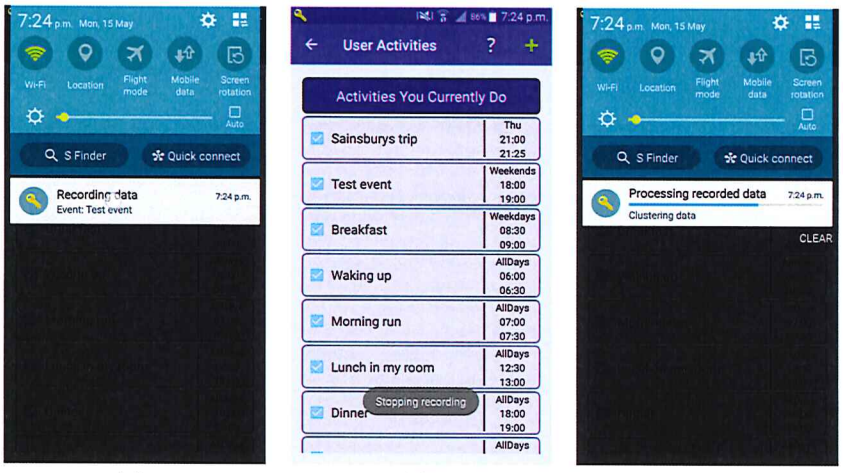
Test 3: When an activity is scheduled to begin, does the user receive a notification to start recording data?



(a) (b) (c)

Figure 25: (a) This notification appears one minute before an activity is scheduled to begin. The device also plays a tone, and the notification LED lights up as orange. Behind the darkened notification dropdown, the Dinner event can be seen as starting at 6pm, and the notification timestamp shows it was displayed at 5:59pm. (b) Clicking the notification shows a toast confirming sensors are recording data and (c) places a permanent notification in the status bar while data is being recorded.

Test 4: Is data recorded and stored when an activity is performed?



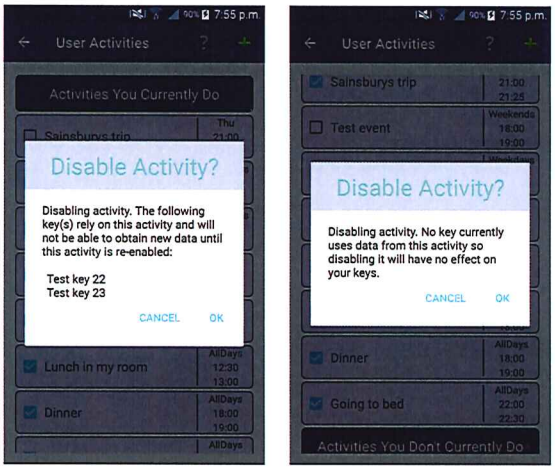
(a) (b) (c)

Figure 26: (a) Clicking the permanent notification stops data recording and (b) displays a toast notifying the user the sensors are being deregistered. (c) It also replaces the notification with another one which displays the progression of statistical analysis on the data and storing data in the database.

Listing 7: A few lines of the log showing database stores are completed successfully when data recording is stopped

```
cluster stored: DBClusterEntry{activityID=19, run=5, sensorID=1,
  centroid=(15.82, -10.55, 13.22), std_dev=5.479272121508821, clusterSize=30}
cluster stored: DBClusterEntry{activityID=19, run=5, sensorID=1,
  centroid=(-18.73, -3.39, -1.44), std_dev=4.077403401319759, clusterSize=29}
cluster stored: DBClusterEntry{activityID=19, run=5, sensorID=1,
  centroid=(-4.02, -9.80, 3.55), std_dev=4.3924547456504355, clusterSize=44}

sensor event stored: DBSensorEventEntry{activityID=19, run=5, sensorID=9,
  point=(-5.97, 1.92, 7.54)}
sensor event stored: DBSensorEventEntry{activityID=19, run=5, sensorID=10,
  point=(5.75, -1.88, 2.26)}
sensor event stored: DBSensorEventEntry{activityID=19, run=5, sensorID=1,
  point=(-0.25, 0.05, 9.77)}
```

Test 5: When a user stops performing an activity, are they notified of the keys this will affect?



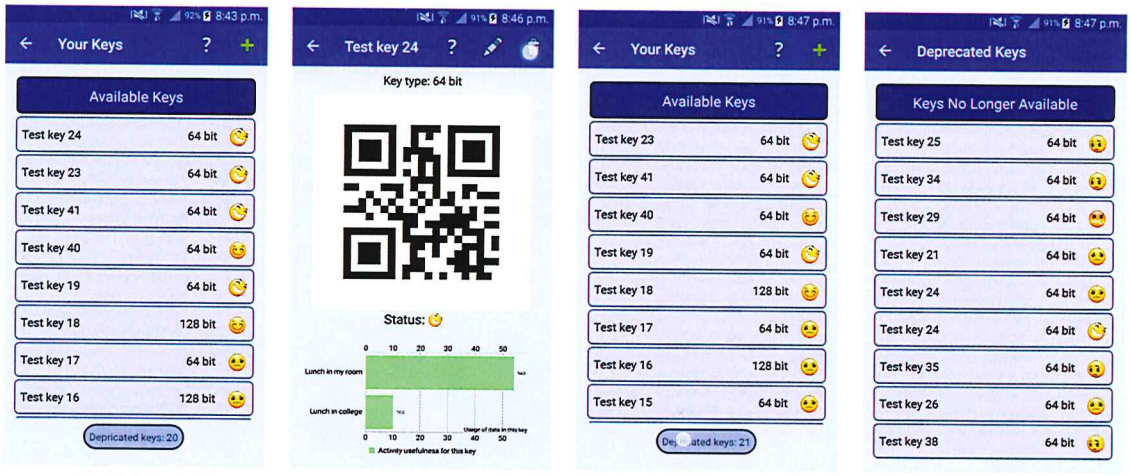
(a) (b)

Figure 27: (a) Unticking "Sainsburys trip" shows a list of keys which use data from this activity. (b) Unticking "Test event" displays a different message as this activity is not used in any key.

Test 6: Is the user able to generate cryptographic keys? This test matches requirements 1 and 2. It can be seen from the graphs displayed below the QR codes that the keys are generated from the data recorded from sensors, satisfying requirement 1. In addition, Figure 28e shows the key being regenerated before being displayed, proving it is not stored, but instead recomputed when needed.

Figure 28: (a) The current list of keys, with the New Key button clicked. (b) The screen for entering details for the new key. (c) Key generation progress is shown in a progress bar at the bottom of the screen. (d) Once key generation is complete, the user is automatically taken to the updated list of keys. (e) Clicking the new key starts the key regeneration process. (f) The key is displayed as a QR code above key metadata.

Test 7: Is the user able to deprecate keys?



(a) (b) (c) (d)

Figure 29: (a) The current list of keys, note the existence of "Test key 24" and the number of deprecated keys at the bottom is 20. (b) Clicking the trash can icon deprecates the key. (c) After deprecating the key, it is no longer shown in the list, and the number of deprecated keys is incremented. The "Deprecated keys" can be clicked to reach (d) a list of deprecated keys including the newly deprecated "Test key 24".

Test 8: Is the key generated the first time the same as the key generated after collecting more data for the related activities? This can only be tested by checking the logs written when a key is first generated and when it is regenerated. I carried out this test for multiple keys and found them to be unchanged even after collecting more data. This satisfies requirement 3, as the key has been reconstructed successfully even after adding some variance to the data by performing activities again.

7.2 Test Set 2

The next set of tests require developer mode. This is explained in Section 6.5. All of these tests have been performed as unit tests while developing the application, but have been redone to document them.

Test 9: Are all sensors able to record data?

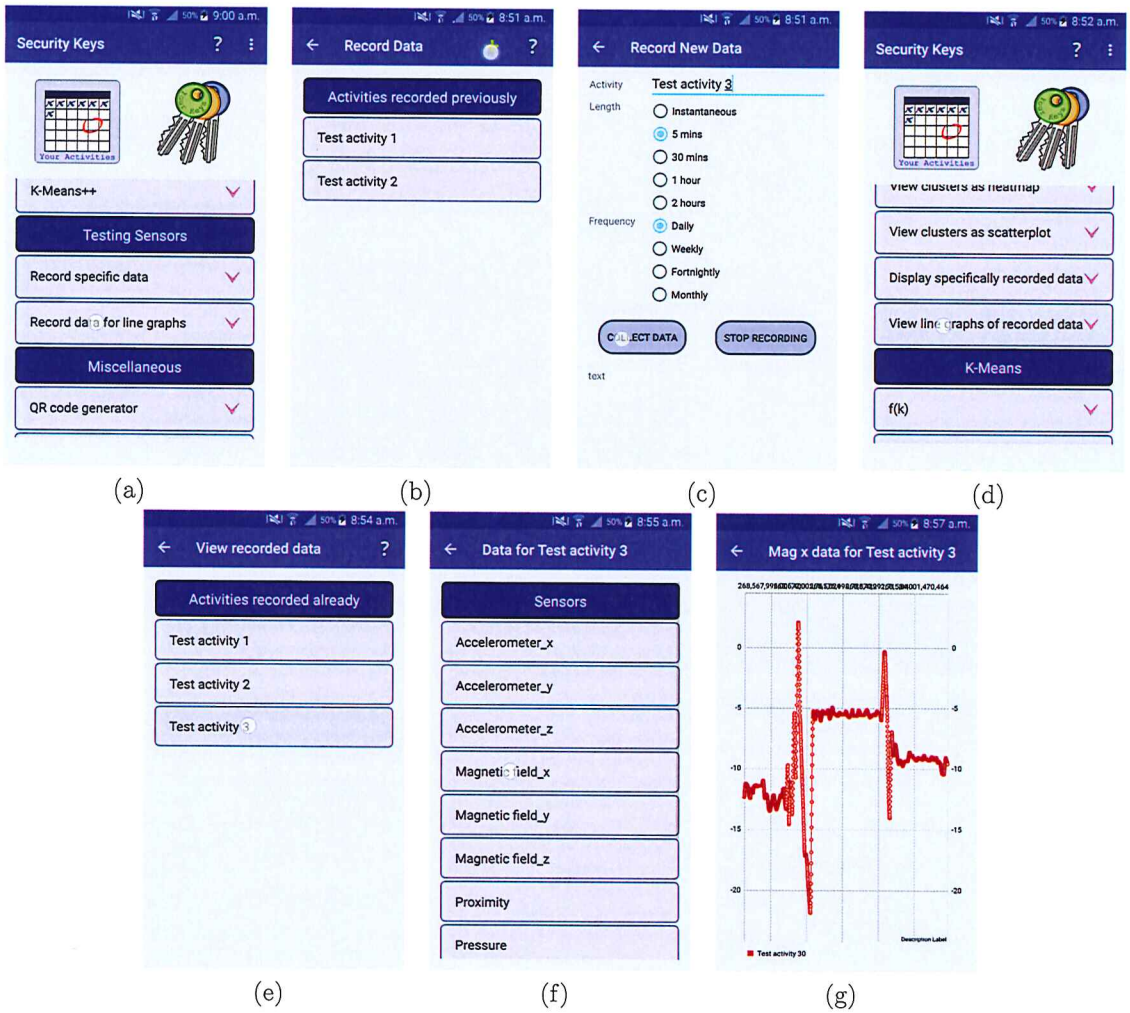
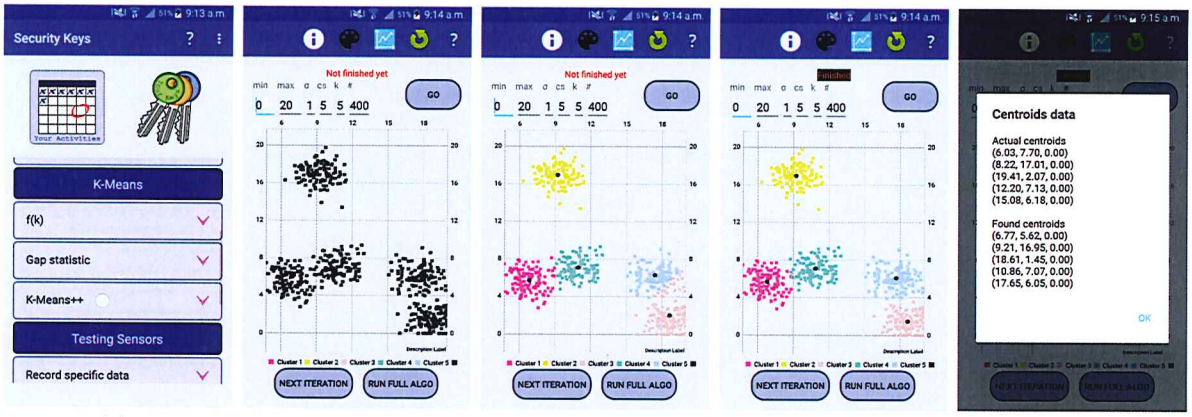


Figure 30: (a) Clicking "Record data for line graphs" in developer mode. (b) A list of activities for which data has already been recorded through this method. Clicking one of these starts recording for that activity; clicking the plus icon records data for a new activity. (c) Starting the recording. (d) Clicking "View line graphs of recorded data" (d) A list of activites for which data was recorded this way, note the new addition of "Test activity 3". (e) A list of all sensors (continues off the screen). (f) The line graph for one of the sensors for "Test activity 3". The other sensors also display their graphs correctly but their screenshots are omitted.

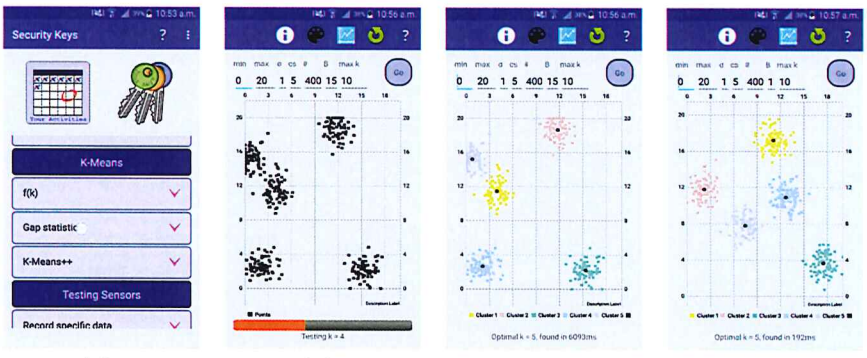
Test 10: Does the implementation of the k-means++ algorithm cluster data as expected



(a) (b) (c) (d) (e)

Figure 31: (a) Clicking "K-Means++" in developer mode. (b) The true centroids and clusters are generated randomly with respect to the parameters at the top of the screen. (c) The initial clusters before running any iterations of the algorithm. The heuristic used by k-means++ is good enough to already have most of the points classified correctly. (c) The result after running the algorithm. (d) Comparing the actual centroids with the centroids found by the algorithm.

Test 11: How do the Gap Statistic and $f(k)$ method compare for finding an optimal k for the k-means algorithm? I performed both algorithms on randomly generated clusters 30 times to find the average time taken and the number of times an incorrect k was found. Correctness was measured by eye in case multiple clusters overlap, i.e. although we expect 5 clusters, if two of them are very close together, we effectively end up with four. The Gap Statistic method found the correct k 27/30 times and the $f(k)$ method found the correct k 26/30 times, so perform equally reliably. Timings show $f(k)$ is superior by far.



(a) (b) (c) (d)

Figure 32: (a) The two methods of finding k are tested using the respective options in developer mode. (b) The optimal k is found for randomly generated clusters. While the algorithms are running, the user sees a progress bar. (c) The Gap Statistic method took an average of 5.94 seconds over 30 runs. (d) The $f(k)$ method took an average of 0.18 seconds over 30 runs.

8 Conclusion

This project has shown it is possible to generate cryptographic keys using user-specific data. These keys can be used by the owner of the mobile device to encrypt the data on their device, or in place of a password, with negligible effort from the user. Even a user whose lifestyle does not have many regular events can benefit from this application by recording the few events that happen to be regular, e.g. waking up at a certain time or having lunch at a certain place. Of course, such a user will be able to generate fewer keys than a user with a more regular lifestyle, but the keys they do generate will be just as secure as if their entire lifestyle was regular. Moreover, the application even encourages users to be more active if there are insufficient bits to generate a key. Also, the keys are specific to each user, so cannot be easily forged. The application is easy to use and provides a graphical user interface to make it accessible to a layman. In addition, time consuming tasks are performed on a background thread, so they do not restrict the flow of the application.

The keys generated have been shown in the testing section to seem like a random combination of high and low bits, but can be reconstructed accurately as long as the user does not drastically change their habits. There is of course natural variance in the recorded data provided by the fact that a human cannot be expected to repeat an activity perfectly every time. However, this variance is not high enough to affect the security key, as the preparation for generating a key described in Section 5.1.7 accounts for this variance. In the tests, the recomputed key was the same as the originally generated key even after an activity had been performed multiple times since the key generation.

8.1 Possible Improvements

Although all of the aims of the project were met, there are improvements that could be made to my methodology and implementation. This section discusses improvements I would make if I were to redo this project, or a similar one in the future.

One of the most important parts of any project is the design section. At the start of this project, I drew up the designs shown in Section 4.2 and made rough notes regarding the implementation and how various classes would fit together. However, these were not thorough enough and I should have spent much longer fleshing out more detailed designs in order to make the implementation easier. The lack of comprehensive designs meant there were instances where I was forced to rewrite code because, wasting valuable time.

Another time sink stemmed from the lack of comments in the code I had originally written. When I first started writing the code for the application, I failed to add comments as I felt at the time that the meaning of the code would be obvious at a glance. However, after taking a break from the coding for three weeks, I came back to the project to find I was struggling to understand the flow of the application. Therefore, before adding any new code, I added comments to all of the code I had written until that point, separating similar methods by whitespace (e.g. methods to affect the interface were close together and methods to react to interactions from the user were close together, but both of these types of methods were separate from each other). Luckily, as this point I had not been working on the project for very long so this did not take as much time as it could have had I been further along. The next time I took a break from the coding, I came back to find I easily understood the stage I was at and the next steps. In the future, I will be mindful to write such readable code from the beginning of any project.

8.2 Personal Development

Through the course of this project, I have developed my skills in various aspects of Computer Science. As expected, I am now much more familiar with the quirks of Android applications. For example, the graphical user interface is defined using XML, and there are many Java classes specific to the Android API which I am now comfortable using. This is demonstrated by the fact that I have constructed a large application which runs smoothly, has not been found to run into errors, and uses background threads to perform arduous tasks.

Finally, I have further developed my knowledge of many courses I have taken by actively applying what I already know to the project. I have developed my skills in Object Oriented Programming through the use of good design patterns, as discussed in Section 6.7. Furthermore, The graphs and heatmaps displayed in developer mode shown in the testing section tie in directly to Visual Analytics, as does the K-means Clustering algorithm implementation described in Section 5.1.1. Coincidentally, this algorithm also links to Machine Learning, of which I have substantially improved my knowledge by understanding how the application analyses the recorded data to predict a range in which we expect the most datapoints to be recorded. Another course that links directly to this analysis is Probability And Computing, from which I applied my knowledge of random variables and manipulation of probabilities to reach conclusions about expected values. Finally, the Databases course was invaluable in this project as all of the data is stored in an SQLite database, for which I had to draw on my knowledge of SQL queries as well as query optimisation, such as through indexing columns and using `EXIST` instead of `COUNT(columnName)>0`.

8.3 Future Work

There is currently no other application available for Android devices which generates cryptographic keys in this way, but there is still room for many new features to be added. This application was simply a proof of concept to show this method can be used to generate cryptographic keys, so there are possibilities for future expansions. The most obvious of these possibilities is to expand the application to work on and in conjunction with a wider range of devices so it has access to more sensors. For example, Android Wear devices have a range of sensors that would be useful to record data which could be sent back to the main device for processing and key generation.

Another future development could be to make the application play a more active role in ensuring the user is performing the activities rather than having them faked by an attacker. For example, if the user has scheduled activity A for a certain time interval, some time during that interval a notification could appear asking the user to confirm they are performing the activity. This would be something simple, such as "You are scheduled to be performing [activity A]. Please take a photo to prove you are at the correct location for this activity." The user would then be taken to a camera internal to the application (similar to how messaging applications allow pictures to be taken from within the application) and would take a picture of their surroundings. This would be analysed in terms of light intensity, colours used, shapes in the image, faces present, or many other possibilities. There is no need to restrict this to only pictures, as it would also work with sound recordings, analysing features such as the pitch, timbre, or volume for background and foreground sounds. This would require reliable analysis of images and sounds and was outside the scope of this project, but is a possibility for the future.

9 References

Arthur, D. and Vassilvitskii, S. (2007, January). 'k-means++: The advantages of careful seeding'. Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 1027-1035. Society for Industrial and Applied Mathematics.

Brownlee, Jason (2016), 'Supervised and Unsupervised Machine Learning Algorithms', Machine Learning Mastery [Online]. Retrieved from: http://machinelearningmastery.com/
supervised-and-unsupervised-machine-learning-algorithms/ [Accessed 23 April 2017]

Google (n.d.). 'Keeping Your App Responsive', Google Developer training article [Online] Retrieved from: https://developer.android.com/training/articles/perf-anr.html [Accessed 20 December 2016]

Gove (2015). 'Using the elbow method to determine the optimal number of clusters for k-means clustering' [Online]. Retrieved from:
https://bl.ocks.org/rpgove/0060ff3b656618e9136b [Accessed 4 February 2017]

Pham, D.T., Dimov, S.S. and Nguyen, C.D. (2005). 'Selection of K in K-means clustering'. Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science, Vol. 219, No. 1, pp.103-119.

PhilJay, (n.d.). 'MPAndroidChart', Github repository [Online]. Retrieved from:
https://github.com/PhilJay/MPAndroidChart [Accessed 31 January 2016]

The Telegraph (2015), 'Emoji is Britain's fastest growing language as most popular symbol revealed' [Online], Retrieved from:
http://www.telegraph.co.uk/news/newstopics/howaboutthat/
11614804/Emoji-is-Britains-fastest-growing-language-as-most-popular-symbol-revealed.html [Accessed: 14 March 2017]

Tibshirani, Robert, Walther, Guenther and Hastie, Trevor, (2001). 'Estimating the number of clusters in a data set via the gap statistic'. Journal of the Royal Statistical Society: Series B (Statistical Methodology), Vol. 63, No. 2, pp. 411-423.

ZXing, (n.d.). 'Zebra Crossing', Github repository [Online]. Retrieved from:
https://github.com/zxing/zxing [Accessed 28 January 2011]