Spring 2018

Using Byte Code to Find Idiosyncratic Android Camera Apps

Christopher Blake Burnley
Bard College

Using Byte Code to Find Idiosyncratic Android Camera Apps

A Senior Project submitted to The Division of Science,

Mathematics, and Computing of Bard College

By

Blake Burnley

Annandale-on-Hudson, New York

May 2018

Abstract

The growing popularity of Android devices has made them an increasing target for malicious apps. Of course, a malicious app is most effective when a user is not aware of its intent. Therefore, they often take the form of ostensibly benign, helpful, and, for that matter, free applications. Our goal is to discover how common this is among free camera applications. Camera applications are a good test case because they ought to be very simple. This paper uses static analysis on the Dalvik byte code of camera applications to search for certain characteristics or identify applications that demonstrate behavior that is not expected by the user.

Acknowledgements

I would like to thank...

- My parents for helping and supporting me through everything
- Professor Robert McGrail for keeping me on track and pushing me every week
- My coaches and teammates for giving me an opportunity each day to take a break from Senior Project and my classes

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Android devices have consistently been targets of applications that seek to behave either maliciously or in a way that does not match its stated purpose. This is due in part to the popularity of Android devices and the ease of publishing an app. In this paper we seek to use byte code analysis to identify those applications that employ features or demonstrate behavior that one would not expect from the purpose of the application.

On the Google Play store alone there are millions of free apps, thousands of which are camera apps.[1] A trend that is noticeable in the Play Store with camera apps, as well as other types, is an over-saturation of free apps. There are countless applications that do the same thing as another app, with no obvious distinction. Potential motivations for the large number of free Android applications are wanting to create apps for resumes, code piracy, and attempting to disguise malicious apps as useful ones.

In the past the Google Play Store has had a problem with the number of malware apps that make their way into the marketplace. These apps appear to be legitimate, sometimes even mirroring popular apps. The creators of these apps want to attract the largest possible number of users so they create free apps in hopes that more people will download them.

This paper will highlight the key aspects to understanding and evaluating Dalvik byte code. It will also explain the process that is needed to get the data from an APK (Android Package Kit) file so that we can understand its nature. The paper will then look at the data it created and explain the conclusions that can be drawn from the data.

1.2 Related Works

The DroidAPIMiner project by Yousra Aafer, Wenliang Du, and Heng Yin from the Department of Electrical Engineering & Computer Science at Syracuse University uses static analysis on the byte code of Android applications to classify those applications. The project sought to build a classifier for Android apps that would overcome the shortcomings of permission-based warning mechanisms. The paper focused on API level information in the byte code, the package levels, and their parameters, in an attempt to classify malware. From this paper we used the Android methods that they identified as having a higher correlation towards malware apps than benign apps.

Chapter 2

Background

2.1 Background

Using the APK files for various camera applications we sought to collect their byte code and compare the invocation of specific Android API level methods. We wanted to look for apps that might be demonstrating suspicious behavior and did so through static analysis of the byte code. Suspicious behavior might include making method calls that are irrelevant or not needed for the purpose of an application.

Malicious behavior in Android applications has been an ongoing issue. An app might seem, on the surface, to be a genuine app, but underneath it might be performing some suspicious activities. An application might go “off the rails” when it starts to demonstrate behavior that we would not expect for a benign app or one of that type. This type of behavior might include bitcoin mining, for example. Clearly the programmer would want to mask their activity, as most users would not consent to this type of behavior on their device. This is why many malware applications appear as legitimate, free applications, because they hope to attract the largest possible number of users in order to take over as many devices as possible.

2.2 Static Analysis

Static analysis is the analysis of computer code. Unlike dynamic analysis, which executes a program and analyzes its behavior, static analysis functions by considering the program before execution. The level of analysis that is used can vary from looking at individual functions to looking at the complete source code. Static analysis can be used to find various errors in a program such as potential coding errors, program-breaking bugs, and potentially malicious behavior.

One of the most common forms of static analysis is type checking. Type checking is performed by the compiler whenever a program is compiled.[2] Type checking is important because it guarantees a few things before executing a program. First it makes sure that the arithmetic operations are valid and computable. For example, in a language like Java, it makes sure that you can not add an integer to a Boolean. It also makes sure that functions are called with the proper number and type of arguments. One of the last things that type checking guarantees is undeclared variable analysis. For example, in Java, this makes sure that an undeclared variable will never be read by the program.

Another form of static analysis that is executed in the compiler is optimization. Optimization is useful in converting code into a format that is better in terms of run-time and memory for the machine. It performs operations like simplifying arithmetic equations into fewer steps and carrying out partial evaluation in a loop during compilation so that it doesn't have to recompute.

Another implementation of static analysis is on a whole program level, not just type checking. Several systems, where a failure during use could have catastrophic effects, rely on static analysis instead of dynamic analysis.[3] These systems include things like medical devices, nuclear power plant software, and aviation software.[4] In all of these systems, static analysis is the preferred method, since dynamic analysis would open the systems up to vulnerabilities or potential hazards at runtime.

2.3 Byte Code

Byte code is the machine language of the Java Virtual Machine (JVM) and Dalvik Virtual Machine(DVM), which are what allow a computer to execute either Java or Dalvik code. Android applications specifically rely on Dalvik byte code and the Dalvik Virtual Machine since the APK file contains the .dex code, which is in Dalvik form.[5]

Dalvik byte code was created specifically for the Dalvik Virtual Machine. This has since been replaced with the Android Runtime Environment (ART). The Android Runtime Environment still uses the Dalvik byte code and .dex files in order to maintain backwards compatibility. In order to understand the Dalvik byte code it is important to understand the DVM and how it works.

The DVM is a virtual machine that is optimized for use on mobile devices. This works by compiling the Java code into byte code for the JVM, which is then translated into Dalvik byte code and saved in a .dex file.[6] The contrast is illustrated in Figure 2.1.

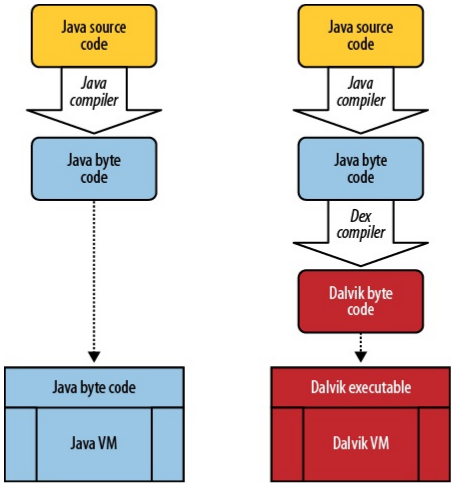
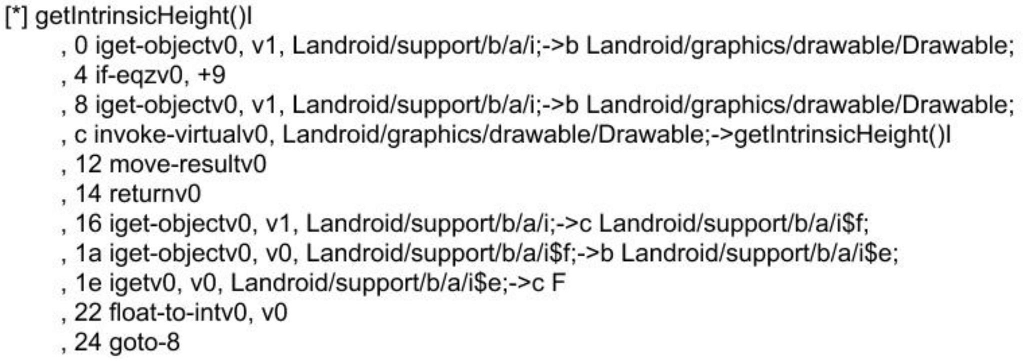


Figure 2.1: Java vs Dalvik Byte Code [7]

The Dalvik virtual machine is register-based and has frames of a fixed size that are set upon creation. Since the virtual machine is register-based and not stack-based, it needs fewer, but more complex instructions.[8] Each frame is made up of only a particular number of registers which is specified by the method.[9] The Dalvik byte code works by feeding multiple opcodes to the virtual machine which serve as a set of instructions. Whenever a virtual machine executes a program it receives a stream of opcodes for each method in the class, which tell the virtual machine how to execute the program.

Figure 2.2 is an example of the byte code for a method. The first line is the method being declared. The next line, 0, has an opcode *iget-objectv0*. In this line the virtual machine reads an object reference instance field into x, or *Landroid/support/b/a/i*, and then the instance is referenced by y, in this case *Landroid/graphics/drawable/Drawable*. The next line, 4, looks to see if $v0 == 0$ and if so it jumps to 9. Line c has an opcode *invoke-virtual*, which is where it invokes a virtual method. The method it is invoking is outline by the pattern, 'class;->method'. In this case the class is *Landroid/graphics/drawable/Drawable* and the method is *getInstrinsicHeight*. The next line moves the result from the previous method to $v0$ and the following one returns the value of $v0$. The next three lines are all similar to line 0. Line 22 converts a float into an integer, in this case it is converting $v0$. The last line tells it to jump back up to line 8.



```
[*] getIntrinsicHeight()I
      , 0 iget-objectv0, v1, Landroid/support/b/a/i;->b Landroid/graphics/drawable/Drawable;
      , 4 if-eqzv0, +9
      , 8 iget-objectv0, v1, Landroid/support/b/a/i;->b Landroid/graphics/drawable/Drawable;
      , c invoke-virtualv0, Landroid/graphics/drawable/Drawable;->getIntrinsicHeight()I
      , 12 move-resultv0
      , 14 returnv0
      , 16 iget-objectv0, v1, Landroid/support/b/a/i;->c Landroid/support/b/a/i$f;
      , 1a iget-objectv0, v0, Landroid/support/b/a/i$f;->b Landroid/support/b/a/i$e;
      , 1e igetv0, v0, Landroid/support/b/a/i$e;->c F
      , 22 float-to-intv0, v0
      , 24 goto-8
```

Figure 2.2: Dalvik Byte Code Example

Since smart phones have less memory than modern laptop or desktop computers the DVM is optimized for low memory requirements and has specific characteristics that differentiate it from other standard virtual machines. For example, the constant pool has been modified to use only 32-bit indices to simplify the interpreter. Standard Java bytecode executes 8-bit stack instructions and local variables must be copied to or from the operand stack by separate instructions. Dalvik instead uses its own 16-bit instruction set that works directly on local variables. The local variable is commonly picked by a 4-bit "virtual register" field. This lowers the instruction count and raises the interpreter speed.[10] These features make it more efficient for use on portable devices, where memory is limited.

2.4 APK Files

APK files, which stands for Android Package Kit, are used by the Android operating system to distribute and install applications onto the devices. APKs are a type of archive file, like ZIP and JAR files. The APK contains all of the program's code including the .dex files, which we use to analyze the byte code. The files are created by compiling a program for Android and packaging all of its components into one file. A user can download APK files from official sources, like the Google Play Store, or from other unofficial marketplaces. A user can also install an APK directly from their computer via third party applications.

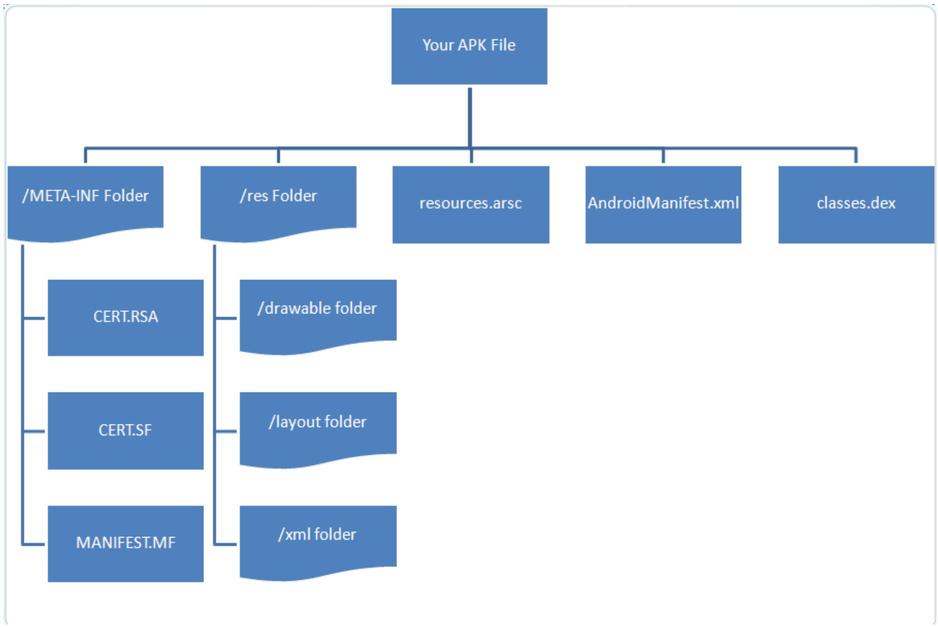


Figure 2.3: APK File Structure [11]

Figure 2.3 shows the file structure of an APK file. At the top level is the APK. Inside the APK file are two sub-directories, META-INF and res. In the META-INF folder is the application's meta data on the contents of the APK, as well as the signature for it. In the res folder are three sub-folders. The first is drawables which is where the .png files are located, which are used to generate the images for the application. The next subfolder, layout, consists of binary xml files. The third subfolder is xml, which has more binary xml files for the application. There are then three files, resources.arsc, AndroidManifest.xml, and classes.dex. The first file, resources.arsc, contains the compressed resources file, which helps direct the application to its various resources. The AndroidManifest.xml file presents essential information about the application to the Android system. This is information the system must have before it can run any of the application's code. Finally the classes.dex file, the one we are interested in, contains the Java code and all the classes converted into Dalvik byte code.

2.5 Androguard

Androguard is an open source APK decompilation program written in Python that allows one to reverse engineer an application.[12] The application allows one to access the .dex file in the APK and the various components that it comprises. A user is able to create a DalvikVMFormat object which then allows them to access the Java class files that make up the .dex file. From here one can retrieve various pieces of information, such as all the classes and methods, the source code, and the Dalvik byte code. For our purposes, we only focused on the byte code and the methods invoked within it.

2.6 Mathematica

Wolfram Mathematica is a technical computing system developed by Wolfram Research. Its functionality covers a wide range of technical computing, including neural networks, machine learning, image processing, geometry, data science, and visualizations.[13] The program is useful for generating various graphs and models, based off of different statistical methods.

We chose Mathematica because we needed statistical inference to examine and compare the APK files. While it is easy to get the raw data, Mathematica does a good job converting the raw data into something useful that we can analyze. We specifically used Mathematica to generate clusters and graphs for our data.

The main functions that we used in Mathematica were *FindClusters* and *ListPlot*. The FindClusters function allowed us to generate clusters for our scatter plot data using different clustering algorithms such as k-means, mean shift, and Gaussian mixture.

Chapter 3

Methods

3.1 Overview

Given a camera application we want to examine its byte code in order to determine whether the application is behaving in a way that one would expect. In order to do so we considered the methods that were invoked in the byte code. For each application we extracted the byte code and counted the number of times that it invoked a particular "suspicious" method. The following sections are sequentially ordered and contain brief explanations of the process at each step. Figure 3.1 is a visual representation of the process we used.

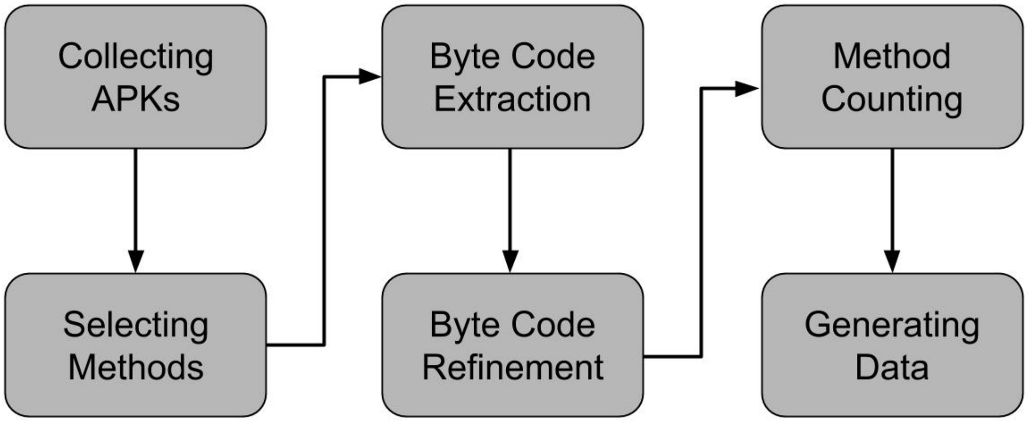


Figure 3.1: Process Flow Chart

3.2 Collecting APKs

The first step was gathering applications to make up the dataset. All of the applications that we used came from the Google Play Store. Since we wanted to focus on camera apps we gathered a sample of 100 unique, free camera apps. We chose free apps in order to control cost and because many malicious apps are made free in order to attract the largest number of users. To download an application's APK file we used the desktop version of the Google Play Store and the website APK Downloader.[14][15] The website allowed us to download an APK file without using a physical Android device. We downloaded each application by hand.

3.3 Selecting Methods

In order to determine whether an app was behaving in a way that we expected we needed a set of specific method calls to look for. We wanted to look for method calls that would be consistent across all Android applications, so we looked at Android API level methods. We also wanted method calls that would be indicative of behavior that we would not expect for an Android camera application. Again we could use the Android API methods for this.

The methods that we searched for came from the DroidAPIMiner paper, which found the method calls that were most frequent in a set of malware apps compared to a set of benign apps. From the paper we selected the twenty method calls that were most frequent in the set of malware applications. Figure 3.2 displays the twenty methods we used and the difference in frequency that were found in malware apps compared to benign apps.

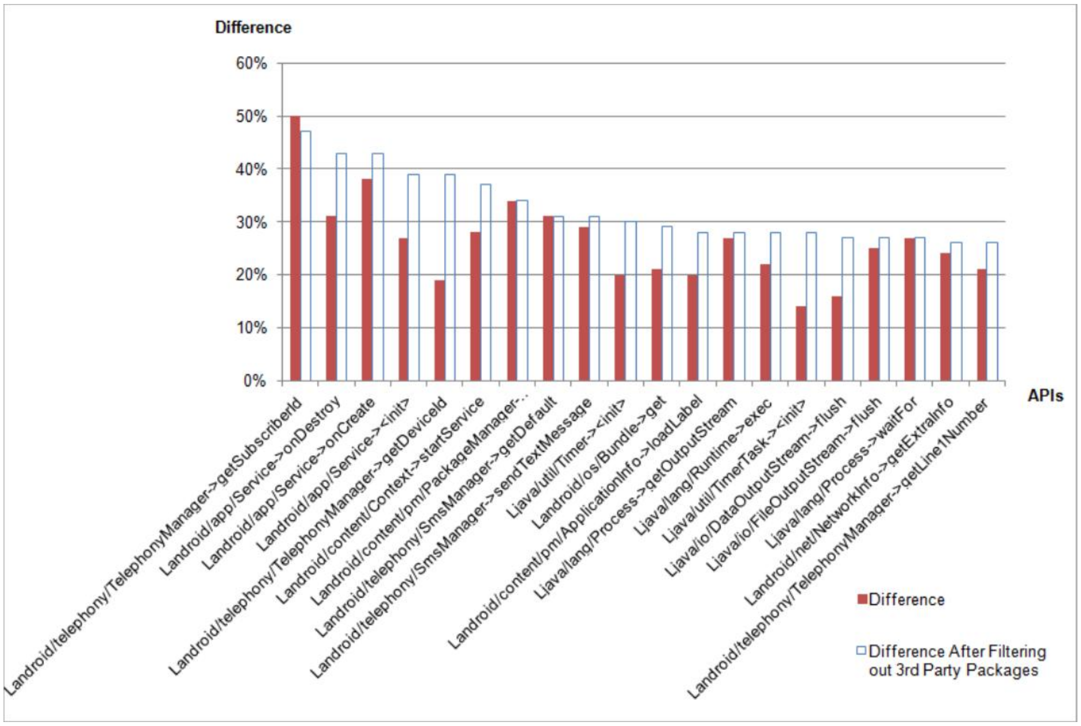


Figure 3.2: Top 20 Methods with the Highest Difference Between Malware and Benign Apps [16]

3.4 Byte Code Extraction

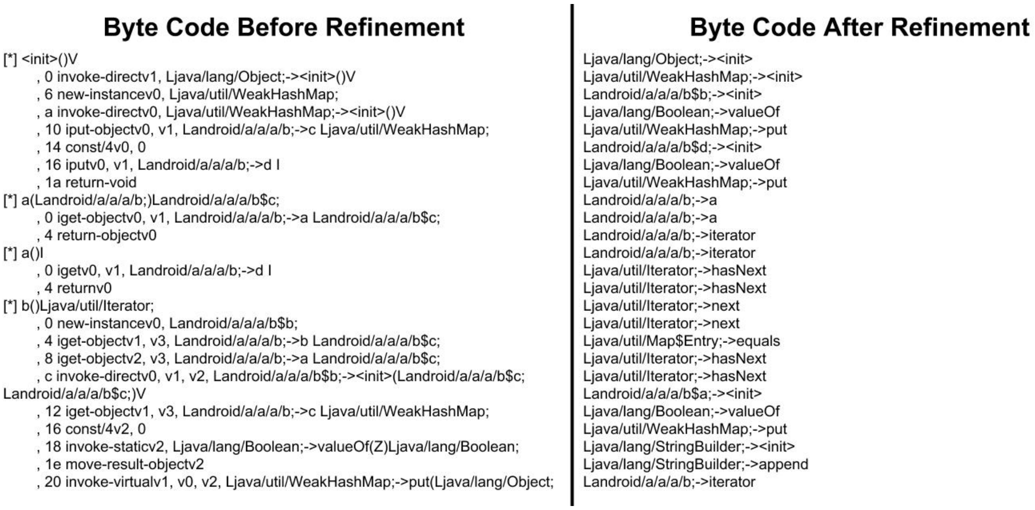
The next step in analyzing an application after downloading the APK file is to extract the byte code. We extracted the byte code from the APK file by writing a script based off of the Androguard platform and the tutorial written by Keith Makan. [12][17]

```
def dump_methods(file, outfile):
    a = apk.APK(file)
    d = dvm.DalvikVMFormat(a.get_dex())
    bcf = open(outfile, "w+")
    for current_class in d.get_classes():
        for method in current_class.get_methods():
            bcf.write("[*]␣"+method.get_name()+ method.get_descriptor()+"\n")
            byte_code = method.get_code()
            if byte_code != None:
                byte_code = byte_code.get_bc()
                idx = 0
                for i in byte_code.get_instructions():
                    bcf.write("\t,␣%x␣" % (idx)+i.get_name()+i.get_output()+"\n")
                    idx += i.get_length()
```

The script takes in an APK file for an application and writes the byte code to a specified file. The script uses the Androguard program to create both an apk and dvm object. With the dvm object we were able to loop over all of the classes of the application and the methods for each class. Each time we looped over a method we generated the byte code for it and wrote the byte code out to a file.

3.5 Byte Code Refinement

Once we generated the byte code for an application, we needed to refine it in order to efficiently and correctly search for specific methods. Removing unnecessary lines of byte code so that it is just a list of the methods that were invoked allows for more efficient searching. Our program loops over a file and, the regular expression *(invoke.+,)(.+)(\(()*, finds all of the instances in the byte code where a method was invoked. It then writes the specific methods' name to a file, removing both the Dalvik opcodes and the parameters used in the method. Figure 3.3 demonstrates the difference in the byte code before and after it is refined.



Byte Code Before Refinement

```
[*] <init>()V
        , 0 invoke-directv1, Ljava/lang/Object;-><init>()V
        , 6 new-instancev0, Ljava/util/WeakHashMap;
        , a invoke-directv0, Ljava/util/WeakHashMap;-><init>()V
        , 10 iput-objectv0, v1, Landroid/a/a/a/b;->c Ljava/util/WeakHashMap;
        , 14 const/4v0, 0
        , 16 iputv0, v1, Landroid/a/a/a/b;->d I
        , 1a return-void
[*] a(Landroid/a/a/a/b;)Landroid/a/a/a/b$c;
        , 0 iget-objectv0, v1, Landroid/a/a/a/b;->a Landroid/a/a/a/b$c;
        , 4 return-objectv0
[*] a()I
        , 0 igetv0, v1, Landroid/a/a/a/b;->d I
        , 4 returnv0
[*] b()Ljava/util/Iterator;
        , 0 new-instancev0, Landroid/a/a/a/b$b;
        , 4 iget-objectv1, v3, Landroid/a/a/a/b;->b Landroid/a/a/a/b$c;
        , 8 iget-objectv2, v3, Landroid/a/a/a/b;->a Landroid/a/a/a/b$c;
        , c invoke-directv0, v1, v2, Landroid/a/a/a/b$b;-><init>(Landroid/a/a/a/b$c;
Landroid/a/a/a/b$c;)V
        , 12 iget-objectv1, v3, Landroid/a/a/a/b;->c Ljava/util/WeakHashMap;
        , 16 const/4v2, 0
        , 18 invoke-staticv2, Ljava/lang/Boolean;->valueOf(Z)Ljava/lang/Boolean;
        , 1e move-result-objectv2
        , 20 invoke-virtualv1, v0, v2, Ljava/util/WeakHashMap;->put(Ljava/lang/Object;
```

Byte Code After Refinement

```
Ljava/lang/Object;-><init>
Ljava/util/WeakHashMap;-><init>
Landroid/a/a/a/b$b;-><init>
Ljava/lang/Boolean;->valueOf
Ljava/util/WeakHashMap;->put
Landroid/a/a/a/b$d;-><init>
Ljava/lang/Boolean;->valueOf
Ljava/util/WeakHashMap;->put
Landroid/a/a/a/b;->a
Landroid/a/a/a/b;->a
Landroid/a/a/a/b;->iterator
Landroid/a/a/a/b;->iterator
Ljava/util/Iterator;->hasNext
Ljava/util/Iterator;->hasNext
Ljava/util/Iterator;->next
Ljava/util/Iterator;->next
Ljava/util/Map$Entry;->equals
Ljava/util/Iterator;->hasNext
Ljava/util/Iterator;->hasNext
Landroid/a/a/a/b$a;-><init>
Ljava/lang/Boolean;->valueOf
Ljava/util/WeakHashMap;->put
Ljava/lang/StringBuilder;-><init>
Ljava/lang/StringBuilder;->append
Landroid/a/a/a/b;->iterator
```

Figure 3.3: Byte Code Before and After Refinement

3.6 Method Counting

With the list of methods invoked we sought to count the number of times that each distinct method appeared in the application. Looping across a file that contained all the methods we were able to create a dictionary for all the methods. Each time the script came across a method that was not in the dictionary it added the method to the dictionary. When it came across a method that was in the dictionary it increased the count for that method by one. When the method finished running it would write out the information to a csv file for analysis.

3.7 Generating Data

After counting the number of times each method appeared we looked for the number of "suspicious" methods in each application. To count the number of "suspicious" methods in each application we wrote a Python script that looped over all the counted methods. When it came across a method that was in the set of "suspicious" methods it retrieved the count for that specific method and added it to the total count of suspicious methods for that application. We ran this script on our 100 apps to generate the dataset.

Chapter 4

Data

For our data we determined which methods each application invoked via the byte code. This required us to extract the byte code from each application and analyze it. We counted the number of times that each unique method was invoked in an app. We used this to make it easier to count the number of times “suspicious” methods were invoked. After we had the total count of method invocations we looked at the number of times an app invoked a “suspicious” method. The methods we considered for were the ones generated from the *DroidAPIMiner* paper, as described in the Section 3.2. From those two values we calculated the proportion of method calls in each app that we classified as “suspicious” methods.

Figure 4.1 is a visual representation of the data. Each dot represents an application. The x-axis is the total number of method invocations for each app and the y-axis is the number of "suspicious" invocations.

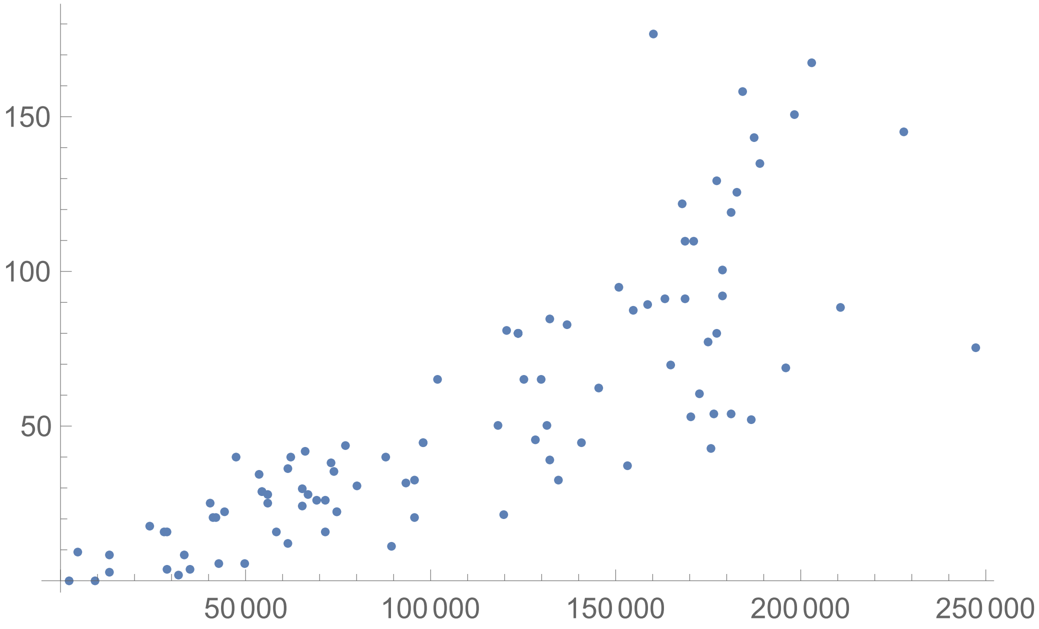


Figure 4.1: Total Number of Methods vs "Suspicious" Methods

We used a scatter plot to represent the data in order to show the relationship between the total number of methods and the number of "suspicious" methods. Plotting this relationship allowed us to understand a few things. The first is that we get a sense of the number of "suspicious" methods that an app invokes compared to the other applications by looking at its y-value. If an app invokes a lot of "suspicious" methods then we might want to examine it further.

We also get a sense of what percentage of the method calls are "suspicious" by comparing it to the total number. Knowing the proportion of method calls that are "suspicious" is helpful because it allows us to see if a particular app makes "suspicious" calls more frequently than we would expect. This can be helpful if there is a small app that might only have few method invocations, but a large proportion of which are considered "suspicious" or a large one that, by its nature invokes a lot of methods, but invokes "suspicious" ones at a rate that we would expect.

Some noteworthy apps from the results are apps 3 and 63, which did not have any "suspicious" method calls. App 66 had the most "suspicious" method calls with 177, but app 62 had the highest proportion of "suspicious" method calls at 0.001833367285. There was also a group of applications that appeared to be identical to each other. Apps 13, 14, 36, 77, and 88 all had the same number of total methods as well as the same number of "suspicious" methods. Table B.1 in Appendix B shows full list of all the applications and the results from the tests.

Chapter 5

Analysis

5.1 Intro

We used multiple clustering algorithms via Mathematica to analyze our data. The various algorithms yielded results that were slightly different from each other which provided distinct interpretations of the data. Clustering the data is a useful step since the clusters as a whole can tell us more than each individual data point can. By looking at clusters of data we get a sense of different types or characteristics that might be in the applications.

5.2 Mean Shift Clustering

In order to cluster the data we used the mean shift clustering algorithm. The mean shift algorithm produced three clusters from our data. The first cluster included the applications that had less than 110,000 method invocations, the second containing those in between 110,000 and 150,000 method invocations, and the third containing the applications with over 150,000 method invocations. The results of this method can be seen in Figure 5.1.

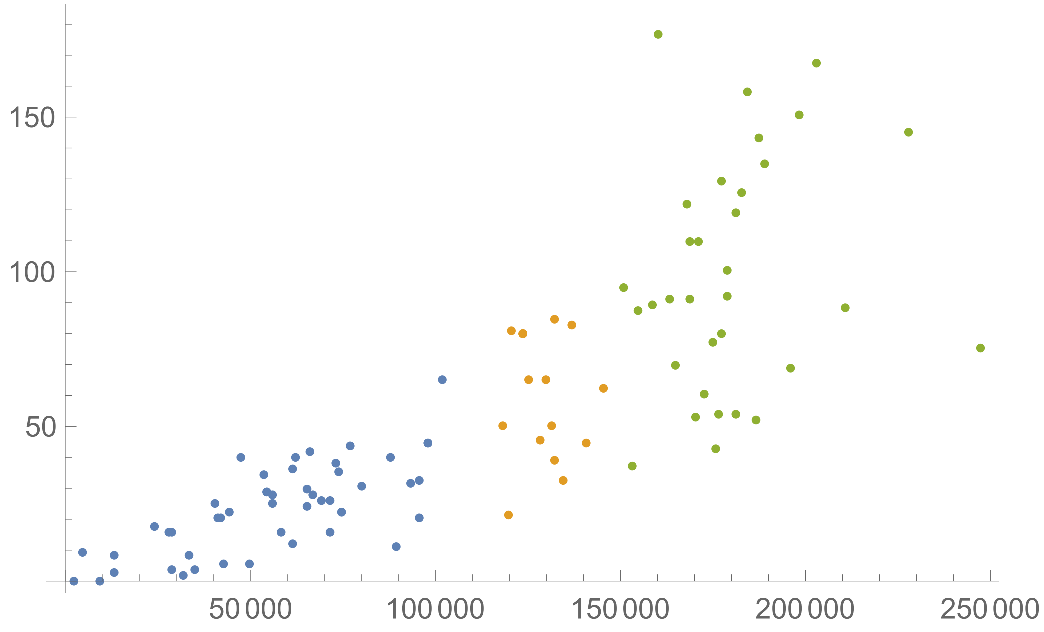


Figure 5.1: Mean Shift Clustering

Mean shift clustering is an analysis technique for locating the maxima of a density function. It is known as a mode-seeking algorithm. The mean shift algorithm works by finding a random point and window W. It then calculates the center or mean of W and shifts the search window to the mean. It continues this process until convergence. To create multiple clusters one creates points at all the data points and performs the mean shift algorithm until convergence. One then merges the windows that end up around the same mode.

5.3 Agglomerate Clustering

The agglomerate clustering method is similar to some of the others since it uses the Euclidean distance to group data points. The algorithm starts by considering each point as an individual cluster. At each step it merges the closest pair of clusters, based on Euclidean Distance, into one larger cluster. The process repeats until the desired number of clusters remain. The results of this method are represented in Figure 5.2

Unlike the mean shift algorithm the agglomerate clustering method resulted in five clusters. A few of these clusters contained only a handful of

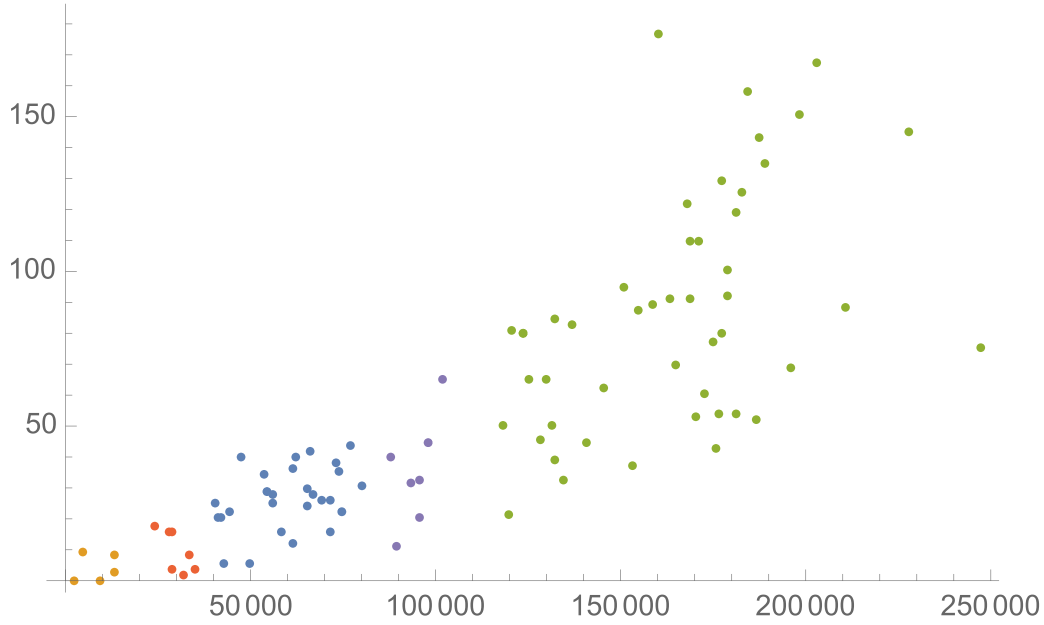


Figure 5.2: Agglomerate Clustering

applications, but the last cluster was by far the largest containing 51 of the 100 applications. The last cluster contained all the applications that invoked more than 120,000 methods. The rest of the applications were divided into four clusters.

5.4 Gaussian Clustering

We used a Gaussian mixture algorithm to see if there were any subpopulations within the data. The model is useful for identifying subpopulations when one is unsure of their existence. This method of clustering differs from the others in that the model is based on probability. Instead of looking to see how close two points are, it looks to see whether a dataset displays a normal, or Gaussian distribution. The resulting cluster is demonstrated in Figure 5.3.

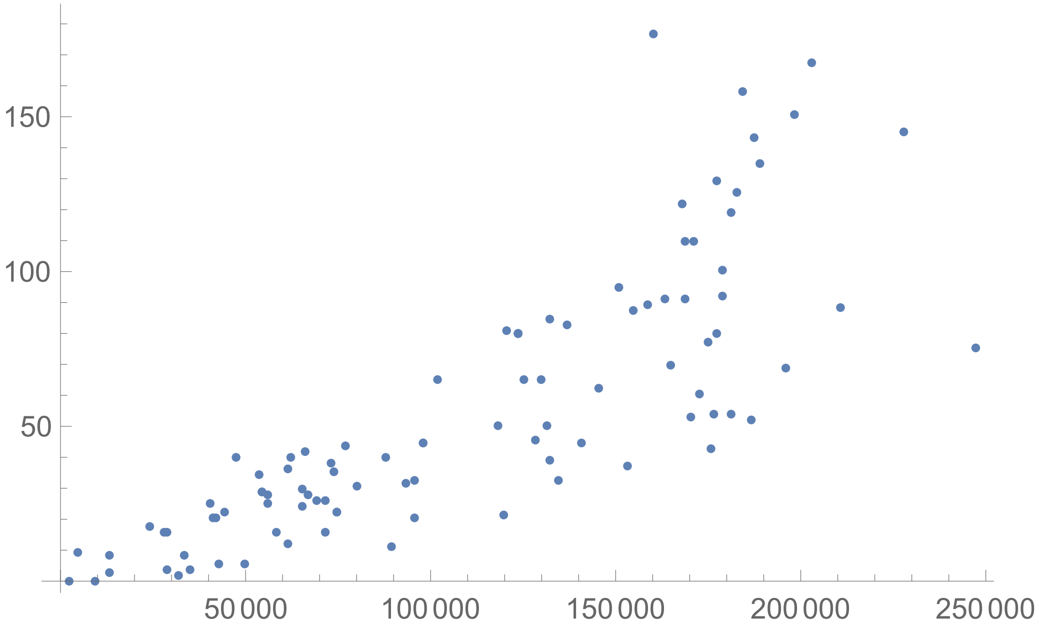


Figure 5.3: Gaussian Mixture Clustering

While the other methods gave us multiple clusters, the Gaussian clustering method only resulted in one cluster. Based off of this model it would appear that there are not any subpopulations in the data.

5.5 K-Means Clustering

K-means clustering is similar to the mean shift and agglomerate clustering methods. The algorithm gets its name since it partitions the data into k different clusters that are based on mean distances. K-means works by creating k initial points and creating clusters around those points based on Euclidean distance. It then moves the points to the center of their respective cluster and recalculates the clusters. It continues this until there is no change in the clusters. The resulting clusters for this method with k values of 3, 4, and 10 are shown in Figures 5.4, 5.5, and 5.6, respectively.

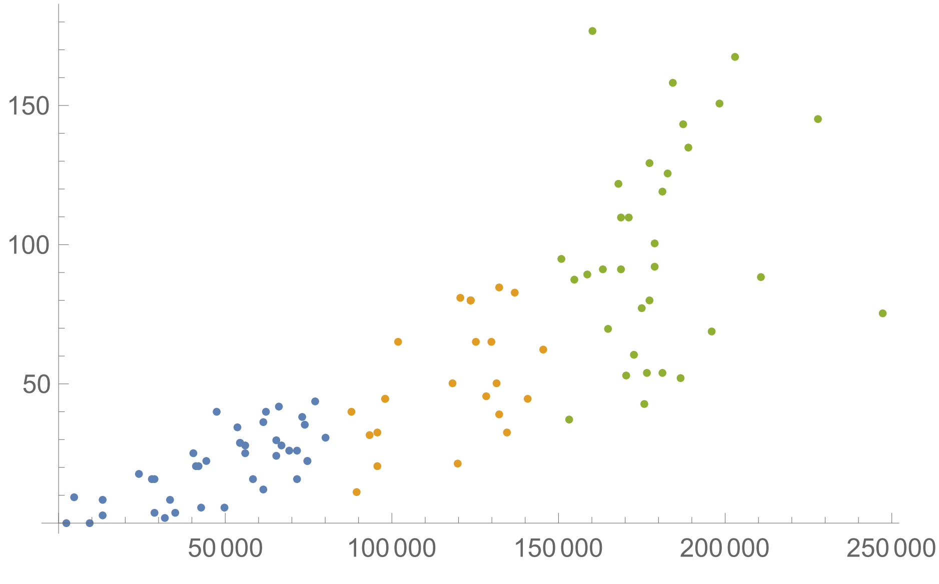


Figure 5.4: K-Means Clustering with $k = 3$

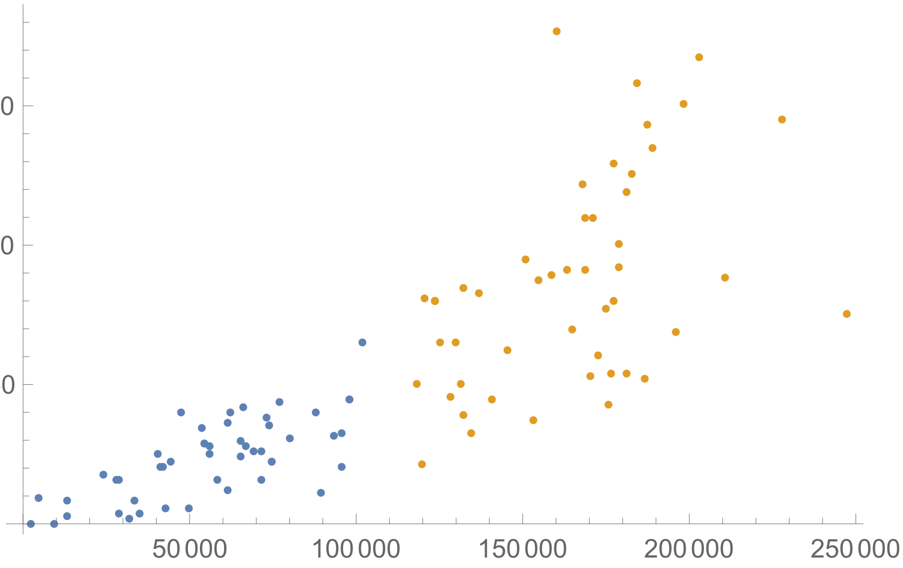


Figure 5.5: K-Means Clustering with $k = 4$

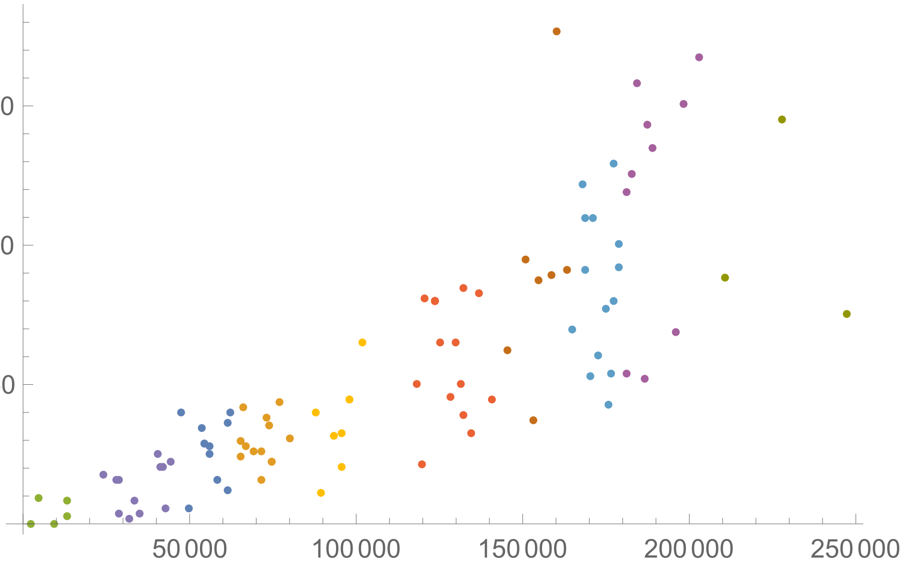


Figure 5.6: K-Means Clustering with $k = 10$

Due to the number of distinct clusters the k-means method yielded, it supplied us with some of the most useful data for our analysis. The different clusters provided us with various interpretations of the data.

5.6 Analysis of Clusters

The Gaussian mixture algorithm only yielded one cluster. In terms of our analysis of the applications this means one of two things; either all the apps are benign or all of the apps are malicious. Considering that some of the apps had 0 “suspicious” method calls it does not make sense for those applications to be classified as malicious. Since the algorithm grouped all the apps together it is safe to assume that, according to the model, all the applications are benign. If we were to compare another application to this data set either it is in the cluster or not. This means that an app would be considered either benign or “suspicious” with no grey area.

The other algorithms, k-means, mean shift, and agglomerate, yielded more interesting results. Through each of these clustering methods we were able to get multiple clusters for the data. Figures 5.1, 5.2, and 5.4 appear to be the ones that best represent a cluster of applications that, in the best case, are “suspicious” and, at worst, potentially malicious.

Looking at Figure 5.6 we can see the results of k-means clustering when $k = 10$. The larger clusters we saw in both Figure 5.4 and Figure 5.5, where we used k-means clustering with k values of 3 and 4, respectively, begin to break up into smaller clusters. These clusters are only comprised of a handful of applications. It is interesting to see how in Figure 5.5 a k value of 4 yielded two clusters, but a k value of 10 yielded ten clusters.

Resulting from all three of these algorithms was a cluster that included all the applications that invoked 150,000 or more methods. These applications consistently had the most “suspicious” method calls. Therefore, there is evidence to suggest that these applications are doing something other than what we expect them to do. There are also several applications in this cluster that have some of the highest proportion of “suspicious” method calls. We consider the applications in these clusters to be the “suspicious” applications, since they make a significant number of method calls that correlate to malware and are not related to camera functions. We would want to further examine these applications and potentially reject them.

Looking at all of the clusters we noticed that there are several of them that start with apps that invoke 120,000 methods. These clusters can be seen in Figures 5.1, 5.2, 5.5, and 5.6. Upon further examination of these clusters we discovered a group of applications that had the same total number of method invocations as well as the same number of “suspicious” methods. Upon further examination of these applications it was revealed that they were created by only two different developers. Each developer had multiple applications under their name, all of which were camera apps.

We ran our byte code analysis program on sixteen of the applications created by the two developers. This revealed that all sixteen of the applications were nearly identical. All of the applications had 123,917 total method invocations, expect one which had 123,914, and each of them had 80 “suspicious” method invocations. These results can be seen in table B.2 in Appendix B.

One possible explanation for this is code piracy, where one developer steals another’s application and only makes a few cosmetic changes so that they appear different. Another possible explanation is that one developer might have several apps that appear different, but are identical. Both of these scenarios could be attempts to make it appear that they have created several applications. This might be an attempt to increase potential monetization opportunities. Whatever the reason, these both contribute to the surplus of free camera applications on the Play Store.

Chapter 6

Conclusions

6.1 Summary

Through static analysis of the byte code of multiple Android camera applications and the use of several clustering algorithms we are able to develop clusters of applications that might engage in “suspicious” or irrelevant behavior. The applications that comprise these clusters are ones that have a high number of “suspicious” method calls indicating that they are potentially doing something unknown to the user or are malware. We would want to “throw out” the applications that make up these clusters and examine them further. Before they would be reinstated we would make sure that they are no longer demonstrating behavior that we would consider suspicious.

Along with applications that display “suspicious” behavior, we want to include applications that seem to be identical to another application. There is no good reason for one application to be exactly the same as another. After investigating into several applications that seemed identical, we discovered over a dozen applications that appeared to be the same. Whenever we discover a cluster of applications that are identical we could flag them and look further into whether there is a legitimate reason for it or if there is something else going on.

This can be useful in the Google Play Store where detecting potentially malicious applications before they are released to the public is an important task. The Play Store could use this approach to filter applications before they allow them to be public. By looking at an application and noticing that it might be doing something that it does not claim to do, they can reject it for revision. This could also be useful for individuals who are curious or concerned about a certain application that they downloaded. When looking at a large number of applications we are able to see that there are some that demonstrate behavior that can be “suspicious”.

6.2 Future Work

With thousands of camera apps on the Play Store, the data set could be expanded several fold. We could run the clusters on all the applications in the Play Store and open it up for others to submit apps to as well. This would give us a sharper understanding of the “typical” number of “suspicious” method calls. We could then reject all the apps that are in the suspicious clusters and periodically check them again to find new applications that emerge.

Another way we could expand the project is to include known Android malware apps, especially those that attempted to mimic camera apps. This would allow us to develop a baseline to which we could compare the “benign” camera apps. We would be able to, with more certainty, correlate certain data points or clusters with malware.

Bibliography

[1] Statista. *Number of available applications in the Google Play Store from December 2009 to December 2017*. URL: `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`.

[2] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* Volume 27 Issue 7 (July 2008).

[3] Benjamin Livshits. "Improving Software Security With Precise Static and Runtime Analysis". PhD thesis. The Department of Computer Science and the Committee on Graduate Studies of Stanford University, Dec. 2006.

[4] BA Wichmann et al. *Industrial Perspective on Static Analysis.* Tech. rep. Department of Computer and Information Science at Linköping University.

[5] Google. *Application Fundamentals.* URL: `https://developer.android.com/guide/components/fundamentals.htmll`.

[6] Sohail Khan et al. *Analysis of Dalvik Virtual Machine and Class Path Library.* Tech. rep. Security Engineering Research Group, Institute of Management Sciences, Nov. 2009.

[7] *What is Dalvik and dalvik-cache?* URL: `https://stackoverflow.com/questions/7541281/what-is-dalvik-and-dalvik-cache`.

[8] Yunhe Shi, Kevin Casey, and M. Anton Ertl and David Gregg. "Virtual machine showdown: Stack versus registers". In: *ACM Transactions on Architecture and Code Optimization* Volume 4 Issue 4, Article No. 2 (Jan. 2008).

[9] Google. *Dalvik bytecode.* Feb. 6, 2018. URL: `https://source.android.com/devices/tech/dalvik/dalvik-bytecode`.

[10] John Rose. “with Android and Dalvik at Google I/O”. In: (May 31, 2008). URL: https://blogs.oracle.com/jrose/with-android-and-dalvik-at-google-io.

[11] Infosec Institute. *Android Application Assessment*. URL: http://resources.infosecinstitute.com/android-application-assessment/#gref.

[12] *Androguard*. URL: https://androguard.readthedocs.io/en/latest/index.html.

[13] Wolfram Mathematica. URL: https://www.wolfram.com/mathematica/.

[14] *Google Play Store*. URL: https://play.google.com/store.

[15] *APK Dowloader*. URL: https://apps.evozi.com/apk-downloader/.

[16] Yousra Aafer, Wenliang Du, and Heng Yin. “DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android”. In: *Zia T., Zomaya A., Varadharajan V., Mao M. (eds) Security and Privacy in Communication Networks. SecureComm 2013. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 127. Springer, Cham* (Sept. 2013).

[17] Keith Makan. *Automated DEX Decompilation using Androguard*. Nov. 5, 2014. URL: `http://blog.k3170makan.com/2014/11/automated-dex-decompilation-using.html`.

Appendix A

Gathering Methods

A.1 dump_methods.py

This code is based off of the tutorial written by [17] and modified to fit our needs.

```
from sys import argv
from androguard.core.bytecodes import apk
from androguard.core.bytecodes import dvm


def dump_methods(file, outfile):
    a = apk.APK(file)
    d = dvm.DalvikVMFormat(a.get_dex())
    bcf = open(outfile, "w+")
    for current_class in d.get_classes():
        for method in current_class.get_methods():
            bcf.write("[*] "+method.get_name()+ method.get_descriptor()+"\n")
            byte_code = method.get_code()
            if byte_code != None:
                byte_code = byte_code.get_bc()
                idx = 0
                for i in byte_code.get_instructions():
                    bcf.write(”\t,_%x_” % (idx)+i.get_name()+i.get_output()+”\n”)
                    idx += i.get_length()
```

A.2 methodFinder.py

```
from regex import *
import re
import csv



#Function that makes new file without lines that dont have pattern
def selectMethods(fileName,pattern,outfile):
    fh = open(fileName,”r”)
    of = open(outfile, ”w+”)

    #loop
    for l in fh:
        if (re.search(pattern,l)):
            m = re.search(pattern,l)
            of.write(m.group(2)+”\n”)
    #close file
    fh.close()
    of.close()

#counts the number of unique methods in each apk
def methodCounter(filename, outfile):
    fh = open(filename,”r”)
    of = open(outfile,”w+”)

    #attributes
    lc = 0              #number of total methods
    dictM = {}          #dictMionary with methods and number of apperances
    #loop
    for l in fh:
        lc +=1

        if l in dictM:
            dictM[l] += 1
        else:
            dictM[l] = 1

    w=csv.writer(of)

    for key in dictM:
        w.writerow([key, dictM[key]])

    fh.close()
    of.close()

#counts the number of suspect methods in each apk
def createData(filename, outfile, dangMFile):
    #files used
    fh = open(filename, ”r”)
    of = open(outfile, ”a+”)
    df = open(dangMFile, ”r”)

    #counters
    totalCount = 0
    dangCount = 0

    #list of all the method calls
    dl = []

    #write all the suspect methods to a list
    for line in df:
        line = line.strip(’\n’)
        dl.append(line)

    reader = csv.reader(fh, delimiter=’,’)
    #loop across the file and see if an methods are in dl
    for row in reader:
        totalCount = totalCount + int(row[1])
        r = row[0].strip('\n')

        if r in dl:
            dangCount = dangCount + int(row[1])

        prop = dangCount/totalCount

    w=csv.writer(of)
    w.writerow([totalCount, dangCount, f'{prop:.10f}'])

    fh.close()
    of.close()
    df.close()
```

A.3 main.py

```
import os
from methodFinder import *
from dump_methods2 import *

#directory with all the apks
directory1 = (”/media/blake/My_Passport/Sproj/apks/”)
directory = os.fsencode(directory1)

for file in os.listdir(directory):
    filename = os.fsdecode(file)
    print(filename)

    #check to see if it is an APK file
    if filename.endswith(”.apk”):

        #filename is actual name of apk, fix confustion
        apk = directory1+filename
        byteCodeFile = apk+"_bytecode.txt"
        methodsFile = apk+"_methods.txt"
        methodCountFile = apk+"_methodCount.csv"

        #check to see if the files already exists
        if not os.path.exists(byteCodeFile):
            #call dump_methods on an apk to get the bytecode
            dump_methods(apk, byteCodeFile)

        if not os.path.exists(methodsFile):
            #Select only the lines that invoke a method
            #Edit the bytecode so that it is just the method names
            selectMethods(byteCodeFile,"(invoke.+,_)(.+)(\()",methodsFile)

        if not os.path.exists(methodCountFile):
            #count all the methods in the filename
            methodCounter(methodsFile, methodCountFile)

    #get the counts of all the methods in question
    elif filename.endswith("_methodCount.csv"):
        createData(directory1+filename, directory1+"dangMethods.csv",directory1+"
            dangMethods.txt")
```

Appendix B

Results

B.1 Data Table

Reference	Total Count	"Suspicious" Count	Proportion
1	54588	29	0.0005312522899
2	73923	35	0.0004734656332
3	9622	0	0
4	119718	21	0.0001754122187
5	76824	44	0.0005727376861
6	120511	81	0.0006721378131
7	31731	2	6.30E-05
8	13058	3	0.0002297442181
9	154591	87	0.0005627753233
10	177610	80	0.0004504250887
11	124971	65	0.000520120668
12	102216	65	0.0006359082727
13	123917	80	0.0006455934214
14	123917	80	0.0006455934214
15	176309	54	0.0003062804508
16	178677	100	0.0005596691236

17	177026	129	0.0007287065177
18	128296	46	0.0003585458627
19	80223	31	0.0003864228463
20	183991	158	0.0008587376556
21	202851	167	0.0008232643665
22	247085	75	0.0003035392679
23	182960	126	0.0006886751202
24	181490	119	0.0006556835087
25	40748	25	0.0006135270443
26	140716	45	0.0003197930584
27	153089	37	0.0002416894747
28	65085	24	0.0003687485596
29	178535	92	0.0005153051222
30	172842	60	0.0003471378484
31	41611	20	0.0004806421379
32	186509	52	0.0002788069208
33	168228	122	0.0007252062677
34	73469	38	0.000517224952
35	95303	33	0.0003462640211
36	123917	80	0.0006455934214
37	170228	53	0.0003113471344
38	198379	151	0.000761169277
39	49800	6	0.0001204819277
40	54588	29	0.0005312522899
41	131032	50	0.0003815861774
42	117950	50	0.0004239084358
43	98192	45	0.0004582858074
44	55835	25	0.0004477478284
45	89113	11	0.00012343878
46	175521	43	0.0002449849306
47	137050	83	0.0006056183874
48	87663	40	0.0004562928488
49	134808	33	0.0002447925939
50	12957	8	0.0006174268735
51	187123	143	0.0007642032246
52	34621	4	0.0001155368129
53	189018	135	0.0007142176936
54	123914	80	0.0006456090514
55	227633	145	0.0006369902431

56	163657	91	0.0005560409882
57	71433	16	0.0002239861129
58	53747	34	0.0006325934471
59	44381	22	0.0004957076226
60	66178	42	0.0006346519992
61	150615	95	0.0006307472695
62	4909	9	0.001833367285
63	1953	0	0
64	58456	16	0.000273710141
65	168420	110	0.0006531290821
66	160539	177	0.001102535832
67	129601	65	0.00050153934
68	69165	26	0.0003759126726
69	62382	40	0.0006412106056
70	65703	30	0.0004566001552
71	42434	6	0.0001413960503
72	33801	8	0.0002366793882
73	28981	4	0.0001380214623
74	61062	36	0.0005895647047
75	175120	77	0.0004396984925
76	71484	26	0.000363717755
77	123917	80	0.0006455934214
78	95918	20	0.0002085114369
79	181097	54	0.0002981827418
80	196206	69	0.0003516712027
81	131979	85	0.0006440418551
82	132538	39	0.0002942552325
83	210351	88	0.0004183483796
84	170789	110	0.0006440695829
85	41186	20	0.0004856019036
86	61819	12	0.0001941150779
87	24262	18	0.000741900915
88	123917	80	0.0006455934214
89	56003	28	0.0004999732157
90	93200	32	0.0003433476395
91	145751	62	0.0004253830162
92	67083	28	0.0004173933784
93	97593	45	0.0004610986444
94	28138	16	0.0005686260573

95	28745	16	0.0005566185424
96	74869	22	0.0002938465854
97	164789	70	0.000424785635
98	168782	91	0.0005391570191
99	158936	89	0.0005599738259
100	47715	40	0.0008383108037

B.2 Similar Applications

Reference	Total Count	“Suspicious” Count	Proportion
13	123917	80	0.0006455934214
14	123917	80	0.0006455934214
36	123914	80	0.0006456091
77	123917	80	0.0006455934214
88	123917	80	0.0006455934214
101	123917	80	0.0006455934
102	123917	80	0.0006455934
103	123917	80	0.0006455934
104	123917	80	0.0006455934
105	123917	80	0.0006455934
106	123917	80	0.0006455934
107	123917	80	0.0006455934
108	123917	80	0.0006455934
109	123917	80	0.0006455934
110	123917	80	0.0006455934
111	123917	80	0.0006455934