Classifying Android Applications by GUI Feature Extraction

Stephanie Rogers, James Huang, Zack Hsi, Casey Lawler, Vladimir Ponomarenko, Huapeng Qi

INTRODUCTION

Our goal is to classify Android Applications based on GUI features, which are to be extracted from a set of more than 2000 applications. We are aiming to create a system, which will ultimately allow people to quickly and efficiently understand which applications are similar or dissimilar by computing and measuring features based on GUI code and layout. Once this is implemented, we need to test it on real data from Android applications and verify that our method does indeed create a realistic representation of applications.

MOTIVATION

Our project is motivated by an increasing need to detect malicious Android applications. A concrete instance of this, reported by Trend Micro[1], is that in just the first three months of 2012, over 5,000 new malicious Android applications were identified. We need an objective method of analyzing, classifying, and comparing applications so that we can detect, for instance, if an application is masquerading as a non-malicious application but is really performing malicious tasks in the background. The common theme is that the malicious application has the same GUI as a regular, benign application but utilizes inappropriate permissions to steal user data. Thus, by accomplishing our aforementioned project goal, we can efficiently identify similar-looking applications and perform further checks to ensure that they are safe for users.

BACKGROUND

Android Applications

One security feature of the Android operation system is isolation. A Java virtual machine, Dalvik, ensures that each application process is allocated separate resources and run as a different user. Communication between applications is done through an application framework to ensure that applications do not interfere with each other in malicious ways. Android applications are distributed under the Application Package File (APK) file format. This file holds all the files needed to run the application including program code, certificates, as well as sound and graphic resources. Stand-alone, it can be installed and run on the Android operating system. APK files can be reverse engineered with the Android APKTool. The decompiled output code is in the smali format, which is assembler code for dex format used by Dalvik. A decompiled application typically comes with 3 main subdirectories: assets, resources, and smali code. The difference between resources and assets is an Android implementation different in how they are accessed inside of the application. Most (but not all) applications have a layout folder in resources that contains xml to specify the appearance of the GUI of the application.

Feature Extraction

The extraction of features related to the GUI of an application can be approached in two directions. Features can both be extracted from the layout xml code of decompiled application as well as from the pixels of a running application. Because smali is more similar to low-level Assembly language, it is hard to extract structure or behavior of the GUI directly from the code. Instead we turn to extracting features directly from the layout xml. For this we use string hashes to turn segments of code into numbers and collects these into a feature vector. Conceptually, application with similar GUI will have similar xml specifying layout and thus similar feature vectors.

To extract features from the actual images displayed to an end-user we utilize an open source computer vision library named VLFeat. We use 2 main algorithms of this library. Firstly, we scale-invariant feature transform to detect and describe interesting local features in images while accounting for the fact that interesting features may have relatively different sizes and locations on a screen. Low contrast regions are filtered out so key points typically lie on perimeters of important shapes. Secondly, we utilize blob detection to identify regions and points that differ in brightness and color to surrounding points. Conceptually, applications with similar GUI will have more matching vision features.

APPROACH

Our approach to solving this problem was to use a rolling hash on a specified number of contiguous bytes to obtain a set of feature bit vectors from each application's layout XML files. Then, using these feature vectors, we could compare two applications and obtain a metric of how similar or dissimilar they are in the form of a score. Additionally, we also performed a screenshot comparison of two applications using an image comparison process called SIFT.

There is a possibility that a malicious application is not developed from the ground up. Instead, malicious code typically is added into already existing applications. Thus, the motivation for classifying android applications by their GUI code would be to identify applications that have similar code to ones we have already classified. We could then check the new applications' source code for malicious behaviour.

Rolling Hash
In implementing an adequate rolling hash function, our goal was to focus on generating feature vectors from layout XML files in each Android application's layout folder and to hash every k number of contiguous bytes.

We created a Python script that extracts feature vectors from the XML files in decompiled android applications that specify their application's layout. After normalizing these layout files for spaces, line breaks and capital letters, the script rolled a k-byte window (of size k = 7-15) through all of the files. For each window, the script hashed the corresponding string using the djb2 hash function. The script instantiated a bitvector (from the bitarray Python module) of variable length and set bits corresponding to the values produced by the djb2 hash. The resultant bitvectors were sparsely populated, and effectively indicated the style and function of the code used to render each application's user interface. The usage of the script is as follows:

'usage: roll_hash.py <path> <window size> <file1>...<file2> '

The Python script took as input: a path *<path>* representing the location of the application, the size *<window size>* of the rolling window as described above, and each of the separate layout-specifying files *<file1>...<file2>*. The script output the application name and one bitvector per layout XML file to standard output. We also built a bash script that crawled a specified directory for applications and produced bitvectors for each with our rolling hash function. This script's parameters included the size k of the rolling window. We ultimately extracted a feature vector for every window of string length k = 7 to 15 and resolved to determine the optimal window size later.

Bitarray Size
We had to determine the optimal bitarray size to minimize the number of hash collisions towards creating sparser bit vectors as outputs. With a few of our own test cases -- in which we counted the number of collisions and took into account those due to the same substring -- as well as the help from our project advisor, we were able to determine that 100,000 was the best order of magnitude. We decided to use a bitvector such that we could work modulo a prime in order to further reduce the number of collisions; hence, the bitvector size is 100,003.

Window Size
We had to decide which window size (from k=7 to k=15) we would use for our rolling hash. Toward this goal, we observed the effects of each window size on our resultant bitvectors using our comparison function. We handpicked similar and dissimilar applications to determine which size produced the most accurate results. Ultimately we decided to go with a larger window size of 15 characters. We predicted that since larger window sizes yield more unique strings, there would be less collisions on average. However, we needed a small enough window size that applications would share certain things in common. After running a few tests we noticed that there were 10% less hashing collisions, as we predicted, meaning that the larger window size is 10% more accurate.

Classifying Applications
We extracted feature vectors from two separate repositories of android applications, each approximately 1000 applications in size. One repository contained popular applications, while the other repository had a smaller set of more obscure applications with multiple versions of each. Using our rolling-hash python script, we extracted bit vectors of length 100,003 by running the rolling hash on the layout folders of each application. *'usage: roll_all.sh'* We ultimately generated a bit-vector (where window

size k=15) for each of about 2000 applications. These bit-vectors gave us the ability to compare the GUI features of new Android applications against those of a repository of analyzed Android applications.

Comparison Function

We defined a metric to determine and measure the similarities and dissimilarities between given applications in the form of a comparison function. We created a comparison function that would take in the file paths of two Android applications and produce a score based on the similarities and differences of the two bitvectors. For our final report, we only used the results of the k=15 window size. This function's usage is as follows:

'usage: compare_vector.py <app1_path> <app2_path> <smallest window size> <largest window size>'

The comparison function returns the sum of the asserted bits in the bitvector produced by taking the bitwise AND of each for each window size in the given range. The fact that our comparison function produced scores that were monotonically increasing did not trouble our project advisor. The scores for the two dissimilar applications were much lower than the scores for the two similar applications, no matter the window size, signaling that our parameter choices are sufficient and that our comparison function is accurate.

Screenshot Feature Analysis

We decided to explore vision processing as a means to extract results from applications' rendered user interfaces. Using an Android emulator and tools such as MonkeyRunner, we attempted to automate the process of taking and processing screenshots of applications' UIs.

For each .apk file we analyzed, we performed the following steps:

1. Decompiled the .apk file to gain access to its Android Manifest
2. Parsed each manifest to extract important information like packet names and list of Activities. We created a directory named by each application's package and created a text file in each that lists the Activities we proceeded to record.
3. The MonkeyRunner then installed the program onto our emulator and ran through each of the aforementioned activities. After waiting a predetermined amount of time for the application to load, the MonkeyRunner took screenshots of each application. These screenshots were all saved in an output directory.

We attempted to implement pairwise comparison for all activities for each application as mentioned above. That, however, resulted in a significant increase of complications caused by activities requiring more functionalities than the emulator could provide. In the end, we decide to compare only the user interface for the first activity of each application.

To analyze the screenshots, we have implemented an image comparison process using SIFT (Scale Invariant Feature Transform).We converted the screenshots them into pgm format, then created .sift files using the sift program. The .sift files contains the locations of key features identified on an image. We took out the very top portion where the battery life and signal strength resides before we compared each pair of .sift files. We compute the score by dividing the number of matched features by the number of features identified in the image that has lesser features and multiplying the result by 100. A score above 50 is an indication that the two images are similar.

KEY CHALLENGES

One of the key challenges that we faced during this project had to do with the computer vision aspect of features extraction. This was a risky endeavour in terms of time and results because our team have no prior experience with extracting features from images. Moreover, in order to extract features from an application through vision it is necessary first to run the application in an Android emulator. The entire process needs to be automated to a large degree for this sort of comparison to work in detecting similarities in the GUIs of applications.

We managed to automate feature extraction for repositories of Android applications, but in certain circumstances we get bad results. Sometimes an application will require internet or GPS or certain certificates. This means that the GUI we extract features from is not reflective of the typical application users see. Other times it will refuse to run because Android applications do not have perfect compatibility with the emulator because we are emulating the wrong version of Android. Additionally, some of the larger Android applications take a very long time to load in the emulator and thus our automated feature extraction doesn't do the right thing.

RESULTS/CASE STUDY

We broke our case study up into three parts. In the first part, we will randomly sample a small set of applications (~200 applications) to see how our bit-vectors cluster applications by performing an all-pairs comparison. After running the comparison, we will manually examine the top 10 similar and dissimilar applications based on the scores they were given from our comparison function, as well as the containment percentage of one application within another. The second part is to look at 4 pairs of applications that are extremely similar and 4 pairs that are extremely different and look at the accuracy of our classification/comparison function. The last part will be an analysis of our work with vision processing of the GUI features.

Phase I: All-pairs comparison

Our first task in approaching the case-study was to select a random sampling of Android applications on which to run our comparison function. We wrote and used the Python script randomSelect.py to minimize any biasing of the data set toward more popular applications. We chose approximately 200 applications from our Android repository and approximately 80 applications from the repository that our project advisor provided for us. By running a bitvector comparison against all 39,600 possible pairs of these Android apps, we generated a listing for each pair in the following format:

'app1/app2/similarity/containment of app 1 in app 2/containment app 2 in app 1 '

To determine the "similarity" value between app1 and app2, we ANDed our bitvectors for app1 and app2 to produce one vector v1, we XORed the bitvectors for app1 and app2 to produce another vector v2, and we finally calculated the difference between the asserted bits of v1 and the asserted bits of v2.

We borrowed the "containment" metric from the JuxtApp[4] presentation produced by the Intel Science And Technology Center on Secure Computing at UC Berkeley. To determine the "containment" value for one application app1 in another application app2, we ANDed our bitvectors for app1 and app2 to produce a bitvector v1. We then divided the count of the asserted bits of v1 by the count of the asserted bits of app1's bitvector. Intuitively, the result of this division is the ratio of features of app1 that exist within app2; as app1 shares more features with app2, this ratio increases toward 1.

We created Figure A1 in Appendix A that shows the similarity value and manual analysis of their similarity for five randomly chosen pairs of applications among the ones that scored the highest and five among the ones that scored the lowest. Four of the five pairs with high similarity scores turned out to be false positives. We identified that the false positives were due to the large size of the layout folder in the decompiled files for the application pairs. Applications with larger layout folder tend to have higher similarity score just because they have more contents, and applications with smaller layout folder tend to have lower similarity scores. This motivated us to create a two-way containment comparison that eliminated the misleading results induced by file size.

We calculated containment values for each application pair to quantify the degree to which one application's GUI is based on that of another. Applications with a large amount of layout specifying code typically have a relatively high similarity metric, thus two-way containment is required to ensure actual similarity. We created Figure A2 in Appendix A that shows the containment value and manual analysis of their similarity for five randomly chosen pairs of applications among the ones that scored the highest and five among the ones that scored the lowest. In contrast to Figure A1, there are no false positive in Figure A2. The pairs of applications with highest containment values are two versions of the same application with very similar GUIs: racing.waijul21j and racing.waijul21e, for instance. The applications that had high similarity score but are dissimilar ended up with low containment values and were not identified as similar with the containment metrics.

Phase II: Accuracy of our classification

The next step was to test the accuracy of our classification, by running our comparison function on 8 pairs of handpicked applications. We identified the following pairs of applications as having extremely similar GUIs based off of manual analysis: Officesuite Professional and OfficeSuite Viewer, Bedside Night Clock and Gentle Alarm, AndroZip File Manager and AndroZip Pro (Root) File Manager, DroidBox and DroidBox Pro. We also identified the following pairs having extremely dissimilar GUIs: Miniplane and Bloons, Devilry Huntress and Facebook, Scan Biz Cards and Alchemy, Binary Converter and Castle Warriors.

Our reasoning behind this is our desire to verify that the relationship between manual (verified correct) analysis and automated analysis is bidirectional. We have already established that there is a high correlation between the two in Phase I, but until this point had not cemented our results. Manual analysis followed by programmatic analysis adds one more degree of validation to our results.

For each of the proceeding application pairs, we briefly examined and noted the specific similarities and dissimilarities between each application in entries as shown in *Appendix C.* These entries don't necessarily deal with the GUI specifically, but should aid us in our analysis of the effectiveness of our classification/comparison function based on GUI features. Furthermore, we collected several screenshots of these applications (approximately four per application), examples of which are shown in *Appendix B*.

Figure A4 in *Appendix A* shows our results, with predicted similar applications having an average score of 8990 and predicted dissimilar applications having an average score of 1558. All of the similar pairs had scores over 6000, while the dissimilar all remained under approximately 3000. This high correlation leads us to conclude that our algorithmic, automated comparison is just as effective at determining similarity as a manual comparison.

Phase III: Vision Processing of GUI features

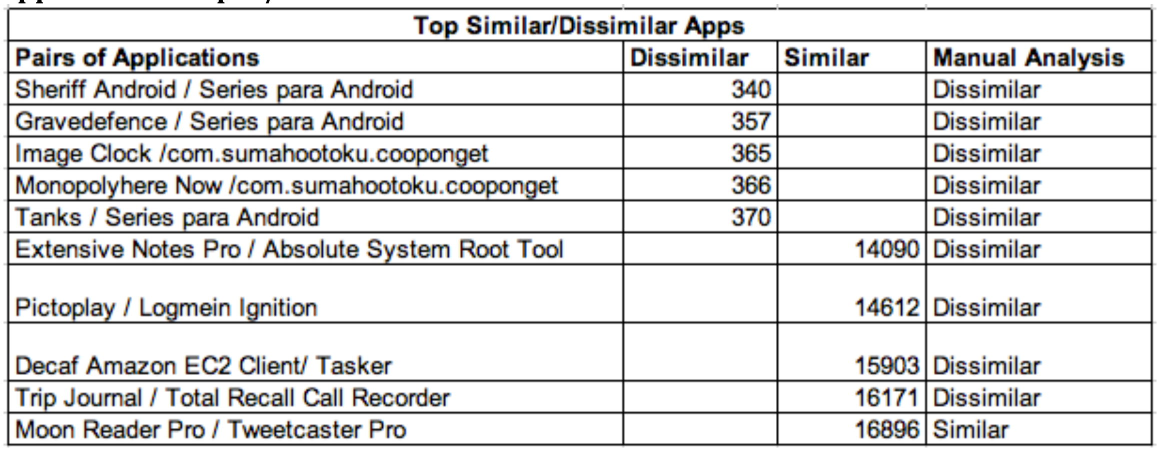
The android emulator was unable to run the handpicked pairs of applications in Phase II except Droid Box and Droid Box Pro. We randomly picked another 13 applications that have size less than 500 KB and ran all 15 of them through our vision processing process. We picked smaller applications to avoid the case when the application loading time exceeds the waiting time after running an application before MonkeyRunner takes the screenshot.

Figure A5 in *Appendix A* shows the results. Each score represents the percentage of identified key image features that are matched between screenshots of the two applications. The results showed that Droid Box and Droid Box Pro has 90% identical key image features, which furthered asserted our result in Phase II about these two applications. Figure B2 in *Appendix B* shows exactly which features are identified and which matches are made between screenshots of the two apps. Figure B3 in *Appendix B* shows the matching for another pair of applications that are identified as "not similar" by our results.

CONCLUSION

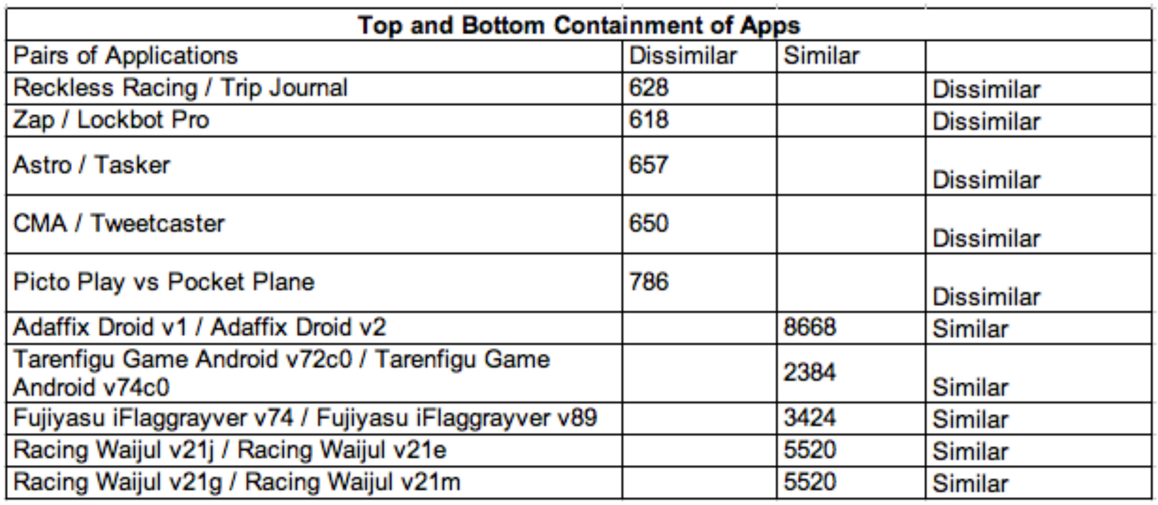
Overall, we achieved a fairly high level of success tackling the challenge of GUI feature extraction from Android applications. Using our comparison function, we were able to somewhat successfully classify the degree of similarity between applications. Unfortunately four out of the five pairs that scored the highest were actually false positives after we performed a manual analysis. However, using a two-way containment comparison all five of the top scoring pairs were not false positives eliminating misleading results caused by large layout file sizes. Additionally, we achieved some success with processing screenshots and doing an image comparison on them using SIFT. The results of our study indicate that there is much potential for further analysis and classification of Android applications and that it is very possible to eventually be able to quickly and efficiently determine if a seemingly benign application is actually performing malicious tasks.

Appendix A: Graphs/Tables



Top Similar/Dissimilar Apps			
Pairs of Applications	**Dissimilar**	**Similar**	**Manual Analysis**
Sheriff Android / Series para Android	340		Dissimilar
Gravedefence / Series para Android	357		Dissimilar
Image Clock /com.sumahootoku.cooponget	365		Dissimilar
Monopolyhere Now /com.sumahootoku.cooponget	366		Dissimilar
Tanks / Series para Android	370		Dissimilar
Extensive Notes Pro / Absolute System Root Tool		14090	Dissimilar
Pictoplay / Logmein Ignition		14612	Dissimilar
Decaf Amazon EC2 Client/ Tasker		15903	Dissimilar
Trip Journal / Total Recall Call Recorder		16171	Dissimilar
Moon Reader Pro / Tweetcaster Pro		16896	Similar

Figure A1: Classification of 5 randomly chosen pairs of dissimilar applications and 5 randomly chosen pairs of similar a pplications. The green represents the pairs our score indicate to be similar based, and the red dissimilar pairs of applications. The bar graph shows the scores of each of these pairs.



Top and Bottom Containment of Apps			
Pairs of Applications	Dissimilar	Similar	
Reckless Racing / Trip Journal	628		Dissimilar
Zap / Lockbot Pro	618		Dissimilar
Astro / Tasker	657		Dissimilar
CMA / Tweetcaster	650		Dissimilar
Picto Play vs Pocket Plane	786		Dissimilar
Adaffix Droid v1 / Adaffix Droid v2		8668	Similar
Tarenfigu Game Android v72c0 / Tarenfigu Game Android v74c0		2384	Similar
Fujiyasu iFlaggrayver v74 / Fujiyasu iFlaggrayver v89		3424	Similar
Racing Waijul v21j / Racing Waijul v21e		5520	Similar
Racing Waijul v21g / Racing Waijul v21m		5520	Similar

Figure A2: Classification of 5 randomly chosen pairs of dissimilar applications and 5 randomly chosen pairs of similar applications. The green represents the pairs our score indicates to be similar, and the red marks dissimilar pairs. The bar graph shows the containment of one application within the other of these pairs.

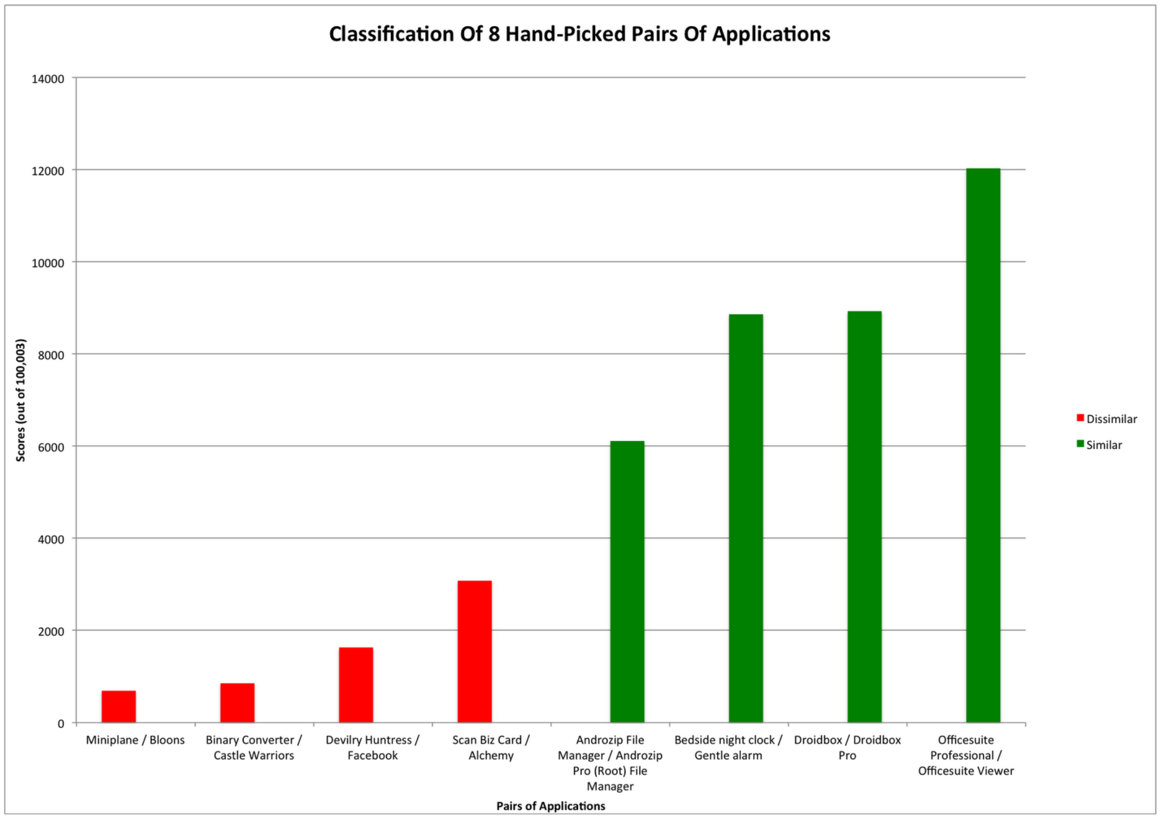
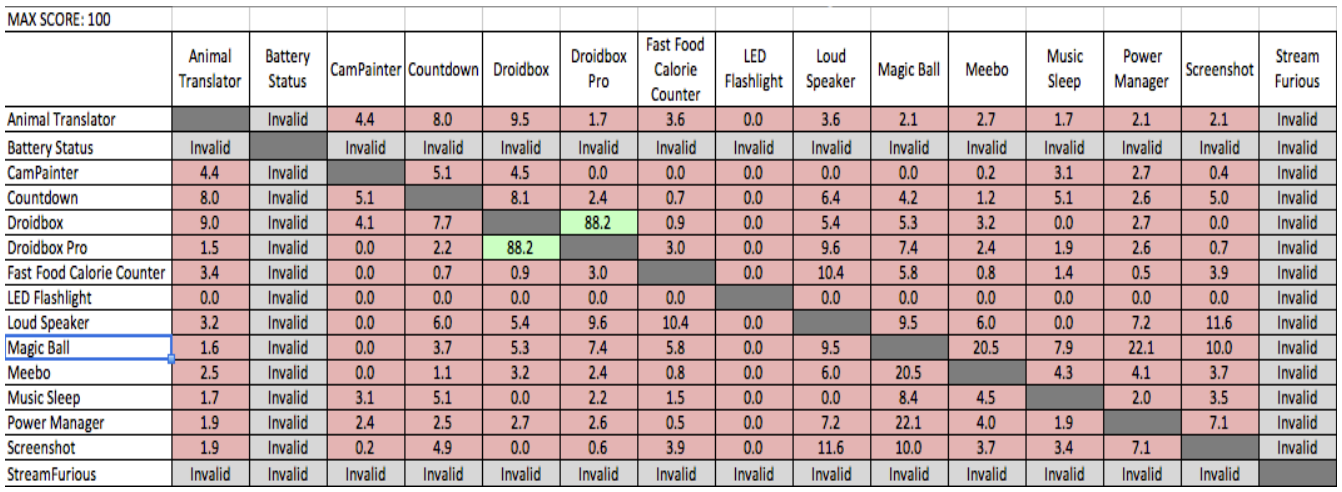


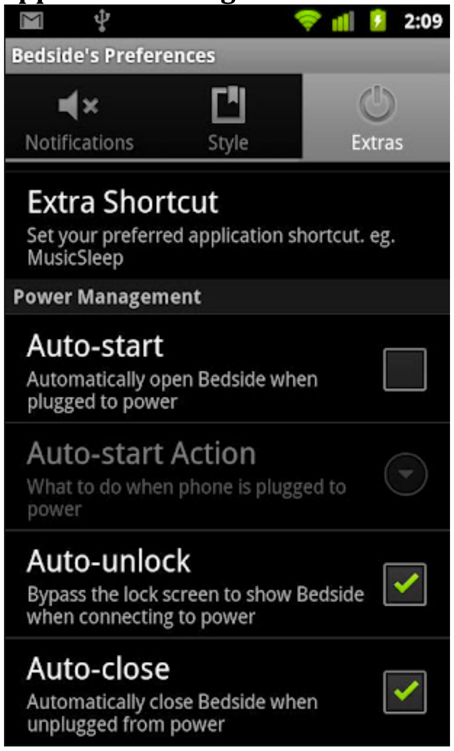
Figure A4: Accuracy of our classification based on 8 hand-picked pairs of applications. The green represents the pairs we predicted to be similar based off of manual analysis, and the red dissimilar pairs of applications. The bar graph shows the scores of each of these pairs.



MAX SCORE: 100															
	Animal Translator	Battery Status	CamPainter	Countdown	Droidbox	Droidbox Pro	Fast Food Calorie Counter	LED Flashlight	Loud Speaker	Magic Ball	Meebo	Music Sleep	Power Manager	Screenshot	Stream Furious
Animal Translator		Invalid	4.4	8.0	9.5	1.7	3.6	0.0	3.6	2.1	2.7	1.7	2.1	2.1	Invalid
Battery Status	Invalid		Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
CamPainter	4.4	Invalid		5.1	4.5	0.0	0.0	0.0	0.0	0.0	0.2	3.1	2.7	0.4	Invalid
Countdown	8.0	Invalid	5.1		8.1	2.4	0.7	0.0	6.4	4.2	1.2	5.1	2.6	5.0	Invalid
Droidbox	9.0	Invalid	4.1	7.7		88.2	0.9	0.0	5.4	5.3	3.2	0.0	2.7	0.0	Invalid
Droidbox Pro	1.5	Invalid	0.0	2.2	88.2		3.0	0.0	9.6	7.4	2.4	1.9	2.6	0.7	Invalid
Fast Food Calorie Counter	3.4	Invalid	0.0	0.7	0.9	3.0		0.0	10.4	5.8	0.8	1.4	0.5	3.9	Invalid
LED Flashlight	0.0	Invalid	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	0.0	Invalid
Loud Speaker	3.2	Invalid	0.0	6.0	5.4	9.6	10.4	0.0		9.5	6.0	0.0	7.2	11.6	Invalid
Magic Ball	1.6	Invalid	0.0	3.7	5.3	7.4	5.8	0.0	9.5		20.5	7.9	22.1	10.0	Invalid
Meebo	2.5	Invalid	0.0	1.1	3.2	2.4	0.8	0.0	6.0	20.5		4.3	4.1	3.7	Invalid
Music Sleep	1.7	Invalid	3.1	5.1	0.0	2.2	1.5	0.0	0.0	8.4	4.5		2.0	3.5	Invalid
Power Manager	1.9	Invalid	2.4	2.5	2.7	2.6	0.5	0.0	7.2	22.1	4.0	1.9		7.1	Invalid
Screenshot	1.9	Invalid	0.2	4.9	0.0	0.6	3.9	0.0	11.6	10.0	3.7	3.4	7.1		Invalid
StreamFurious	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	

Figure A5: Screenshot pairwise comparison based on 15 randomly selected applications from a pool of applications with size less than 500KB . The green represents the pairs that are similar based off of screenshot analysis, and the red dissimilar pairs of applications. The score represents the percentage of key features that are matched between the screenshots of two applications. Application that caused error or didn't run are labeled as Invalid.

Appendix B: Images


2:09

Bedside's Preferences

Notifications | Style | Extras

Extra Shortcut
Set your preferred application shortcut. eg. MusicSleep

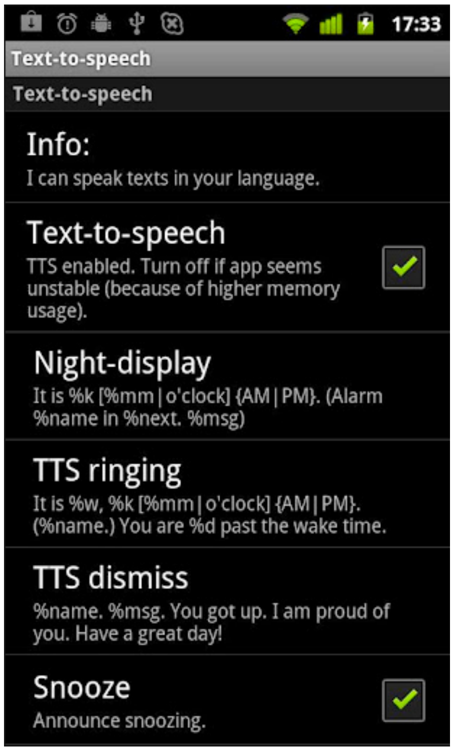
Power Management

Auto-start [ ]
Automatically open Bedside when plugged to power

Auto-start Action
What to do when phone is plugged to power

Auto-unlock [✓]
Bypass the lock screen to show Bedside when connecting to power

Auto-close [✓]
Automatically close Bedside when unplugged from power


17:33

Text-to-speech

Text-to-speech

Info:
I can speak texts in your language.

Text-to-speech [✓]
TTS enabled. Turn off if app seems unstable (because of higher memory usage).

Night-display
It is %k [%mm | o'clock] {AM | PM}. (Alarm %name in %next. %msg)

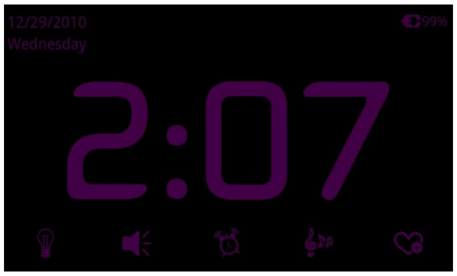
TTS ringing
It is %w, %k [%mm | o'clock] {AM | PM}. (%name.) You are %d past the wake time.

TTS dismiss
%name. %msg. You got up. I am proud of you. Have a great day!

Snooze [✓]
Announce snoozing.

Bedside night clock

Gentle Alarm



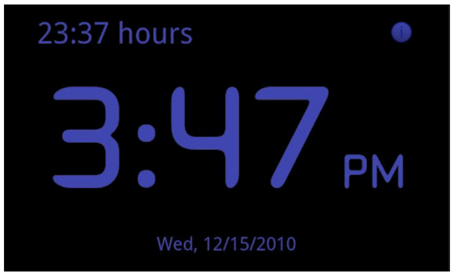


Figure B1: A juxtaposition of two screenshots of both Bedside Night Clock (left) and Gentle Alarm (right), two alarm clock Android applications.

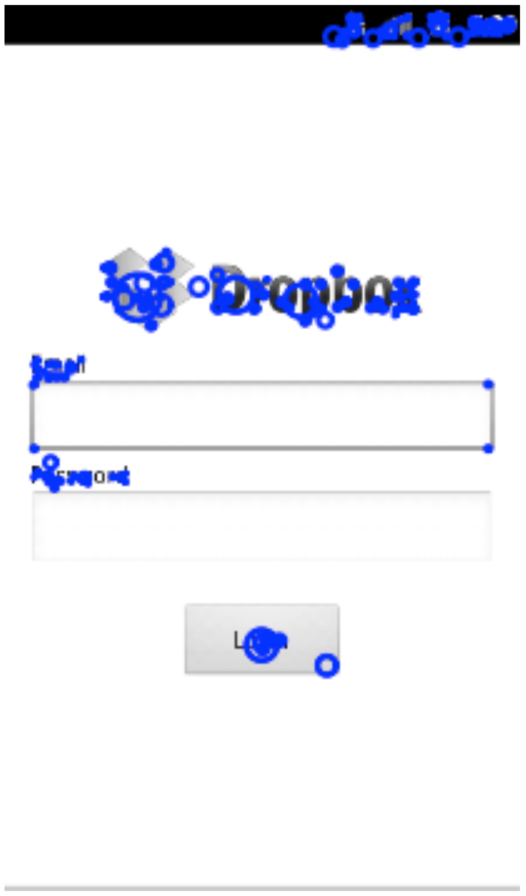
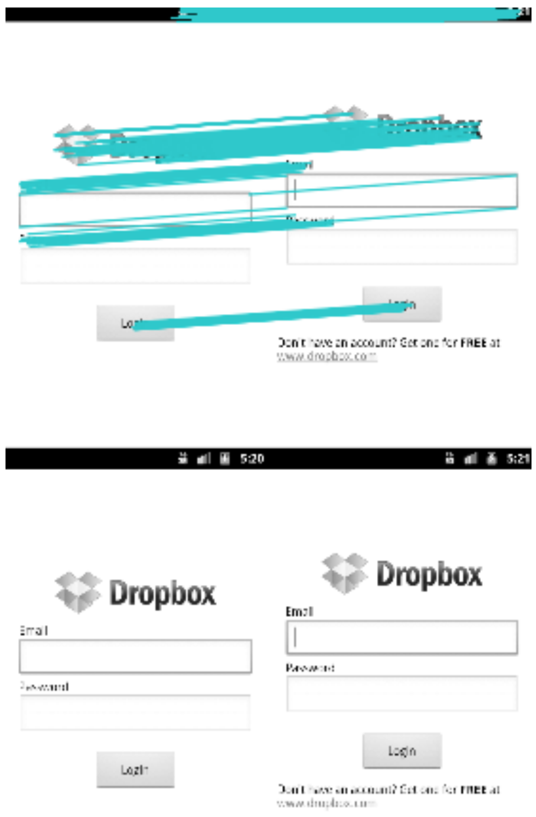



Figure B2: The feature extraction of Droid Box application (left) and the feature matching between Droid Box and Droid Box Pro(right).

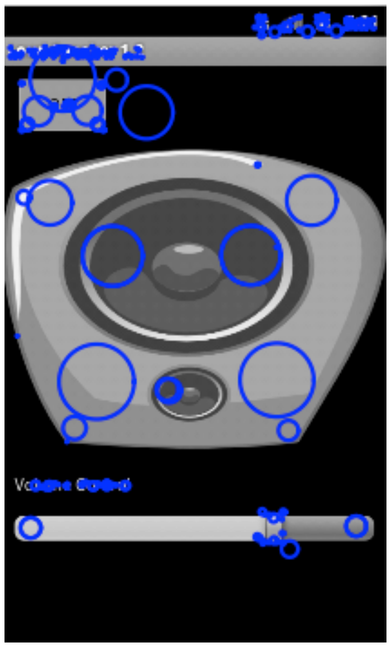
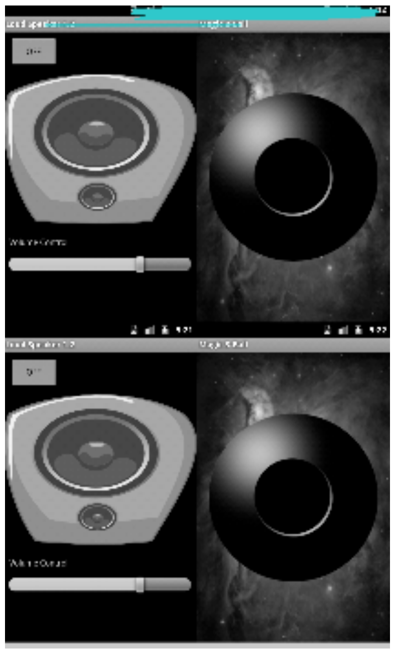



Figure B3: The feature extraction of Loud Speaker (left) and the feature matching between Loud Speaker and Flashlight(right).

Appendix D: Manual Analysis of Hand-picked Applications

Similar Apps

Officesuite professional and Officesuite viewer
Score:
12025

Similarities:
Made by the same developer
Category: Business
Type: File manager
Deals with office documents on the Android phone
AD Free
Compatible with the most commonly used desktop document formats

Differences:
Professional: view, create and modify office documents
Viewer: ONLY open/view office documents

Bedside night clock and Gentle alarm
Score:
<insert later>

Similarities:
Category: Tools
Type: Alarm clock
Night displays and bedside mode
Ability to choose own colors/font size and backgrounds
Text-to-Speech/speech input and output
Dock support

Differences:
gentle alarm: Snooze/math equations
silent alarm with slowly increasing vibration

Androzip File Manager and Androzip Pro (Root) File Manager
Score:
<insert later>

Similarities:
Category: Productivity
Type: File Manager
Made by the same developer
Enable extraction of zip, rar, gzip, tar, bzip2 files
Enable creation of zip, tar, gzip files
Send files via email

Differences:
One works on root filesystem while the other doesn't

Droidbox and Droidbox Pro
Score:
<insert later>

Similarities:
Category: Productivity
Type: DropBox Accessor
Made by the same developer
Enable users to upload and download files from their Dropbox accounts

Differences:
Free application offers a 20 file upload limit -- GUI appears to be the same for both

Dissimilar Apps
Miniplane vs. Bloons (might not actually have the score -- check this later)
Score:
1809

Similarities:
Category: Games
Type: Alarm clock
Many different levels/scenes

Differences:

Miniplane: action game that requires constant interaction
Bloons: strategy game consisting of phases of action and viewing

Devilry Huntress vs. Facebook

Score:
<insert later>

Similarities:
Both are used for entertainment

Differences:
Devilry Huntress: game category
Facebook: social networking category
Devilry Huntress: runs without Internet connection
Facebook: relies heavily on constant data transfer
Devilry Huntress: single player
Facebook: "multi player"

Scan Biz Cards vs. Alchemy

Scores:
<insert later>

Similarities:
Both have full internet access permission

Differences:
Alchemy: game category
Scan Biz Cards: business category
Alchemy: does not require others to be fully utilized
Scan Biz Cards: requires a lot of people to be fully utilized

Binary Converter vs. Castle Warriors

Scores:
<insert later>

Similarities:
Both do not require special permissions to run

Differences:
Castle Warriors: game category
Binary Converter: tools category
Castle Warriors: highly interactive, requires constant attention and focus when using
Binary Converter: utilized only when you need to, does not require constant attention

References

1. "Q1 Threats Go Mobile." 17 April 2012. Trend Micro. Date Accessed: 1 May 2012. http://blog.trendmicro.com/q1-threats-go-mobile/

2. djb2 Hash Function. Lukáš Lalinský. 03 April 2010. Stackoverflow. Date Accessed: 20 March 2012. http://stackoverflow.com/questions/2571683/djb2-hash-function

3. Android Developers: Security and Permissions. 01 May 2012. Google/Android. Data Accessed: 23 February 2012. http://developer.android.com/guide/topics/security/security.html

4. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. 12 Jan 2012. Intel Science And Technology Center on Secure Computing at UC Berkeley. Data Accessed: 6 May 2012. http://www.berkeley.intel-research.net/~hling/research/juxtapp_ling.pdf

Tools Used

Bit-array Python extension **- http://pypi.python.org/pypi/bitarray**
SIFT - **http://blogs.oregonstate.edu/hess/code/sift/**
SIFT - **http://www.cs.ubc.ca/~lowe/keypoints/**
Automating the screenshots **http://testdroid.com/tech/54/automated-ui-testing-android-applications-robotium**
apk->xml decompilation **http://vividmachines.com/dexdump.tgz**