Key Recognition in Android using OpenCV

Sheikh Arouge Mehdi
B.A.I. Engineering
Final Year Project, April 2013
Supervisor: Dr. Fergal Shevlin

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

_________________________________________  _________________________________
Name                                                Date
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this project upon request.

______________________________  ______________________________
Name                          Date
Acknowledgements

I would like to thank Dr. Fergal Shevlin for agreeing to supervise this project, for all his guidance and help throughout, without which these past six months would have been much more difficult.

I would also like to thank my friends and family for their support during this time.
Key Recognition in Android using OpenCV

3rd April 2012

Abstract
This project’s goal was to develop an Android application, which would record the video of a key. Using the OpenCV computer vision libraries for Android, the aim was to determine the model number of the key using algorithms based on several different factors of the key (length, thickness, angle, etc.). The test app ended up applying two of these factors correctly; however, more factors need to be added for a successful application.
# Table of Contents

1. **Introduction** .................................................................................................................. 6  
   1.1 What is the report about? .............................................................................................. 6  
   1.2 The issue at hand ......................................................................................................... 6  
   1.3 How will it be tackled? ................................................................................................. 7  
   1.4 What are the project goals? .......................................................................................... 8  

2. **Software** ......................................................................................................................... 10  
   2.1 Android......................................................................................................................... 10  
   2.2 Camera Intent.............................................................................................................. 10  
   2.3 OpenCV & OpenCV4Android ..................................................................................... 11  
   2.4 JavaCV ......................................................................................................................... 11  
   2.5 Eclipse Configuration ................................................................................................. 12  

3. **Video Processing** .......................................................................................................... 14  
   3.1 Frame Extraction .......................................................................................................... 14  
   3.2 Pixel:cm Ratio ............................................................................................................ 15  
   3.3 cvCanny ....................................................................................................................... 17  
   3.4 Key Length .................................................................................................................. 18  
   3.5 Angle Measurement .................................................................................................... 20  
   3.6 Database/Key Model Matching .................................................................................. 22  
   3.7 Displaying Results ...................................................................................................... 23  

4. **Application Evaluation** ............................................................................................... 25  
   4.1 Analysis ....................................................................................................................... 25  
   4.2 Performance ............................................................................................................... 26  
   4.3 Issues Faced .............................................................................................................. 27  
   4.4 Usability Features ...................................................................................................... 28  
   4.5 Bugs ........................................................................................................................... 31  
   4.6 Security Considerations ............................................................................................. 32  

5. **Conclusions** .................................................................................................................... 36  
   5.1 Personal Reflection ..................................................................................................... 36  
   5.2 Further Development ................................................................................................. 36  

6. **List of Figures** .............................................................................................................. 38  

7. **References** .................................................................................................................... 39  

8. **Electronic Resources** ................................................................................................... 40
1 Introduction

1.1 What is the report about?

The purpose of this project is to explain the approaches taken and the decisions made in the development of this application. The ultimate goal is to have an app (that can run on a variety of Android devices) which scans a key as a video, and then determines the model number of that key, by comparing it to a catalogue of a single brand (in this case, the Silca key brand). It will describe the algorithms used, and try to explain as clearly as possible what occurred at each stage of the process, both in the app itself, and the development over the past few months.

The remainder of the Introduction over the next few pages will give a better understanding of the problem; give a background on how the project aims to solve it, and what would be considered a successful result. Section 2 gives an overview of the various software being utilised, as well as the reasoning for choosing one environment, application or set of libraries over the other. Section 3 covers the application algorithms, and the Computer Vision processing that the app is based on. Section 4 is a summary of the final app, evaluating what was done well, what was done poorly, and what could have been improved upon and how. Finally, Section 5 will sum up the project as a whole, comment on what had been achieved, and discuss how this project could be expanded in the future.

1.2 The issue at hand

In order to cut/copy a key, the key-cutter needs to know the model number of the key being cut. That way they can select an appropriate blank, with which to copy the key. However, the key-cutter will usually have blanks from a single manufacturer, while the keys brought in are all different makes and models. While the common keys are simple enough to identify, there are hundreds of keys available, and it is just not possible to memorise all of them. The only way to match a key to a blank is to look through all available blanks manually until a match is found – a cumbersome task.

Therefore, to make this matching process bearable, a program/application that scans a key and then compares it to a key catalogue would be far more efficient. Moreover, while many stores which carry out key cutting do not have a computer & webcam to scan the key, most people have smartphones, so it allows for a powerful, easily accessible solution.
1.3 How will it be tackled?
When copying a key, the key-cutter needs to first figure out which key blank to use. Since not all manufacturers have the same model number, the key needs to be examined. There are several factors before you can make a decision:
1) Length of the key
2) Shape (angle) of the tip
3) Head shape
4) “Identifier” symbols on front and back (shown in figure 1.3 (b) below)
5) Thickness of the key (not intended to be a part of analysis)

In most real world cases, using the identifier on the front and the length will be enough to determine the key (though sometimes you would also need to check the angle of the tip, or the identifier on the back). In some cases, the head shapes can be unique to a certain type, so only head shape and length would be required. However, in all situations, getting the length of the key is paramount – without length, you can get into the correct “family” of keys, but not necessarily the matching one. Therefore, it was clear that being able to calculate the length of the key in the video was going to be the number one priority for the application. Otherwise, while the application could analyse the “type” or “family” of the key, it would not be able to distinguish further to pick out a single model.

Deciding the next step was a choice between the angle of the tip of the key, and the shape of the key head. The reason to choose the angle of the tip was in order to differentiate between two otherwise similar keys. You could get some keys with the same length, same thickness, similar head shape and similar identifiers (on one side at least), and the only way to tell one from the other is to either check the identifiers on the back (meaning a second video would need to be recorded and analysed), or the profile of the tip. Alternatively, analysing head shape would allow unique or uncommon keys to be identified quickly (for example, model UNI 3, as shown in figure 1.3 (a) below).

If the project could recognise this shape, it would only need to calculate the length of the key in order to determine the model number. In this example, UNI 3 is not part of a family; it is the only key in the Silca range with this head shape, so length would not actually be required, but for most uncommon head shapes, there is a range of lengths. Therefore, head shape would be a good option if the goal were to get the widest range of keys analysed. The trade-off is that not only are these “unique” keys less commonly used (so the application would not increase the useful size of its database), but most key-cutters can find key models of unique shape by eye. If no other key in the world has this head shape, then there is very little need for an application to say that.

The application ended up prioritising the angle of the tip (rather than the head shape) after the length. The reasoning was that while analysing the head shape would give better results, it would only be for uncommon keys. Moreover, in the case of truly unique ones, such as UNI 3 above, there is very little need for an app – you can tell the shape by eye. In addition, while most
head shapes will be similar across manufacturers (non-Silca models of UNI 3 would still have the same head shape), that is not true in all cases. So going with the angle of the tip would allow for greater a better analysis of common keys, and would be consistent across multiple manufacturers.

As mentioned earlier, the identifier symbols (shown below in figure 1.3 (b)) are the primary way to determine model number by eye. However, it would be a particularly difficult criterion to measure. As will be shown later in Section 3, getting a good image of the symbols was not always possible, depending on the lighting. In addition, when using the Canny edge-detector (Section 3.3), the lighting caused additional edges to be found within the grooves themselves. That, and along with the fact that a second video would be needed for the reverse side, meant that the project would focus on the earlier factors (length, tip angle, head shape), before attempting to work with the symbols.

1.4 What are the project goals?

It would not be realistic to expect the application to recognise the whole catalogue of Silca keys - there are thousands available (as shown in figure 1.4 (a)). So the idea was to have as few key models as possible, and to ensure the application could accurately determine each of them, before looking at implementing a full (or fuller) database, closer to the full range of >5,000 keys.
Therefore, a successful application is one that will be able to accurately measure the length and measure the degree of the tip of a key, and show the results correctly on screen for the user. Additional features that were possible to supplement the application included the ability to view past results, showing the closest match in cases where no exact match can be found, and the ability to recognise when no key is available (as opposed to a key’s model number not being accurately determined). Ideally, the application would be able to produce a result in less time than it would take someone to look up the key manually, which can range from 5s to 60s. The aim is to get the process time down to ~20s if possible, though efficiency will not be a priority unless all four factors listed in Section 1.3 are successfully implemented. Usability features (like a menu with the option to delete temporary files, or define how many frames are analysed and at what interval to extract them from the video.

Overall, the goal is an application that can differentiate between the set of 3-5 keys accurately, and run the analysis in a reasonable period. It should optionally allow the user to specify how much of the video to analyse, and allow them the ability to view past results.
2  Software

2.1 Android

Android is the open-source operating system (OS) developed by Google for use in smartphones, tablets, and other similar electronic devices. The Android platform has 70.1% of the worldwide smartphone OS market share (compared to 21% from Apple’s iOS) as of Q4 2012. [1] The Android environment is a common ecosystem – a software compatible with a wide array of devices, so inter-operability is maximised. It also gives access to many input sources, such as cameras, sensors (gyroscopes, accelerometers, magnetometers, etc.) and touchscreen input, which give multiple options to developers to build applications.

2.2 Camera Intent

One of the first things that needed to be decided upon was how the video that would be analysed would be recorded. The Android environment allows developers to either access the camera directly – meaning they would need to configure screen previews, encoding rules and handle data – or to call an “intent”, a request to the device to carry out a certain task. [2] Therefore, there are various benefits and drawbacks to both options, which were all considered when making the decision.

One method is to access the camera directly. This does involve more work – first, the developer needs to handle the input stream and display it on the main screen (as a SurfaceView). This is the preview used to record the video. Along with this, the encoding settings also need to be defined, usually using the MediaRecorder class. [3] Some of the available options are the video encoding (H.263, H.264 and MPEG_4), audio encoding (AAC, HE_AAC, AMR_N and AMR_W), output format (.mpg, .mp4), frame rate, resolution, as well as other constraints such as the maximum duration (in ms) and the maximum file size (in bytes). It allows for some customisation down the line too – if, for example, the developer wished to have an outline of a key while recording a video (to help the user position the camera correctly), it would be simple to add an image on top of the SurfaceView. After all this, the video file needs to be saved to a location. This means the max size and max duration of the video (based on the available space on the SD card or internal storage) would need to be defined within the code.

The alternative is to use an intent in order to record video. The benefit of this approach is that the programmer does not need to worry about the camera specifics, nor how the phone handles the video file. All camera work is handed over to the device itself, allowing the application to tap into the default Camera application on the smartphone/tablet. So while some of the features that come with interacting directly with the hardware are lost, it means recording the video to be processed becomes a rather simple task, and more time/effort can be spent on the vision part of the application. However, since the device is in control of the recording it also saves the file itself. As with most digital cameras, it saves to the default DCIM folder, with a filename of VIDxxx, where xxx refers to how many videos have been previously saved. This would result in two recordings having different filenames, and no way to determine which filename is next in the list. [1] The way around this is to “hijack” the data stream and save to an intermediate location. The intent continues to save the file as normal (which can be used as a backup), and the app now has a consistent location from which to access the video.

I decided to use the intent in this project. While it was clear that this meant giving up some functionality (specifically, the ability to overlay an outline key shape to help the user centre the video), and could lead to encoding inflexibility (which it did), I did not think the lack of those options would affect development. The purpose of the video was simply to extract frames for processing; any processing or filtering could be done during the extraction period. So the intent

1 Technically Intent.putExtra (MediaStore.EXTRA_OUTPUT, fileURI), should allow you to define another location, but is bugged in the current version of the Android SDK.
was used, which still afforded the option (through the default app) to specify the resolution, which I felt would be enough customisation for the application, and could devote more time to the actual extraction and processing of the video.

2.3 OpenCV & OpenCV4Android

OpenCV (Open source Computer Vision) is a commonly used programming library, which contains over 2,500 optimised algorithms spanning the full range of Computer Vision techniques. The OpenCV libraries include the methods that will be used in the application, allowing for features such as Canny edge detecting, the Hough transform, as well as pixel manipulation and RGB value extraction.

The OpenCV libraries are implemented in C/C++, but since Android applications use Java, OpenCV needs to be compiled specifically for the ARM architecture (since most Android smartphones use the ARM architecture rather than the Intel x86 architecture computers do). One way to do this is to use the JNI (Java Native Interface), a sub-feature of the Java language, which allows code to be accessed from within a virtual machine (called Dalvik). [4]

![Figure 2.3 (a) – JNI Implementation](image)

An alternative was to use OpenCV4Android. OpenCV4Android initially started out as a port of OpenCV, run separately to the main project, and has since evolved to become a part of OpenCV as a whole (and as a result, gets updates alongside the main release). There are two levels to this distribution: basic and advanced. The basic level uses OpenCV’s Java API, which requires you to just import the classes and begin using them, without using the NDK. The benefit is that since all computations are performed natively, the “cost” is similar to one JNI call (though this can become an issue if there are too many calls to OpenCV).

The advanced level is a slightly more complex approach. In situations where basic level is too limited (you’re calling OpenCV multiple timers per image, and the JNI cost begins to become large), advanced allows you to merge all OpenCV calls into a single C++ class, and call that class per frame, instead of multiple OpenCV instances. It requires a more detailed knowledge of the Android NDK, and can be a bit daunting to begin with, but it gives access to all of the OpenCV API. This not only allows for the best performance, but, since it is in C++, it can be considered cross-platform too. [5]

2.4 JavaCV

In the end, the option settled on was to use JavaCV – the Java interface for OpenCV. JavaCV provides wrappers to the commonly used libraries in OpenCV (as well as other similar libraries, such as ffmpeg, OpenKinect, OpenGL, etc.), which allow developers to use OpenCV methods natively in a Java environment. It also comes pre-compiled for Java, so there is no need to use the Android NDK (Native Development Kit), and can instead be imported as if it were a standard Java library (this is done by importing the com.googlecode.javacv.cpp package). As with
the earlier decision with regards to using the camera directly, or calling an intent to use it, the trade-off here is that using the JNI allows for more customisability; it allows for modifications to the methods being used (the code would need to be recompiled in this case). JavaCV, on the other hand, is simpler to use, and requires very little modifications to access the OpenCV methods.

So ultimately, the ease of use of JavaCV (especially considering this was my first real development project, and certainly first using Java and Android) won out over the added features and flexibility of using OpenCV directly.\(^2\) The application began using OpenCV4Android (basic), but once I learnt about JavaCV, it was just a much simpler set of libraries to use. In addition to being a Java wrapper of the native OpenCV code (rather than going through an API), it allows the ability to select packages to import a bit more discretely. Rather than importing all of the packages that include the IplImage class, the developer could specifically import IplImage only (as an example). It also resulted in reducing the .apk file size down from ~20mb to less than 10mb.

2.5 Eclipse Configuration

The development environment chosen for this project is Eclipse Juno. In order to configure Eclipse for this project, three things needed to be installed:

1) Android SDK

The Android SDKs for the versions the application is being developed for (in this instance the lowest version supportable is 2.3.3, but the build target is the latest 4.2). This was relatively simple to achieve – once the SDKs are downloaded, Eclipse simply needs to be pointed to the correct directory.

2) OpenCV Libraries

As stated above, initially the project used OpenCV4Android, which required the OpenCV project to be imported into Eclipse, and set as a library for use by the main project. It also required .so files to be added to the project/libs folder, and specified whether it is for x86, ARM or ARMv7.

The library files for JavaCV need to be included in the project directory so they can be accessed at compile time. Once downloaded, the .jar files (javacpp.jar and javacv.jar, version 0.3, in this instance) need to be placed in the project/libs folder, which can be imported into the project. They are also cross-platform, so there was no need to have multiple .jar files for each potential platform on which the application could be used.

3) Android Device Drivers

For testing purposes, the test Android device (a HTC One S) needed to be connected and configured for the Eclipse environment. This was necessary to allow the IDE to access and run applications on the device directly. To do this, the device drivers needed to be downloaded from the manufacturer’s website (HTC). Once recognised by the computer (and debugging mode enabled on the device), Eclipse gives the option to install and run the application on the phone, rather than using an emulator.

Eclipse also allows the use of Android Virtual Devices (AVDs) for testing purposes. AVDs are essentially Android emulators, which can be modelled on devices of different hardware (screen resolutions, SD card space, CPU power, etc.), as well as different software (Android OS versions from the original v1.0 up to the latest v4.2 (Jelly Bean). This allows (limited) testing of an

\(^2\) As mentioned in Section 2.3, the ability to “merge” multiple OpenCV calls into a single call would go some way to improving the efficiency of the application. This would be an interesting route to take for further project development.
application of different devices to test interoperability, as well as how the device runs on different Android versions.

Once these three items were properly configured, it was a simple matter of compiling the code and directly installing and running the application on the device. There was also some testing done on AVDs in the early stage of the application, when the design was still being tweaked, and a small amount at the end to test how the layout was handled on larger devices (4.7” smartphones, or tablets, etc.). It was also used for error reports using LogCat.  

This is commented on more fully in Section 4.3.
3 Video Processing

3.1 Frame Extraction

The first step in processing the video is to extract the frames, on which we can perform the Computer Vision/OpenCV methods. To achieve this, we utilise the MediaMetadataRetriever class – an interface to extract frames and metadata from media (audio or video) files. [6] It allows access to information such as the media title, resolution, artist, and duration. Of these, the duration is of import – by knowing how many seconds in the video file, we can determine the maximum number of frames available to be extracted. Frame extraction is carried out with the following code:

```java
Bitmap bmp = vidFile.getFrameAtTime(100000*i, MediaMetadataRetriever.OPTION_CLOSEST);
```

The video file (saved on the SD card as “video.mp4” in the “KeyAnalyser” folder) is loaded into vidFile as a variable of class MediaMetadataRetriever. This lets us extract the information and frames. The `getFrameAtTime()` method extracts frames at a given millisecond; since we wish to extract every 0.1s, this value is set to go up in increments of 100,000, depending on how many times the application loops (which is user defined). The OPTION_CLOSEST flag define the type of frame to be extracted. To explain this option, it is necessary to explain the structure of the video file (mp4 format, using the H.264/MPEG_4 AVC video codec) first. The video has a frame rate of 29, meaning there should be one new frame approximately every 0.03 seconds. However, video codec do not generate these frames equally – the majority are “standard”, while one frame in every 50 or so will be of higher quality. This frame is known as the “keyframe” or “i-frame”. By default, the `getFrameAtTime()` method will attempt to extract the keyframe, which is closest to the value of milliseconds entered. If the extraction interval is too low, you could end up getting multiple frames, but they would all be the keyframe. In order to change this, we use the OPTION_CLOSEST flag, which allows you to extract the frame at the exact time specified, regardless of whether it’s a keyframe or not.

(It should be noted that since we are using the camera intent, this is something that I needed to accept. As I stated earlier, letting the device handle the camera interaction also means I have very little say in the recording of the video. If I was accessing the camera directly, it is likely that I would have been able to change the interval i-frames are created, since I would be able to change encoding settings for the camera. The fact that I could not do this did lead to an issue concerning extracting non i-frame frames from the video, which will be discussed more fully in Section 4.5.) Once the frame has been extracted from the video, it is saved as a Bitmap on the SD card in “KeyAnalyser\1_raw_images”. This is only for development/debugging use, since for the completed application, there is no need to store the image file. Converting it for use with OpenCV (either cv:Mat, or IplImage, as I use) would suffice, as it could be processed directly, rather than reloading the same image into an IplImage variable.

Figure 3.1 (a) – Sample extracted frames

*Based on empirical testing, not from documentation*
Above are examples of the extracted frames, taken at 0s, 0.5s and 1s, to show the differences in frames as the camera “swipes” over the key. With the 0.1s extraction rate, these images would be #0, #5 and #10 respectively, with the intermediary frames showing less significant changes. One of the options being considered was that longer videos would have less frequent extractions. So a 5s video would extract, for example, 10 frames, one every half second, while a 20s video would extract 10 frames, one every two seconds.

However, it was decided that having more frames (certainly at this stage of the application’s development) was a better idea, to give as many sources of information as possible while refining the calculations. Also, as will be discussed in Section 4.5, a 20s video doesn’t necessarily mean 20s worth of frames can be extracted, so changing the extraction interval may result in fewer frames available than would be ideal.

3.2 Pixel:cm Ratio

The first step in processing the extracted frames is to define the scale of the image. This is an essential step in determining the length of the key, which has already been concluded to be the highest priority in this analysis. Since the application can be run on a variety of devices, and each could potentially have a different resolution, a method independent of the hardware itself needs to be used.

The technique used for this is to have a known object in the frame. In Computer Vision, including an object of known shape, size, colour or orientation can be referenced and used to understand the rest of the image. For this project, this is achieved by specifying that the key must be recorded on a sheet with a red rectangle printed on it (as shown in figure 3.2 (a)).

![Figure 3.2 (a)](image)

Figure 3.2 (a) – The sheet on which key must be recorded.

The size of the rectangle could be as large or small as the developer wishes - as long as it allows all keys to fit within it, and leaves some white space at the edge.\(^5\) Since the longest keys are ~7cm in length, adding 5cm above and below resulted in the height of the rectangle being set as 17.5cm (the width is not important for the application so is not specifically defined).

---

\(^5\) This is because when recording the video, the user must include the entire red rectangle, but not go beyond the bounds of the white space. The more white space available, the easier this becomes.
The developer also has the option of which colour to use for the rectangle. In this situation, rather than choose a colour with a mix of RGB values, I decided to use only red (255, 0, 0), green (0, 255, 0) or blue (0, 0, 255). The program also recognises these red pixels (as opposed to the white pixels of the background, or the non-red pixels of the key) using “thresholding”. Thresholding in this case means that if the RGB values of a certain pixel are within a certain range, it can either be considered to be a part of the rectangle or not. While testing each option, it became clear that red was the best of the three colours. While blue stood out of the page more than red, it was difficult to detect efficiently in darker lighting conditions. Similarly, green (while not as distinct from the page) was not as easily picked up in situations where the lighting was strong. Red was a good compromise, in that it could consistently be recognised regardless of whether the lighting was good or poor. The next stage was deciding on the thresholding values. A value of 255 in the red channel does not actually mean the pixel is red, just that the red channel is saturated. If the green and blue channels were also 255, then the pixel would actually be white (and if the values were 255, 0, 255 the pixel would be magenta, etc.). It was less about the absolute value of the red channel, and more about the values of the three individual channels.

Before deciding on the thresholding values, I tested some extracted frames to see what the RGB values were when analysed. This is because while the image was strictly 255, 0, 0, it is unlikely the camera would pick up the printed image as such. Depending on the lighting, the red channel ranged from 140-190, while the green and blue channels ranged from 30-65. Since there is always the possibility of shadows becoming involved, rather than apply relative thresholding (pixel is red if green is less than 50% red and blue is less than 25% red) the application uses absolute values. Giving a buffer value of 10 on the tested data, the application will consider a pixel as red if the red channel is greater than 130 and the green and blue channels are both less than 75. This helps alleviate the issue with relative thresholding not staying true when there are both dark and light sections on a single image, but allowing enough leeway that there would not be any false positives.

The processing procedure by which the application determines the scale is described in pseudo-code below.

---

**Pseudo-code to calculate pixel:cm ratio**

- Load image
- Starting at first column, check each pixel, row by row
  - Tally the number of pixels that are red (defined by the threshold)
  - Set variable “maxRedCount” to this value
- Go to next column, repeat process
  - If this column’s red count is greater than previous maximum red count, set maxRedCount to this new value
- Else move on
- After last column, divide 17.5 (height of rectangle) by maxRedCount. This is the pixel:cm ratio (to be used later)

---

**End of pseudo-code**

For the remainder of the application, only the area within the rectangle is of interest. Therefore, during this section of the processing, the boundaries of the rectangle are also measured. As the image is traversed, the first column/row with at least 10 red pixels and the last column/row with 10 red pixels define the shape of the red rectangle. This allows later section of the application to limit the processing to inside that area, rather than the entire image, decreasing the runtime of the application.

---

Some would question the efficiency of using thresholding for an analysis like this; to determine the column with the highest amount of red pixels, every single pixel of the image needs to be analysed. This will be commented on later in Section 4.
3.3 cvCanny

The application’s next step is to pass the image through the Canny Edge-Detector (by using the cvCanny method in JavaCV). The theory behind edge-detection is that there are two forms of processing images which aren’t efficiently analysed using simple (binary) algorithms—they are generally either region based, or edge based. It analyses the image to attempt to identify the points in the image where the luminous intensity of the image changes significantly (i.e., a discontinuity has occurred). It then “marks” these edges as a single pixel, and outputs the image. The result is similar to what is shown below.

![Figure 3.3 (a) – cvCanny Example [7]](image)

Canny was developed with the three primary criteria in mind:

- Detection
- Localisation
- Minimal Response

The edge-detector will recognise as many edges as possible in an image (detection) and it would mark out edges it found in a position as close as possible to the edge in the raw image (localisation). One of the main benefits of Canny is that it has minimal response; it will attempt to mark an edge in the image only once, so the resultant image does not have multiple pixels marking a single edge. Another bonus is that Canny attempts to filter out image noise by convolving the image with a Gaussian filter before analysing, to reduce the number of false positives found.

Using the Sobel operator in the Canny detection means employing a gradient based analysis. The first derivative for both the X-axis and Y-axis are calculated independently, and then the derivative along the X and Y-axes are approximated by convolving with kernels below. [8]

![Figure 3.3 (b) – Sobel Kernels](image)
This analysis could also be further refined to determine the orientation of the edge by using the second order derivative, if required. The analysis of edges using first and second order derivatives are shown in figure 3.3 (c).

![Image](image.png)

Figure 3.3 (c) – First and second order derivatives [9]

The purpose of this step of the analysis is to reduce the amount of information in the image. From this point on the key is being analysed, but only its shape, not any of the features of the key itself. Therefore, it is important to find where the object begins and ends, the outline of the head, or the profile of the key tip. Canny allows the application to go from a full colour image to a binary image – where a pixel is either a background pixel, or an object pixel.

3.4 Key Length

Once Canny has been applied to the image, the application needs to ensure that the edges of the rectangle are not included. This can be an issue if the video is not straight when recording, so an offset is added to the boundaries measured in Section 3.2. This leaves a black background with the outline of the key in the centre (there will also be some edges detected within the key itself, but they can be ignored).

Since we have already calculated the pixel:cm ratio, we can now calculate the length of the key. We simply need to count the length of the key in pixels. To do this, we employ a similar algorithm as the one to count the pixels in the red rectangle. The difference is that for the rectangle, we needed to count the pixels per row; for key length, we can simple check each row to see if an object pixel exists. Finding the first and last rows that contain an object pixel should – ideally – define the length of the key. For the red rectangle, this would not always be true; on occasion, the rectangle could be rotated, giving a false measurement. The algorithm is defined in the pseudo-code as follows.
Pseudo-code to calculate length of key

- Load cvCanny image
- Start at the first row, and traverse each pixel by column
  - If a white pixel exists, set firstRow to current row
- Continue to next row, and repeat
  - If a white pixel exists
    - If firstRow is not set, set firstRow to current row
    - Else, set lastRow to current row
- If firstRow = lastRow, or lastRow is not set, key length cannot be determined -> handle failure (no key, or unknown key?)
- Else, calculate rowDelta = lastRow - firstRow
- rowDelta*pixel:cm = length

End of pseudo-code

Essentially, the application measures the first key row and last key row (figure 3.4 (a)) to calculate rowDelta – the height of the key in pixels. This is then multiplied by the pixel:cm ratio calculated in Section 3.2, which is the length of the key in cm. If no object pixel is found at all, then the firstRow variable will not be set – we can assume the video is invalid and no key is present. If the firstRow is set, but lastRow is not, then again, an assumption can be made that there is no key present in the video (or the entire red rectangle is not visible, causing the boundaries to be incorrectly defined). This is handled when comparing results to the database of keys in Section 3.6 below.

![Canny image and rowDelta calculation](image)

The application now has a value with which to decide on a key model from a small range of keys. However, for a complete application, it is not enough to decide between the multiple different keys with just the length – there are too many models available. At least one more factor is needed to differentiate between various models with any degree of usefulness.
3.5 Angle Measurement

The application next attempts to measure the angle of the tip of the key - the shape at the very bottom of the key – to further determine the model number. First, we need to limit the image further to isolate the tip of the key. To do this we will first use the values for the first and last red columns, from Section 3.2 when we were defining the rectangle boundaries. In addition, from each of those columns, we will contract the image by the a third of the width of the original image. This reduces the width to ~3cm of measurement, and since the largest keys have a width of 3cm, it should be sufficiently to show the tip of the key.

The row restriction is far simpler – the last row with an object pixel has already been calculated from Section 3.4. By including the row directly below this (so the key edge is not considered out of the bounds of the image), and then including the x rows above, the application can process just the remaining pixels. Assuming there are 20 pixels between the edge of the image to the red rectangle columns, this leaves 35*142 (4,970) pixels to calculate, compared to the full image of 544*960 (522,240) pixels. This is a reduction of 99% of the pixels, compared to the first part of the analysis. The section to be processed is shown in the red box in the below images:

The actual measurement of the angle was not ideally completed at the time of writing. The best measurement of the angle would be calculated by using the inner product, taking the ends of the two intersecting lines, and then measuring the angle between them at the point of intersection. In figure 3.5 (c), the end points of the tip are points A and B, while the point of intersection is point C, which is where the angle would be measured. However, I could not accurately get the value of the three points in order to calculate the inner product.

---

7 The test device used a resolution of 544*960, so width/3 would be ~181 pixels.
8 x is defined as 7*(height/200), which is 7*4.8 = ~34 rows.
The alternative was to use the lines available, rather than their points. Take the lines $|AC|$ and $|BC|$ from figure 3.5 (c) as the two lines of the tip. We will use the Hough lines algorithm from OpenCV to find the lines, and use those values to find the angle $C$. The Hough transform can be considered a way of taking the edge points form the Canny algorithm and forming some useable information from it. Canny results in a sequence of pixels corresponding to edges. While it is possible to loop through the pixels and form some understanding, it is a complex and inefficient task. The Hough Lines transform gives a geometric representation of those points, allowing the application to determine the slope or intersect points of the lines.

A line is essentially a collection of points, and handling a collection is a harder task than just a single point. Therefore, the first step is to represent a line as a single point, but while retaining all relevant information about the line. This is achieved by storing a line in “m-c space”.

![Figure 3.5 (d) – m-c Space](image)

The operation of the Hough transform is essentially to find points in the image (in x-y space), and convert them to lines in m-c space. As multiple points from the same line are mapped to m-c space, they will intersect. The intersection point is the m-c space representation of the line, including the parameters of the line. An example is show below in figure 3.5 (e).

![Figure 3.5 (e) – Example of m-c space](image)
Using the Hough Line transform, the application finds the lines \(|AC|\) and \(|BC|\). For each line, it can determine the angle to the X or Y-axis. We can thus find the tip angle by one of two methods.

1) If comparing to the X-axis, we will find the angle between \(|AC|\) and the X-axis, and \(|BC|\) and the X-axis. This the angle \(C\) would be calculated by \(180^\circ-(A+B)\).\(^9\)

2) If comparing to the Y-axis, we would find the angles \(X\) and \(Y\) in figure 3.5 (f) below, and would simple need to add them together to find the angle \(C\).

![Figure 3.5 (f) – Hough lines angles](image)

This value would then be averaged out for every frame analysed, resulting in the degree factor for database matching in Section 3.6.\(^{10}\) The efficiency and accuracy of this measurement will be commented on in Section 4.1 later.

### 3.6 Database/Key Model Matching

Now that the application has two values to work with, it attempts to choose the relevant key model from the database. Since there are only two factors, there would be no point in creating a database of dozens of keys. In the end, the database was made with four key models:

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Length (cm)</th>
<th>Angle of tip (°)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS 2</td>
<td>4.2</td>
<td>90</td>
</tr>
<tr>
<td>UL 050</td>
<td>5.5</td>
<td>87.5</td>
</tr>
<tr>
<td>UNI 3</td>
<td>5.75</td>
<td>97.5</td>
</tr>
<tr>
<td>UL 054</td>
<td>6.0</td>
<td>92.5</td>
</tr>
</tbody>
</table>

![Figure 3.6 (a) – Key models](image)

\(^9\) As we know, a straight line has an angle of 180°.

\(^{10}\) The application returns the angle in radians by default, so it also needs to be converted to DMS.
The key models are all stored in a “Key” class, which stores the three variables (name, length, and angle). The model matching process is done in the main Activity since it is rather simplistic now, but as more models are added to the database, it would be a better idea to create a keyMatch() method in the Key class. This would certainly make the task easier for the developer when attempting to match keys when not all factors agree with a specific model and the application is trying to find the closest match.

Two things needed to be considered before matching the factors to key models. First, would there be any bias in the matching? Since it is unlikely the length will be measured precisely (± 15mm is expected), some threshold is allowed for. For the length, an offset of 2.5% is allowed (0.15 cm for a 6cm key, so on target for keys of length 6cm or shorter), in both the forward and reverse direction. While this is fine for a key like MS 2, where there are no keys of a length within that threshold, it can cause conflicts with keys of similar length, like UL 050 (5.5cm) and UNI 3 (5.75cm). We can see that the forward threshold of UL 050 (5.63cm) and reverse threshold of UNI 3 (5.61cm) will conflict. So this would be handled by either having different thresholds for forward/reverse or for each different key, or instead giving priority to certain keys.

In the end, the application gives preference to specific keys, based both on the knowledge that the application typically adds length during measurement, and the fact that some keys are simply more likely to be correct (i.e. some keys are more common than others are). Also (with a full application in mind) a key such as UNI 3 would be recognised by head shape, not length, so again, giving preference to UL 050 would a valid decision to make.

The other concern was how to handle situations where only not all factors gave a match (i.e. an angle matching but not the length, or vice versa). For the current application, measuring the angle of the tip was to filter out common results, not to determine a key itself, so of the two factors it had the lower priority. So in the case of a degree matching but a length not matching, then it would be considered a failure. However, when a length is found but not the angle is not, the application could show the closest results to the measured length (and optionally the measured angle too).11

3.7 Displaying Results

The final step for the application is to show the results of the analysis. The results are shown on the main screen of the application, under the “Video” and “Process” buttons, in two parts. The first is a TextView, which includes the actual information determined:

• Key model chosen (if any)
• Length calculated
• Tip angle measured
• Processing time (in seconds)
• Passes - the number of frames analysed

Under this is an ImageView, which either shows the model of the key recognised, or “failed analysis” image. If a key is found, there is also the option to long-press the image to find ideal parameters of the key model, as well as a link to key model in the online catalogue. The ImageView is handled by a separate class called “SetImageView”. This class handles the input image and scales it to the correct size (if the ImageView is smaller than the image), and sets the ImageView to the correct image. The images are included in the application (hence the low quality – when the images were on the SD card, they were much clearer), so the SetImageView class needs to be able to access the resources of the application (images are saved as R.drawable.ic_ms2, for example).

11 With such a small sample size, this was not implemented in the application. If another factor (head shape) was included and there were more keys in the database, this would be part of the next stage of development.
If multiple analyses are carried out, there is also the option to view previous results (as shown by the second screen in figure 3.7 (b)), as well as options to clear the screen and delete the images being processed. These will be explained further in Section 4.4.
4 Application Evaluation

4.1 Analysis

The aim of the project was to create an Android application that could analyse at least two factors of a key from video. Since it was always meant to be a building block to be used for further development, a perfect application is not the expected outcome at this stage. The ideal position to be in is that the application can accurately measure the factors included (length and angle of tip), and can use those factors to choose from a selection of keys. It is not a complete database, but should show the potential for this application if some more work was done to develop it. Another important element of the application was to allow some user-friendliness. These will be discussed in Section 4.4 below.

The application measures the length of the key rather accurately; it is accurate to 0.15cm (97.5% accurate). Since the length is the most vital parameter of the key in the analysis, it was always important to get this section to be as accurate as possible. The angle measurement, however, is not as efficient as it could be. As mentioned earlier, the ideal method would have been to calculate the inner product using the points in the image (figure 3.5 (b)), rather than using the angle between the lines and the X/Y axes (figure 3.5 (e)). It is less accurate than the length analysis, and has an accuracy of 80-85%. This becomes an issue due to the range of angles of key tips. For example, UL 050 has a tip angle of 87.5°, while UNI 3 (a key of similar length) has an angle of 97.5°, which is a difference of 10%. In addition, the angles for the other keys (UL 054 and MS 2) are even closer, which means the degree is not a particularly efficient method by which to sort key models.

Part of the reason for this is the resolution of the video. The resolution used by the device was 960*540, rather than the higher resolutions 1280*720 or 1920*1080 available. This was done to reduce the amount of work required – going from 960*540 to 1920*1080 would increase the number of pixels to be analysed by 300% (522,240px to 2,073,600px). By using cvCanny, the resolution was not necessary to be so high for the key length measurement, and if the inner product were used, it would have been enough to measure the angle of the tip accurately. On the other hand, since that method was not used, the accuracy of angle measurement was not good enough for the application.

A workaround would have been to record the video in the highest resolution available (device dependent), and then scale the image down for the first sections. This would reduce the workload where high fidelity is not essential, (pixel:cm and key length calculations), while still allowing the high quality images to be available when needed. This would require some recoding in Section 3.5, since resizing the image for angle measurement would be based on the scaled down images, not the full resolution images. So in order to increase the accuracy of this facet of the application, there are two potential solutions available to be expanded upon.

The key matching procedure is also incomplete. As stated, with such a small number of keys in the database, there was very little need to include a method to show the closest match in cases where an exact match could not be found. With only four keys in the database, it would be a fair assumption to make that it was either found, or not. However, more keys and/or more factors would require a more robust matching process than is currently used by the application.

Overall, I feel that the application achieved the minimum requirements, in that it can measure two factors of the key, even though the angle of the tip measurement isn’t quite accurate enough at the moment); with a relatively small amount of further work this would be solved. However, while the application does not use any of the other two factors (head shape, and identifier symbols) this is somewhat offset by the addition of user-friendly features for the rest of the application. So while the project started out being a Computer Vision project that happened to be developed on the Android platform, it ended up being closer to a 50:50 project. Part was developing the Computer Vision processing, and part was creating an application that
was targeted at the user level – many ease-of-use features were added as necessary to make the process simpler for the user, which actually did not influence the analysis itself.

4.2 Performance

While the application’s video processing of the key is one aspect, the application’s performance is one of the main things the user would notice. However, bearing in mind that the project only deals with two key factors and a complete application would need to deal with at least two in addition, runtime was not considered high priority. This becomes clear when actually running the processing – the application runs on the main thread for the entire process, causing the app to remain unresponsive. This means the user simply waits (the button remains highlighted to show something is still occurring), until the processing is complete. Ideally, adding a progress feature (either “x% complete”, the Android spinning wheel, or text pop-ups confirming when a frame has been processed) would allow for some true feedback during processing. This also results in other drawbacks to the application, such as not being able to change settings or view past results during processing, and it also means there is no “cancel” button during the processing. Since the application is unresponsive during processing, there is no way to interrupt it, aside from returning to the Android homepage and manually force closing the application. While these do result in a poor experience during the video processing itself, none of it actually influences the analysis – it is simply less interesting for the user during this time.

Concerning process runtime, one of the areas that could be significantly improved upon is the calculation of the pixel:cm ratio. As described in Section 3.2, the application goes through every single pixel (checking every column, row by row) in order to find the column with the most red pixels. However, look at figure 4.2 (a) for an example of why this can be a poor algorithm. The column the application is looking for is in the first 10% of the image, so 90% of the processing being done at this stage is redundant. But, if this image was passed through the Canny edge-detector, we could instead have the image next to it, and rather than cycling through every pixel,
the application would go row by row until an object pixel is found. At that point, it travels down the column until it finds a second object pixel.\footnote{Since the object pixel first found will either be at the left or right of the image (rather than in the centre), it would be unlikely that the key would give a false positive object pixel hit, unless placed incorrectly.}

As we can see, the application traverses each pixel until the yellow row (approximately 51*544 pixels), and from this first object pixel, it travels down along the column until the next object pixel, another 811 rows below. This method requires 28,555 pixels to be analysed, compared to the current method needing to analyse all 522,240 pixels - a reduction of \(~94.6\%\) of the analysis for this section. Since the application was not focusing on runtime reduction, the project tried to focus on other aspects of the application. However, for future development, where efficiency is an important aspect, this would significantly cut down process time per frame.

4.3 Issues Faced

Most of the issues faced during the development of the project were related to the steep learning curve of Android programming itself, and picking up the tricks and tools available. The biggest setback was not being aware of the simple things; this resulted in the first few weeks of development being very slow. The main issues were:

1) Not getting the Android device linked to the computer. To begin with (based on the limited Android knowledge I had going into this project), I was testing the program on an Android Virtual Device (AVD) – an Android emulator in Eclipse. This was fine for the very early steps, when the layout of the application was being changed, and I was getting to grips with Android. However, the first job was to get the camera intent working, but the AVD does not support cameras in emulation, so there was no way of knowing whether it was working, or saving to the SD card correctly. Therefore, in this instance, I had to export the installation (.apk) file for the project, install it and actually test it each time a significant change had occurred. This was incredibly inefficient.

Once the camera intent was working correctly, the frame extraction was the next step. Using the Android Debugging Bridge (adb), a sample video file was copied to the AVD, which was then used for extraction. However, while it did work, the emulator was very slow – simply starting the device took a couple of minutes, and it took the best part of 15 minutes to complete the extraction. In addition, when errors occurred, I had no idea if it was an issue with the code/methods being used, or a result of the limitations of the AVD. I eventually found help online and was instructed to install the Android Device drivers onto the computer and connect it to Eclipse. Once I had learnt this, development started in earnest.

2) The second major issue was the debugging of errors. Errors occurred frequently in the early stages of development. Correctly calling the OpenCV algorithms (especially when using OpenCV4Android – JavaCV made them much simpler to access) was causing issues, due to the OpenCV project not being correctly set as a library for projects by default. A small step, but one that was made much harder to diagnose without correct error outputs. The application simply failed, with no explanation. Therefore, I needed to find some way for Eclipse (or the application itself if possible) to create an error log on failure. Thankfully, Eclipse has a built-in debugging tool, known as “LogCat”. Figure 4.3 (a) is an example of the output shown when video processing is attempted, but no video file is available.
The sectioned highlighted in blue gives the hint as to why the error occurred. The `setDataSource()` method failed, which would imply either the code is incorrectly written, or there is something wrong with the video (which case it actually is can be determined by checking the rest of the log). Knowing this allows the user to quickly check to make sure the code is the issue, or the video file, and it wouldn’t take long to realise that there is no video file present on the SD card (or, that the SD card is connected to the computer, not the device, which can be another source of error).

To avoid having to resort to LogCat for each of these issues, it’s good practice to have the application signal to the user an error has occurred. In this situation, the application has a pop-up stating that there is no video file available for analysis (and a different message if the SD card is not mounted, etc.).

3) The third issue was the logging of variables. The success of the image being restricted to the key tip for angle measurement (Section 3.5) was completely dependent on the boundaries of the red rectangle being correctly determined (Section 3.2). This was easier said than done, however, since the first few attempts at this caused the image to go out of bounds. The only method I knew of checking these variables was to create a pop-up on the application itself, showing the values. But since the application runs on the main thread, there was no way of showing these pop-ups sequentially – they would all pop-up together at the end of the processing, so I had to comment out the code after the pop-up was created to get measurements at the right points. This was an incredibly lengthy process, and did not always help pinpoint values correctly.

Thankfully, I was not held up for too long with this method, and found the logging system for Android & Eclipse. [11] This form of logging allowed me to output variable values at any stage during runtime to the LogCat screen. These outputs could also be filtered out by using the flag defined in the method.

Until these three obstacles were dealt with, development was slow and painful. Had I known these workarounds when the project started, I have no doubt the application would be in a much more polished state than it is now. At the very least, the angle measurement and head shape would also be efficiently analysed. Far too much time was spent dealing with minor issues that could have been simply solved by checking error outputs and constant logging of variables throughout development. It would have saved a great deal of time and effort early on in the configuration of the programming environment.

4.4 Usability Features

Initially the plan was for the application to decide on the parameters for analysis automatically, basing its decision on the length of the video. Therefore, a long video would have

---

**Figure 4.3 (a) – Example LogCat output**

<table>
<thead>
<tr>
<th>LogTag</th>
<th>Event</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>E 31500 31500 * * *</td>
<td>*</td>
<td>java.lang.RuntimeException: android.media.MediaDataSource failed: status = -9223372036854775808</td>
</tr>
<tr>
<td>E 31500 31500 * * *</td>
<td>*</td>
<td>android.media.MediaDataSource.getMediaSource(time=453800454,mmap=true,offset=0,bufferSize=0,flags=0)</td>
</tr>
<tr>
<td>E 31500 31500 * * *</td>
<td>*</td>
<td>android.media.MediaDataSource.getMediaSource(time=453800454,mmap=true,offset=0,bufferSize=0,flags=0)</td>
</tr>
<tr>
<td>E 31500 31500 * * *</td>
<td>*</td>
<td>android.media.MediaDataSource.getMediaSource(time=453800454,mmap=true,offset=0,bufferSize=0,flags=0)</td>
</tr>
<tr>
<td>E 31500 31500 * * *</td>
<td>*</td>
<td>android.media.MediaDataSource.getMediaSource(time=453800454,mmap=true,offset=0,bufferSize=0,flags=0)</td>
</tr>
<tr>
<td>E 31500 31500 * * *</td>
<td>*</td>
<td>android.media.MediaDataSource.getMediaSource(time=453800454,mmap=true,offset=0,bufferSize=0,flags=0)</td>
</tr>
<tr>
<td>E 31500 31500 * * *</td>
<td>*</td>
<td>android.media.MediaDataSource.getMediaSource(time=453800454,mmap=true,offset=0,bufferSize=0,flags=0)</td>
</tr>
<tr>
<td>E 31500 31500 * * *</td>
<td>*</td>
<td>android.media.MediaDataSource.getMediaSource(time=453800454,mmap=true,offset=0,bufferSize=0,flags=0)</td>
</tr>
</tbody>
</table>
| E 31500 31500 * * * | * | android.media(MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDataSource,MediaDat...
less frequent frame extractions than a shorter video, always leaving a constant number of frames to analyse (which would create a uniform process time regardless of the video). While this may be the result of the application, instead the application allows the user to decide the number of frames to extract, using a dialog as in figure 4.4 (a). The maximum number of frames is determined by checking the length of the loaded video, and the frames specified must be within this range.

Figure 4.4 (a) – Set frames to be extracted

Another option added is the one to delete any temporary images or video used during the analysis. The complete application would do this by default (and leave no files) or simply not save files at all. Nevertheless, for debugging purposes, the files are saved and not deleted on process completion. The option below gives the user the ability to delete the images, or images & video (since sometimes you would like to keep the video for future analysis).

Figure 4.4 (b) – Option to delete temporary files

Landscape mode is also supported, which allows for better flexibility for tablets and larger devices. While landscape mode alone was straightforward to create, the ability to switch between portrait and landscape (and vice versa) once the application had started took some work. In Android, by changing from one screen orientation to the other, the application Activity is “destroyed” and recreated anew. While this does not mean anything to the user, it means the variables used by the application are also reset. In order to change orientation without errors, these variables need to be saved and restored during the Activity’s destruction.

This was successfully done; however, the application can run into errors in specific, uncommon situations. Specifically, if the application has done multiple analyses, the user is viewing one of the “middle” results (not the first or the most recent, but one which allows both buttons to be visible, as in figure 4.4 (c) below), and then changes orientation, the application can reset all variables, or simply crash.
Saving the results to be viewed later required creating a new class called “ResultStorage”. It contained the variables for each result (model, length, angle, frames used, time take for processing). It also included a method to set the TextView with this information (and handle whether the result was successfully found or not). Since care had to be taken when the application destroyed Activities, the application could not store every result’s variables individually. Creating a new class and storing in an array was the most efficient way to achieve this. The array used is static (only the most recent 10 results are shown), which is not the most efficient method. I would have preferred to use an ArrayList to allow the array to resize itself dynamically. However, at this stage of the application, there was no need to implement one.

When a successful analysis has been complete, the application shows the info and key model in the empty space of the screen. If the user longpresses the key image, a pop-up appears containing the following.

- A link to model page on the online Silca catalogue
- The length of the ideal key model
- The angle of the tip of the ideal key model

This can be used to confirm the analysis chose the correct key from the information measured, and links to a resource with more information if necessary.
There is also the ability for the user to clear the screen (by selecting the option from the expanded menu). This removes the previous results from the screen, though they can be viewed again if necessary by using the “previous result” buttons that appear once a result has been completed. The application also has a “close” button (which can be seen as a red “X” in landscape mode), which allows the user to exit the KeyAnalyser application, and gives the user the choice of deleting the temporary images created during analysis.

Finally, in the case of a failed analysis, the application shows the information measured (or not measured), and shows an “X” in the image frame, to make it clear the analysis was unsuccessful. As can be seen in figure 4.4 (f), the length was not measured, and the angle of the tip was not even picked up. However, the information is still shown, so good analyses by no matching key models in the database can be manually determined even at this stage of the application’s development.

4.5 Bugs

There is one major bug in the application as it stands, which affects the frame extraction portion of the application. As described in Section 3.1, the frames are extracted using the MediaMetadataRetriever class, with the OPTION_CLOSEST flag enabled. However, for an as-yet undetermined reason, this method begins failing after 1.9s of the video. After 19 frames have been extracted, the video will return an empty frame, which will cause a “NULL Pointer Error” to occur.\textsuperscript{13}

\textsuperscript{13} It is usually after 1.9 seconds, though can also be after 2.4s or 3.5s. 3.5 seconds is the further without an error that has occurred though.
This remains true regardless of the length of the video, or the resolution used, and increasing the buffer size does not help. Without being able to find any help online, there were two options to deal with this issue:

1) Accept the error, and handle when an empty frame was extracted. Nineteen frames should be enough get a relatively accurate measurement, so this was the easiest method, and the one the application employs.

2) Rather than use the OPTION_CLOSEST flag, the application could use the OPTION_CLOSEST_SYNC flag instead. This would extracted the closest keyframe to the time specified. However, this would result in many frames being duplicated in a short video with 0.1s extractions. So the video would need to be longer (perhaps 10 seconds long, recording the key multiple times from the same angle) in order to give enough frames for the application to work with. However, since this would result in the processing taking longer, and it could not be guaranteed that this would solve the “NULL Pointer Error” issue, it was not used.

Another issue, which was touched upon in Section 4.2 is that when processing is being carried out, the application is unresponsive. Ideally, the application would handle the processing in the background, leaving the UI available for the user to interact with (again, changing settings, or viewing past results).

4.6 Security Considerations

While security was not a major theme of the application (partly because there is very little information for an attacker to steal, and partly because the attacker would generally learn more by stealing the device than the information on it), it is still an important factor to bear in mind for any application.

**Systems Security**

This topic involves making sure that the applications (and the devices it is run on) are used only for the purpose for which it is designed – an attacker should not be able to exploit the device by running the application “out of bounds”. In addition, no attacker would be able to stop the device from running other applications (or in the case of an online app, to stop other users from using the application or connecting to its resources).

**Inappropriate usage - The kind of questions and situations considered are:**

- Is it possible to crash the application by recording too large a video? How does the application handle a video file larger than the remaining space on the SD card/internal memory?
- What permissions does the app request? Does it have access to contacts or text messages? Is it able to access the internet directly?

For the first case, it is less of a security issue as it is a development issue. Thankfully, the Android environment is extensive enough that there is no need for the programmer to directly access the device camera, and hence have to handle the data itself. By calling a camera intent, the application hands over control to the device itself, using the native camera application to record video. Due to this, the application and developer do not need to worry about the record procedure being abused by an attacker.

For the second situation, the application is only permitted to save files to the SD card and access the camera. Therefore, an attacker cannot use this application to access any personal information (unless, of course, they utilise a vulnerability in Android itself). In addition, all online interaction in the application is handled by the device’s browser (rather than any in-app connectivity, links are used). So again, any attacks would be through the browser being used, rather than the application itself.
Denial of service - An attacker may not want to steal your information, but instead render your device/application unusable. Therefore, the questions that need to be considered are:

- Is it possible to cause the application to become unresponsive by running an extremely long calculation (video being several minutes long, highest resolution, etc.)?
- The application is coded to run on main thread (so all other activities in application are unavailable) – on a device with a slow CPU (or low on RAM), can this be exploited to cause phone calls/text messages to be delayed and received later?

Active Attacks

The application, as it stands, does not access any online database – the information for various keys are stored as “Key” class files in the app. However, a more efficient solution would be to host a catalogue/index of keys online on a server. This way, as the key database increases, the app itself would not change. Now, while the application does not deal with any particularly sensitive information, it is still good practice to be able to handle potential attacks, as long as they do not affect the app too negatively. If the database was to be moved onto a server then active attacks become a possible option for attackers:

- If a password log-on was implemented, one of the main attacks to watch out for is password sniffing – since the device will usually be connected using WiFi or 3G/4G, you are depending on the security of your ISP, and any other networks between sender and receiver.
- Man-in-the-Middle attacks would also been a possible issue – an attacker could listen in to the response from the server and copy it for himself.
- Message insertion, deletion and modification attacks are additional valid attacks that should be considered, and against which defences could be planned. Each attack would result in the application user not receiving the true response from the server or, alternatively, to change the application request to the server, so the user receives a reply directly from the server, but not to the request that it originally sent.

Passwords - As a standalone method of user identification, passwords are becoming easier and faster to crack. The reason for this is two-fold:

1) There seems to be a lack of information as to what exactly makes a good password. It is a common misconception that adding a number (1, 01, etc.) at the end of your “normal” password makes it harder to crack. The truth of the matter is that this only adds 10 potential values – using a capital letter (thus including lower and upper case letters) adds 52 potential values, and adding a standard symbol (&, %, etc.) adds 95 additional values. Another fallacy came about after an XKCD comic became popular online (figure 4.5 (a) below). It makes the argument that using common but random words is both easy to remember, and difficult to brute-force, compared to a word made up of random letters/symbols. An argument against this is if an attacker was to attempt to brute-force your password using full words (as opposed to single characters), then a four-word passphrase is little better than a four-character password.
2) Past attacks have put huge numbers or real passwords on the internet. Incidents such as the RockYou attack in 2009 [12], or the LinkedIn attack in 2012 means 14.5m and 6.5m password hashes respectively were lost to hackers. As a result, attackers now have both dictionary based attacks and lists of actual passwords used (sample in Figure 4.6 (b)). In addition, since the majority of internet users keep the same password (or a variant of it) on all of their websites, attacks are continuing to occur, and computers are getting faster and better at brute forcing, passwords are less unpredictable as they once were.
To combat these potential threats, one way to establish the identity of the application use is using password authentication, or link the user to each individual device. So a combination of MAC address and username/password would allow for a more secure connection to the server. When taken separately, both identification methods are vulnerable, but by combining the two, you can make it harder for attackers. As in the first example above, by mixing the two methods together (e.g. uncommon words with random symbols, or incorrect spelling), you can generate a far more effective password. Another option is to use TLS (TLS 1.2, the current successor to SSL) when connecting to the server for analysis results. Designating a specific port on the server for TLS connections, would be a valid way to allow the TLS protocol to be used. A secure session is established using a simple handshake procedure, wherein the server is authenticated by its certificate. [13]

**SQL Injection**

Since any online database would likely be using SQL, an avenue an attacker could use would be to modify the database. Due to their numerous types, large attack surface, and the complexity needed to defend against them, they are amongst the most common and successful attacks available. Protecting against SQL injection attacks involves validating any queries to ensure that it is in the correct form. Any data that fails that is simply rejected.

An alternative way to protect your database from attackers is to limit the permissions for database users (known as the “Least Privilege” principle). All database users are (by default) only given permissions, which are essential for them to complete their tasks. This way the attacker – even if he gains access to the database – is limited in the queries he can perform, such as dropping tables or changing other users’ privileges. [14]

**Cross-Site Scripting (XSS)**

Another form of attack is XSS – injecting a script (often JavaScript, but also including ActiveX, VBScript, Flash, etc.) into a web application. It is a particularly dangerous form of attack since any script run by the attacker seems to be a legitimate executable from the web application. This can include anything from causing an executable to run in the background, monitoring your keyclicks (and work as a keylogger), UI redress (altering the UI of a web application, to trick users to click infected links/buttons), etc.

Defending against these attacks can be tricky – they need to be constantly maintained, especially if Input Validation is used. As a defence mechanism, it is limited to only filtering out any untrusted inputs, and as such, cannot accurately predict future untrustworthy inputs. [15]
5 Conclusions

5.1 Personal Reflection

I feel that the project reached the bear minimum expected – the plan was to have a couple of factors, and essentially develop a platform that could be expanded on later. While I am disappointed that the angle measurement was not as accurate as it could have been, I feel it was compensated enough by the additional features that were added as a substitute. There was never an intention to allow as much customisability for the user – the plan was for the application to decide on how many frames to analyse and to handle the files and deletion. Nevertheless, once it became clear there was no time to add a third factor for analysis, a bit more emphasis was put on the style and feel of the application than the hard Computer Vision processing that was intended.

I am content with this though – it gave me a good opportunity to learn about different parts of Android, and put a bit more emphasis on the front-end of the application; the part the user interacts with, rather than focussing solely on the processing hidden in the background. Therefore, instead of simply being a Computer Vision project, I have a greater understanding of both Vision processing and of Android development. Again, the only regret is that it took so long to get up to speed with using Eclipse and efficient logging during development. Aside from that, it was a great experience in a longer-term development situation than I had ever been a part of before, as well as a project, which was not ever defined or limited by anyone else. If I felt something should be added or was needed, I was able to implement it (or attempt to at any rate), and was not forced to compromise on things I wanted to see in the finished product.

5.2 Further Development

From this point, there are a few potential options for future development. An option would be to implement the online database commented on in Section 4.6. Though I had not intended to apply this myself, due to the massive amount of work that would go into learning how to do it, it could be an interesting separate project. Taken from the point of view of mobile application <-> web server interaction, or even as a lesson in Computer Science security, to show that security can be implemented even on a system with very little apparent need. However, there is no actual need for this in the application - it would be more of a theoretical project, with little to no real world use.

One of the steps that must be carried out before claiming the application is successful is to continue with the Computer Vision processing. There are still two factors to be programmed and analysed to increase the scope of the application (and one factor in need of enhancing). Head shape is an interesting route to go down next. There is such a variety of different head shapes, that there is a lot of scope for further development. Some heads are predominantly circular (as three of the four included in this application are), while others are square-headed, and others still (like UNI 3) do not follow geometric shapes at all. So including both Hough transforms (for circles, triangles, straight lines and/or corners) as well as image template matching (which means checking if a specific, uncommon shape occurs anywhere in the image) would be used to filter keys by head shape, and then move on to further refinement with key length and tip angle.

Analysing the identifiers would be an ideal next step, but the true difficulty here is getting a high quality image that the application can handle. The raw images extracted are not high resolution enough to allow for recognition – perhaps the recording could be made two-phase, where one is a full recording of the key, and the second is an up-close view of the identifiers? However this issue is dealt with, if analysis could be done on the identifiers, it opens up an enormous range of additional keys to add to the database, when including the factors already analysed by the application.
Finally, another step that must be done before the application is complete is to improve the performance of the processing. Process runtime is certainly one of the weaker parts of the application as it stands, and it would not be a useable alternative to manually matching keys unless that time can be reduced. As discussed in Section 2.3, multiple calls to the OpenCV library can result in a slowdown in processing, which we know to be true based on the application description in Section 3. Therefore, one possibility would be to use the native OpenCV code (using OpenCV4Android – advanced), utilising the NDK and defining a class with all OpenCV calls. This would condense the multiple calls made currently (Section 3.3 and Section 3.5) into a single call, reducing the import cost to the equivalent to a single JNI call.

The efficiency of each individual section could also be looked at. The potential improvement for Section 3.2 (pixel:cm ratio) has already been discussed, but there is still a potential for further streamlining. Now in order to define the boundaries (first and last row/column), the application cycles the entire image twice (once row by row to define the rows, then column by column to define the columns). At the very least the same improvement could be done here by converting the image to cvCanny, and cycling from left to right until the first column is found, then from right to left to find the last column. An alternative (that I have not yet looked at) is to use the Hough transform to find the corners of the rectangle, and use those four points as the boundaries of the rectangle. The processing involved in this is less than cycling through each pixel manually, so it would offer a more efficient (and potentially more accurate) definition of the location of the rectangle.

In addition, as has already been mentioned, the application currently extracts a frame, converts it using cvCanny, then resizes it for the key tip, and saves in each occasion. Meaning for every frame to be analysed, it is saved and reloaded three times. Converting the file directly from an extracted Bitmap file to an IplImage variable would cut down on this too.14 Of course, concerning Android, moving the processing into the background is an essential step before the application can be completed; no completed application should saturate the UI thread during processing. So moving the processing to behind the scenes allows the user to interact with the application, and allows the developer to update the user on the processing situation (running analysis of length & angle per frame, notification for each completed frame, etc.).

Therefore, while the goals of the project have been met, the application itself still has a great deal of potential additional work. It seems like much of my work has been on understanding Android and getting all the various parts to work together. With that out of the way, and a framework for the application now complete, it allows the project to move to another level – a level of delivering an actual application with real-world uses, rather than the proof of concept it currently is.

---

14 Not ideal with an unfinished application, since you can’t view images in case of debugging, so probably best left until all CV processing has been completed.
6 List of Figures

1.2 (a) A labelled key-cutting machine
1.3 (a) Unique head shape of UNI 3 key
1.3 (b) Identifier symbols on reverse side of UL 050 key
1.4 (a) One page of Silca catalogue key index (R-S)

2.3 (a) Description of the role of the JNI in a Java environment
2.5 (a) Launching application on connected Android device

3.1 (a) Sample extracted frames at 0 seconds, 0.5 seconds and 1 second
3.2 (a) Sheet on which object key must be recorded
3.3 (a) Example of Canny edge detection
3.3 (b) Sobel edge-detection convolution kernels for X and Y axes
3.3 (c) First & second order derivatives of edges
3.4 (a) cvCanny output image of key, with first and last object rows in red
3.5 (a) Restricted section of key tip; area to be analysed
3.5 (b) Close up of key tip to be analysed
3.5 (c) Inner product of key tip (ideal)
3.5 (d) Representing a line as a single point in m-c space
3.5 (e) Mapping points of a line to a single point in m-c space
3.5 (f) Measuring angle using Hough Lines Transform
3.6 (a) Key models in database (MS 2, UL 050, UNI 3, UL 054)
3.7 (a) Result screen of application (single analysis)
3.7 (b) Result screen of application (multiple analyses)

4.2 (a) Current pixel:cm method
4.2 (b) Example of improved pixel:cm method
4.3 (a) Example output of LogCat when no video file is available.
4.4 (a) Dialog where user sets the number of frames to be extracted/analysed
4.4 (b) Option to delete temporary video and/or images
4.4 (c) Application in landscape mode
4.4 (d) Screen with “clear screen” and “close application” options
4.4 (e) Key information on longpress of image
4.4 (f) Result screen on failed analysis
4.6 (a) XKCD “Password Strength” comic – popular fallacy about passwords
4.6 (b) Example of 30 passwords stolen in the RockYou incident
7 References
8 Electronic Resources

The attached CD/DVD includes the following:

• Current KeyAnalyser (v0.8) Eclipse project, “KeyAnalyser v0.8\src\mehdis\KeyAnalyser”, containing:
  Key.java Class for key model information
  KeyAnalyserActivity.java Main activity for application
  ResultStorage.java Class to store results for callback
  SetImageView.java Class to handle ImageView post-analysis

• “KeyAnalyser v0.8\libs”, containing:
  javacpp.jar and javacv.jar JavaCV methods

• “KeyAnalyser v0.8\libs\armeabi”, containing
  All .so files (compiled versions of JavaCV files) for application

• “KeyAnalyser v0.8\res\drawable-xxhdpi”, containing
  Images for key models for ImageView

• Application icons, saved as:
  “App Icons” (“Orig” and “Black” no longer used)

• Project images, saved as:
  “FYP Images”

• Full RockYou password list (133MB), saved as:
  “RockYou Password List”.txt

• PowerPoint presentation for project demo, saved as:
  “Project Demonstration – 20th March 2013”.pptx

• Sample video file used for testing, saved as:
  “video”.mp4

• Excel spreadsheet of eight keys, with length for all and degree of tip for four (the four used in the application), saved as:
  “KeySizes”.xlsx

• Backups of all previous project versions v0.1 – v0.7 (including raw code and .apk installation files), saved as:
  “KeyAnalyser v0.7 Backup”.zip

• A changelog of all version from v0.1 - v0.8, saved as:
  “Changelog v0.8”.txt

• Two copies of this report, in Word 13 format, Word 97-2003 format and as a PDF, saved as
  “Sheikh Mehdi (09498389)”.docx, “Sheikh Mehdi (09498389)”.doc and “Sheikh Mehdi (09498389)”.pdf

  Total size = 551 MB