

A Demographic Survey of Selected Third-Party Android Markets

by

William Symon

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
August 4, 2012

Keywords: Android, Third-Party Markets, Demographic Overview, Reverse Engineering, APK files,
Google Play

Copyright 2012 by William Symon

Approved by

David Umphress, Chair, Associate Professor of Computer Science and Software Engineering
Dean Hendrix, Associate Professor of Computer Science and Software Engineering
Hari Narayanan, John H. and Gail Watson Professor of Computer Science and Software Engineering

Abstract

In 1998 less than 10 percent of the global population held cell phone subscriptions. By the end of 2011 this number had increased to 87 percent, accounting for over 5.9 billion subscribers worldwide. Due to this high rate of adoption cell phones have become one of the most pervasive pieces of technology in society, changing the way we think about communication and influencing our daily lives. Society as a whole currently stands on the precipice of mass adoption of the next generation of cellular communication, the smart phone. These devices hold just as much, if not more, ability to change the way we think about interpersonal communication.

149 million smart phones were sold in the fourth quarter of 2011 with close to half of these devices running the Android operating system. Smart phones grant end users the ability to install third-party software on their devices customizing the functional of a device to their specific needs. These third-party programs, or apps, are generally hosted on markets which allow a user to easily browse, purchase, and download apps of their choosing. While Google hosts an official Android market known as Google Play many third-party markets have been established as well. Android users have heavily utilized these markets, installing an average of 35 apps on each device owned and downloading more than 10 billion apps from Google's official market alone.

Yet for all the popularity of the Android operating system and the app markets which accompany it very little demographic information can be found on these markets as a whole. In this work we present a process for acquiring a large body of Android apps, reverse engineering them, and then extracting demographic information from this data. We then compile a demographic overview of four different third-party markets examining attributes of these markets such as app permissions and SDKs used,

localization utilization, the employment of monetization schemes, and many others. Finally, we compare the results across all four markets looking for trends in that emerge in our data set.

Acknowledgments

I would first like to thank my advisor Dr. David Umphress for his time, patience, and guidance during the completion of this research. Your mentorship has been invaluable to my efforts and I cannot thank you enough for investing yourself in this work. I would also like to thank my other committee members, Dr. Dean Hendrix and Dr. Hari Narayanan, for their input and instruction on this work. I've enjoyed working with all of you and wish you all the best in the future.

Next, I would like to thank my parents, Pops and Rufus, for all the support and love they have given me throughout my education. You instilled a love in me for education, raised me to pursue what I was interested in, and taught me to always apply myself 100%. The lessons you have taught me will last a lifetime. The ways in which you've supported me are too numerous to list here but know that I am thankful for each and every one.

Finally, I would like to thank my fiancée Kristin. Love, you've been patient and understanding with me as I completed this work even in the midst of trying to plan a wedding, move us to another state, and start a new job. I love you very much and cannot wait to make you my wife.

Table of Contents

Abstract	ii
Acknowledgments.....	iv
List of Tables	vii
List of Figures.....	viii
List of Abbreviations	x
Chapter 1 – Problem Description	1
Chapter 2 – Previous Work	11
Chapter 3 – Solution	28
Methodology	28
Results	46
Chapter 4 – Solution Validation	79
Chapter 5 – Conclusions and Future Efforts	85
Conclusions	85
Future Work	86
References	88
Appendix 1 – Source Code	93
Apps For Adam Targeted Crawler	93
App Town Targeted Crawler	95
And App Online Targeted Crawler.....	97
Slide Me Targeted Crawler.....	99

Decompiler.....	101
Main Data Extraction.....	102
Data Base Writer.....	109
Language Localization Extraction	112
Drawable and Layout Localization Extraction	113
Permission Extraction	115
File Information Extraction.....	119

List of Tables

Table 1 – Apps For Adam Targeted Crawler Findings	35
Table 2 – And App Online Targeted Crawler Findings	35
Table 3 – App Town Targeted Crawler Findings	35
Table 4 – Slide Me Targeted Crawler Findings	35
Table 5 – Total Apps Downloaded by Markets	35
Table 6 – Permission Not Utilized in Any App Examined	58
Table 7 – Android Localization Language Abbreviations	71
Table 8 – Most Popular Permissions Per Study	81

List of Figures

Figure 1 – OS Market Share 4 th Quarter 2011	2
Figure 2 – App Availability 1 st Quarter 2012	4
Figure 3 – Google Play Informational Side Bar for a Random App	29
Figure 4 – Google Play Update Notes for a Random App	29
Figure 5 – Google Play Permission Information for a Random App	30
Figure 6 – APK file Directory Structure	38
Figure 7 – Sample res Directory Structure with No Localization	40
Figure 8 – Sample res Directory Structure with Localization	40
Figure 9 – Smali Source Code for a Standard Hello World Program	41
Figure 10 – Required AndroidManifest.xml File Structure	43
Figure 11 – Number of Apps Per Market After Pruning	46
Figure 12 – APK File Size Sorted Linearly	48
Figure 13 – Range of File Sizes Per Market	49
Figure 14 – Apps with Different Versions Per Market	50
Figure 15 – Multiple Apps by the Same Publisher Per Market	52
Figure 16 – Percentage of Market Published by a Nonunique Publisher.....	52
Figure 17 – Apps Published by a Single Publisher	53
Figure 18 – App Crossover by Market	54
Figure 19 – App Crossover as a Percentage of the Smaller Market	54
Figure 20 – Percent of Each Market Requesting at Least One Permission	56
Figure 21 – Range of Permissions Requested Per Market	56

Figure 22 – Permissions Utilized in More Than 5% of all Apps Analyzed	57
Figure 23 – Percentage of Apps Examined Requiring Minimum SDK level	59
Figure 24 – Percent of Each Market Utilizing External Libraries	61
Figure 25 – Libraries Used as a Percent of All External Libraries Utilized	61
Figure 26 – Range of Services Declared Per Market	62
Figure 27 – Range of Providers Declared Per Market	63
Figure 28 – Range of Receivers Declared Per Market	65
Figure 29 – Percent of Services, Providers, and Receivers by Market	66
Figure 30 – Rang in Number of Activities Per Market.....	67
Figure 31 – Percent of Each Market Utilizing String Localization	69
Figure 32 – Range of String Localizations Per Market.....	70
Figure 33 – Most Popular String Localizations	70
Figure 34 – Percent of Each Market Utilizing Layout Localization.....	72
Figure 35 – Range of Layout Localization Per Market	72
Figure 36 – Most Popular Layout Localizations.....	73
Figure 37 – Percentage of Markets Utilizing Drawable Localizations	74
Figure 38 – Range in Number of Drawable Localizations Per Market.....	75
Figure 39 – Most Popular Drawable Localizations Used	75
Figure 40 – Percent of Each Market Utilizing Advertising	77
Figure 41 – Percent of Apps Utilizing Each Marketing Scheme Examined.....	78
Figure 42 – Percent of Each Market Utilizing More Than One Marketing Scheme	78
Figure 43 – Percent of Each Market Utilizing a Given Localization.....	84

List of Abbreviations

AAPT	Android Asset Packing Tool
ADB	Android Debug Bridge
API	Application Programming Interface
APK	Android Package File
APP	Android Application
CAPTCHA	Completely Automated Public Turing Test to Tell Computers and Humans Apart
DEX	Dalvik Executable
DPI	Dots Per Inch
DVM	Dalvik Virtual Machine
GB	Gigabyte
HTML	Hyper Text Markup Language
JAR	Java Archive
JVM	Java Virtual Machine
J2ME	Java 2 Platform Micro Edition
OS	Operating System
PC	Personal Computer
SDK	Software Development Kit
SOM	Self Organizing Map
XML	Extensible Markup Language

CHAPTER 1 - Problem Description

According to data published by the International Telecommunications Union in 1998 less than 10 percent of the global population held cell phone subscriptions. By the end of 2011 this number had jumped to 87 percent, accounting for over 5.9 billion subscribers, including 79 percent of the population in developing countries [ITU 11]. Mobile cellular has been the most rapidly adopted technology in history and today is the most popular personal technology on the planet [ITU 09]. This rapid adoption of cell phone communication has transformed the way our species interacts with one another on a daily basis and revolutionized what we once thought was possible with respect to interpersonal communications. We currently stand on the precipice of the mass adoption of the next generation of cellular communication, the smart phone. This family of devices holds as much, if not more, ability to change the way we communicate with one another as the first generation of cellular devices did.

149 million smart phones were sold in the fourth quarter of 2011, a 47 percent increase from the same time period of the previous year. During the year of 2011 nearly half a billion smart phones were sold accounting for 31 percent of all mobile device sales, up from just under 20 percent in 2010 [GARTNER 11]. The amount of the mobile device market share accounted for by smart phones has increased steadily since their mainstream appearance in 2000. In recent years, as more diverse and robust smart phones have entered the market, this rate has begun to increase more rapidly. With nearly one out of every three mobile devices sold being a smart phone there can be little doubt that these devices hold the power to drastically effect our daily communications.

Smart phones differ from their “dumber” counterparts in the fact that they are designed to be used as a mobile computing device, not just a means of mobile voice or text communication. Because of this, one of the main factors that differentiate these devices from one another is the operating system that is installed on the device. The four main operating systems currently in use in the smart phone market are

Android, iOS, Symbian, and Research in Motion. During the fourth quarter of 2011 over half, 50.9 percent, of all smart phones sold came with the Android operating system installed. Apple's iOS accounted for another 23.8 percent of smart phone sales during the same time period. Both Symbian and Research in Motion held significantly less of the market share with 11.7 and 8.8 percent respectively. With the advent of the new line of windows phones, Microsoft has made a bid to join this group, but as of the fourth quarter of 2011, they have not been able to achieve much traction in the market with only 1.9 percent of the market share [GARTNER 11].

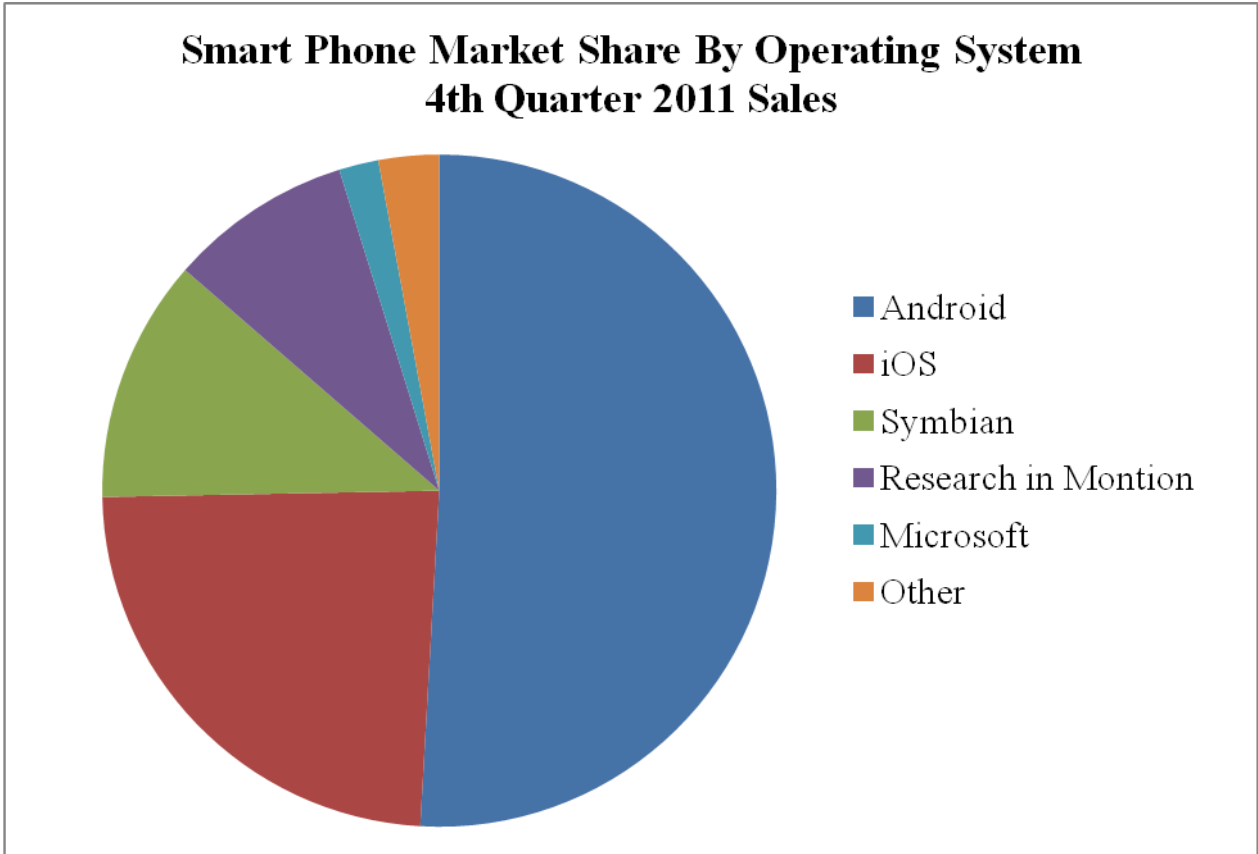


Figure 1: OS Market Share 4th Quarter 2011

Because of the processing power, variety of user input mechanisms, and the display technologies employed on smart phones a demand has arisen from customers to have the ability to customize the functionality of these devices by installing a variety of software programs, commonly known as apps, to meet their personal needs. This need has been met by the four major OS developers through the creation

of separate app marketplaces. These marketplaces provide a common location in which apps can be hosted by developers and then browsed and downloaded by device owners. The apps hosted on these marketplaces are usually developed by independent developers not associated with the company which developed the operating system of the smart phone. Apps can be priced at the developer's discretion with the most expensive apps currently selling for \$999 although the majority of apps are either free or under \$1.00. An entire economy has sprung up around these app markets with success stories such as the company Rovio, developer of the popular game Angry Birds for smart phones, who reported a profit of nearly 68 million dollars in 2011 alone [ROVIO 11].

The four main app markets, the ones maintained by the operating system developers, have all been hugely successful. Apple's App Store is currently the largest market and hosts more than 500,000 apps ranging from entertainment titles to business software to specialized niche applications [APPLE 12a]. Google Play, the officially supported app store for the Android operating system, trails closely behind Apple's App Store with more than 450,000 apps on offer. OVI, Nokia's app store for the Symbian operating system, and App World, Blackberry's app store for the Research in Motion operating system, host over 100,000 [NOKIA 12] and 60,000 [BLACKBERRY 12] apps respectively. While these two markets have fallen behind Apple's and Android's offerings they are by no means a failure. All of these market places have found success not only in finding developers to create apps but also in the quantity of apps which smart phone users have downloaded for their personal use. In March of 2012, Apple reported that over 25 billion apps had been downloaded for use on devices running iOS [APPLE 12b]. Shortly before that, in December of 2011, Google announced that over 10 billion apps had been downloaded for Android devices [GOOGLE 11a]. Even the smaller markets such as App World have met success, with Blackberry announcing that it had achieved over 2 billion downloads from its market during their 2012 developer conference in Europe [BLACKBERRY 12]. With this amount of activity from both end users and developers, these markets clearly warrant careful scrutiny from not only the business community but the academic community as well.

The task of deciding what external software is available to be installed on smart phones has largely been left to the operating system developers. The strategies that have emerged to deal with this issue are best exhibited by the way Apple and Android manage their app market places. These two companies have chosen vastly different market management schemes and each has shown to have its own strengths and weaknesses.

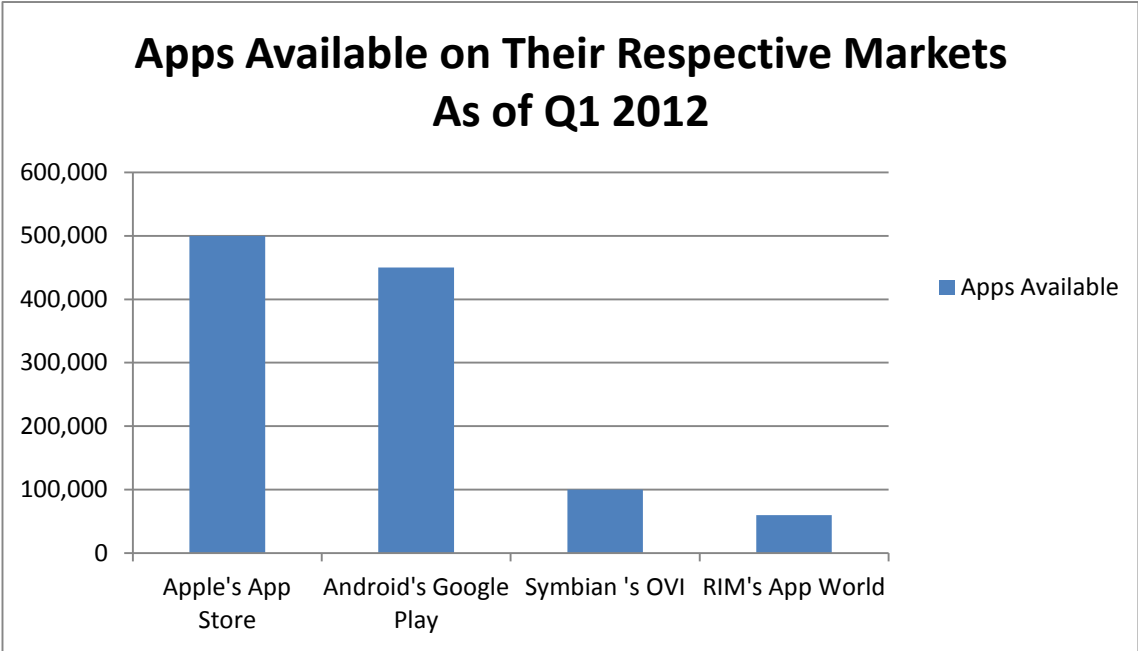


Figure 2: App Availability 1st Quarter 2012

Apple takes a walled garden approach to the management of its App Store market place [NETANEL 07]. In this approach Apple has built a wall around their devices by making the decision that only apps from their own App Store will be allowed to be installed on their smart phones. Apple then controls the gates to this garden through a strict approval process developers must go through to get their app hosted on this market, thus allowing Apple to provide a curated App Store for the end user. In addition, any developer which wishes to publish to the Apple App store must first pay a fee, currently \$99 a year, to sign up for a developer account with Apple. This fee allows Apple to offset some of the costs associated with the overhead of this walled garden approach.

Apple states that their review and approval process is in place to ensure that applications are reliable, perform as expected, and are free of explicit and offensive material. They review each app submitted to the App Store for approval based on a set of technical, content, and design criteria. These criteria cover a wide range of development issues such as user interface design, functionality, content, and the use of specific technologies [APPLE 12c]. This screening process is a formalized process which all app developers must go through for each app they wish to publish. According to statistics released by Apple to third party developers this process is completed in less than a week in the majority of cases and in under 14 days for 95 percent applications submitted [MCDANIEL 10]. If an app is rejected from the market place, the developer is given notice as to why and can resubmit the app once the appropriate changes have been made. Apple believes that this process allows them to present a higher quality product to end users which have chosen to use its devices [APPLE 12d].

Android, on the other hand, has adopted a wild west approach to the management of what apps can be installed on devices running its operating system. In this approach there is no strict review process before apps are made available for public consumption. In general, apps are still hosted on market places but even this does not have to be true. Android has made it as easy as possible for external software to be installed on its devices, relying on the security features of the operating system itself to prevent damaging effects of malicious software instead of a formal review process.

The Google Play market place is administrated by Google and is the official market place for Android devices. Google describes their publishing process for Google Play as a straightforward process that can be done in a few simple steps – register, configure, upload, and publish [GOOGLE 12a]. As with the Apple App Store developers must pay a fee, currently a onetime \$25 amount to register with Google and be allowed to publish apps to Google Play. The next two steps in Google’s publishing process, configuring and uploading, can be completed in minutes through tools provided by the Google Play service. After all of this is complete, publishing an app is as simple as pressing a button in the developer’s console and waiting for the app to show up in the market place which, according to Google, should not take more than a few hours [GOOGLE 12a].

While there is no formal review process for the Google Play market, there are still guidelines as to what content can be published. These guidelines state that a developer's application should not contain things such as illegal content, material not suitable for persons under the age of 18, content that interferes with the functioning of any services or other parties, content designed to invade personal privacy or violate the rights of publicity, pornography, obscenity, nudity, or promotions of hate or incitement of violence. Google reserves the rights to remove applications that violate these guidelines without the consent of the developer [GOOGLE 12b]. Since apps are not required to be reviewed to these guidelines before they are released, Google relies on the market community to police itself and bring to attention any apps that do not comply with these terms. In the past, this practice of removing apps after they have been published has led some to question the true openness of the Google Play market [HOLZER 11].

Android also made the choice not to limit a consumer's choice of market places to just the officially supported Google Play market. This further extended its reach to external software, a move that has been hugely popular with both device users and app developers of Android software. This policy has manifested itself in two key ways with the sharing of individual .apk files and the rise of third party Android app market places.

An Android app is contained within an .apk file. This file contains all the information that is needed to install and run an app on an Android device and is comparable to the way we think of modern software programs. By simply changing a setting on their smart phone and then using the Android Debug Bridge, another piece of software provided by Android, users can install .apk files directly to their Android devices without going through a market. This ability has led to a community growing up around the sharing of these .apk files, allowing users to install any app they want even if it would not be sanctioned by established markets.

The second way we see this policy manifested is in the rise of third party Android markets. These markets come in two general flavors. First, a third party market could simply be a composition of Android .apk files collected in one place to be downloaded by users. The apps made available on these

types of markets consist of anything from high quality production apps similar or the same as those found on the Google Play market place to low quality or even malicious software. Apps are made available on these markets for free or at a cost just as with Google Play. Markets of this nature have become very popular with Android smart phone users and many different sizes of markets can be found such as freeware lovers [FREE 12], a collection of over 1500 apps of various types, appsforadam [APPS 12], a small collection of around 100 apps for business and communication purposes, and appslib [LIB 12], a collection of 38,771 apps.

In addition to independent third party markets, official markets supported by well-known companies or for specific devices have also arisen. One of the best examples of this type of market is Amazon's Appstore [APPSTORE 12]. Amazon introduced its Appstore with the launch of their Kindle Fire device, an Android-powered tablet. This device comes preinstalled with access to the Appstore but not the Android marketplace. This allows Amazon to steer users in the direction of its store first when looking for apps. In addition it also allows Amazon, or any other company setting up this type of market, to screen the apps offered to their customers anyway they want. While this eliminates the concerns of some as to the safety of the apps they are installing on their devices, it also troubles some parties that these markets are not as open as their original counterparts.

One of the main reasons that these two paradigms of market place management have come about is the underlying design of the operating systems themselves. In the Android operating system, each app is run in its own virtual machine. This isolates the app from other parts of the system, creating a safe environment for it to execute in while minimizing the amount of damage a malicious app can do to the system. In addition, Android systems require that a user must grant an app permission to use specific system resources the first time it is installed on the device. In this way Android has taken some of the responsibility for the tasks than a piece of software is allowed to perform out of the operating system itself and into the hands of the users [BUTLER 11]. In contrast Apple's iOS allows apps to access many system resources by default and does not make the system user approve which data or hardware devices

an app has access to. These two different security management schemes reflect highly on the underlying way in which the security of the operating system itself is managed.

Though Android apps run in a sandbox environment and rely on user input to help control the security of the device, this by no means makes Android a completely secure operating system. Malware has been found both “in the wild” as well as on the Android market place. Examples that have been found, such as AdSMS and DroidDream, act like modern computer trojans and rely on poorly informed users granting access to these apps that they either should not normally possess or request explicitly for malicious purposes [HUSTED 11]. Once users have granted these apps these permissions the apps can then perform malicious acts such as stealing personal information from the device, tracking the device without the user being aware of it taking place, or send premium rate text messages at the expense of the user.

Both Apple’s and Android’s position on market place management have been praised and criticized. Apple’s walled garden approach limits content available to users, especially since Apple’s supported market is the only option open to them, yet provides a way for apps to be screened in order to ensure that they are safe to run on their devices. Android’s wild west approach makes more apps available, yet the quality and security of the apps are uncertain. Google has been actively searching for a way to ensure that high quality apps rise to the top of their market, but, as of yet, have not found a satisfactory solution [BUTLER 11].

This walled garden versus wild west approach that is currently playing out with smart phone markets has many parallels to the same types of paradigms adopted in the gaming world concerning console games and personal computer games [HIGA 08]. Console systems have taken a walled garden approach to software for their devices since their inception, while personal computers have always had a wild west mentality, leaving it up to the user to decide what software could and should be run on the system. In the gaming industry, this has caused an uneasy tension for many years and no solution satisfying all parties has been found as of yet. One trend that has been seen over the past few years is the loosening of both parties’ stance. Console manufactures have started to allow games to be distributed

through online channels to their systems with a much more streamlined review process than traditional hard media releases, while personal computers have seen the rise of markets offering prescreened, approved software guaranteed to run safely and correctly on your computer. As the smart phone markets continue to mature, much could be learned from studying the decisions made in the console and personal computer gaming struggles of the past.

Smart phone app markets represent large bodies of software, yet as we began our research we could discover little if any information concerning the quality of a smart phone app or the quality of a market as a whole. This perceived deficit led us to initially steer our research in this direction. We began by looking into the issue of determining the quality of a smart phone app. This problem quickly proved to be a highly complex topic which intermingles with most disciplines of software engineering [LOCHMANN 11]. This complexity was increased due to the fact that smart phone app development has seen an astronomical increase over the past several years and is still an emerging discipline in its own right. The fact that this discipline is still in its infancy has led to major challenges in program understanding and software maintenance as well as determining quality that will have to be answered as the field continues to mature [SYER 11]. For this reason we decided that determining the quality of a smart phone app was outside of the scope of our work and decided to focus on discovering more fundamental information about apps in a marketplace.

In particular, aside from being unable to discover any information on the quality of an app market place as a whole, we found little to no demographic information about smart phone market places. Essentially the only demographic information we could unearth on the market places we examined were simple data such as the number of apps hosted on the market place or the number of different categories (e.g. games, business, communication) apps were divided into on a market. Demographic information on these market places would be highly beneficial to the maintainers of the market places, end users purchasing apps from the market, as well as future researchers looking into to attributes of these markets. Maintainers of market places could use information of this nature to better understand what type of apps are being hosted on their market as well as identify trends among popular apps. End users, on the other

hand, could benefit from demographic information when determining what market to use to obtain apps for their devices. Finally, this demographic information will be highly useful for future research concerning smart phone app markets as work of a demographic nature could serve as a backbone to further more advanced research efforts in this field.

When deciding which markets to focus on when gathering this demographic information we chose to start with Android market places. This was done for several reasons. First, the open nature of the Android platform has led to the development of many app markets for Android devices, not just the officially supported market Google Play. This in turn led to multiple sources of data for us to examine, an option we would not have if we had chosen a different operating system. Next, again due to the open nature of the Android platform, a large community has evolved around the development and implementation of software for the Android operating system. While it is true that communities do exist around the development of software for other smart phone platforms, we initially found the Android community to be the largest and most active of the ones researched. We believed this community would prove an invaluable resource during the initial phases of this work. Finally, the Android platform was chosen because it is a personal interest to the main parties conducting this research.

Initially, we planned on using the Google Play as one of the markets studied in our research as it is the officially supported marketplace of the Android platform, but limitations placed by Google on its use proved to be prohibitive for our purposes. We attempted to contact Google to see if we could work around these limitations for academic purposes but were not successful in being able to obtain this permission. Since we were unable to obtain this access, we turned to using solely third party markets for our data gathering process. There are hundreds of these markets available and this turned out to be a very beneficial decision for our research as it allowed us to compare results across markets which resulted in some interesting results.

The main goals of the research completed for this thesis were two-fold. First, we wanted to determine if data of a demographic nature could be obtained from an Android app's .apk file, and, if so, what type of data could be recovered from the .apk file. Our second goal followed from the first in that

we wanted to determine if a demographic overview of an Android market as a whole could be compiled based on the information we were able to recover.

Chapter 2 – Previous Work

When conducting our research we could not uncover any previous work specifically relating to the gathering and compilation of comprehensive demographic information from a body of Android apps. This was not due to a lack of research interest in the Android platform or development process, though, as we did find a large amount of work currently being conducted by a variety of researchers in this area, including some with limited demographic information such as the number of permissions used per app. This further confirmed our beliefs that a demographic survey of an Android market would add value to the community as a whole. The topics we found currently being researched involving Android devices were highly varied and ranged from the analysis and evaluation of user interfaces [HENZE 11] to case studies of specific apps [POSTOLACHE 11] to using Android devices in education at a college level [HECKMAN 11] to reverse engineering apps to aid in the analysis of Android security [BERGER 11]. While none of the techniques or methods used in these studies presented an alternative solution to our topic we found much of this body of work invaluable as we moved forward with our own research. In this chapter we will examine many of the works which influenced our research and explain their importance to our work.

During our initial phase of research we looked briefly into the history of the mobile operating system environment. Sharon Hall's and Eric Anderson's paper "Operating Systems for Mobile Computing" served as a springboard for this investigation. In this paper, they began with early personal digital assistants of the 1990s and explain major developments through Research in Motion's Blackberry and Microsoft's Pocket PC, eventually arriving at Apple's iPhone and the Android operating system in the mid 2000s. They mentioned that the rising number of web searches being made on mobile devices was Google's main motivation for entering the mobile market with the purchase of the company called Android and the starting of the Open Handset Alliance to aid in the development of the Android operating system [HALL 09]. Prior to the Android operating system Google was having trouble gaining revenue from these mobile web searches since many closed operating systems at the time did not facilitate easy access to the Google search engine, routing users instead to their own search engine. Hall and Anderson

concluded that Google was able to head the Open Handset Alliance and offer the Android operating system for free due to the increased revenue gained from advertising that these additional mobile web searches using the Google search engine and originating from Android devices produced. They also stated that as of 2009 Android appeared to be gaining ground on Apple's iOS due to its open nature, the fact that apps are developed in Java, and the availability of Android development tools for many personal computer platforms.

Next our research focused on the Android platform in general to gain an understanding of the main components of the Android operating system and to start exploring what demographic information we might want to extract from apps. Adam Dutko presented a good overview of the layer structure of the Android operating system explaining the Android hierarchy in four main layers: the Linux kernel, the provided libraries and Dalvik virtual machine, the application framework, and the applications themselves [DUKTO 08]. This understanding of the layer structure of Android was critical to our understanding of how the Android operating system functioned. Dukto also briefly discussed the way in which Android applications were handled within the operating system while they were running. He discussed how Android apps are designed to be run in a state in which they cannot adversely affect other apps or services on a phone. He states that the only way in which an Android app can alter this design is to explicitly request permission to interact with other components of the mobile device or if the original programmer of the app explicitly shares components of the app with other apps on the system. Exceptions of this type are one of the many aspects of an Android app which are controlled through the apps manifest file. The information stored in this manifest file would become a critical focus of our research as our project progressed.

In another paper, Jeremy Andrus and Jason Nieh highlighted some of the underpinning mechanisms of the Android operating system itself as they were championing using Android to teach a undergraduate college level course on operating systems [ANDRUS 12]. They discuss parts of the actual Linux kernel used in the Android operating system such as system calls and process, scheduling, and the file system. Due to the nature of this paper none of these topics were covered in depth on a technical

level but a good amount of useful information could still be gathered as to how Android handles these issues. In the end this information turned out to be to low level to have an immediate effect on the direction of our research but it was still good information to possess.

The best source for information on the Android platform that we located was Google's developer's guide for Android [GOOGLE 12c]. This guide was the most consolidated, accurate, and up-to-date source of information we could locate on the Android operating system. The developer guide contains information on nearly every aspect of the Android operating system and was referenced often during the course of our research. It was from this guide that we were able to determine the structure of an Android app's .apk file, the file type in which all Android apps are distributed as. This allowed us to determine where specific information should be located within an app's directory structure when we began automating the extraction of data from our collection of Android apps. In addition, this guide was also the source of information on exactly what information was stored within an app's manifest file and how this information is used when an app is installed and run on a system. This information would help us decide what information we extracted from our collection of apps in order to build our demographic overview of an Android market.

From the beginning of our research we knew that we wanted to examine a relatively large body of apps when extracting demographic information. To this end we next turned our attention to other work which had been done on large collections of apps to examine how these collections were obtained for analysis and what data was analyzed from each app in these collections. We found a good number of studies of this nature that have been completed by a variety of researchers. The majority of these studies focus on the Android permissioning system or some other topic related to a security issue. To our knowledge no studies have been completed to provide a broad range of demographic information on a body of Android apps.

In a 2012 study, a team examined a collection of 20,500 new and 650 popular Android apps to investigate the risk signals inherit in apps which intrude on a user's privacy [CHIA 12]. Instead of actually obtaining the each app's .apk file this group used the application information page from the

Android market to obtain the data they analyzed. This data included the app installation count, average community rating of the app, number of ratings, developer URL, price, content maturity level, and the permissions requested by the app.

With this data in hand they then divided the permissions available in the Android operating system into three categories: safe; dangerous; and dangerous and information relevant. They classified dangerous permissions as permissions for actions which could be potentially harmful to the user, while dangerous and information relevant permissions were a subset of the dangerous permission category including all permissions which allow access to sensitive user information. Any permission which did not fall into one of these two categories was considered safe for the purposes of this study. They then attempted to draw correlations between the number of permissions requested in each of these categories and the other information they had collected about each app. They concluded that at this time no reliable risk signals currently exist to identify apps which invade a user's privacy, at least not from the information available on the Android market app pages. They suggest that this is primarily because users are trained to accept the requests for permissions from these apps, causing the permissioning system to become less effective over time.

In other work, a group from the University of Hong Kong analyzed a body of 1179 apps to check for a known privilege escalation attack on the permissioning system of the Android operating system [CHAN 12]. This group obtained apps for study by downloading them from AndroidFreeware a third party Android market specializing in free software for Android devices. For this study, the group obtained the actual Android package (.apk) files that are used to install an app on a device. They then used a series of reverse engineering tools to extract the data in which they were interested. First, a tool was used to convert the .apk files to .jar files. Once in this format the manifest file for the app (AndroidManifest.xml) was extracted and converted from a binary format to a human readable XML format. This file is required for all Android apps and contains a wealth of information including what permissions are requested by the app. Once this manifest file was in a human readable form, it was then parsed for each app to determine if the app used at least one permission and there existed an activity or

service component that did not require any permissions and was publicly visible. If both of these conditions evaluate true, components satisfying the second condition hold the potential the leak capability and lead to the privilege escalation attack these researchers were looking for. Of the 1179 apps this group examined 6 apps which were previously thought safe were shown to exhibit this privilege escalation vulnerability, including the popular Adobe Photoshop Express app.

This study influenced our research in several ways. First, this was the first study we examined in which a large body of apps was obtained from a third party market. This proved to be a promising idea, one we would run across multiple times as our research continued and eventually use ourselves. Additionally, this was the first work we encountered in which the .apk file for an application was reverse engineered and then the resulting artifacts used to evaluate features of an app. This immediately seemed to be a promising idea as we could obtain much more information from successfully reverse engineering an app than we could from its compiled state or from the market place app pages as used in the previous study. Finally, this was the first occurrence we discovered of information being extracted from an Android app's manifest file. As our research continued it would become apparent that this file should be a central focus of our efforts due to the large amount of valuable information it contained.

We next looked at a study which attempted to determine if an Android app was using the policy of least privilege as suggested by the Google Android developer's guide [FELT 11a]. This policy suggests that developers only request permissions from the user that are actually required for the app they design. This helps eliminate security concerns but is challenging to check for in an automated fashion. These researchers wrote a program entitled Stowaway to perform this task. To accomplish this goal they first had to obtain a body of Android apps and build a permission map for each method in the standard Android API. To build this permission map, automated testing tools were used to extract what permissions were required for each API call. For a body of apps this group chose to download 940 apps from Google's officially supported Android Market.

Once these components were in place these researchers used a third party tool to disassemble each app's Dalvik bytecode or .dex files. These disassembled files were then parsed and all calls to the standard

Android API were identified and extracted. The Android permissions required for these API calls were then determined by comparing calls found with the previously constructed permission map. Finally, the permissions requested by the app in the manifest file were compared to this list of required permissions to determine if overprivileging had taken place. This study found that of the 940 apps examined roughly one third of them were in fact overprivileged.

This research again proved the feasibility of extracting information from an Android app after modifying the original .apk file through decompilation or reverse engineering and also suggested some third party tools which could be used for this purpose. In addition, the fact that automated testing methods were used to determine properties of an Android app was also an interesting concept. This provided another avenue for us to explore when we determining how to perform data extraction in our own research.

A final paper of this type that we examined concerned the act of detecting repackaged Android apps [ZHOU 12]. This study was impressive for several reasons, but one of the main reasons was the sheer volume of apps analyzed. These researchers examined six third-party Android markets, two from each the United States, China, and East Europe. From these six markets they obtained 22,906 apps. The main goal of their research was to determine how many of the apps found in these third-party markets were actually repackaged apps originally from the Google Android Market. To this end they also obtained 68,187 apps from the Android Market itself.

In order to determine if one of these third party apps was a repacked app the Dalvik bytecode was obtained using an existing disassembler. This byte code was then stripped down to just opcodes since operands, such as string values, could easily be changed during the repacking of an application. Finally a technique of fuzzy hashing was used to generate a fingerprint for app. This process was repeated for each app in the Android Market as well. These fingerprints were compared in a pairwise manner and each of the over 88 million resulting pairs was given a similarity scoring. When an app received a similarity score which was greater than a predetermined value, the third party app was determined to be a repacked app. Using this method, these researchers concluded that 5% to 13% of apps found in these third party

markets were actually repackaged apps from the Android Market. They concluded that the most common reason for repacking an app was to either remove advertising placed in the app by the original developer or to replace this advertising scheme with a different one to redirect revenue generated from in app ads to another party. They also found a few cases in which malicious code had been inserted into the app during repacking.

The volume of apps used in this study confirmed two things in our minds as we moved forward. First, third party Android markets could provide our research with a large enough data set to perform the type of analysis we were looking to conduct and, secondly, that it was feasible to conduct a large-scale study on source code obtained from Android apps. The Dalvik executable decompiler used in this work, baksmali, would also prove to be the decompiler we used to in our own work. For this reason, it was beneficial to see how these researchers processed and used this decompiler in their own research before we began to explore ways in which we might do the same. Finally, the ability of third parties to change the behavior of these apps during repacking proved as a proof of concept that information could be extracted from apps in this state. By being able to change an advertising scheme or insert malicious code into a repackaged app these third parties demonstrated an ability to take an app which was packaged for distribution and gain at least a minimal understanding of the original source code. This gave us reason to believe that as our research progressed we should be able to do the same, extracting data of a demographic nature instead of changing the behavior of the app.

After examining research that had been conducted on large bodies of Android apps we focused our attention on the topic of reverse engineering Android apps in order to allow us to extract information from them. At this time our main goal was to determine what methods existed for reverse engineering apps and what data could be extracted from apps once these methods had been applied. Since the actual construction of a reverse engineering toolset for Android apps was outside the scope of our research, we hoped instead to find a method already in existence that would allow us to extract the maximum amount of demographic information from each app. From our previous research on the Android operating system we knew that at a minimum we wanted to be able to examine each app's manifest information as well as

the app's directory structure. Access to the original or close to the original, source code of the application would be beneficial as well.

One of the first tools of this type that we discovered was Baksmali and its sister program, Smali [GRUVER 12]. Inside of each Android app's .apk file is a classes.dex file. This file contains the classes of the Android program compiled into the dex (Dalvik executable) file format which is understood by the Dalvik virtual machine, the virtual machine on which apps are run on an Android device. The .dex files are equivalent to Java's .class files while the Dalvik virtual machine serves the same purpose as the Java virtual machine (JVM). The .dex file format and Dalvik virtual machine were designed by Android to be a specialized format highly suitable for devices with limited memory and processing power such as smart phones or tablets. Baksmali converts the classes.dex file into a format that is similar to assembly code. This makes the code human readable, albeit much harder to understand and follow than the original Java code. From a demographic standpoint, it is possible to extract, with some effort, most of the structural information of an Android app's source code from files in this format. In addition, this code maintains all of the functionality of the original program. The sister program of Baksmali, Smali, takes the code and then recompiles it back into the .dex format for use on an Android device. Using these two tools in tandem therefore allows an interested party to make changes to the code of an Android app and then run this modified code on an Android device.

Another tool we discovered was Androguard [DESNOS 11]. Some disassemblers make mistakes during the disassembly process which could lead to lost data. For example Smali sometimes has problems correctly recovering numbers in the source code. Androguard claims that they have been able to overcome this problem [DESNOS 12]. Androguard is more robust than Smali and has the ability to read and write .dex and .class files into full Python objects for manipulation. Androguard also maintains a database of known Android malware and allows you to check if an Android app you are manipulating is part of this database. Finally, Androguard offers a tool to transform the AndroidManifest.xml file from its native binary form into a human readable form allowing information to be extracted from it as well.

Having the ability to read the `AndroidManifest.xml` file was particularly important to our research and would be a feature that we would require of any tool we decided to use.

A different type of tool we encountered were tools which attempted to decompile files from the dex format to Java. One of these tools, `ded`, was presented by a team from Pennsylvania State University and was used to aid in their analysis of Android application security [ENCK 11]. The `ded` decompiler's ability to produce java source code was important for two reasons. First, the team wanted the ability to use tools already in the community to analyze Android source code once it had been decompiled. They found that many tools of this nature already existed for Java source code yet none existed for the analysis of dex bytecode. The second reason that the acquisition of Java source code was important was that the team wanted the ability to identify false-positives in their research by manually inspecting the source code after analysis had been completed. They decided that this would be nearly impossible to perform on dex bytecode, making the Java source code necessary.

The `ded` decompiler works in three phases: retargeting, optimization, and decompilation [OCTEAU 10]. The first phase of this process retargets an Android app's `.dex` file into Java classes and is by far the most complicated of the three. To accomplish this, variable typing information is recovered from the Dalvik bytecode. This is challenging because this bytecode deals with register declarations and sometimes does not contain enough information to determine the exact type of a variable or a constant. For this reason, type inference must be used during this phase of decompilation. Next, the format of the Android `.dex` and Java's `.class` constant pools are different so this must be compensated for. In a `.dex` file a single constant pool is maintained for the entire application while in Java a constant pool is maintained for each class in the program. Additionally, Java bytecode uses the constant pool for references to most primitive type constants while dex byte code places these primitive constants directly into the bytecode itself. Once these differences have been adjusted the final stage in this step of the decompilation process is the translation of the actual method code. This is handled by a two phase process in `ded`. First, the dex bytecode is preprocessed to identify structures in the code which cannot be directly translated into bytecode for the Java virtual machine. Then, once these issues have been solved, the remaining dex

bytecode is linearly traversed and translated into Java bytecode, .class files, which is able to be processed by the JVM.

Once code is in this state the decompilation process can now be completed by taking the .class files and decompiling them into their equivalent Java source code. The only problem with this is that the .class files produced in the first phase of this operation are unoptimized. One example of how this lack of optimization might present itself is in the handling of for loops. Most decompilers convert for loops into infinite loops with break instructions. While this code is functionally the same as the original for loop, it can be harder to read and understand, especially when embedded loops are used. For this reason this group used the Java decompiler Soot to finish the decompilation process. Soot has the ability to optimize Java bytecode during its decompilation, eliminating many of these problems.

For us, this study served as a comprehensive overview of how the nuts and bolts of the decompilation of an Android app could take place. In addition, it also demonstrated that the original Java source code for an app could be recovered with a fairly high success rate, just over 94% of the time in this study. From this we concluded that it was an option to examine the actual Java source code of an app for demographic information, presenting up with another possible option for gathering data in our own study if we deemed it necessary. This would make the process of extracting any demographic information from the source code much easier than if we were to use the Davlik bytecode produced by the other decompilers we had seen up until this point.

Further research into this area produced another decompiler, dex2jar, with the ability to produce the original Java source code for an Android app's .apk file [PAN 12]. To our knowledge, no information has been made available about the inner workings of dex2jar except for the release of its own source code. From the developer's description of the program dex2jar has the ability to decompile an .apk file into a Java archive (.jar) format. Once in this format a user can view or modify the Java source code of an app. Dex2jar also includes the functionality to translate this Java code back into an .apk file format.

After researching source code decompilers we began to investigate any additional options to convert the binary format of an Android app's manifest file into a human readable form. Since an app's

manifest file contains a large amount of demographic information about the app we knew that a tool of this nature was a high priority for our research. To this end we uncovered three different options. The first, AXMLPrinter2, was a tool that had been in development in 2008 as part of a larger effort to convert Android app code into J2ME code [SKIBA 12]. As of the latest information on the developer's website this project has been put on hold but this program is still available for download. Since no one is actively working on this project no documentation could be found on the actual functionality of the AXMLPrinter2 other than its brief description. Even though this project lacks and current development effort or support we still found this program to be used in at least one of the other research efforts we came across [CHAN 12].

Another tool we found for this purpose, axml2xml, was part of a collection of programs targeted at helping Android developers [GUILFOYLE 12]. This program had the same functionality of the other programs we researched but was interesting for that fact that its developers suggested using it to discover the types of layouts being used by other Android developers. We found this intriguing because it was in line with our own goals of finding a program to allow us to extract information of this sort from an app's manifest file.

The final tool of this nature that we found was the program apktool [WISNIEWSKI 12]. This program was different from the other two in the fact that it actually marketed itself as a tool to be used for the complete reverse engineering of Android apps. It possesses the ability to decompile resources into nearly their original form, generate smali code for an Android app, and convert an app's manifest file into human readable form. In addition this tool also had the ability to reverse this process, converting all of these materials back into an .apk file which could then be loaded onto an Android device. This tool appears to tie together several other tools, including some features of the Android SDK such as the Android asset packaging tool (aapt). This was the most comprehensive tool suite for aiding in the reverse engineering of Android apps the we came across during our research though it did lack the ability to produce the Java source code, instead of smali byte code, of an Android app.

Finally, we turned our attention to research that had been performed examining individual characteristics of an Android app. The main goals of this portion of our research were to determine what aspects of Android apps had been examined in the past and to aid us in deciding which properties of an app we would examine in our own research. The majority of the research that we reviewed in this area concerned itself with Android's permissioning system. One example of such work is the short paper out of the University of Toronto concerned with taxonomizing permissions by three characteristics: the level of control given to users, the amount of information which the permissions convey, and the level of interactivity the permissioning system grants to the user [AU 11]. For their research they defined a permissioning system as any system which allows a user to define a per-application policy that constrains what resources an application may access on their phone. They also state that the permissioning system may communicate to the user of an app what hardware or software resources the app may access now or in the future. They examined six different mobile operating systems, including Android, to determine the number of permissions available on the platform as well as the overall control, information, and interactivity provided by each operating system's permissioning system.

For the Android operating system they found that 137 different permissions existed at the time of their study. These permissions are divided into four categories: signature, system, normal, and dangerous. Only normal and dangerous permissions can be used by third party apps with signature and system permissions reserved for apps that have been developed by a firmware manufacturer or come preinstalled on a device. For this reason they did not consider signature and system permissions during their analysis of the Android permissioning system. This left them with 75 permissions to analyze which could be used in any third party application.

These researchers rated the Android permissioning system as medium in control, high in information, and low in interactivity. They found that even though Android had by far the largest number of permissions in any of the permissioning systems they examined, end users only had the option to either allow or deny an app all of the permissions it requests in bulk. Due to the lack of any fine grained control mechanisms they felt that Android only provided a medium level of control to users. Again due to the

large amount of permissions available as well as their descriptive names they felt that the level of information provided by the permissioning system was high although they did note that this large amount of information could be confusing to the average user. Finally, they rated the interactivity of the permissioning system as low since users only interact with the system when an app is installed. Once a user has granted an app a permission it has the ability to use this permission until it is uninstalled from the device. We found the practice of dividing the permissions into groups that could and could not be used by third party apps to be an interesting approach and one that might be useful to our research as we moved forward.

Another interesting piece of information that came out of this research was the amount in which the permissioning system on Android devices has changed over time. They found that on average 4 permissions were changed per each major release of the Android operating system from version 1.0 up to version 2.3, the latest release when this work was completed. In this context a change means that a permission was either added, removed, or depreciated. In fact, only one Android release during this time frame, version 2.0.1, did not make any changes to the permissioning system. Version 2.3 of the Android operating system was released two and a quarter years after version 1.0 and by this time 32 changes had been made to the permissioning system. This was applicable to our work as any research involving permissions used on an Android device would have to take this high level of permission churn into account.

Another study along the same lines sought to determine how effective the Android permissioning system actually was [FELT 11b]. Researchers studied 100 paid and 856 free applications from the official Android Market. They then looked at the permissions that each of these apps requested paying special attention to dangerous permissions. These permissions have been classified as dangerous by Android because they hold the ability to be potentially harmful to the end user. This group includes permissions which allow the app to perform functions such as recording audio or video, opening network connections, or even disabling the device itself. They found that over 90 percent of the data set they analyzed requested at least one of these dangerous permissions and that 10 percent requested at least

seven. Additionally, they found that a small subset of these dangerous permissions was requested by a large number of the apps they examined. The permission INTERNET was found to be the most requested permission showing up in 86.6 percent of all free apps and 65 percent of all paid apps examined. The next three most prevalent dangerous permissions were WRITE_EXTERNAL_LOCATION, ACCESS_COARSE_LOCATION, and READ_PHONE_STATE all showing up in more than 30 percent of the free apps examined.

From the high percentage of apps they found requesting dangerous permissions, this group drew the conclusion that users are accustomed to installing apps which request dangerous permissions. This diminishes the effectiveness of the permissioning system because users become accustomed to agreeing to these requests and then do so out of habit without considering if the developer of the app is trustworthy or not. They also noted that some permissions, such as INTERNET, are so prevalent across the Android Market that users no longer consider its warning anomalous making them essentially useless in the permissioning scheme. Examining permissions in this manner, determining what percentage of apps requested certain permissions and the prevalence of permissions in a market as a whole, was directly in line with the type of information we hoped to extract from our collection of apps when completing our demographic overview of a market. Therefore as our research progressed and we decided to exclusively use third party markets as our data source this study would be a valuable comparison point as we could see how the data we collected compared to data which had been collected from the official Android Market.

Two other pieces of research that we discovered gathered information of a similar nature. First, a group from Carleton University examined the permissions requested from a group of 1,100 Android apps by using Kohonen's Self-Organizing Map (SOM) algorithm, a type of neural network algorithm, to look for groupings of permissions that reoccur throughout the data set [BARRERA 10]. The data analyzed consisted of the top 50 most popular free apps in each of the 22 categories of the official Android Market in December of 2009. Apps from the communication category of the Android market requested the highest number of different unique permissions, 62, while apps in the theme category requested the

lowest, 1. Permissions were considered to be unique if they were requested by at least one app in the category, so if two or more apps in a category requested the same permission it was only counted once. They, like the previous group, also found INTERNET to be the most prevalent permission, being requested by 62.4 percent of all apps they examined. Finally, they analyzed which permissions were more prevalent in certain categories and uncovered several pairs of permissions which were frequently requested together such as ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION or WRITE_SMS and READ_SMS.

A second group from Smobile Systems analyzed 48,964 apps to in an attempt to uncover malware apps based on the permissions they requested [VENNON 10]. To accomplish this, researchers first isolated a set of Android apps known to be malicious and then examined and recorded the permissions that they requested. Next, for each new app they examined, they compared it to this collection of permission lists to see if a match was generated. If a match was found the new app was examined in detail to determine if it was actually malware itself. They found that of the apps they examined roughly 20 percent of apps requested permissions to access private or sensitive information that could potentially be used for malicious purposes. Also found were 29 apps with permission signatures exactly matching that of known malware and 8 apps that requested the permission BRICK, allowing it to completely lock the device at will. In addition, and more interesting to our own research, this group compiled a list of the top 20 permissions requested in the apps they examined. Again, INTERNET was the clear leader found in 34,636 apps or 71 percent of their data set. The next four most popular permissions which they examined were ACCESS_COARSE_LOCATION, WRITE_EXTERNAL_STORAGE, ACCESS_FINE_LOCATION, and READ_CONTACTS with 25, 16, 15, and 9 percent of their data set respectively.

In addition to studies which examined the permissioning system and the permissions requested by individual apps we found several other studies which examined information of demographic interest in a market. First, during their work on detecting repacked apps, which we discussed when looking at research which had been performed on large bodies of Android apps, a group from North Carolina State

University also looked at market crossover of apps from the official Android Market to six different third party market places [ZHOU 12]. That group found that of the nearly 23,000 apps they examined, 13.5 to 30 percent of each third party market were apps for the Android Market that have been redistributed. This piqued our interest and influenced us to attempt to determine how many of the apps we examined were found in more than one of the markets in our own data set.

A final paper that we examined which had direct application to our own work was a paper exploring the security risk posed when using third party advertising libraries while monetizing an Android app [GRACE 12]. This group of researchers analyzed 100,000 apps from the official Android Market in early 2011 to determine what libraries were being used for this purpose and how many, if any, were posing risks to developers using them or end users of the app itself. Their research focused on the development of a system known as AdRisk to systematically identify these potentially harmful libraries. In the end, they concluded that a variety of potentially harmful libraries exist and suggested that their results clearly showed a need for more regulation in the way in which third party libraries are integrated into Android apps. More interesting to our research, however, was the fact that they identified 100 representative libraries used for the purpose of monetizing apps. Additionally, they found that over 50 percent of the apps they examined had at least one of these libraries in use. This data would lead us to look into the advertising schemes used in the apps in our own data set, exploring how many used some of the more popular advertising libraries.

Additionally two other research projects we discovered suggested some directions our research could take in the future. The first project of this type looked into the human computer interface side of Android apps in an attempt to determine what types of apps Android device owners were using and when they were using them [BOHMER 11]. This study resulted in several interesting findings such as the conclusion that on average users spent just under 60 minutes per day on their Android devices. They also found that the average session with an app was less than a minute although longer averages of close to six minutes were seen between the hours of four and five am. These early hours of the morning experienced the least amount of users with approximately 30,000 users in their sample using their devices while the

evening hours of three to eight pm experienced the most with around 170,000 active users. Finally, they found that communication apps were the apps most often used first and earliest in the day on a user's phone while games were more frequently played later in the day. While this research had no direct effect on our own research it would be an interesting direction to see our research take in the future.

In another paper Steve Gold, while talking about Android security, presents some interesting data compiled by the Nielson Company in 2010 concerning the state of the mobile app marketplace [GOLD 12]. In a survey conducted they asked smart phone users to report on what categories of apps they had used in the past 30 days. 61 percent of the group surveyed reported that they had used a game app in the past 30 days. The next three highest reported categories were maps/satnav at 50 percent, social networking at 49 percent, and music at 42 percent. All remaining categories they asked about had been utilized by less than 40 percent of the survey group in the previous 30 days. Again, while this research did not apply directly to the type of demographic information we were currently looking to collect it would be another interesting area to look into in the future.

Chapter 3 – Solution

Methodology

In order to meet our goals of determining what demographic information could be extracted from an Android app's .apk file and using this information to construct a demographic overview of an Android market we first had to acquire a body of Android apps to use as our data set. To this end three different viable options presented themselves. First, we could use the official Android Market, now known as Google Play, to download the apps used for our research. The next option we explored involved using the Google search engine to search for open directories on the internet which contained .apk files and then download files directly from these different sources. Finally, we could use third-party markets and download apps from these in much the same manner as with the Google Play market.

We first explored the option of using the official Google Play market to obtain a body of apps on which to perform the research. This was a natural source of apps as it is the market that is readily available on the majority of Android devices and the market officially supported by Android. The Google Play market currently hosts more than 450,000 Android apps [GOOGLE 12d]. These apps are divided into categories such as casual games, business applications, or education. Additionally, all apps are divided into free or paid sections of the market and ranked based off of their popularity which is determined by aggregating rankings provided by users of the market.

Of all the markets we examined Google Play also made the most demographic information about each app in its market available to their users. For each app they include a side bar, shown below in Figure 3, which contains the rating of the app, the last date it was updated, the current version, the Android SDK version required to run the app, the category the app has been assigned to, a range for the number of installs the app has received, and the app's size, price, and content rating. Also supplied are any changes made to the app during its latest update, Figure 4, and a list of the permissions which the app requires, Figure 5. Due to the nature of our work, attempting to form a demographic overview of a market, the availability of this information was highly of interest to us.

Although this market provided a wealth of additional information in addition to each app's .apk file we ran into several obstacles when attempting to collect data from the Google Play market. The first, and most prominent of these, was the fact that while the Google Play market is designed to be

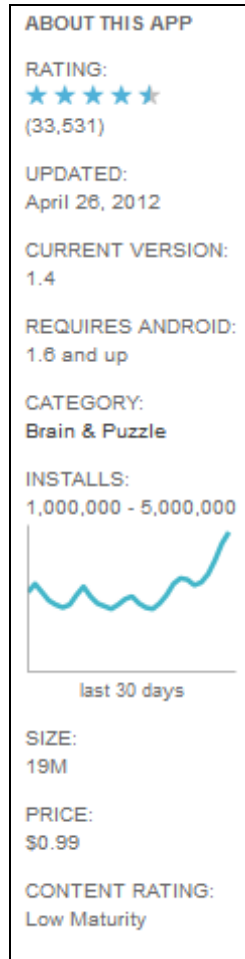


Figure 3 – Google Play Informational Side Bar for a Random App [GOOGLE 12e]

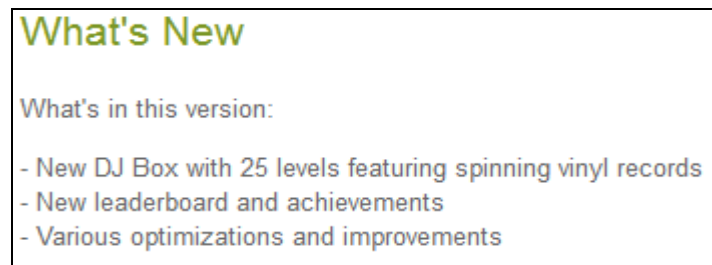


Figure 4 – Google Play Update Notes for a Random App [GOOGLE 12e]

Permissions

THIS APPLICATION HAS ACCESS TO THE FOLLOWING:

YOUR LOCATION

COARSE (NETWORK-BASED) LOCATION
 Access coarse location sources such as the cellular network database to determine an approximate tablet location, where available. Malicious apps may use this to determine approximately where you are. Access coarse location sources such as the cellular network database to determine an approximate phone location, where available. Malicious apps may use this to determine approximately where you are.

NETWORK COMMUNICATION


FULL INTERNET ACCESS
 Allows the app to create network sockets.

PHONE CALLS

READ PHONE STATE AND IDENTITY
 Allows the app to access the phone features of the device. An app with this permission can determine the phone number and serial number of this phone, whether a call is active, the number that call is connected to and the like.

STORAGE

MODIFY/DELETE USB STORAGE CONTENTS MODIFY/DELETE SD CARD CONTENTS
 Allows the app to write to the USB storage. Allows the app to write to the SD card.

 Hide

NETWORK COMMUNICATION

VIEW NETWORK STATE
 Allows the app to view the state of all networks.

DEFAULT

MARKET BILLING SERVICE
 Allows the user to purchase items through Market from within this application

Figure 5 – Google Play Permission Information for a Random App [GOOGLE 12e]

accessed from either a web interface or mobile app. It only allows users to download apps to their mobile devices, blocking any attempts to download an app directly to a desktop computer. This would prove to be an issue due to the fact that we needed the .apk files of the apps on a computer in order to complete our analysis of them. To solve this problem we first looked at downloading apps to a computer using the

Android emulator, thereby removing this obstacle. We were able to install the Google Play market on an emulated device and download apps in this fashion but two other problems arose. First, the memory space that the emulator uses cannot be accessed directly outside of the emulator itself. Essentially, we could download apps directly to our computer but now had no way to access these apps once they were downloaded. The next issue resulted from the way in which the Google Play market is designed. Google has designed this market in such a way that it is extremely difficult to automate the downloading of apps. We concluded that this was done to help prevent attacks on the markets available bandwidth and help address piracy issues but it was also successful at stopping our more benign efforts as well.

We also looked at the possibility of downloading apps to a mobile device and then moving their .apk files to a computer afterwards for analysis. Moving an apps .apk file from a mobile device to a computer is not an issue thanks to the capabilities of the Android debug bridge (adb), a tool provided by Google for communicating with and debugging apps on a device. The issue again arose in the automating of the downloading of these apps. We looked into the possibility of writing an app to accomplish this but ran into the same issue we had met before, namely the intrinsic design of the Google Play market. As a final effort at obtaining data from this market we made several contacts with different employees at Google to see if they could provide us with a body of apps for academic research purposes. As of yet we have not been able to contact anyone who is willing or has the authority to release this type of information. Due to these complications we decided to abandon this means of data collection for the time being and explore other avenues.

The next solution we looked into involved using a normal Google search coupled with some advanced operators to find open directories on the internet which held .apk files. To achieve this the following Google search was used: `-inurl:htm -inurl:html -inurl:php -inurl:jsp intitle:"index of" apk`. The hyphen in the search string narrows the search by excluding results with certain characteristics.. Used in conjunction with the *inurl* operator, we are able to remove search results that consisted of htm, html, php, or jsp pages. The final *intitle* operator ensures that the string following itself, in this case index of, is found in the title of the pages displayed in the search

results. By using the string “index of”, we increase our chances of finding directory listings since this is a default naming scheme. Finally the *apk* term acts as a normal Google search term. Because of the way this search is constructed, it could easily be modified and used as a method of finding files of a different type, say mp3 music files, by replacing the apk term with mp3.

This search query returned 42,000 results for pages matching our requirements. From manually checking several of these results, the amount of false positives -- results yielding no apk files -- was relatively low and positive results returned pages which contained anywhere from just a handful to several thousand apk files each. This method clearly could provide a significant amount of data for our research but again problems arose when we began to look at automating this process. It is actually against Google’s terms of service to use its search engine in an automated fashion [GOOGLE 12f]. We discovered this fact when we attempted to use a web crawler aimed at the results of a Google search of this nature to seek out apk files and download them. After receiving notification from Google that their service could not be used in this manner we found ourselves in the same situation as before: we had located a source of a significant amount of Android apps but had no way in which to download them except by downloading each app manually. Due to the time constraints of our project and the availability of other sources of data we decided to continue looking for a source which would allow the downloading individual apps to be automated.

We next turned our attention to third-party Android markets. These markets serve the same function as the official Google Play market, providing a place for Android developers to host apps and Android users to obtain them either for free or at a price, but unlike Google Play, they are not regulated or endorsed by Google. We found a wide range of this type of market. Some specialized in a specific type of app such as games or communication programs. Others were simply a collection of apps that one person or a group had found useful. While still others featured markets containing many different types of apps for many different purposes. The format in which each of these markets presented their apps and how users were expected to download the apps varied greatly, making only some of these markets suitable for our purposes.

When examining these markets we looked at several attributes to determine if a given market was a candidate for our data collection process. First we determined if the market contained a variety of different types of apps. Since we were attempting to construct a demographic overview of several markets as a whole and then compare them to one another, we did not want to collect data from a market only specialized in one type of app, such as a market only offering games. While checking what types of apps were hosted on the market, we also determined if the market charged the user for the ability to download apps. This was important for our research since we had no funding for the purchase of apps and planned on conducting our research on only free apps. Next, we checked to ensure that the process of downloading apps from the market to a computer could be automated. Several factors were found that hindered us in this area. Markets requiring an account and log in before downloading apps made the automation process much more complicated. In the same vein, markets which used captchas, presented data in a non-html format, or included excessive amounts of ads or timers in their download process complicated matters as well. Additionally, several of the third party markets we discovered were designed to work as the Google Play market, only allowing users to download apps directly to a device. These markets had an app that had to be installed on a device before any additional apps hosted on the market could be installed. This app functioned nearly identically to the Google Play app, allowing users to browse and purchase apps for their device.

For us an ideal market was one which had free apps from a variety of different categories, did not require a user to log in to use its services, presented information in simple html without the extensive use of intrusive advertising or captchas, and allowed apps to be downloaded directly to a computer. We examined 46 third party markets and found several potential candidates which met all of our qualifications. From these candidates four markets were chosen based on the size of the market, measured by the number of apps present. We decided that it would be beneficial to conduct our research on several different sizes of markets allowing us to compare one to another. By doing this we hoped to begin to determine if the size of a market had any correlation to its demographic makeup.

The four markets chosen for our research were Apps For Adam (<http://appsforadam.tk>), And App Online (<https://www.andapponline.com>), App Town (www.apptown.com), and Slide Me (<http://slideme.org>). Once we had decided on these markets, a targeted web crawler was written for each market to discover each app hosted on the market and, if possible, download a copy of each app's apk file to an external hard drive. Targeted web crawlers were used instead of traditional crawlers because of the strict structure each of these markets used in their html code. This process allowed us to save time during the app acquisition phase of our research by taking the shortest path to each download link.

These targeted crawlers used the same base code to which slight modifications were made. First the crawler was directed to start crawling at a page containing a listing of apps available on the given market. The crawler would then parse the html code for the page and find all links which lead eventually to a download link for an apk file. The targeted crawler differed from a true web crawler in that instead of following each link found on the page, the only links followed were links which were guaranteed to lead to a download link on some later page. This was only possible due to the structure of the websites for these markets. Regular expressions were used to identify each of these links and these regular expressions changed for each different market. Once a link was found it was followed the new html code was parsed and the process was repeated until the link for the apk file was found and downloaded. The exact number of times this process had to be repeated before the download link was found also differed depending on which market was being processed at the time. Once an apk file had been downloaded the crawler moved on to the next app listed in the market and the process repeated until all apps listed in the market had been downloaded. The source code for these and all other programs written for this research can be found in Appendix 1.

The four markets yielded nearly 14,000 apps for analysis. Tables 1 through 4 show information on the results of crawling each market. Additionally, the final tallies from each of these markets as well as the total amount of apps acquired can be seen in Table 5.

Apps For Adam Crawler Results	
Apps Examined	154
Unique Apps	125
Duplicate Apps	29
Errors	0

Table 1 – Apps For Adam Targeted Crawler Findings

And App Online Crawler Results	
Apps Examined	685
Unique Apps	685
Duplicate Apps	0
Errors	0

Table 2 – And App Online Targeted Crawler Findings

App Town Crawler Results	
Apps Examined	2987
Unique Apps (free)	2260
Duplicate Apps	610
Pay Apps	117
Errors	0

Table 3 – App Town Targeted Crawler Findings

Slide Me Crawler Results	
Apps Examined	15531
Unique Apps (free)	10815
Duplicate Apps	355
Pay Apps	4351
Errors	10

Table 4 – Slide Me Targeted Crawler Findings

Apps Downloaded Per Market	
Apps For Adam	125
And App Online	685
App Town	2260
Slide Me	10815
Total	13885

Table 5 – Total Apps Downloaded By Market

Note that neither the market Apps For Adam or And App Online hosted any paid apps that had to be processed during this step. For this reason the pay apps row is left out of their respective tables. In nearly all of the cases shown above, a duplicate app consisted of an app that fell into one of two categories. It was either an app that was being promoted by the market and therefore listed in more than one place throughout their website, or it was an app for the market itself which could be installed on a user's device to give access to the market without having to go through a web interface. If a market had an app of this type it was often listed repeatedly as a means of self promotion. Finally, the errors recorded during the downloading of apps from the Slide Me market were all bad download links for apk files which could not be followed.

Now that we had a significant body of apps to work with our next goal was to get these apk files to a point that we could extract demographic information from them. To accomplish this we first had to decide on a tool to use to reverse engineer the Android apps, allowing us to obtain as close as possible the original source code of the apk files. We found several tools that were suitable for performing pieces of this process but only one which could reverse engineer the entire apk file and produce all of its original components in some form. The program, apktool, is an open source project hosted on Google code (<http://code.google.com/p/Android-apktool/>). It takes an app's apk file and reverse engineers it produce the resources, assets, libraries, smali source code, and AndroidManifest.xml file. Several tests were run with this tool to determine its effectiveness by taking the original source code for an app, compiling it to an apk file, and then using this tool to decompile the apk file which had been generated. The artifacts produced from the running of apktool were then compared to the original files to determine what, if any, information was lost in the process. In each of the tests we performed no information of interest was lost for the resources, assets, or AndroidManifest.xml files. The libraries and original source code of the app were reproduced as smali code instead of Java code, which was to be expected. Smali code is essentially assembly code for the Dalvik virtual machine (DVM). This smali code is human readable, consisting of operands and operators, but is extremely hard to follow or alter. The decision was made that smali code would be sufficient for our purposes.

Once we decided that this tool would perform the task we needed it to another program was written to automatically traverse the directories which held our collection of apk files and reverse engineer them using the apktool program. The reverse engineering of our collection of apk files took roughly 10 hours and resulted in 72.6 GB of data stored in over five million files. The program used to accomplish this can be found in Appendix 1.

With our app collection now to a state in which we could begin extracting demographic information from it, we next had to decide exactly what information we wanted to extract. An apk file is divided into five key components: the META-INF folder, res folder, resources.arsc file, AndroidManifest.xml file, and classes.dex file. Each of these key components must be found inside of an app's apk file. Other directories such as ones for additional assets or external libraries may be found as well but are not required by the Android specification.

The META-INF directory of an Android app contains metadata about the app itself as well as certificate information of the app. This directory contains three files: MANIFEST.MF, CERT.SF, and CERT.RSA. The MANIFEST.MF and CERT.SF files contain name/SHA1-Digest pairs matching the names of components in the rest of the apk package to their SHA1-Digest. The final file in this directory, CERT.RSA, contains the certificate information for the application in a binary format. When examining these files we found little information that would be useful from a demographic standpoint therefore this directory was not used as we continued with our data collection process.

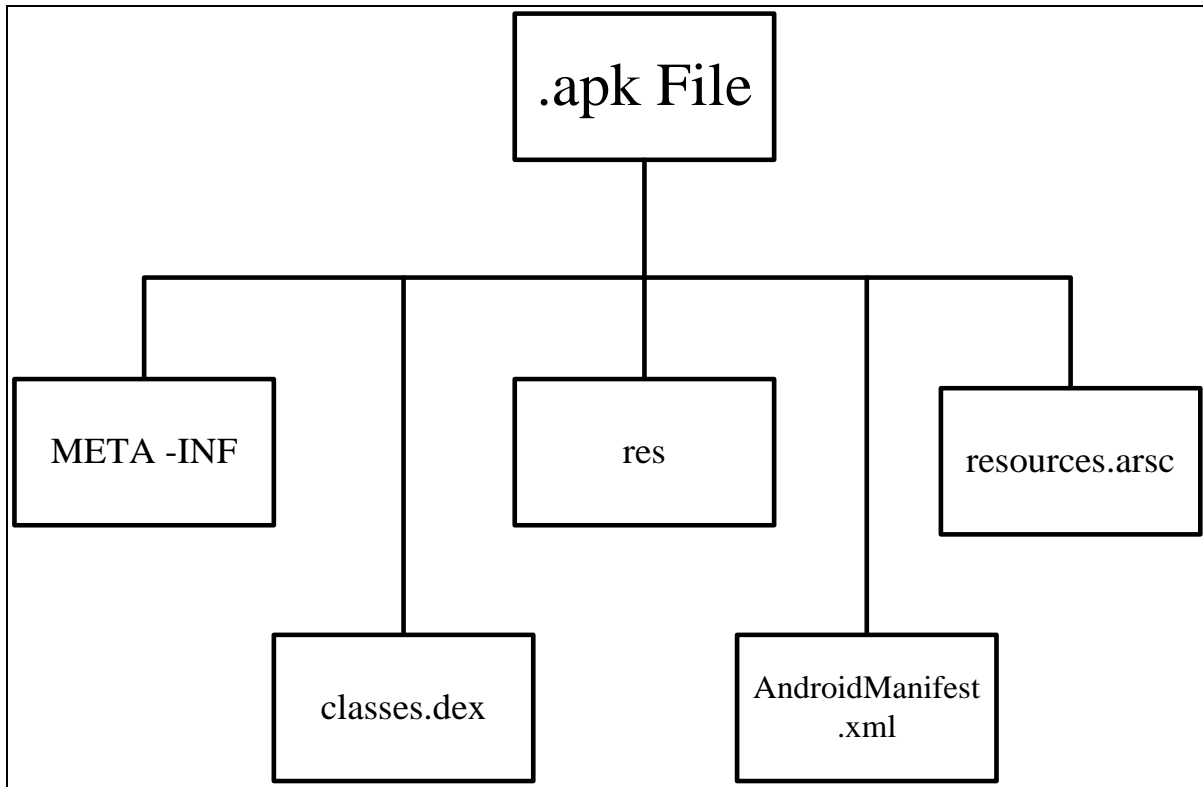


Figure 6 – apk File Directory Structure

The next item in an apk file is the res directory. This directory contains all resources used in the app which have not been compiled into the resources.arsc file. A variety of resources can be stored in this directory, so to aid in their organization, this directory is usually subdivided. Several examples of subdirectories that may occur are drawable, layout, and raw. Both the drawable and layout subdirectories are fairly common. The drawable subdirectory holds image files or xml files which describe a drawable shape, while the layout subdirectory holds xml files which describe the layout of an app on a device's screen. The raw subdirectory serves as the storage area to which any arbitrary asset files are assigned. All resources found in these subdirectories can be referenced in an Android app by their filename.

When developing an Android app, another subdirectory, values, is always found in the res directory of the project. This subdirectory is unique in the fact that it acts as the storage area for XML files which are compiled into many different resources when the apk file is created. An example of the type of information which is found in an app's values subdirectory are the string literals used throughout

an app. XML files in this subdirectory are compiled into the R class of an Android app during the creation of an apk file. For this reason they are referenced through this class instead of by their filename when used in an Android program. In addition to compiling these resources into the R class of an app, the Android build process also produces a resources.arsc file, which contains a resource table as well as other information for these compiled resources.

The tool we elected to use to reverse engineer the apk files in our study has the capability to take the res directory, resources.arsc file, and R.class file and return the original res directory to the state it was in before compilation. In addition to being able to see exactly what types of resources and the resources themselves that developers were using in the apps, this directory proved important to our demographic survey because of the way in which Android handles resource localization. Each of the resource types found in the res directory is able to be localized based on a variety of factors such as the country the device is being operated in or the dots per inch (dpi) of the screen on which the app is being viewed. To accommodate these types of localizations Android has developers use a naming scheme within the res directory itself to distinguish which resources should be used and when. For instance if a user wanted to localize their values subdirectory in the res directory to use different values for users speaking Chinese or French they could simply name their subdirectories values-zh and values-fr respectively. Moreover, if a developer wanted to further localize the Chinese language values he includes with his app to accommodate users of dialects from China and Twain he could use an additional regional qualifier. In this case he would have two subdirectories named values-zh-rCN and values-zh-rTW. A more extensive example of this localization scheme in practice can be seen in the directory structures in Figures 7 and 8 below. We decided that it would be worth our while to explore the demographics of this localization scheme in practice. To this end the resource directories become one of the two components of our decompiled apk collection that we focused most heavily on. From this directory, we were able to determine what localizations were being used most frequently for the values, layout, and drawable resources, the amount of the market taking advantage of localizations, and how these values compared across multiple markets.

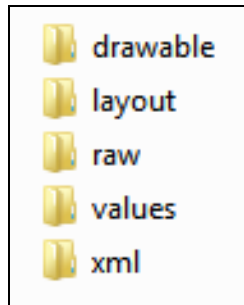


Figure 7 – Sample res Directory Structure with No Localization

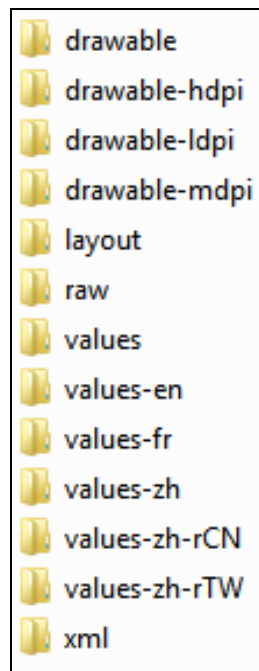


Figure 8 – Sample res Directory Structure with Localization

During the compilation of an Android program all Java source code files are compiled into Java byte code or .class files. These files are then optimized to run on an Android mobile device. The files that are the result of this optimization are known as dex files or Dalvik executables which are, in essence, instructions for the Dalvik virtual machine (dvm). The Davlik virtual machine is comparable in function to the Java virtual machine except for the fact that it is designed to run this optimized file format instead of a normal Java class file. All of the dex files for an Android app are gathered together into the classes.dex file found in the app's apk file.

When reverse engineering the classes.dex file, we produced smali source code for the program. Smali code is essentially assembly code for an Android device. Due to low level nature of smali code, it is extremely hard to understand and follow. In addition, since this code operates at such a low level, the amount of code produced for even a small program can quickly surpass what is able to be interpreted without the aid of a computer. For our research, we decided not to look into the actual source code of an Android app, be it smali or Java code, at this time. We do believe that some very interesting demographic information could be gained by exploring apps in this nature but we felt that it was currently outside the scope of our research. An example of smali source code for the classical hello world program can be seen below in Figure 9.

```
.class public LHelloWorld;
.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System; ->out:Ljava/io/PrintStream;

    const-string    v1, "Hello World!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream; ->println(Ljava/lang/String;)V

    return-void
.end method
```

Figure 9 – Smali Source Code for a Standard Hello World Program [SMALI 12]

The AndroidManifest file contains a wealth of information about the app and holds information which the operating system requires before it can install or run the app. In its native form, this file is stored in a binary xml format. During our reverse engineering process we converted this file back to human readable xml. This file is required to be a part of all apk files, have the exact filename of AndroidManifest.xml, and be at the root of an app's file structure. Types of information in the AndroidManifest file include the name of any permissions the app must have to execute; the minimum Android API level under which that the app can operate; information on the services, providers, and receivers that the app declares or uses; declarations of hardware devices that the app uses; and many other critical pieces of information about how the app interacts with the Android device and/or other programs

[GOOGLE 12g]. The manifest file became the main focus of our data gathering activities because of the large amount of information it contained.

In addition to containing this wide variety of information, the AndroidManifest file is also strictly structured due to the fact that the Android operating system specifies that this file must be a well-formed XML document. XML is a tagged markup language consisting of elements beginning with a start-tag and ending with a matching end-tag. The content of an element in XML may include additional elements, forming a nested structure. In addition, each tag in an XML document can have attributes attached to it providing another way to further define the elements declared by these tags and embed more information into the document. Finally, all well-formed XML documents have a declaration included at the top of the document describing information about the document itself such as which version of the XML standard is being used throughout the document.

The Android operating system utilizes all of these components of XML in the AndroidManifest file. XML elements in the AndroidManifest file correspond to actual components of the app while attributes declared in each tag further define these components. These attributes are often used to specify exactly which one of the several valid values of a component is being used or to attach a name to a new component which is being declared for the first time. For example, a tag such as `<permission android:name = "com.example.project.DEBIT_ACCT" \>` is a permission element being utilized to declare a new permission for the app. The attribute `android:name` is used to name this new permission to `com.example.project.DEBIT_ACCT`. This scheme of declaring an element and then using attributes to further define it is seen consistently throughout the AndroidManifest file.

```

<?xml version="1.0" encoding="utf-8"?>
<manifest>

  <uses-permission />
  <permission />
  <permission-tree />
  <permission-group />
  <instrumentation />
  <uses-sdk />
  <uses-configuration />
  <uses-feature />
  <supports-screens />
  <compatible-screens />
  <supports-gl-texture />

  <application>

    <activity>
      <intent-filter>
        <action />
        <category />
        <data />
      </intent-filter>
      <meta-data />
    </activity>

    <activity-alias>
      <intent-filter> . . . </intent-filter>
      <meta-data />
    </activity-alias>

    <service>
      <intent-filter> . . . </intent-filter>
      <meta-data />
    </service>

    <receiver>
      <intent-filter> . . . </intent-filter>
      <meta-data />
    </receiver>

    <provider>
      <grant-uri-permission />
      <meta-data />
    </provider>

    <uses-library />

  </application>
</manifest>

```

Figure 10 – Required AndroidManifest.xml File Structure [GOOGLE 12g]

Figure 10 shows the how an AndroidManifest.xml file is structured. Only the elements seen in this Figure are allowed to be present in the manifest file but not all of these elements must be utilized in each manifest file. The only elements required to be present in each manifest file are the manifest and application elements, each of which can only appear once per manifest file. The manifest element is the root element of each AndroidManifest file and must contain an application element. The manifest element must also declare the Android namespace as well as the package to which the app belongs. The namespace of an app should always be set to `http://schemas.android.com/apk/res/android` while the developer is allowed to assign any package name they wish. In addition, the manifest tag can contain other attributes which apply to the entire app such as the version code or install location of the app. The application element declares the app itself and serves as a container for each element contained within the app. Again, this element can contain elements which apply to the entire app such as the location of the icon to be used for the app, the name of the app which will be displayed to the user, or the name of a permission which another program must have to interact with the app. All other elements and all attributes are optional, although an app that only utilized a manifest and application tag would not be very interesting. Many of a tag's attributes are only technically optional as they have default values when not explicitly declared.

We decided that several elements of the AndroidManifest.xml file contained information which would be interesting to study from a demographic standpoint. The data that we extracted from this file was stored in the attributes of various elements and included information about the permissions used and declared by the app, the minimum SDK level required to run the app, the services, providers, and receivers utilized by the app, the activities established by the app, the publisher of the app, and the advertising schemes used within an app. A program was written in Python utilizing the `xml.dom.minidom` library provided with Python to parse the XML files and extract the information of interest to us from each decompiled Android app. The code for this program can be seen in Appendix 1 of this report.

Once we had decided what data we wished to extract from our decompiled Android apps, the next step in our research process was to determine a way in which to persist this extracted data so that analysis could be performed. We decided that the most efficient way to accomplish this task would be to use a database system. This would not only allow us to persist our data after our programs executed but would also provide us with a way in which to interface with this stored data and perform queries on it, a feature which we needed in order to meet our goal of building a demographic overview of these markets. To this end we decided to use MySQL for our database needs. This decision was made for several reasons. First, MySQL is the world's most popular open source database management system [MYSQL 12]. Due to the amount of attention that this brings to the product a large body of documentation has been produced to support individual efforts to implement this database system. We felt that this documentation, and the community that produced it, would be valuable as we implemented a system of this nature to meet our own needs. Next, MySQL, at least in the capacity that we utilized it, is a free product which allowed it to fit within the financial constraints of our work. Finally, we wanted to find a database system which we could interface with directly from Python. All of the coding we had conducted during our research up until this point had been completed in Python and we wished to maintain this consistency. A third party open source library known as MySQLdb was found that provided this functionality [MYSQLDB 12]. Once this library was integrated with our existing source code it was a simple process to have our program write the information extracted from the apps to our database instead of printing it to the standard output stream.

Results

With all of these pieces in place we proceeded to extract the data we were interested in to our newly established database and then perform analysis on it. The first set of data we looked at was the number of apps we had from each of our four markets and the percentage each market contributed to our final data set. Once we had pruned all of the files we downloaded by removing duplicate apps as well as apps which did not decompile correctly or were downloaded incorrectly we had a total of 13,320 apps for analysis.

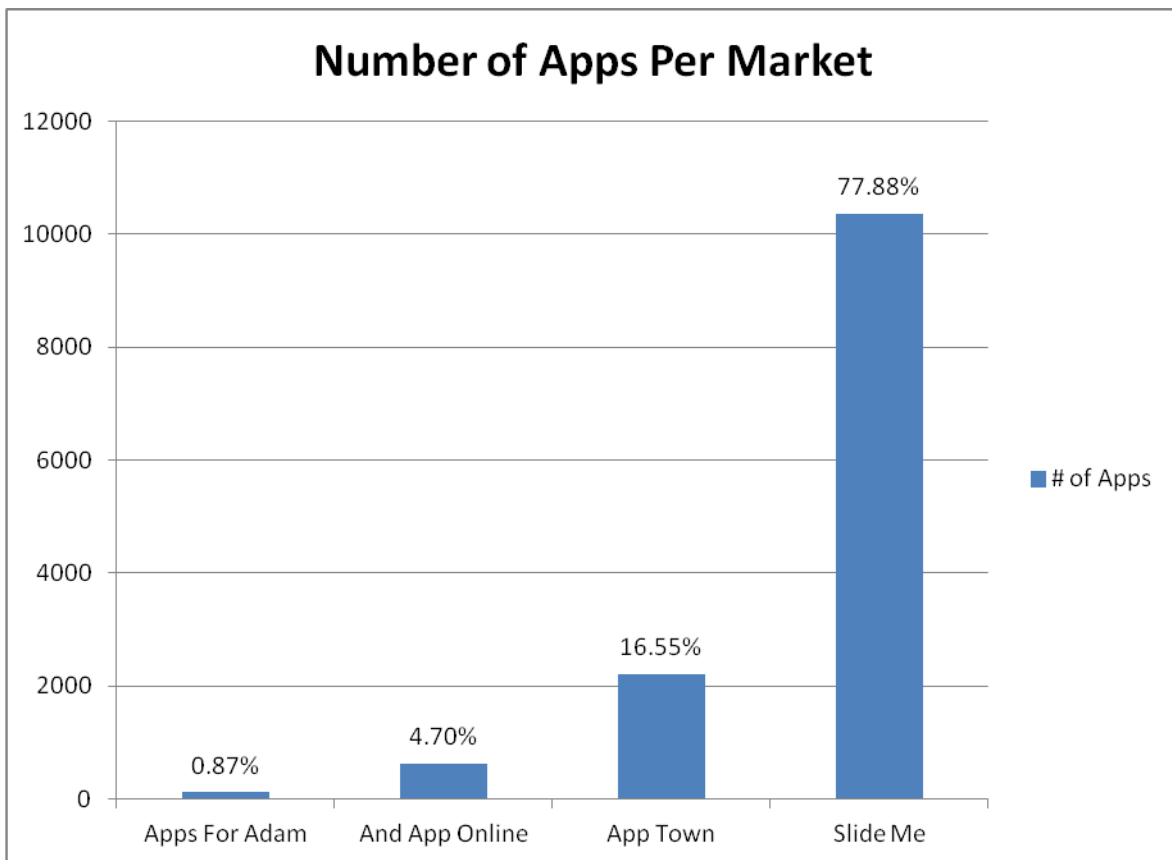


Figure 11 – Number of Apps Per Market After Pruning

This data was gathered simply by counting the unique apps for each market in our database after extraction had been completed. We knew that each app we examined was unique due to the naming/keying scheme we used when extracting data from our decompiled apps. Each app we downloaded had a unique name consisting of the market it was downloaded from followed by the name of

the app itself. This unique filename was then used as the key for any information about that app which was entered into our database system. The Slide Me market contributed the majority of the apps we examined with 10,373 total apps. The App Town market was the next largest contributor with 2,205 apps, these were followed by And App Online with 626 apps and Apps For Adam with 116 apps. The percentage that each market contributed to our body of apps was then simply calculated as the number of apps from a given market divided by the total number of apps in our database system.

We next looked at the first true demographic for our collected data by examining the file size of each app's apk file we had downloaded. First, we looked at the file size for each app as seen in Figure 12 below. This was done by simply extracting the file size of each app, sorting this data in nondescending order, and then plotting it on a scatter graph. From this we were able to get a general feel for the amounts of apps of different sizes that we were studying. There was a nearly linear increase in the size of apps from the smallest apps we came across up until the app size reached around 8 MB. At this point the curve took on a more exponential shape showing that less of these larger size apps are being produced. This trend continued with the amount of apps being produced decreasing significantly as size increased with the largest app we examined being around 48 MB. A slight anomaly was seen around the apk file size of 24 MB which had a significantly larger amount of apps than the file

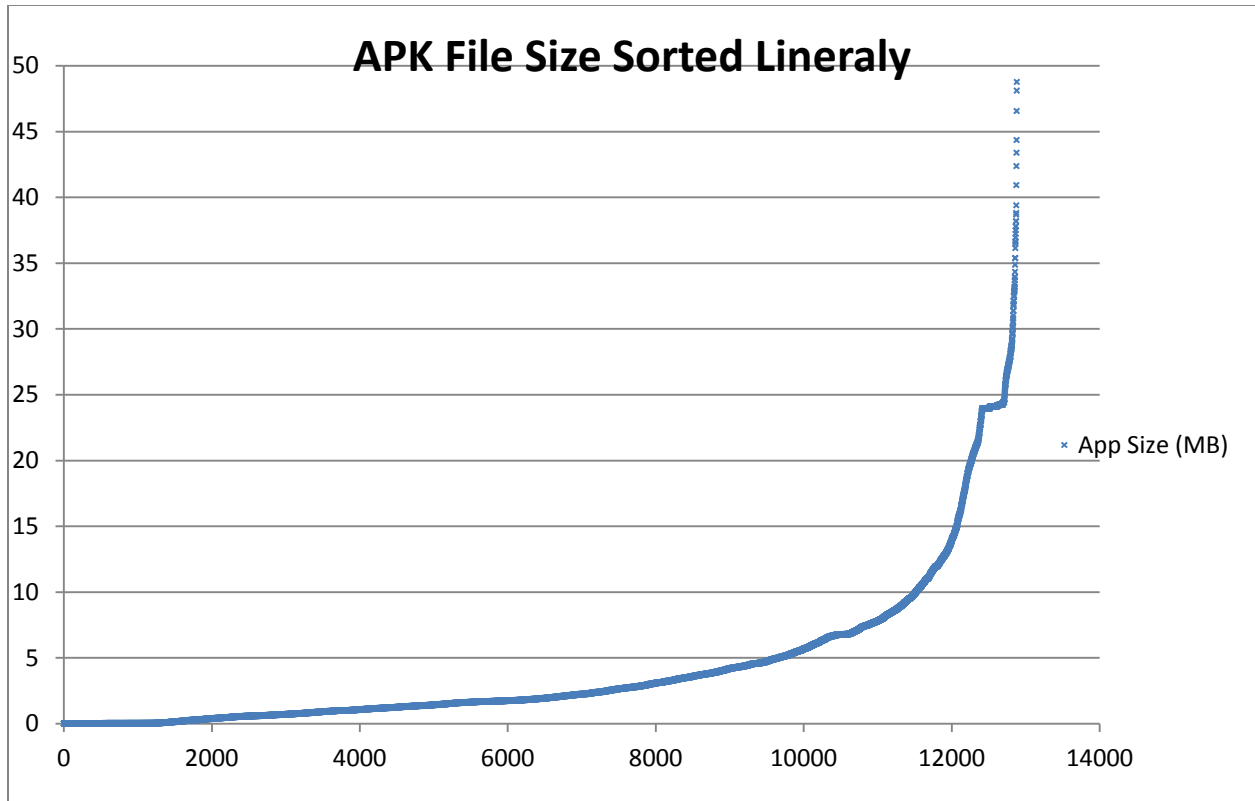


Figure 12 – APK File Size Sorted Linearly

sizes surrounding it. This anomaly was due to a series of related apps found in the Slide Me market. These apps were a family of approximately 200 apps used to track the stats of baseball players or teams. The similarities of these apps lead to their close size causing this linear section to be seen in the exponential section of the graph.

We next looked at the minimum, maximum, and average file size per market. All of the minimum file sizes were fairly consistent coming in at under 0.2 MB for each market. Upon closer examination these small apps were usually consisted of simply the shell of an app which then linked the user to an external website. The only market in which this was not the case was Apps For Adam. This market was unique in the fact that it was a small collection of tools a user had compiled for their own personal use before sharing them online. Therefore, no apps of this nature were found as they were not found to be useful by the group’s collector. This is also the reason for the average file size in the Apps For

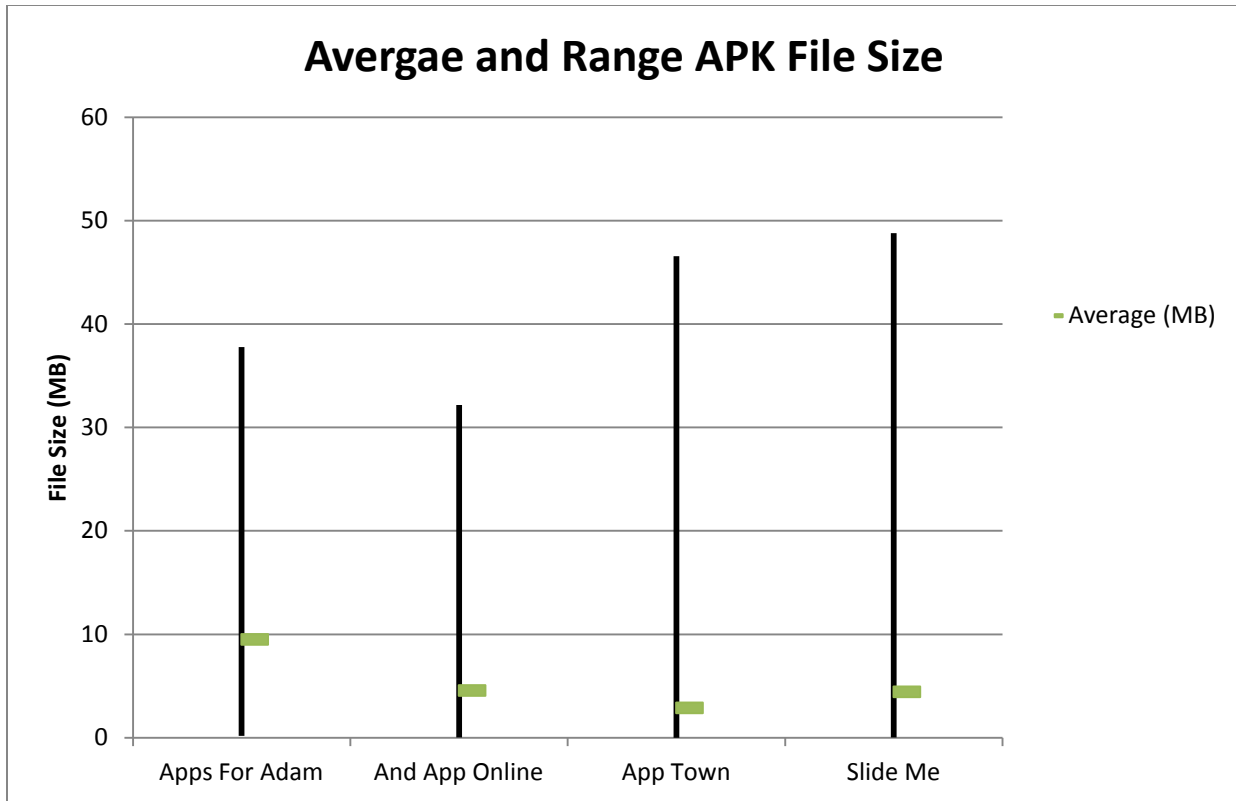


Figure 13– Range of File Sizes Per Market

Adam market being so much higher than the others. The apps in this market generally consisted of business or communication apps and therefore were larger than most. Finally, it was interesting that the two larger markets contained the largest apps while our two smaller markets did exceed 40 MB for any app at all. We feel that this is due mainly to the scarcity of apps at these larger sizes as shown in Figure 12. The smaller amount of apps at these larger sizes lower the chance of finding one of these apps in a smaller market. Surprisingly, these larger apps did seem to be spread fairly evenly across multiple categories of apps instead of being clustered in one category such as communication or games as we originally thought they would be.

We next looked at how many apps on each market were different versions of the same app and not actually a new app. This was determined by examining the fully qualified name stored in the package attribute of the manifest element of the AndroidManifest.xml document. According to the Android

standards this fully qualified name should not be changed when new versions of the app are released, only the version number should be different. We found that each market had at least two apps of this nature

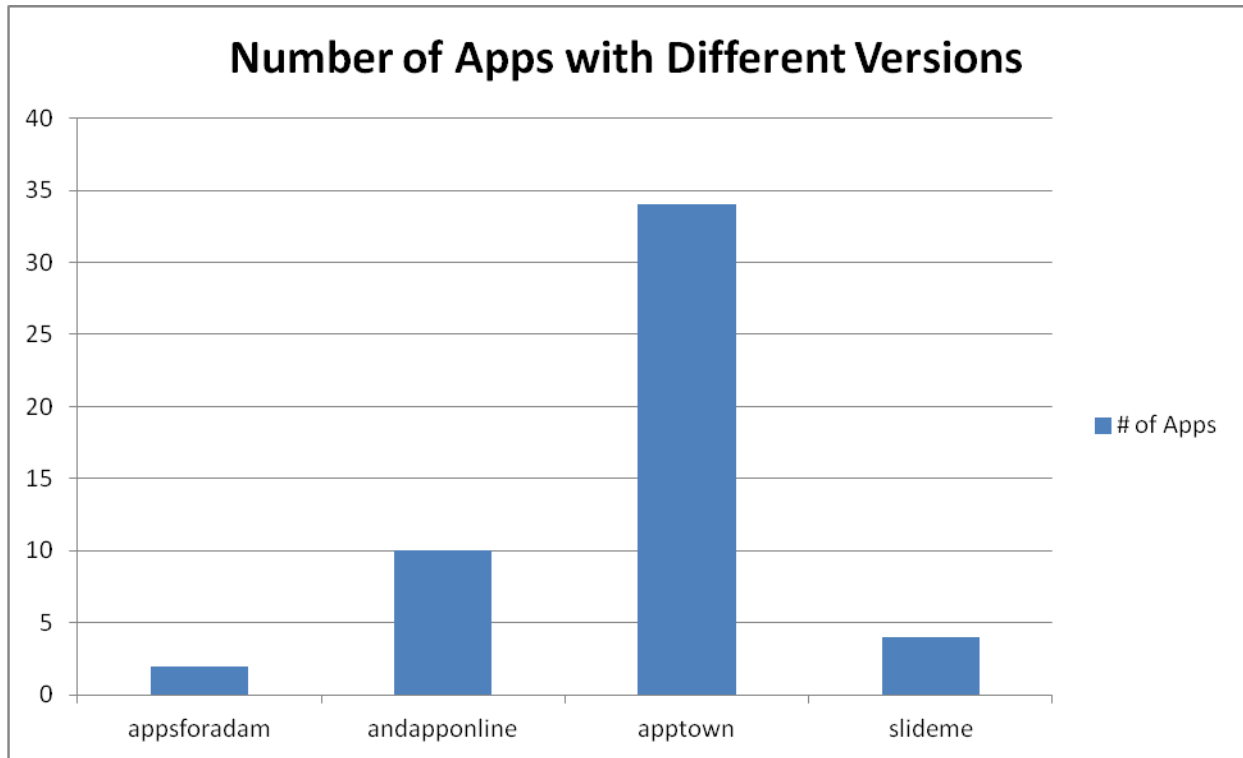


Figure 14– Apps with Different Versions per Market

with the largest amount found in any market being 34. The number of apps with different versions constituted less than 2 percent of any given market and made up less than 0.5 percent of our total body of apps. We believe that this is mainly due to the policies in place in each market. These markets only want to host the latest version of each app and each of them, except Apps For Adam, have policies to this end. When examining the apps that were duplicated most of these scenarios were a case of the app being hosted under a different name, therefore making it appear to be a different app, and not a case of a market knowingly hosting two different versions of the same app. It should be noted that when we speak of versions here we mean purely old and outdated versions of an app versus the newest version of an app not a free and paid version of the app or any other scheme of this nature. These apps were counted as two separate products for the purpose of this data.

After examining duplicate versions of the same app we moved on to examine the number of different apps published by the same developer or company. This information was again extracted from fully qualified name of the app similar to the process used to determine if multiple versions of the same app existed in the market. In this case only a substring of the fully qualified name was used as the Android specifications state that the fully qualified name of an app should be the internet domain the individual developer or company developing the app owns listed in reverse order followed by the name of the app they are publishing. For example the Google maps app could be com.google.maps. The Android developer specifications prescribe using the same starting sequence for all apps that are developed by the same person or company. Because of this property we were able to strip the first part of the fully qualified name off each app and then use these to determine if a publisher was publishing multiple apps to one market. Using this data we found that each market had at least a few apps of this nature. In fact the percent of apps in each market were published in this fashion were very similar, hovering around 8 percent, except for the And App Online market which jumped up to nearly 12 percent. The maximum number of apps we found published by and one developer or company was 297 and was found in the largest of our markets, Slide Me.

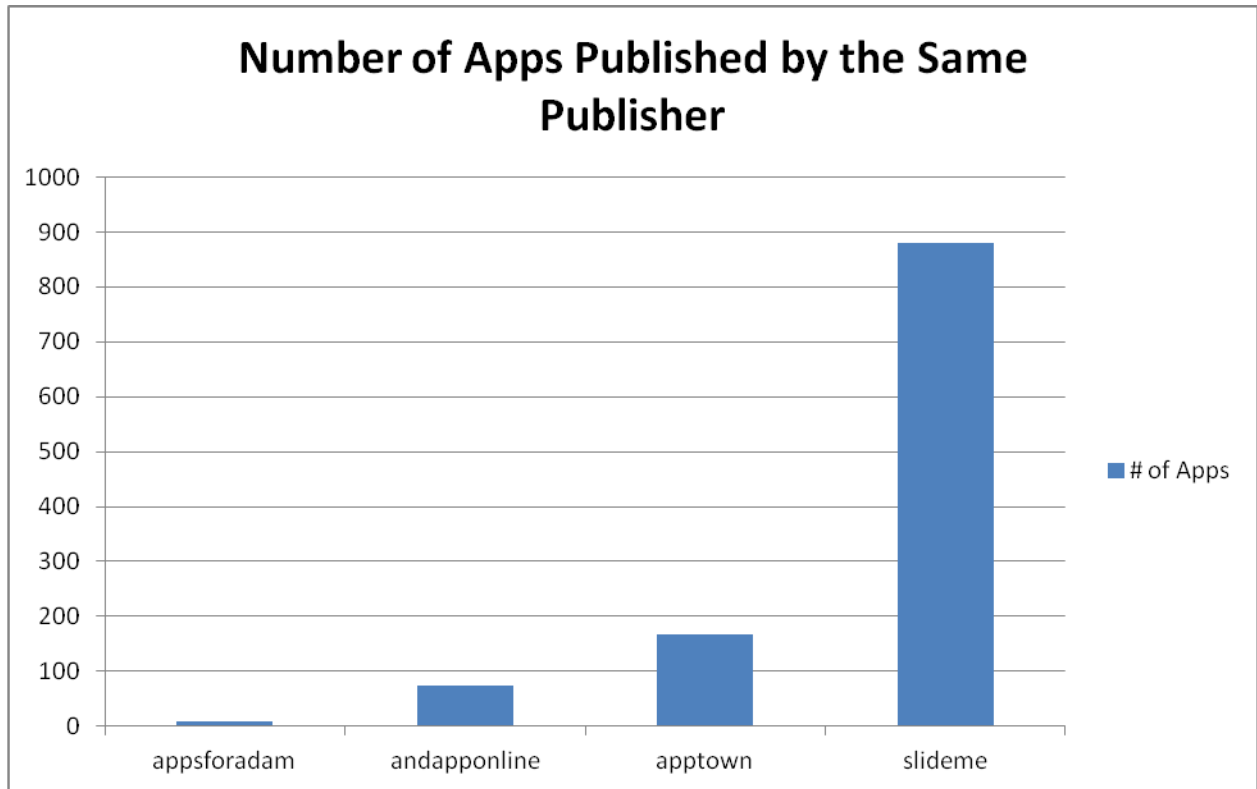


Figure 15– Multiple Apps By The Same Publisher Per Market

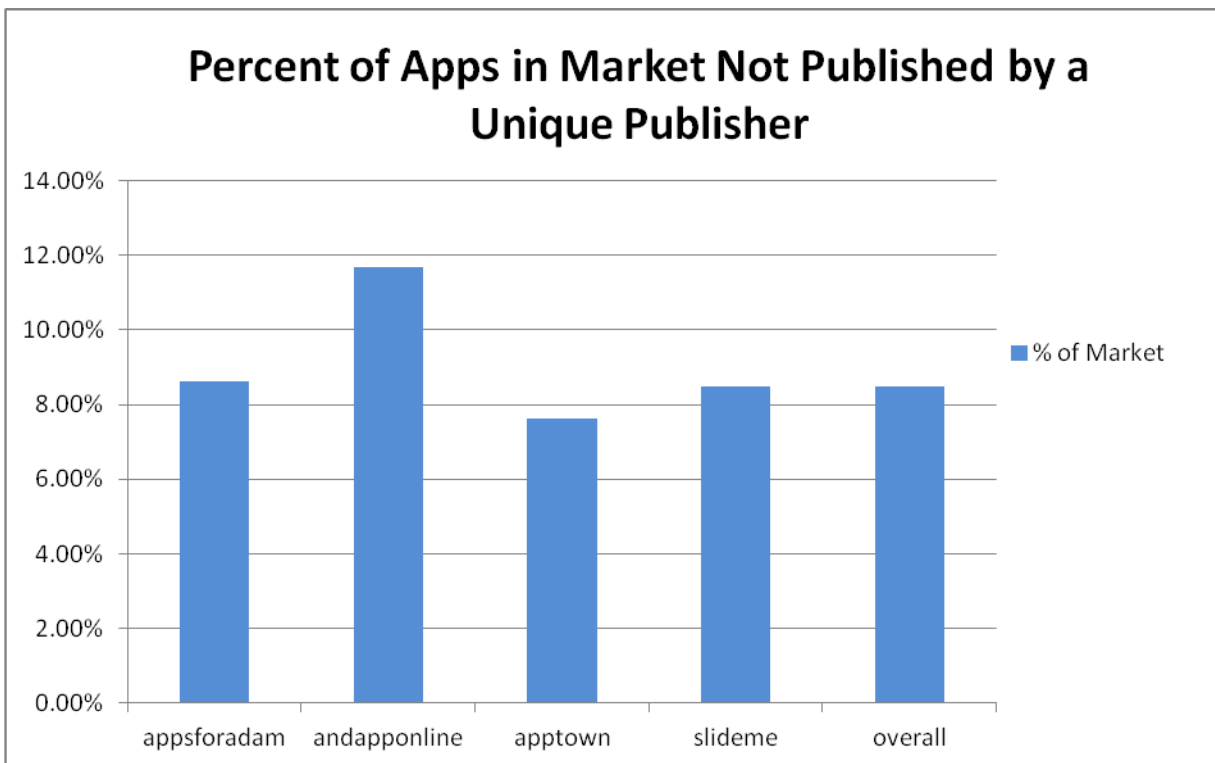


Figure 16– Percentage of Market Published by a Nonunique Publisher

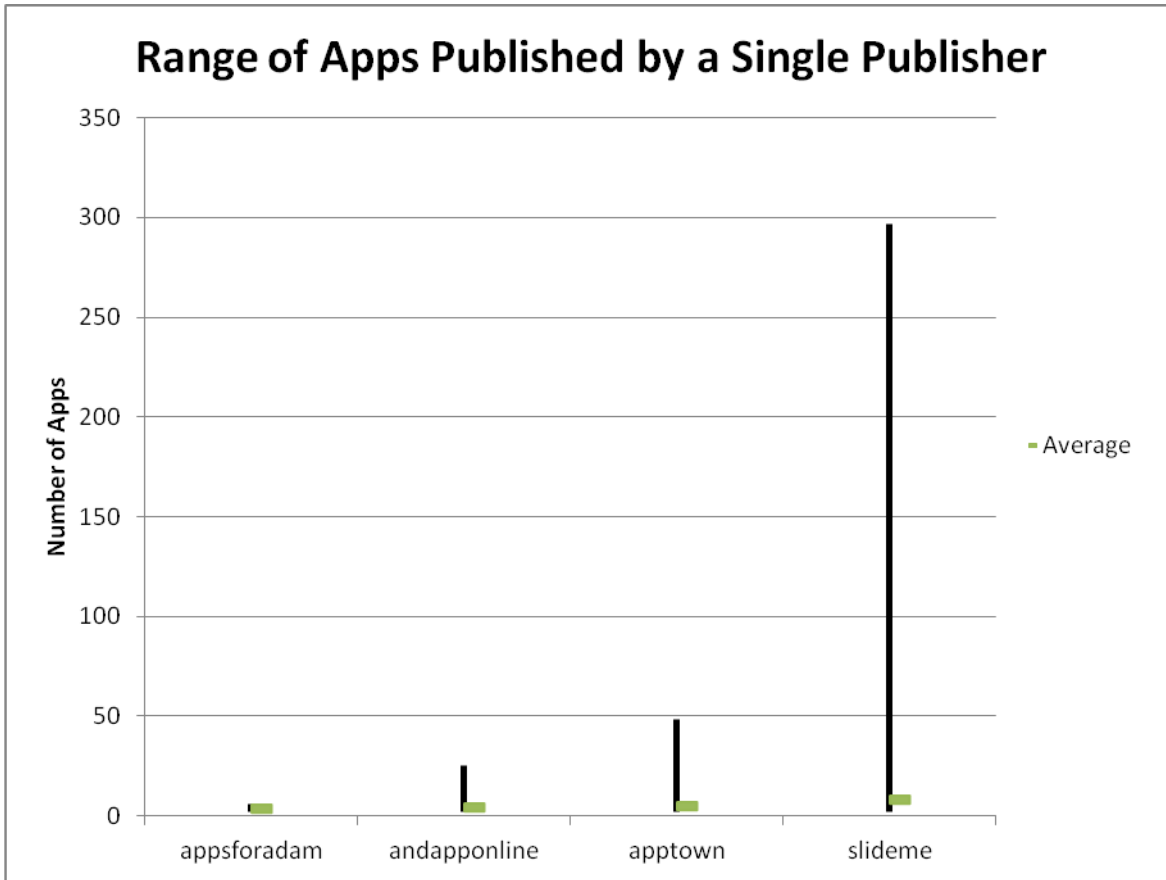


Figure 17– Apps Published By a Single Publisher

The final piece of data of this nature that we looked at was the amount of app crossover between the markets in our study. We defined app crossover to be the number of apps found in any two or more markets. Of the markets we studied And App Online and Slide Me had the strongest correlation among their two markets with 419, or just over 40 percent, of the apps found in And App Online being found in the Slide Me market as well. We expected a large percentage of the smaller markets in our study to show up in the larger markets we studied. In this respect, it was interesting to see that 40 percent was the highest crossover we found and that this was significantly higher than the amount of crossover found in the other market pairs. It was also interesting that none of the apps we examined were found in all four of the markets that we studied. The crossover data is presented in Figures 18 and 19 below. Note that to save space the market names have been

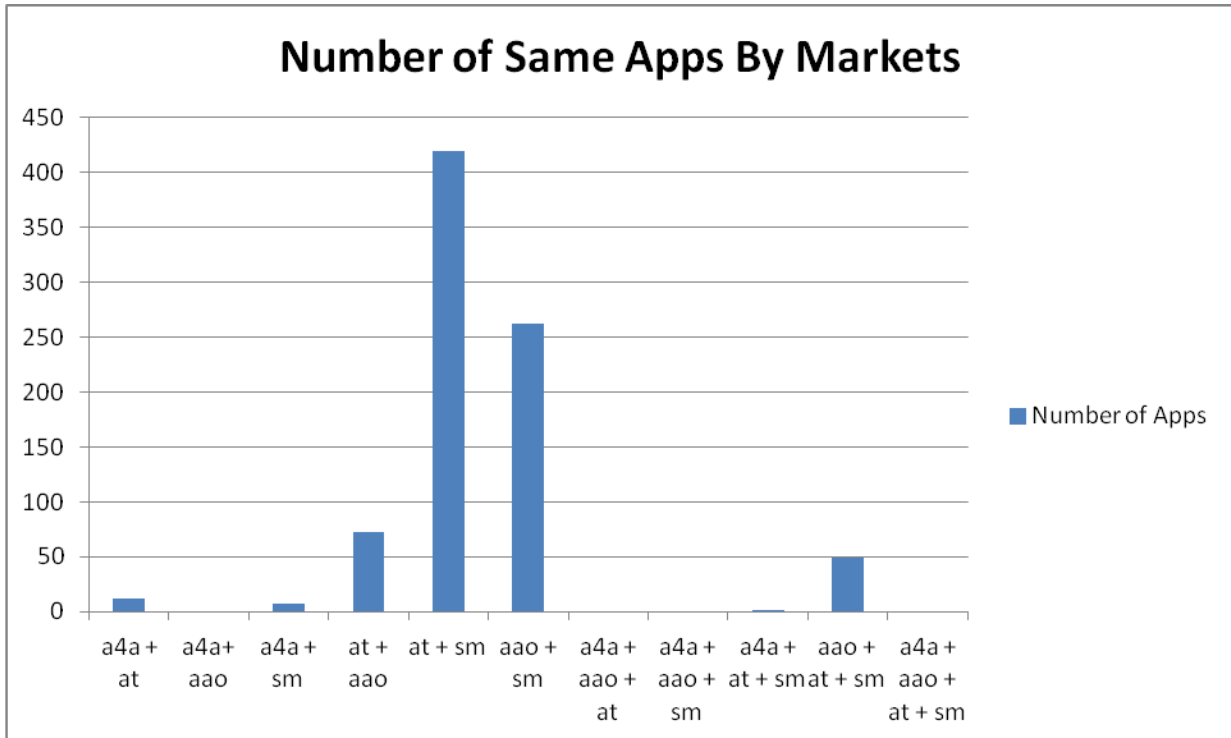


Figure 18– App Crossover By Market

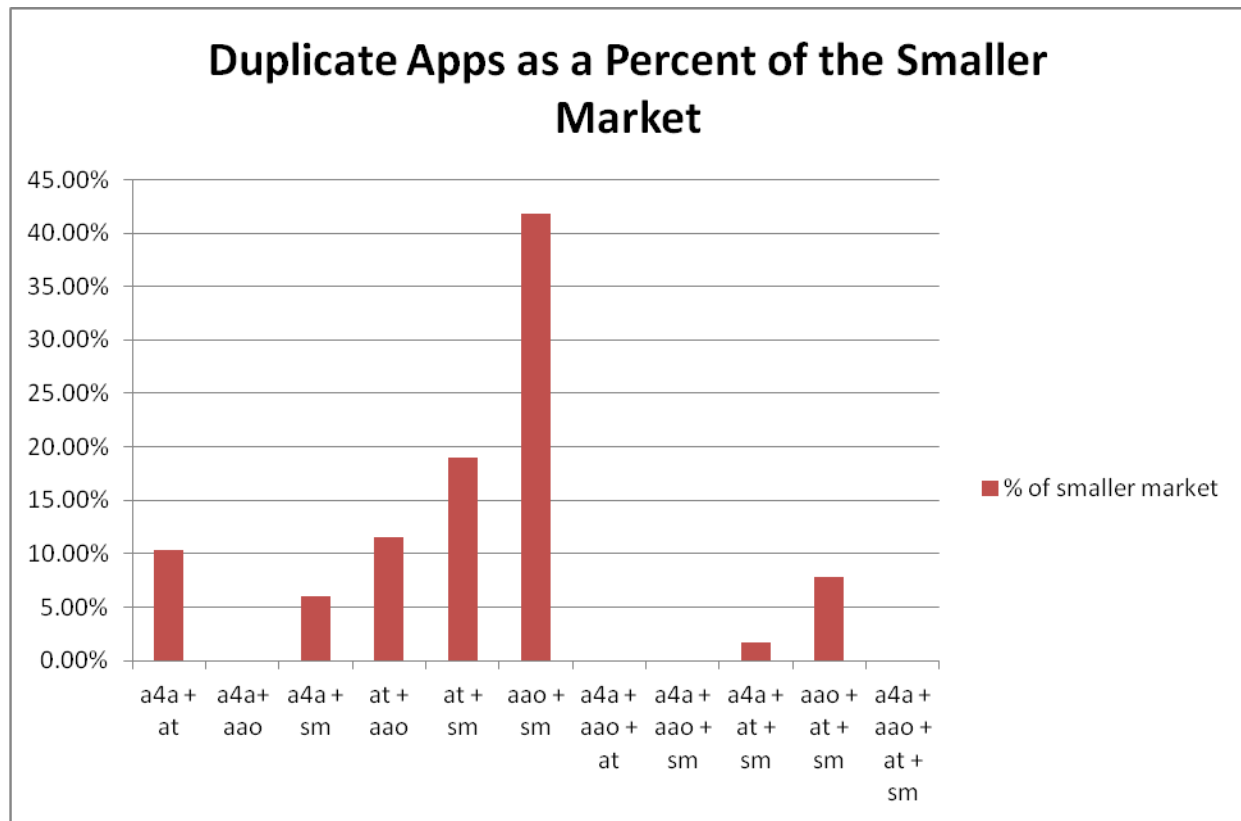


Figure 19– App Crossover as a Percentage of the Smaller Market

abbreviated in these graphs as “a4a” for Apps For Adam, “at” for App Town, “ao” for And App Online, and “sm” for Slide Me.

We then moved on to examining the permissions each app requests when being installed on a device. Each permission that an app requests must be listed within the AndroidManifest file inside of a uses-permission element. The android:name attribute of each of these elements includes a string value stating which of the predefined permissions a user must grant the app before it can be installed on the device. It is not necessary for an application to request any permissions but the majority of interesting behavior that an app can exhibit is controlled by these permissions. We found that overall approximately 91 percent of the apps we examined requested at least one permission. App Town was far lower than the other markets with only 67 percent of apps in this market requesting permissions. No significant reason could be found when examining the apps from the App Town market as to why this was the case.

Of apps from these markets that did request permissions the minimum number requested by any app in each market was one permission while the maximum ranged from 26 permission in the And App Online market to a maximum of 49 apps in the Slide Me market. The average number of permissions requested across all markets was approximately 5.5 permissions per app. The app in the Slide Me market which requested 49 permissions is a security app with various functions such as the ability to remotely lock/locate your phone, optimize the device it is installed on, monitor network traffic, backup and restore the phone, as well as providing antivirus protection. Due to the large amount of functions that this one app is trying to achieve it is not unusual that it would request so many permissions at install.

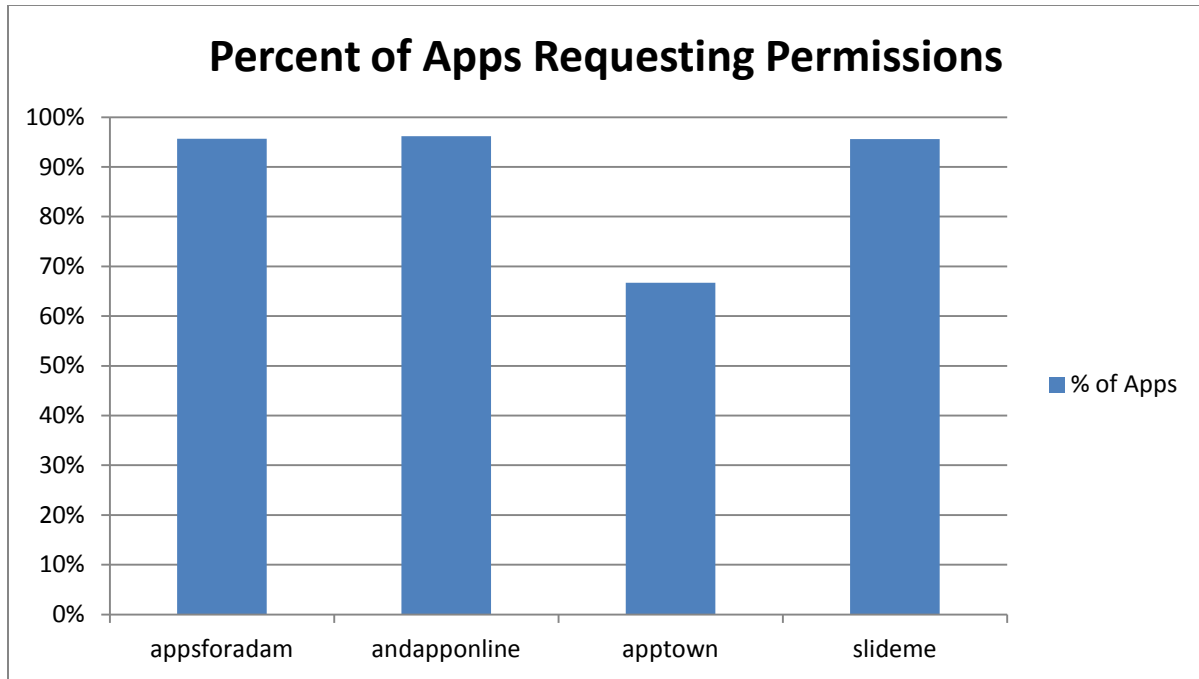


Figure 20– Percent of Each Market Requesting at Least One Permission

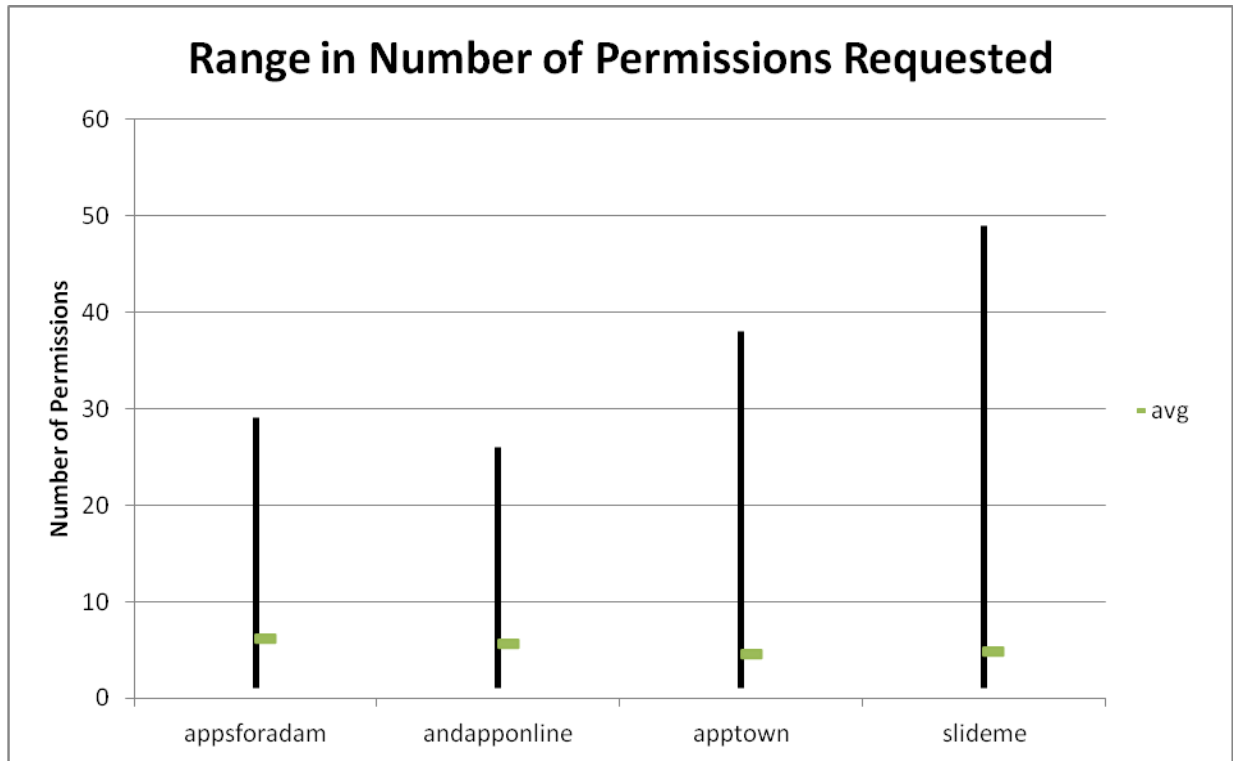


Figure 21 – Range of Permissions Requested per Market

While examining permissions we also looked at the frequency of permission utilization which led to some of the most interesting data from our study. We found that of the 124 permissions currently offered and documented by the Android operating system only 31, or 25 percent, of these permissions were utilized in more than one percent of all of the apps we examined. In addition only 9 permissions were utilized in over 10 percent of all apps. Of these top nine permissions INTERNET, the permission which allows an app to access the internet, was far ahead of all the others being found in 90 percent of all apps that we examined and 99 percent of all apps which requested at least one permission. The next closest permission was ACCESS_NETWORK_STATE at 62 percent of all apps and 68 percent of apps that requested at least one permission. It was not surprising that INTERNET was the most requested permission as one of the main advantages of smart phone platforms such as Android is their ability to integrate connectivity into many of the tasks they perform. It was surprising to find this permission being requested so much more frequently than any other permission as well as the relatively small number of permissions requested on a regular basis.

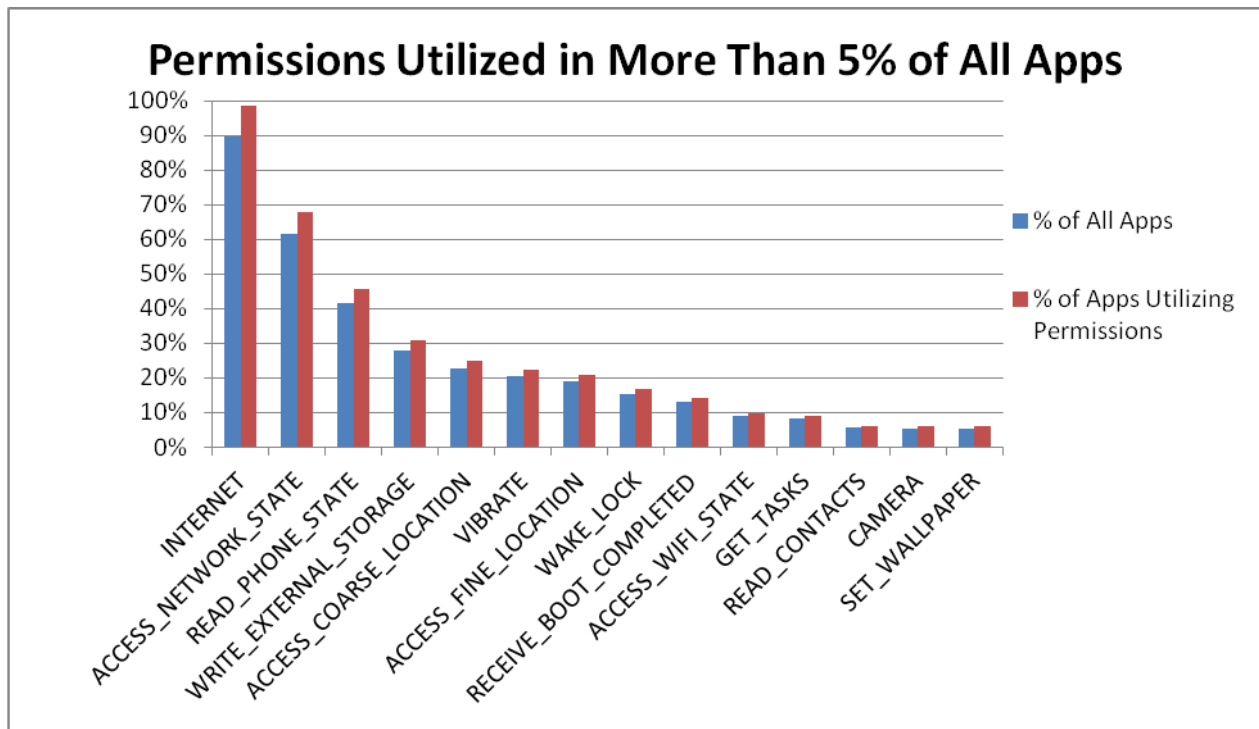


Figure 22 – Permissions Utilized in More Than 5% of all Apps Analyzed

The final piece of data we collected concerning permissions was a list of permissions that were not utilized in any of the apps we examined. We found 22 permissions meeting this criterion, listed below in Table 6. Some of these permissions such as FACTORY_TEST, DUMP, and DIAGNOSTIC are designed to be used during the design and testing phases of an app, if at all, and should never be seen in an app that has been released to the general public. Others such as WRITE_SOCIAL_STREAM were not implemented until later versions of the Android operating system, API level 15 in this case, and had not been deployed yet in any of the apps we examined. Finally, still others such as BRICK, which grants an application the ability to completely brick a phone without any interaction on the users part, are considered extremely dangerous permissions and should not be used lightly.

Unused Permissions
ADD_VOICEMAIL
BIND_REMOTE_VIEWS
BIND_TEXT_SERVICE
BIND_VPN_SERVICE
BRICK
BROADCAST_PACKAGE_REMOVED
DIAGNOSTIC
DUMP
FACTORY_TEST
FORCE_BACK
READ_HISTORY_BOOKMARKS
READ_PROFILE
READ_SOCIAL_STREAM
REBOOT
SET_ALWAYS_FINISH
SET_DEBUG_APP
SET_POINTER_SPEED
SET_PROCESS_LIMIT
SIGNAL_PERSISTENT_PROCESSES
WRITE_HISTORY_BOOKMARKS
WRITE_PROFILE
WRITE_SOCIAL_STREAM

Table 6 – Permissions Not Utilized in Any App Examined

After examining permissions we next turned our attention to the minimum SDK levels required by each app. This information is again found in the app’s AndroidManifest file. It can be found as a

property of the uses-sdk element inside of the manifest element. It does not have to be declared in which case the Android operating system assumes that the application can run on any version of Android. If an Android device's SDK level is lower than the SDK level defined by the developer in this attribute the operating system will prevent the end user from installing the app on their mobile device. From our finding SDK levels 3 and 4 were the most popular minimum SDK levels across all markets. All SDK levels above level 8 were requested at a rate of less than 0.2 percent. SDK levels 3 and 4 correspond to the earliest versions of Android released to the public, 1.5 or cupcake and 1.6 or donut respectfully. From our previous research we expected the SDK levels of 9 and 10 to be higher than they were since these are the SDK levels which correspond to the Android 2.3.x or gingerbread releases. The data we gathered suggests that either the apps we were examining were dated slightly or that the features added that require these SDK levels to operate without crashing have yet to be utilized on a large scale basis by developers.

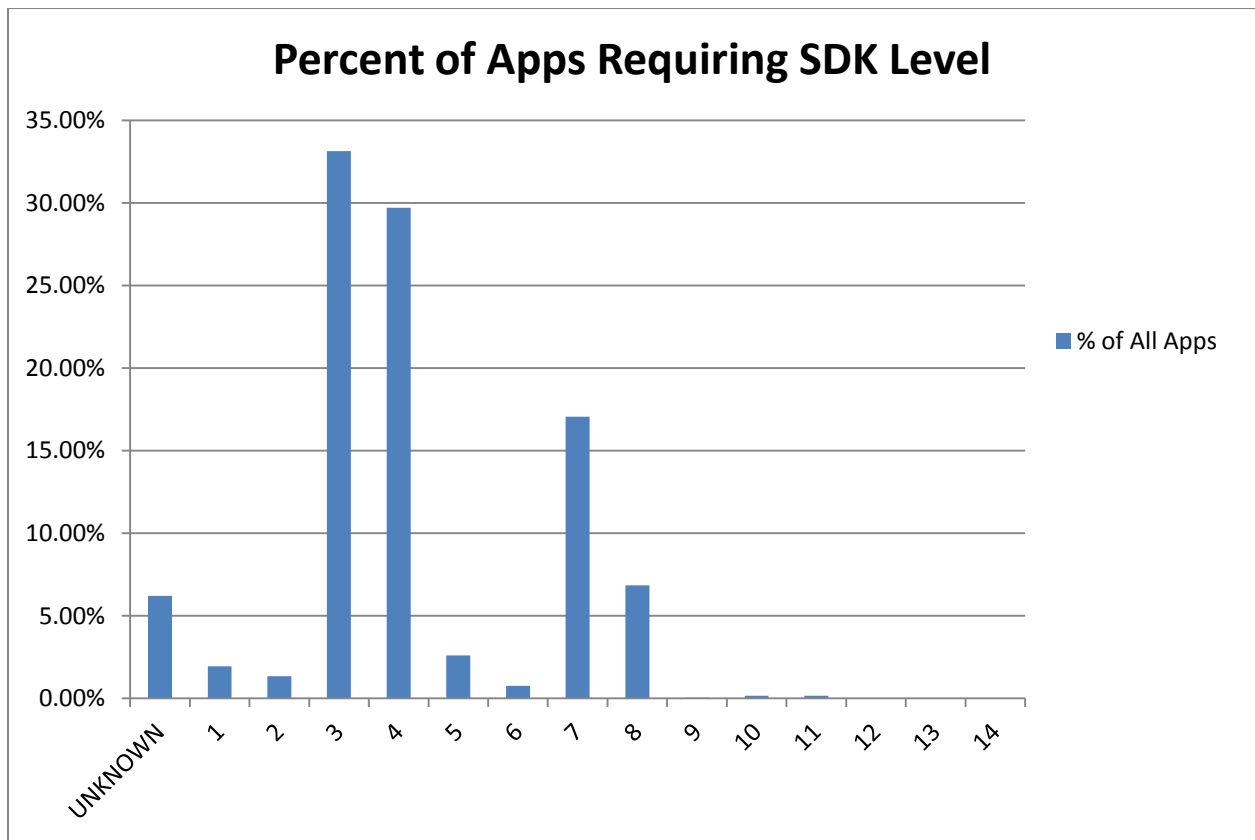


Figure 23 – Percentage of Apps Examined Requiring Minimum SDK Levels

Next we began to look at the external libraries that an application must be linked against. These are libraries whose code must be included in the class loader for the package but do not include external third-party libraries which have been defined by a developer for specific use in one project. Android includes several libraries such as `android.view` and `android.widget` in the default library that all apps are automatically linked against and therefore they do not have to be listed here. This information is found within the `uses-library` element of the `AndroidManifest` file. The specific library which is being linked can be found by examining the `android:name` attribute of this element. This element can also prevent an app from being installed on a device. In the `android:required` attribute of this element is declared as `true` yet the external library is not installed on the user's device the Android operating system will not allow the app to be installed.

We found that only 4 percent of all of the apps we examined requested to use an external library in this fashion. No market we examined had more than 5.5 percent of apps that utilized this feature while Apps For Adam was the lowest with only approximately 2.5 percent of all apps in this market requesting an external library. Of the apps that utilized this feature, 95 percent did so in order to make use of the Android Google maps library, which is not included in the general library on Android devices. In fact, only 9 external libraries were seen over all the markets we examined. Of the 5 percent left after accounting for the Google maps library the `android.test.runner` library accounted for another 3 percent with the remaining 2 percent being split between 7 different libraries. It is important to remember that this data does not suggest that only 4 percent of Android apps are making use of libraries since these numbers do not include the general Android libraries or libraries custom defined by the developer of an app.

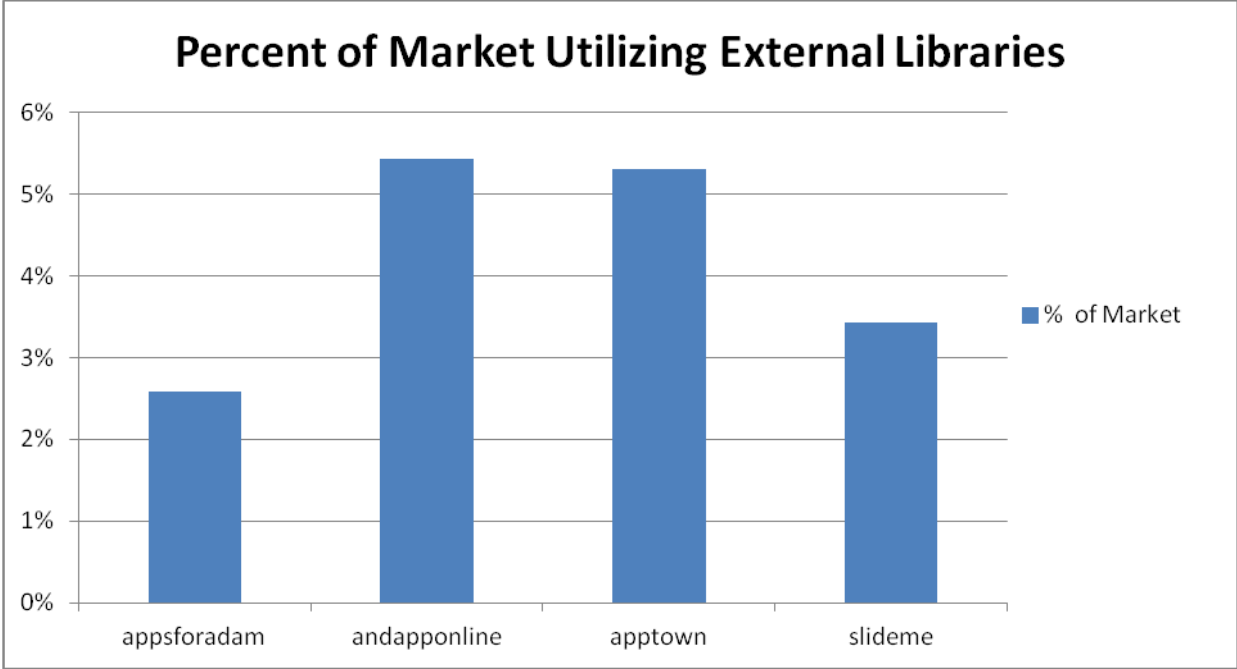


Figure 24 – Percent of Each Market Utilizing External Libraries

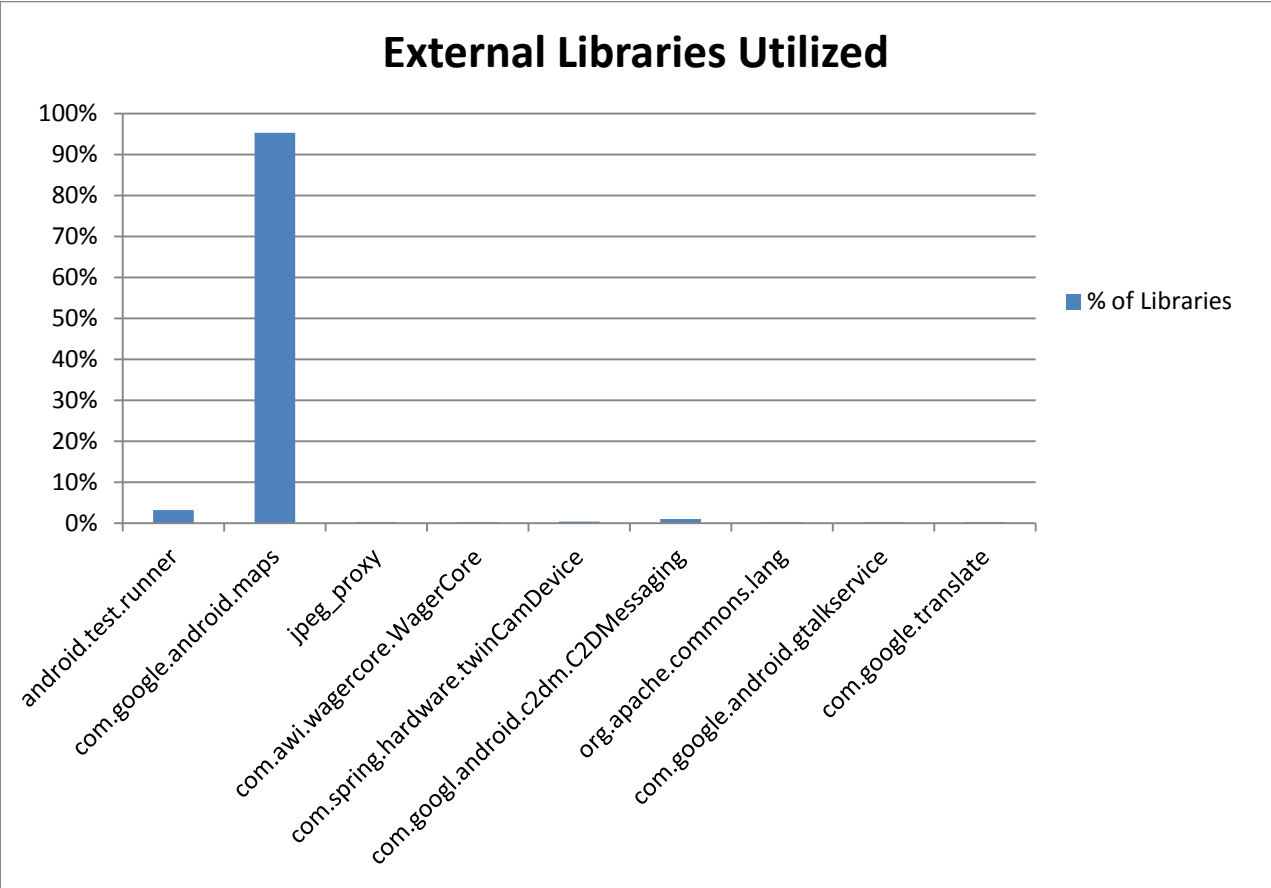


Figure 25 – Libraries Used as a Percent of All External Libraries Utilized

Each Android app also has the ability to set up services, providers, and receivers which each all the app to exhibit additional functionality in a specific way. Services are programs that run in the background of an application and do not normally present a interface to the user. Services can be started by any component of a given application and in some cases can even be started from an application outside the one in which they reside. These services typically continue to run even after a user exits the application that started the service until they are explicitly stopped. A component of an app can also bind itself to an established service allowing interaction with the service. This is one way in which the Android operating system allows interprocess communication to take place. Services must be declared in the AndroidManifest file of an app. The service element is contained within the application element of this file and a new service element must be defined for each service that the app uses. We found that for the body of apps we studied if the services feature was utilized an average of 1.4 services were declared per app. The most services we found declared for one app was 24 in a notification app found in the And App Online market.

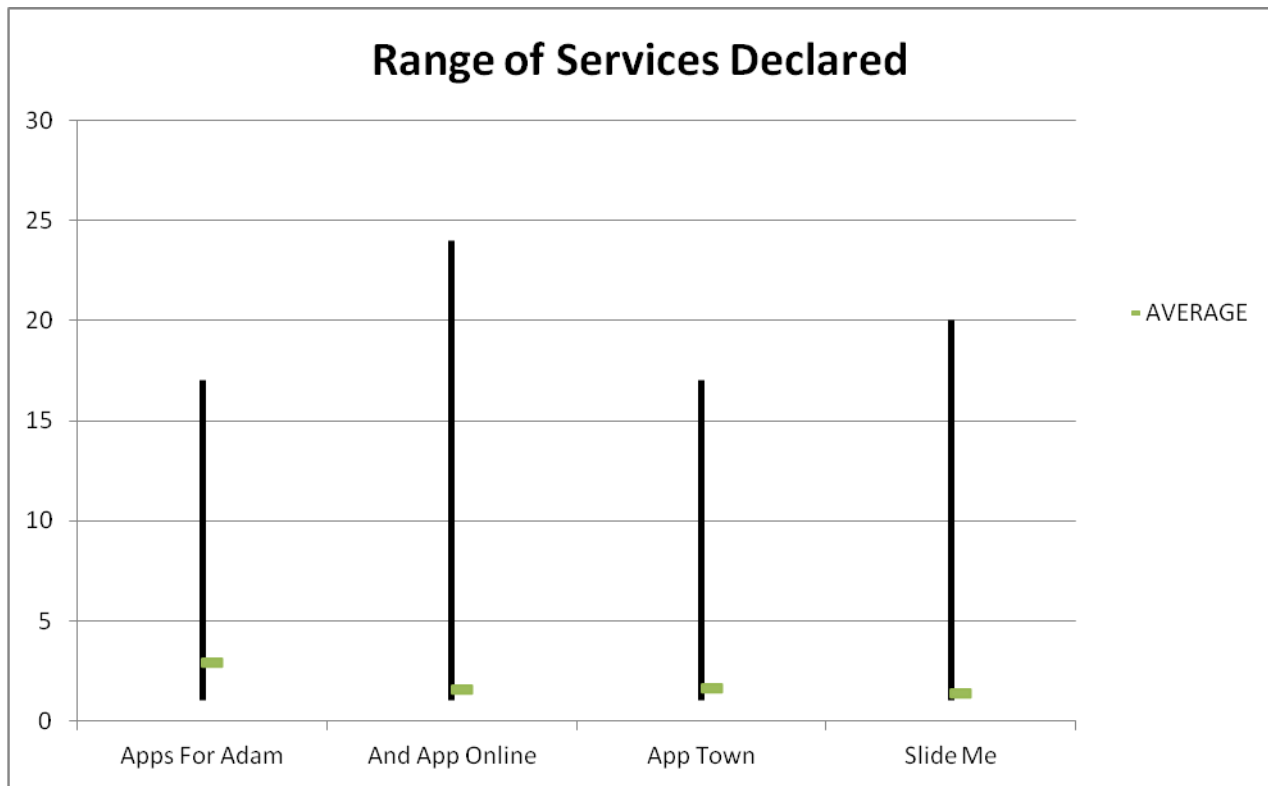


Figure 26 – Range of Services Declared Per Market

Apps can also establish providers which function similar to services. Providers establish an interface for accessing structured data found within one app from another app and are the main means for accessing data across apps. Providers are not required to be present in an app and in fact if the app in question has no data that needs to be accessed from an external source then it should not establish a content provider. As with services, each provider must be declared separately in the AndroidManifest file and all information about a provider is encapsulated in a provider element inside of the application element of this file. We found if an app established a provider they established 1.5 of them on average. The highest number of providers established was much lower than with services. The maximum number of providers we found in one app was 11 and was found in an app from the Slide Me Market. This app was a news app and designed to share its data with many other apps so this number of providers was not unusual.

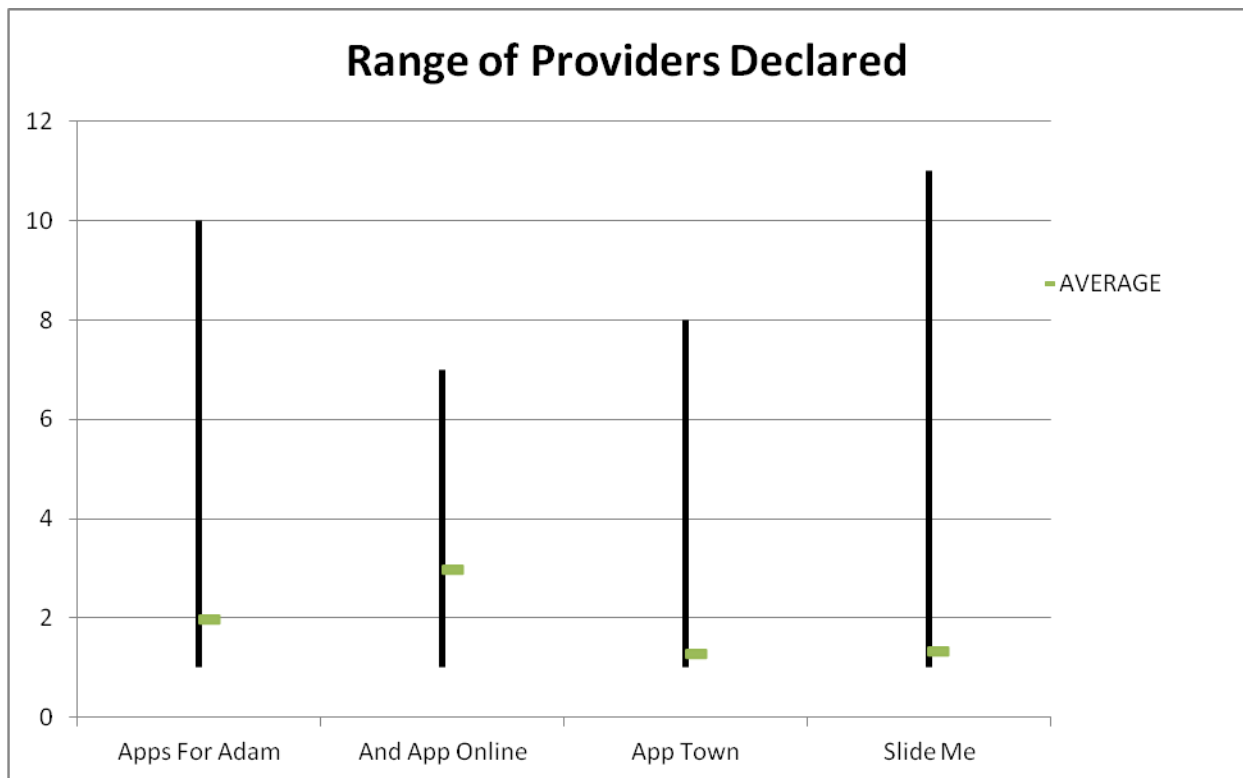


Figure 27 – Range of Providers Declared Per Market

Receivers, also known as broadcast receivers, allow an app to receive intents broadcast by another app or the Android operating system itself. These receivers are similar to services in the fact that

they function even with the app is not active but in the case of receivers the receiving app can actually take an action on this information instead of just running idly in the background. While the Android development guide suggests that all receivers be declared in the AndroidManifest file they do not force the developer to do this. Receivers can also be declared dynamically within the source code of the application itself. If an app utilized receivers, they utilized more of them on average than either services or providers with an average of 2.2 receivers per app. For our research, we only examined receivers declared in the manifest file. These receivers can be found in the receiver element within the application element in the AndroidManifest file. Again, as with services and providers, a separate receiver element must be declared for each receiver utilized by the app. The highest number of receivers we found in an app was 32 in the Axe Googly app from the App Town market. This app is a compilation app which allows a user to perform many different tasks such as getting on Facebook, checking cricket scores, and chatting with friends in one app. It appears that instead of implementing each of these features itself this app relies on other underlying applications to achieve many of these tasks. This would explain why the number of receivers for this app is so high.

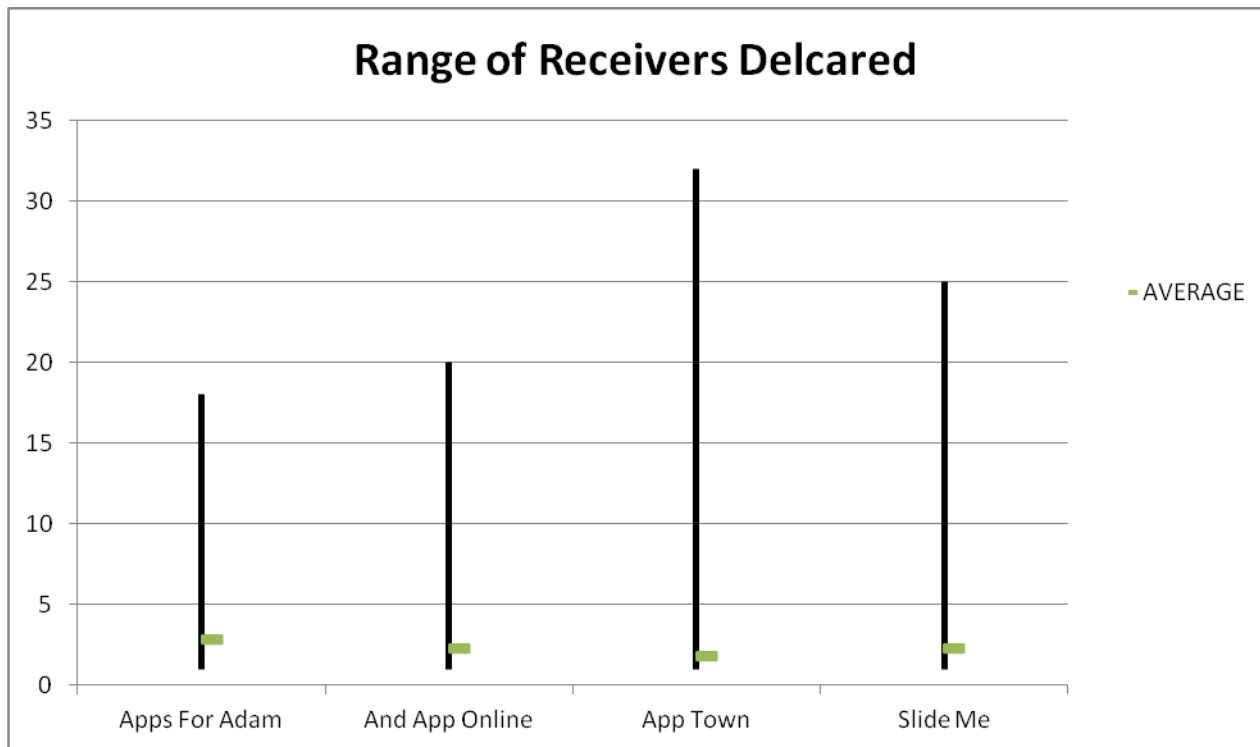


Figure 28 – Range of Receivers Declared Per Market

Of these three features of Android apps we found that receivers were the most frequently utilized, being found in 32 percent of all of the apps we examined. Services were the next most popular with 26 percent of all apps declaring at least one service. Providers were the lowest on the three with only 6 percent of apps supplying a provider for information within the app. Providers ranking the lowest of these three made sense as most applications do not need to share information with other apps due to the fact that they are self-contained entities. From our personal experience we found the amount of apps utilizing services to be reasonable as many apps like to continue to run in the background once the interface to the app has been dismissed. The fact that receivers ranked highest of the three did not surprise us as well since the apps we examined are designed to run on mobile devices on which they often need to respond to some event such as an incoming call, the user entering or leaving a location, or a new text message being received.

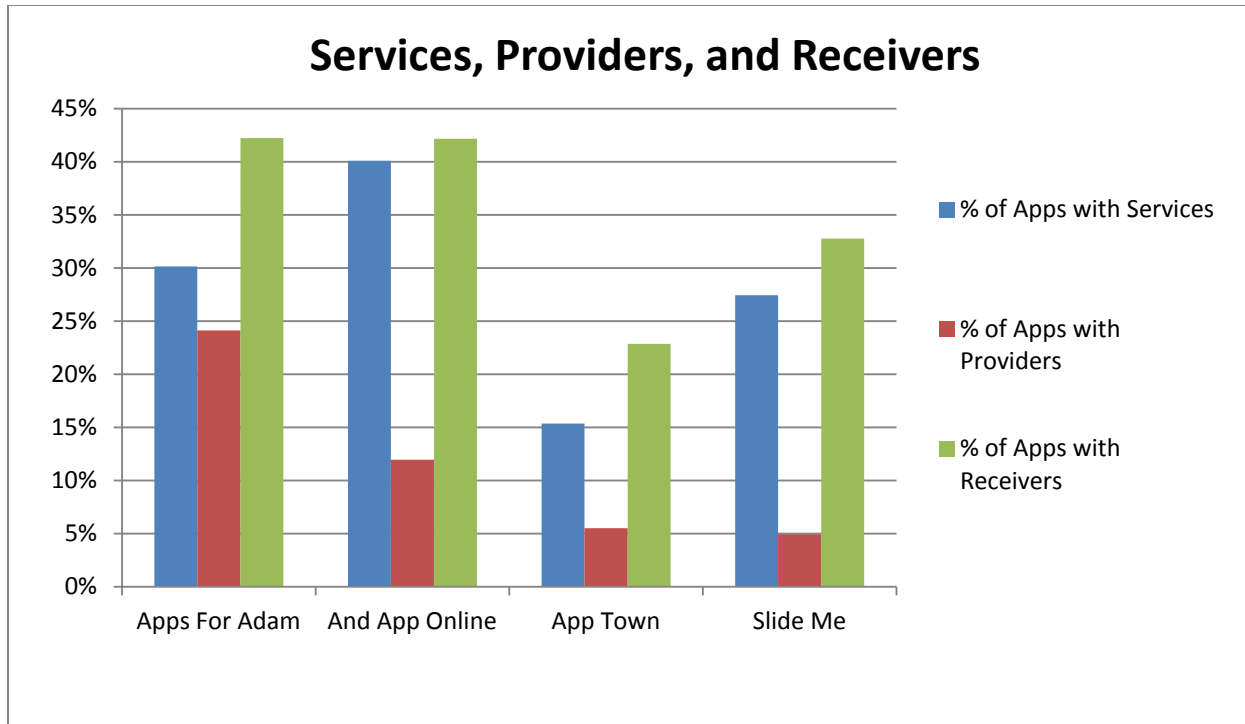


Figure 29 – Percent of Services, Providers, and Receivers by Market

In addition to services, providers, and receivers Android apps also implement another component known as an activity. Activities are the components of an Android app which provide a user with an interface which they can interact with. A typical app will consist of at least several of these activities that are loosely coupled to each other. For example a messaging app might consist of 3 of these activities: one for the home screen displaying all of the current conversations, one for the settings screen allowing users to change the settings of the app, and one for an active conversation which allows a user to view and reply to messages from another party. Only one activity for an app is ever active at one time. All activities than an app utilizes must be declared in the app's AndroidManifest file. The information for these activities can be found in the activity element within the application element of the manifest file. We found that on average each app we examined contained just over 6 activities. Since these essentially correspond to the different views presented to the user this data seemed reasonable. The most activities we found in one app was 113 from the NetQin security app found in the Slide Me market. As mentioned

before, this app is a compilation of many different features which therefore explains the high number of activities it contains.

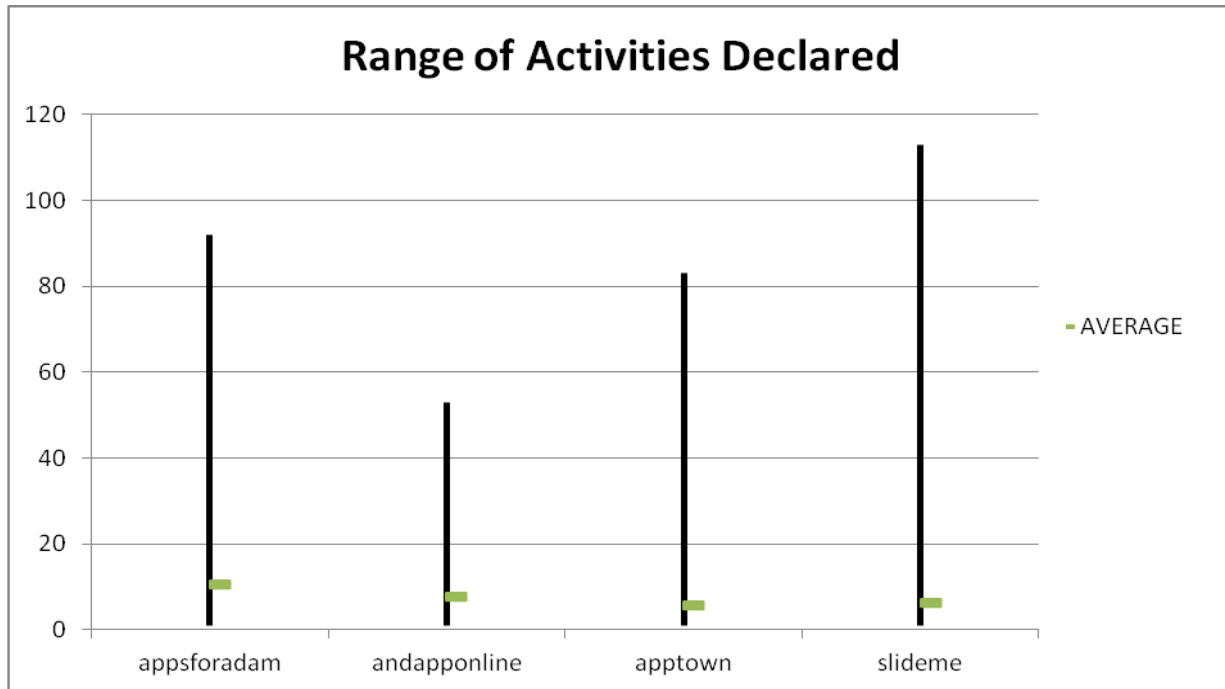


Figure 30 – Range of Number of Activities Declared Per Market

The Android operating system also provides a mechanism to allow users to localize the data in the apps based on different parameters. Depending on the developer's needs three different parts of the app can be localized: the strings used in the app, the layouts used in the app, and the images used. We next focused our attention on determining what percentage of the apps we examined were taking advantage of each of these three localization schemes and also in detecting which localization options were most popular. All localization data was gathered from the file structure of the decompiled apk files. Android uses a naming scheme within its directory structure to facilitate the use of localization. Inside of the res directory, which contains all of the resources for the app, are directories labeled drawable, layout, and values containing the drawable items, xml documents describing layouts used in the app, and string values used in the app respectively. When localization is utilized additional directories are added to the res directory. These directories have the same base name as these three default directories but use additional qualifiers to describe when the items in these directories are used. For example layout-land

would contain layouts which are utilized when the device is in landscape mode while values-zh would contain alternative string values to be used when Chinese was selected as the default language on the device. Once these directories have been set up in this manner the Android operating system handles all of the overhead of selecting which values to use. This is accomplished by following a simple hierarchy scheme checking for the most specific values possible based on the current parameters of the device and then working backwards until an appropriate set of values have been found or the default values for the application are reached.

The first of these localizations that we examined was the values directory which provides the ability to localize strings found in the app. This is useful in an app because it allows a developer to switch all or some of the strings in their app depending on the default language selected on the device on which the app is installed. This provides an easy way for a developer to set up a multilingual app. We found that overall approximately 21 percent of the apps we examined utilized some form of string localization. Of all of the markets we examined, Apps For Adam utilized string localization the most with nearly 35 percent of all apps in this market taking advantage of this feature. If string localization was utilized, we found that the app was localized for 5 different languages on average. The largest number of string localizations we saw in one app was 59 found in a music app in the Slide Me market. Upon closer examination this app used some of the more advanced abilities of the string localization scheme. This developer only localized for 20 languages but they also had different string values for some strings in each of these languages based on if the on screen keyboard was displayed in the app or not.

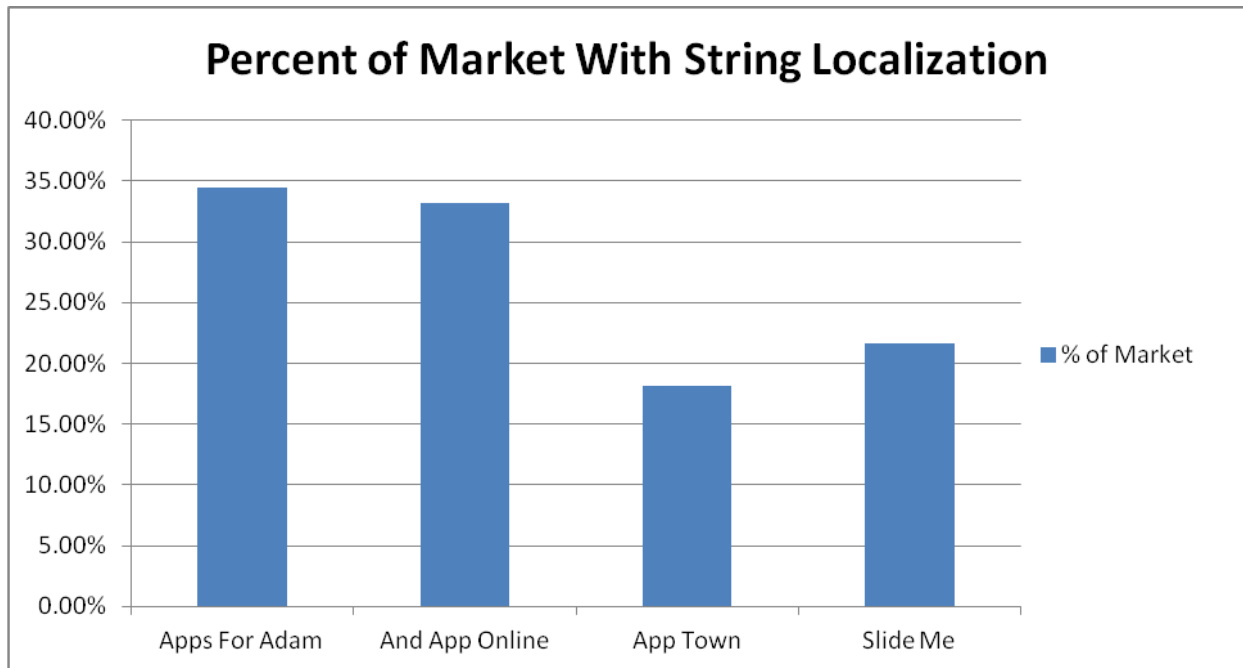


Figure 31 – Percent of Each Market Utilizing String Localization

In addition we also examined which languages or features the strings were localized for most frequently in all of the apps we examined. We discovered that only 19 different localizations were utilized in more than 1 percent of our body of apps. For this data, only apps that utilized at least one string localization were included. All of these 19 localizations were language localizations. Other localizations for string values did occur such as localizing for the device’s orientation or for the visibility of the onscreen keyboard but each of these were seen only a limited number of apps in our survey. Of the languages used, Chinese was by far the most popular being found in over 19 percent of all apps utilizing string localizations if all dialects were combined. Interestingly we found that English was only localized for in approximately 1.5 percent of these apps. When looking into this in more detail we found that English was normally the default language for an app and therefore developers were not providing localization data specifically for it.

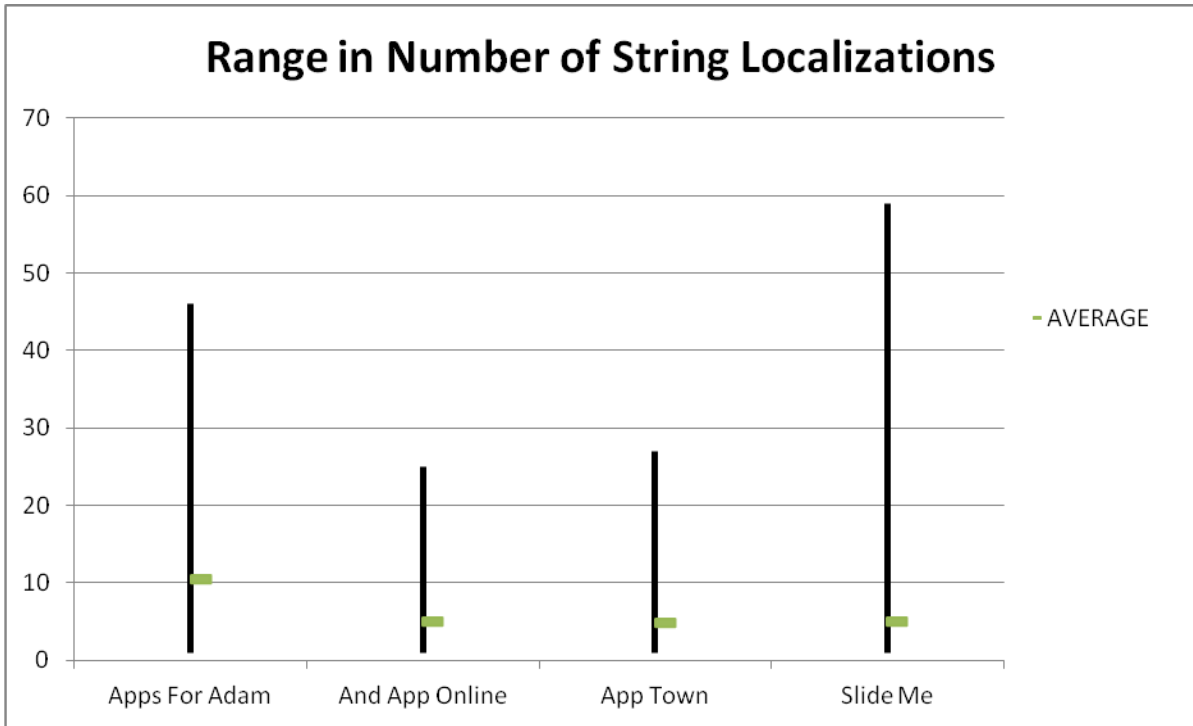


Figure 32 – Range of String Localizations Per Market

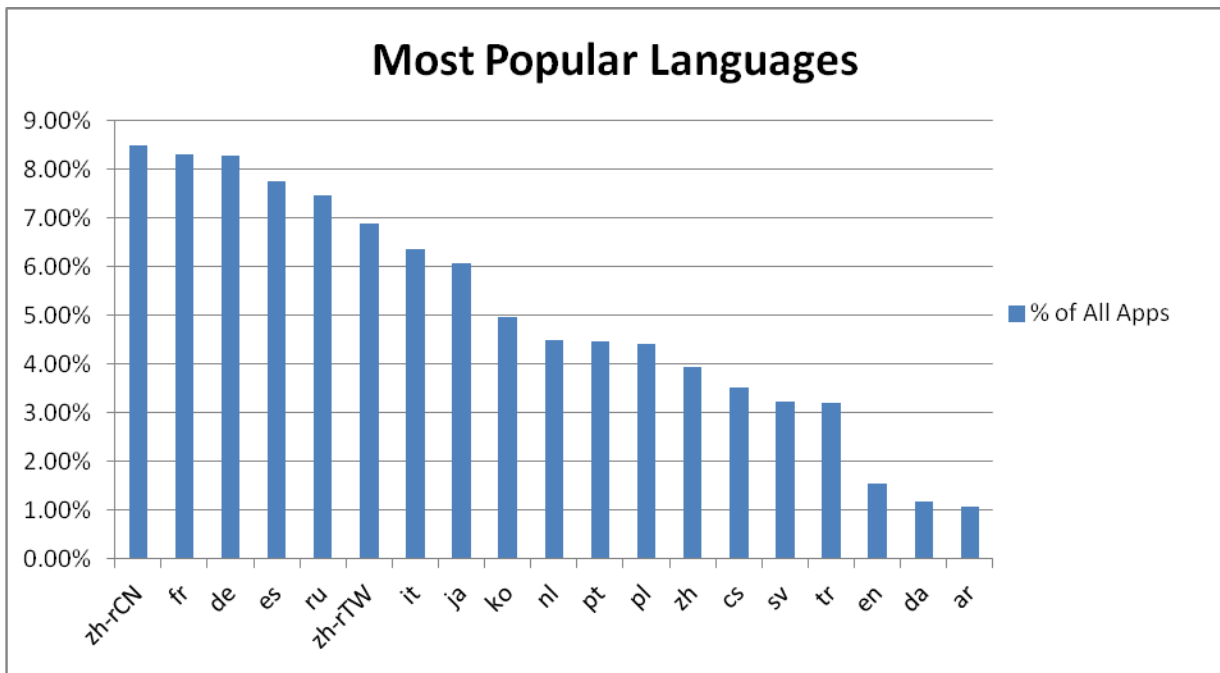


Figure 33 – Most Popular String Localizations

Language Abbreviations	
zh-rCN	Chinese-China
fr	French
de	German
es	Spanish
ru	Russian
zh-rTW	Chinese-Twain
it	Italian
ja	Japanese
ko	Korean
nl	Dutch
pt	Portuguese
pl	Polish
zh	Chinese
cs	Czech
sv	Swedish
tr	Turkish
en	English
da	Danish
ar	Arabic

Table 7 – Android Localization Language Abbreviations

Layout localization information is found in the layout directory of the AndroidManifest file. Layouts can be localized for a variety of reasons but the most common reason is to display data in different ways depending on the orientation target device. For example the layout in which the app is displayed might change from a very small screen to a large screen. We found that on average 19 percent of all apps we examined used alternative layouts to account for differences of this nature. Of the reasons for using alternative layouts account for a device being held in a landscape position as opposed to a portrait position far outweighed the others being used over four times more than its nearest competitor. If an app utilized layout localizations they did so at a lower rate than with string localizations with an average of 1.7 layout localizations per app and a maximum of 12. These lower numbers were to be expected because there are fewer common values for the attributes which layouts are localized on.

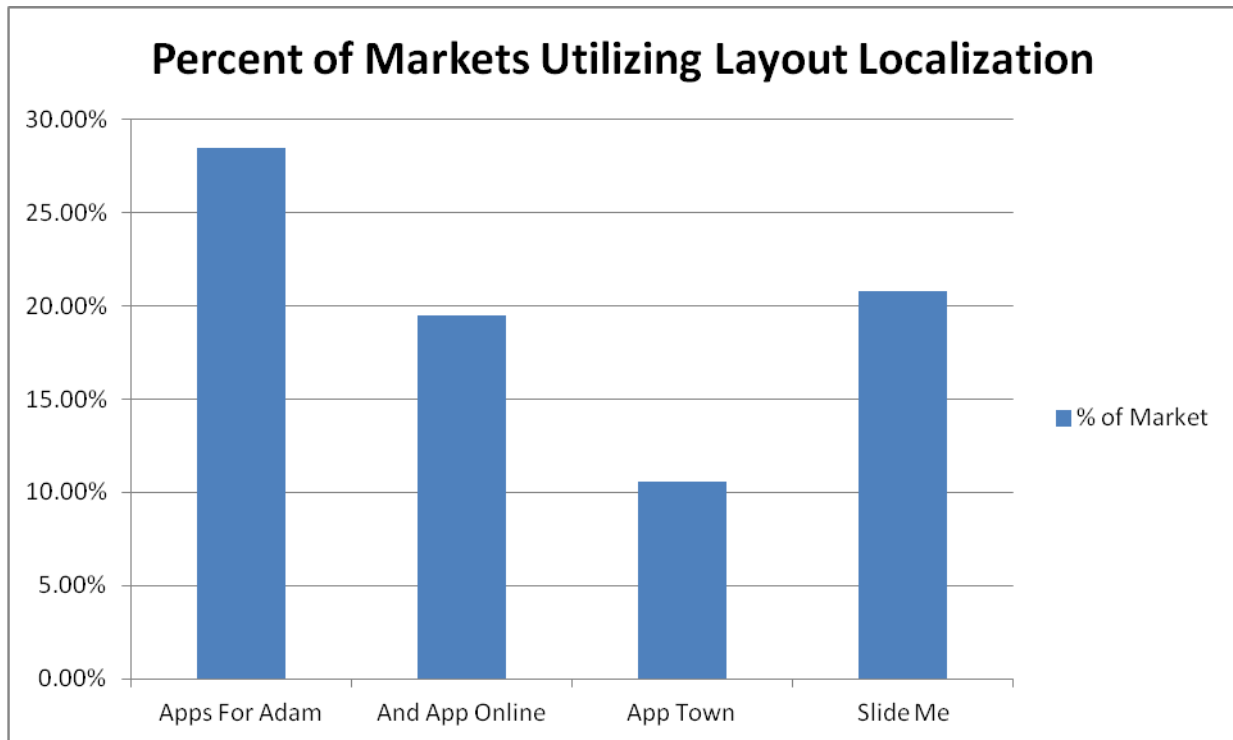


Figure 34 – Percent of Each Market Utilizing Layout Localization

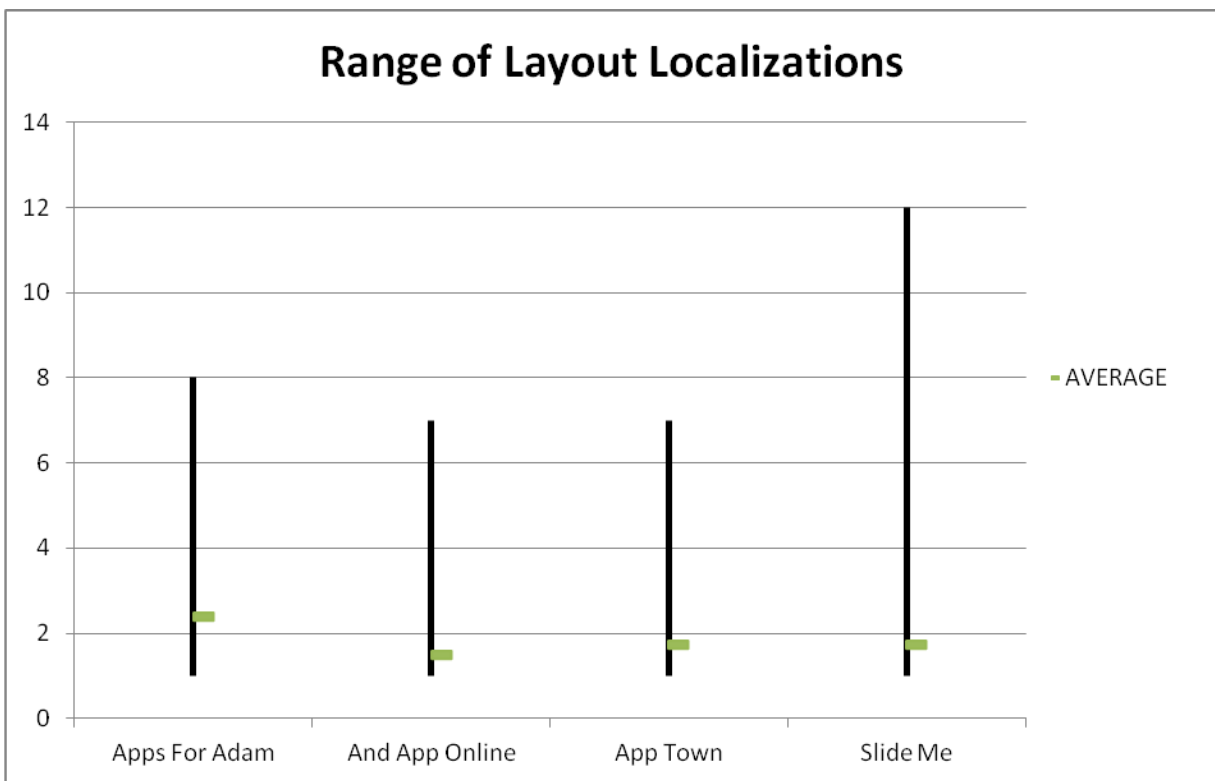


Figure 35 – Range of Layout Localizations Utilized Per Market

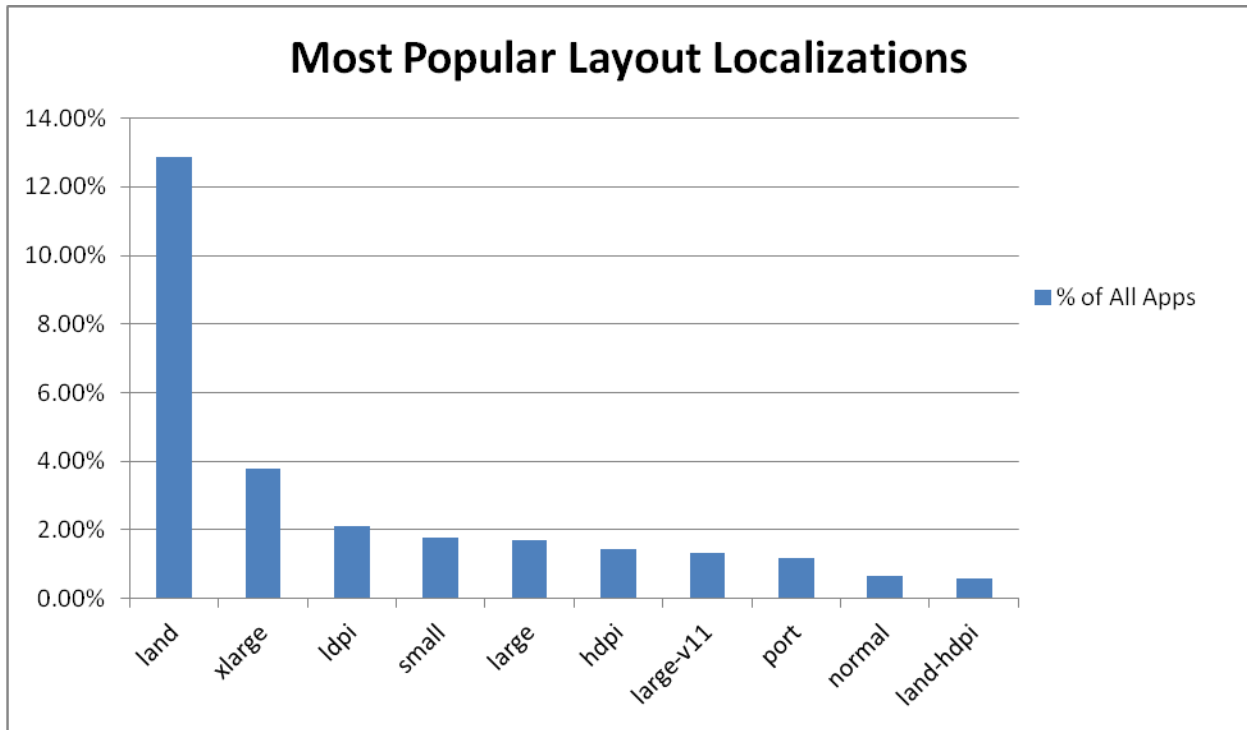


Figure 36 – Most Popular Layout Localizations

The last type of localization information we examined was the localization of images used in each app. These images are found in the drawable directory of the AndroidManifest file. The most common reason for a developer to localize images in their app is to take advantage of different screen resolutions, for instance using a higher resolution image only on a screen that can actually support it and display it correctly. We found that of all the apps we surveyed nearly 60 percent of them took advantage of drawable localizations in some manner. This percentage is high when compared to the other localization techniques because since version 1.6, commonly known as Donut, the Android SDK has automatically set up three localizations for the drawable images: hdpi, mdpi, and ldpi. These localizations correspond to the amount of dots per inch the screen of the device features. Due to this fact, these three localizations were also the top three most popular localizations for the drawable assets all being found in approximately 50 percent of all apps that localized the drawable assets in any way. The next most popular localization was xhdpi, the localization designated for screens with extra high dots per inch, but was only found in 4 percent of apps. As to be expected, if an app utilized any drawable localizations at all they averaged around 3 localizations per app. The most drawable localizations that we saw in any app

was 29 and belonged to an app in the App Town market which displayed the Bible in several languages. When we investigated this in more detailed it was determined that some of the text being used in this app was actually images. Therefore the developer had localized these images by language as well as dots per inch of the device screen resulting in the higher than usual number of localizations.

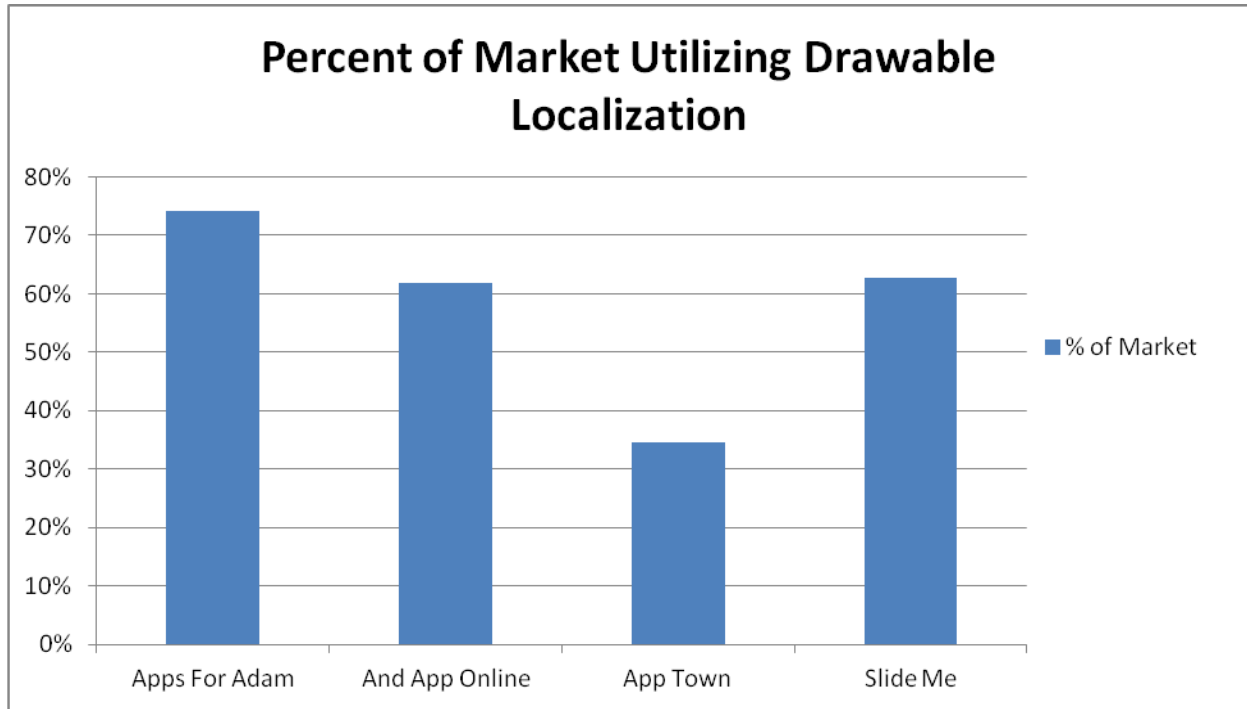


Figure 37 – Percentage of Markets Utilizing Drawable Localizations

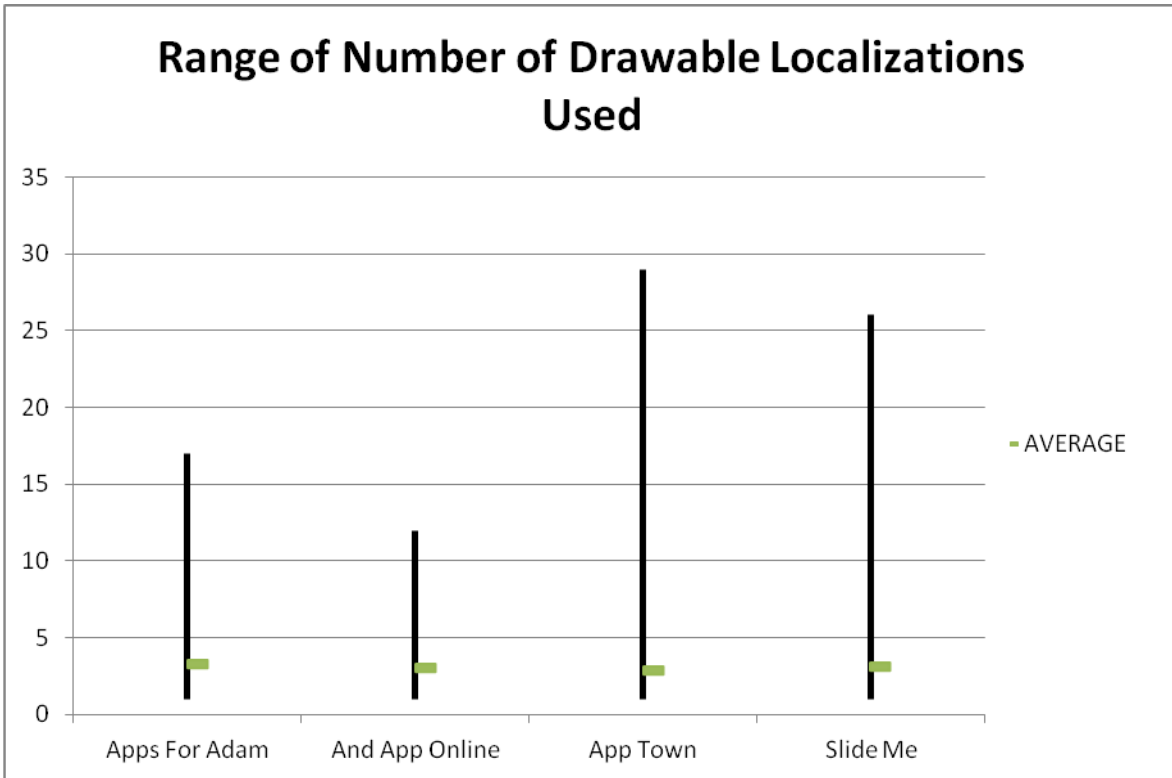


Figure 38 – Range in Number of Drawable Localizations Per Market

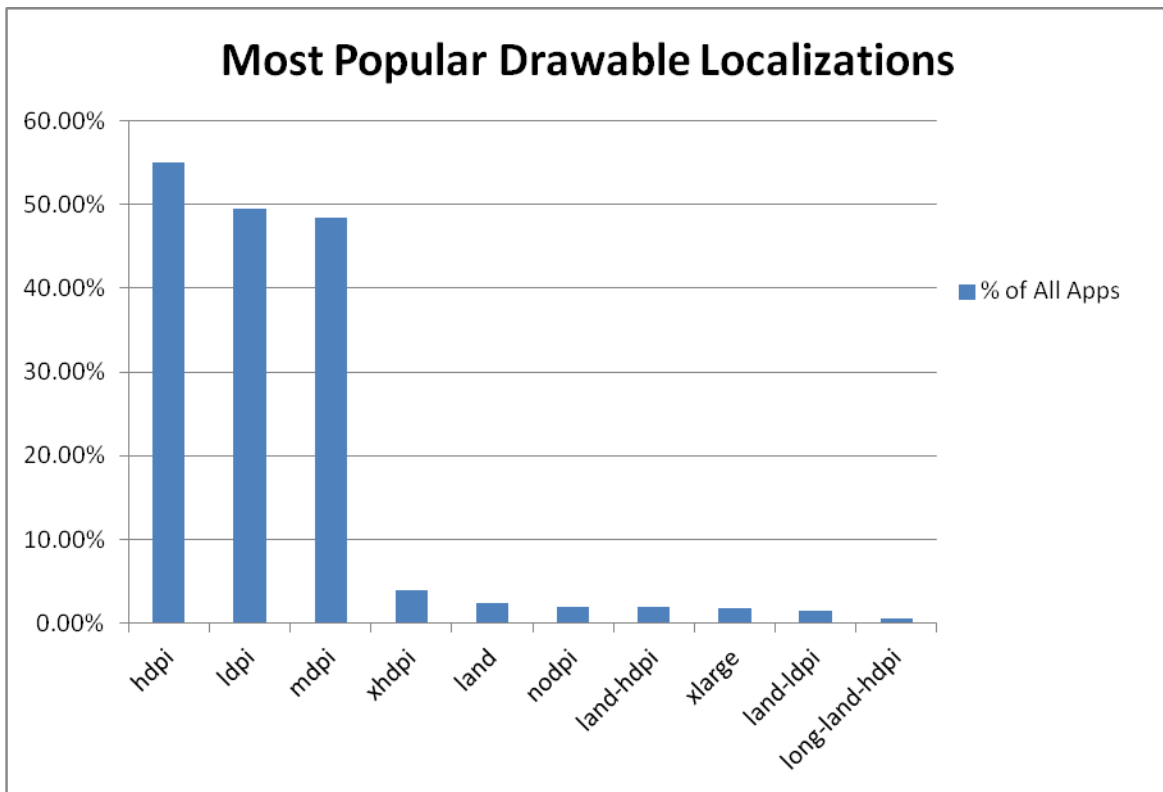


Figure 39 – Most Popular Drawable Localizations Used

The last characteristic of our body of apps that we examined were the advertising schemes which they utilized. When a developer decides to monetize their app they have two main options. First, they can charge a fee when the user downloads the app from a market for the first time providing the developer with a onetime instantaneous income. The second option they have is to provide the app for free and then include advertising in the app to provide income a source of income. This process usually provides a lower income up front, but has the potential to meet or even surpass charging a onetime fee in the long run. While advertising schemes of this nature are normally found in apps that are distributed for free, there is no reason that advertising could not be included in a paid app as well.

These advertising schemes are normally managed by third parties. The most common way in which they are implemented is through the use of libraries which are included in a developer's project. Normally, once these libraries have been set up a developer receives a unique identifier which is linked to their app as well as their account with the advertising company. The developer then includes a specific activity within their program which sets up and displays ads. The sophistication of these advertising schemes vary from storing a few default ads within the app itself to tracking a user's activity on a device and presenting ads targeted directly to them. When a user activates an ad, an acknowledgement of some sort is sent to the advertising firm containing the developer's unique identification as well as information about the app in which the ad was placed. This allows the developer to get paid for the ad and the advertising firms to study which ads are most effective in what type of apps.

Due to the way developers must implement these advertising schemes in their apps most of these schemes leave behind a unique fingerprint within the AndroidManifest file. For our research, we examined four advertising schemes of this nature: AdMob, Google's advertising platform, mMedia, airPush, and InMobi. It is important to note that all of the apps we acquired for this study were free and therefore the statistics gathered here would most likely not map directly to a body of apps which included both free and paid apps. We found that just over 43 percent of all of the apps we examined used at least one of these four advertising schemes. The Slide Me market contained the most app which had been monetized with 49 percent of the apps in the market containing advertising. Google's advertising

platform, AdMob, was by far the most popular of the four platforms we studied being found in over 35 percent of all apps in our study with the next closest being mMedia with 7 percent. Interestingly, we also found that 8 percent of all apps in our study utilized more than one of these four advertising schemes. Again the Slide Me market was had the most apps with more than one advertiser with just under 10 percent utilizing at least two of the four schemes we studied.

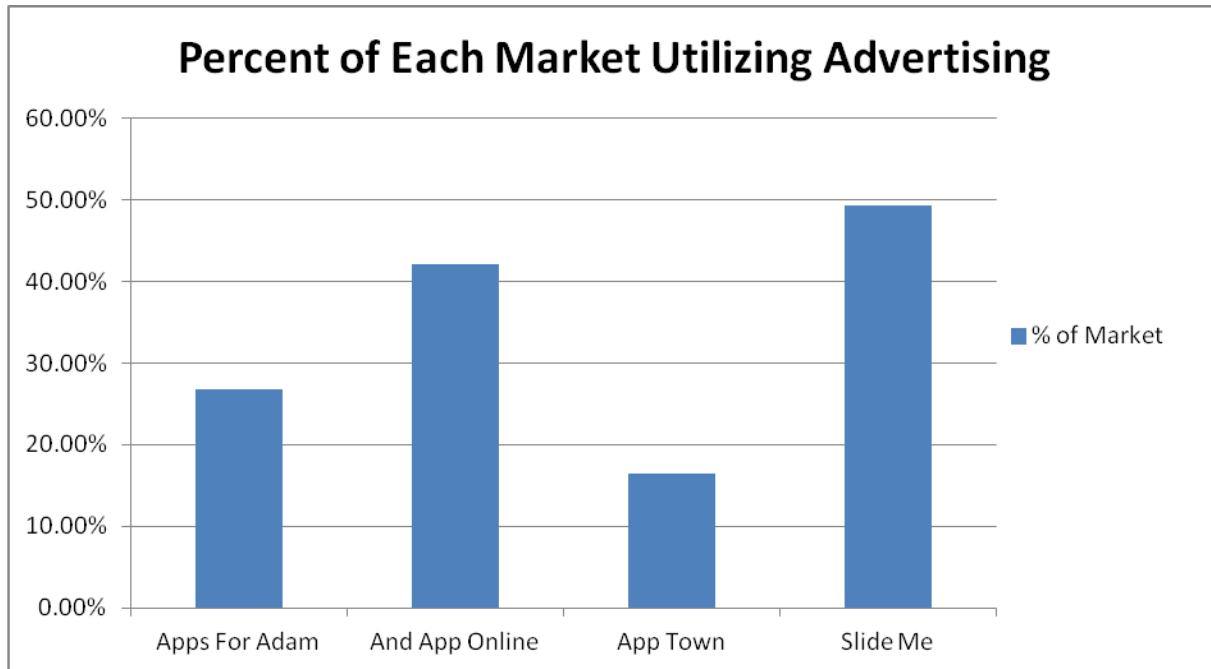


Figure 40 – Percent of Each Market Utilizing Advertising

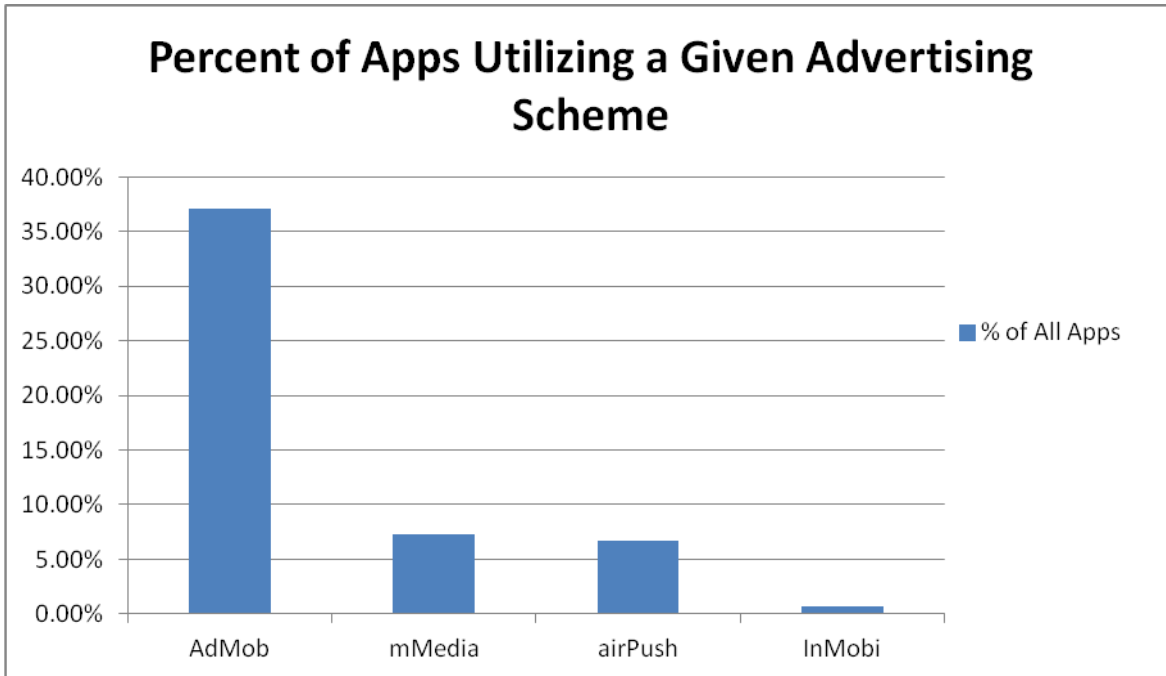


Figure 41 – Percent of Apps Utilizing Each Marketing Scheme Studied

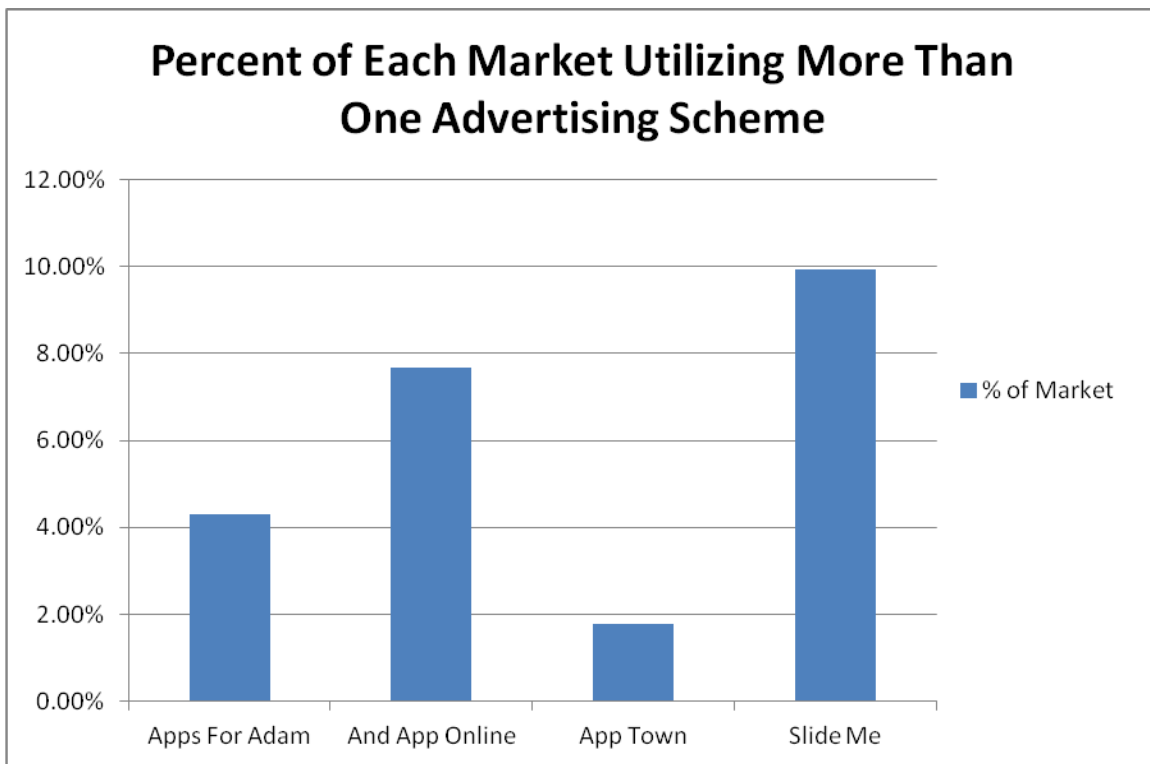


Figure 42 – Percent of Each Market Utilizing More Than One Marketing Scheme

Chapter 4 – Solution Validation

In completing this research we had two main aspects of our work needed to be validated. First, the process we used to reverse engineer our Android apps needed to be validated to ensure that it was operating correctly and that the artifacts produced as a result of this process were consistent with the original source code of the app. This was accomplished by testing this process before we applied it to our body of collected apps as well as implementing an error logging feature to be used when this processes was executed on our app collection. These logs allowed for any problems encountered to be investigated in detail at a later time. Secondly, we needed to validate the demographic information we collected from our body of apps to ensure that the results we found were reasonable. Three different techniques were employed to accomplish this goal, all of them involving comparing our results to other sources of data. We first compared the results of our study to other research efforts in related areas. Next, we compared the values of our data to standards published by Android to ensure that the two complimented one another. Finally, we were able to compare the data we collected from one market to the data we collected from another market to ensure that no major discrepancies existed that could not be logically explained. Through employing these techniques we were able to ensure that our data was valid allowing us to meet our goals of collecting and compiling demographic information for an Android market.

To validate the reverse engineering process used in our research the source code for three separate apps were either written or obtained. These apps were then compiled into .apk files and then installed and tested on an Android device. Once we were sure that these were valid apps and functioning as we intended these apps were then reverse engineered using a process identical to the one used in our study. We then compared the artifacts produced from this reverse engineering process to the original source code for the apps. Since our work focused on the AndroidManifest file and the internal directory structure of the resource directory only these artifacts were compared to the original files. In addition, since the reverse engineering process we used produced Smali code instead of Java code we knew that the source code of the original and decompiled files would be different and therefore there was no need to compare them. In all three cases these two artifacts, the reverse engineered AndroidManifest file and res

directory, were identical to their original counterparts. The structure of the apps used for testing was designed to be significantly different from one another so these results instilled in us a confidence that this process was functioning correctly and would continue to do so when applied to our larger body of apps. As an additional way of validating our process an error logging system was designed into the programs used to obtain and reverse engineer our body of apps. This error logging feature recorded any anomalies that occurred during the execution of these programs and recorded them in a text file saved locally. After these programs were run the produced error log was examined. The only errors encountered during this examination were either broken download links for .apk files or .apk files which when downloaded would not install on a device due to corrupted or invalid data and therefore should not be included in our study.

One method of validation we used on the data we collected was the act of comparing our data to data collected in previous studies. Most of these studies concerned Android's permissioning system but one study [ZHOU 12] looked into percentage of apps in third party markets that were repackaged apps from the official Android app store. In this study six third party markets were analyzed and it was found that between 5 and 13 percent of the apps in these markets were repackaged apps. While in our research we did not study this exact statistic we did examine the amount of crossover in apps between the markets we studied. We expected the number of apps crossing over from one of these markets to another to be at least as high as the amount of repackaged apps found in the previous study. We came to this conclusion because while we were not comparing our app collection to the official Android markets none of the markets we studied explicitly prohibited repackaged apps. We found this to be the case finding between 0 and 42 percent of the apps in any given pair of markets we studied were actually the same. Only one pair of markets Apps For Adam and And App Online had no crossover and only one pair of markets in our study experienced a crossover higher than 18 percent. The remaining pairs were all between 6 and 18 percent, very close to the numbers previously reported.

Of the papers that concerned Android's permissioning system several of them ([CHIA 12], [FELT 11b], [VENNON 10]) presented the most popular permissions found in the markets they were

studying. In all of these studies except our own only dangerous permissions, those with the potential to have unintended results for an end user, were examined. Due to this fact, some of our most popular permissions were not included in data from these other studies. Table 8 lists the top permissions from each of these studies and how they compared to the same permissions in our own study. The top permissions found in each study were very similar as well as the percentage of each market in which these permissions were found. In fact, the permission INTERNET was the most popular permission in all four of these studies. Our top permissions differed very little in either name or percentage from these previous studies leading us to believe that the data we collected for this section of our research was indeed valid.

PERMISSION	FELT 11b	CHIA 12	VENNON 10	OUR DATA
INTERNET	86% (1)	77% (1)	71% (1)	90% (1)
WRITE_EXTERNAL_STORAGE	34% (2)	51% (2)	16% (3)	28% (4)
ACCESS_COARSE_LOCATION	33% (3)	10% (6)	25% (2)	23% (5)
READ_PHONE_STATE	32% (4)	45% (3)	X	42% (3)
WAKE_LOCK	24% (5)	23% (4)	X	15% (8)
ACCESS_FINE_LOCATION	23 % (6)	14% (5)	15% (4)	19% (7)
READ_CONTACTS	16% (7)	7% (7)	9% (5)	6% (12)
WRITE_SETTINGS	13% (8)	X	X	4% (15)
GET_TASKS	4% (9)	6% (8)	X	8% (11)
CAMERA	X	6% (9)	X	6% (12)
READ_LOGS	X	6% (10)	X	1% (29)
RECORD_AUDIO	X	6% (11)	2% (9)	2% (22)
CALL_PHONE	X	5% (12)	5% (6)	3% (18)
SEND_SMS	X	X	3% (7)	3% (17)
READ_SMS	X	X	2% (8)	2% (20)
READ_CALENDAR	X	X	1% (10)	.4 % (44)

Table 8: Most Popular Permissions Per Study

The next source we looked at to validate our data were the standards set out in the Android developers guide. These standards prescribe suggested best practices and give standard values for several of the attributes of apps that we examined. We believe most developers are trying to follow these best practices and therefore expected this to be reflected in the data we collected from the apps in our study. The first attribute of this nature was the file size of the apk files we studied. Until March of 2012, apk files were held to a strict 50MB file size [GOOGLE 12h]. After this time apps were allowed to take up to

4GB of storage by utilizing two expansion files. In this case, the actual apk file is still limited to 50MB but two expansion files of up to 2GB each could be attached to the apk file. Because of this, we did not expect to see any apk files greater than 50MB in our study as all of our apps were collected before this expansion file scheme was implemented. After extracting the file size of each apk file, we found this to hold true. From this, we were able to determine that all of the files in our collection at least met this requirement of being a valid apk file.

The next standard we used to validate our data involved the practices of the markets on which the apps we obtained were hosted. Each of these markets except Apps For Adam have a policy that they do not host multiple versions of the same app. By multiple versions these markets mean that they will not host old versions of an app once it had been updated. This policy does not include free and paid versions of the same app, slight differences, or any other scenario of this nature. When examining our data we did in fact find apps that were hosted several times in each of the markets we studied. Upon closer examination each of these duplicate apps fell into one of two categories. Almost all apps of this nature that we found were apps which had been reposted, usually by a different user, with a different name to the market therefore making it appear to be a different app. The few remaining apps fell into the second category which appeared to be human error. In these cases, it appeared that the old version of the app had simply been overlooked on the market and allowed to remain alongside the new version. Finding a low number of duplicate apps and then being able to justify the ones we did find made us more confident in this data.

Another attribute that we attempted to validate in this manner was the minimum SDK level of the apps in our study. According to Google version 2.3, API level 10, of the Android operating system is currently installed on 65 percent of Android devices while all later versions only account for 7 percent of devices [GOOGLE 12i]. For this reason, we expected most of the apps we examined to be designed for devices operating with a minimum SDK level of 10 or less. When examining our data, this was the case. Apps requiring an SDK level greater than 10 accounted for less than one percent of the total body of apps we examined. Interestingly, SDK levels 3 and 4 were the most popular in our study, accounting for a

combined total of over 60 percent of all apps we examined. We believe this is due to one of two factors: First, it is possible that the apps we are examining are slightly dated. SDK levels 3 and 4 correspond to versions 1.5 and 1.6, cupcake and donut, of the Android operating system. These versions of the operating system were release in May and October of 2009. Therefore if the majority of apps on the markets we were examining were several years old this would explain the peak seen in the data at this point. The second option is that developers are not yet utilizing the features added to later versions of the SDK. This seems like a more plausible explanation as all of the markets we examined were active markets with new apps being posted regularly.

A final piece of data that we validated using the best practices described in the Android developer's guide was drawable localization data. The developer's guide suggests a best practice of localizing drawable images in at least three subdirectories based on the dots per inch of the display screen of an Android device [GOOGLE 12j]. While it is not necessary to utilize any subdirectories within drawable, we expected to find these three suggested localizations, hdpi, mdpi, and ldpi, more than any other when we examined our data. This theory was correct with each of these localizations being found in close to 50 percent of all apps that we examined. For comparison, the next closest localization for drawable objects was xhdpi, extra high dots per inch, and was found in only 4 percent of all apps we examined. Since this data matched the documentation found in the Android developer's guide we had a higher level of confidence that it was valid.

The final method we used to validate our data was comparing the values for attributes found in one market to the values of the same attribute found in a different market in our study. Since we studied only free apps and all of the markets we examined contained apps from a variety of different categories we expected the data we collected to be similar across each of these markets. One type of data which we validated in this way was the percentage of markets utilizing different localizations schemes. As can be seen in Figure 44, the data we collected followed a very clear trend with drawable localizations being far above any other type of localization. Language and layout localizations occurred in close to half as many apps as drawable localizations did and language localizations were utilized slightly more frequently than

layout localizations. The fact that this trend could be seen in each market we examined was encouraging and again lead us to believe that the data we were collecting was valid.

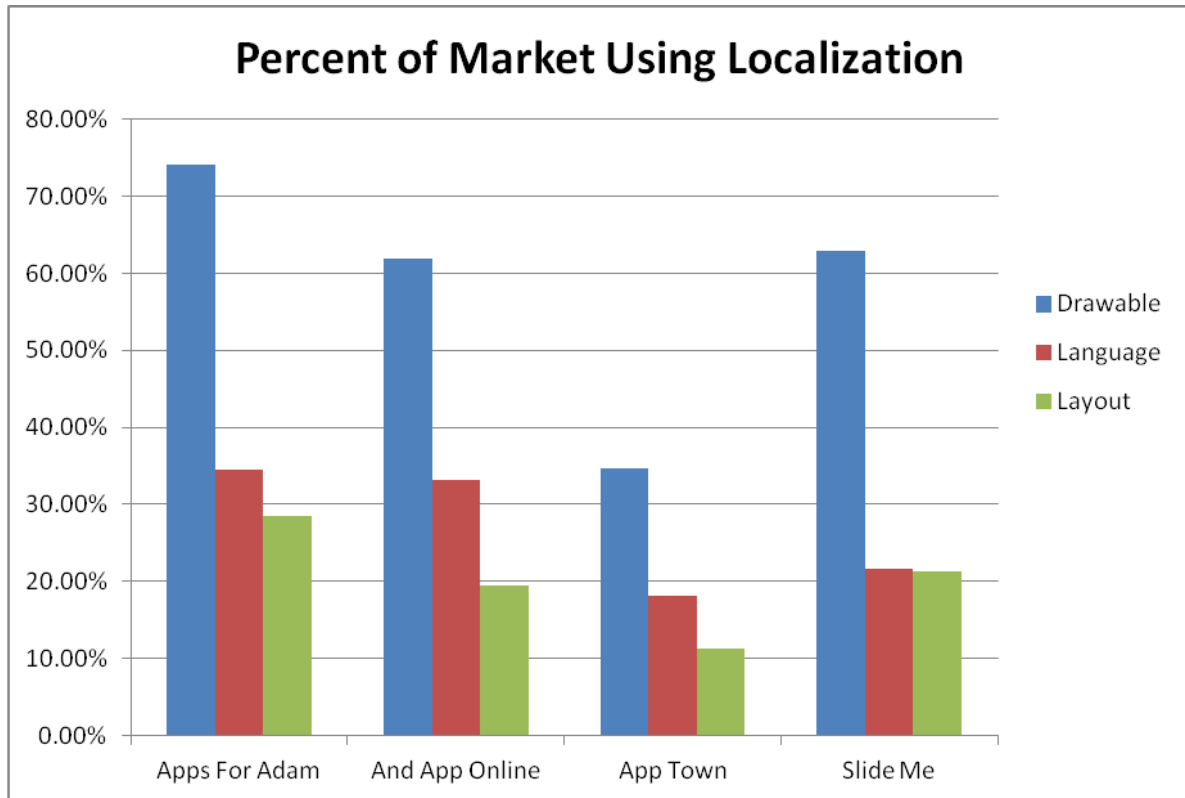


Figure 43 – Percent of Each Market Utilizing a Given Localization

Our validation process validated both the method we used to collect our data as well as the data collected. Our collection process was validated through a combination of trail runs, comparing the results our process produced to known values of what these results should be, and the use and examination of error logs during data extraction. Once our data was acquired we then verified the results of our analysis of this data by comparing it to previous work in the area, critiquing it with adherence to the standards and best practices put forth in the Android developer’s guide, or looking for repeating trends across our different data sets. Using these validation methods ensured that the data we used was consistent and explainable therefore allowing us to compile a valid demographic overview of the Android markets we examined.

Chapter 5 – Conclusions and Future Efforts

Conclusions

We had two main goals for this research. First we wished to determine if data of a demographic nature could be obtained from the .apk file of an Android app. Secondly, we wanted to determine if data of this nature, collected over a large body of Android apps, could be used to compile a demographic overview of a market as a whole. In order to accomplish these goals it was necessary to first demonstrate that a large body of Android apps could be collected in an automated fashion. We proved that this was possible by gathering a total of 13,885 apps from across four different third-party markets. Next, we displayed that demographic information could be extracted from these apps by using existing tools available in the marketplace in conjunction with programs we wrote ourselves to reverse engineer and extract information from the apps we acquired. Using this process we were able to extract information concerning the file size of an app, the number of different versions of an app in a given market, the minimum sdk level and permissions requested by an app, the apps developer, the external libraries used by an app, information on the services, providers, receivers, and activities an app declared, information describing how an app utilized localization, and the advertising schemes employed in an app. Many other attributes of an app would be accessible with slight modifications to the process we utilized but doing so was outside of the scope of our current research.

Finally, we proved that using this demographic information we extracted from our body of apps we could assemble a demographic overview of a market. By analyzing this data we were able to come to conclusions such as the INTERNET permission was the most requested permission in all four markets we examined, that Chinese was the most popular language to localize an app for, and that providers are the least utilized functionality of an app out of services, providers, and receivers. By looking at the markets as a whole we were able to identify these and many other demographic trends. The discovery and act of recording these demographics have helped us to form a more holistic view of the markets we studied and allowed us to better understand their current state.

Future Work

As with any research effort as we explored these topics in an attempt to answer our original inquiry many areas worthy of future research were discovered along the way. When we first approached this problem one of the first questions that arose was if we could determine the quality of an Android app or the overall quality of a market. We quickly realized that this was outside the scope of the work we were attempting to complete but it would be an interesting area to research in its own right. In our brief investigation of this subject, we saw very little information on how the quality of an Android app could be determined. Research efforts could refine what traditional metrics for software quality could be applied to mobile apps in general and which of these, or new metrics altogether, could be applied to Android apps specifically. Additionally, mobile app markets as a whole could be examined from a quality standpoint. Metrics could be investigated to construct an overall health or quality profile of a market and possible warning signs of unsafe markets could be identified.

Another interesting research area would be to see how the demographics of a given market changed over time. As new SDKs and features of the Android operating system are implemented research of this nature could determine how fast these features were adopted by developers and seen by the public. One could also attempt to examine which features were falling out of favor with developers as a market continued to mature. Collecting data of this nature could help architects of the Android operating system determine what features developers valued the most and which features might be unneeded or misunderstood without relying on direct interaction with the developers themselves. This information could then be used to shape the operating system in future releases. Finally, having an overview of a market with the ability to look at snapshots of its demographics at given time periods would have its own value.

During our research efforts we did not perform any analysis of the actual Java or Smali source code of an app. Another potential research area would be to begin to focus on extracting information from this source code in the same fashion that we did from the `AndroidManifest` file and the directory structure of the app. The process we used to reverse engineer apps produced the Smali code for each app

and there are currently tools available which can either convert this Smali code to Java or produce Java code directly from an app's apk file.

As we examined the advertising schemes of the apps in our collection, we found the monetization of Android apps to be a very interesting topic. App developers currently have two main ways in which to make money from their apps. They can either charge a onetime upfront fee for the app or they can add advertising to their app and then make money from the advertising revenue over time. We found many companies which provided advertising packages for Android. It would be interesting to study these different packages, determining how exactly they differed from each other, how profitable each one was, as well as how they all compared, in respect to profitability, to charging an upfront fee for the app.

The Android developer's guide suggests many standards and best practices for developing apps for Android devices. We used several when examining the data we collected to determine if the values we were seeing made sense logically. An additional area of research could isolate a series of these best practices and then determine what percentage of a given market was complying with these. This could potentially be incorporated into a study of the quality of an Android market if it was determined that adherence to these best practices was a good quality metric.

Finally, all of the markets we examined were third party markets due to the difficulties we encountered when attempting to use the official Android market as a source of data. During the completion of our work, we did find other research that had been conducted on large bodies of apps from this official market proving that this type of work is possible. Since Google does not take a walled garden approach with their app market, it would be interesting to perform a demographic study on both the official Google market and third party markets and then compare the results.

References

- [ANDRUS 12] Andrus, J.; Nieh, J.,; “Teaching Operating Systems Using Android”, *Proceeding of the 43rd ACM Technical Symposium on Computer Science Education*, pp 613 – 618, 2012.
- [APPLE 12a] Apple App Store <http://www.apple.com/iphone/from-the-app-store/>
- [APPLE 12b] 25 Billion Downloads Thanks <http://www.apple.com/itunes/25-billion-app-countdown/>
- [APPLE 12c] App Store Review Guidelines
<https://developer.apple.com/appstore/guidelines.html>
- [APPLE 12d] Publishing an App in the App Store
http://developer.apple.com/library/ios/#DOCUMENTATION/General/Conceptual/ApplicationDevelopmentOverview/DeliverYourAppontheAppStore/DeliverYourAppontheAppStore.html#//apple_ref/doc/uid/TP40011186-CH8-SW1
- [APPS 12] Apps for Adam <http://www.appsforadam.tk/>
- [APPSTORE 12] Amazon App Store For Android <http://www.amazon.com/b?node=2350149011>
- [AU 11] Au, K.; Zhou, Y.; Huang, Z.; Gill, P.; Lie, D.,; “Short Paper: A Look at Smartphone Permission Models”, *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 63 -68, 2011.
- [BARRERA 10] Barrera, D.; Kayacik, H., Oorschot, P., Somayaji, A.,; “A Methodology for Empirical Analysis of Permission-Based Security Models and Its Application to Android”, *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pp. 73 - 84, 2010.
- [BERGER 11] Berger, B.J.; Bunke, M.; Sohr, K.,; , "An Android Security Case Study with Bauhaus," *Reverse Engineering (WCRE), 2011 18th Working Conference on* , vol., no., pp.179-183, 17-20 Oct. 2011
doi: 10.1109/WCRE.2011.29
- [BLACKBERRY 12] BlackBerry DevCon Europe Keynote
http://www.youtube.com/watch?v=J7ID1go1k_I&feature=youtu.be
- [BOHMER 11] Bohmer, M.; Hecht, B.; Schoning, J.; Kruger, A.; Bauer, G.,; “Falling Asleep with Angry Birds, Facebook and Kindle: a Large Scale Study on Mobile Application Usage”, *Proceedings of the 13th Conference on Human Computer Interaction with Mobile Devices and Services*, pp. 47 – 56, 2011.
- [BULTER 11] Butler, M.,; , "Android: Changing the Mobile Landscape," *Pervasive Computing, IEEE* , vol.10, no.1, pp.4-7, Jan.-March 2011 doi: 10.1109/MPRV.2011.1
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5676144&isnumber=5676140>

[CHAN 12] Chan, P.; Hui, L.; Yiu, S., “DroidChecker: Analyzing Android Applications for Capability Leak”, *Proceeding of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 125 -136, 2012

[CHIA 12] Chia, P.; Yamaoto, Y.; Asokan, N., “Is This App Safe?: A Large Scale Study on Application Permissions and Risk Signals”, *Proceedings of the 21st International Conference on World Wide Web*, pp 311 -320, 2012.

[DESNOS 11] Desnos, A.; Guegue, G., “Android: From Reversing to Decompilation”, *Blackhat*, 2011.

[DESNOS 12] Desnos, A.; et al, “Androguard Reverse Engineering, Malware and Goodware Analysis of Android Applications”, <http://code.google.com/p/androguard/> May 2012.

[DUTKO 08] Dutko, A., “Domo Arigato Mr Androidato an Introduction to the New Google Mobile Linux Framework, Android,” *Linux Journal*, vol. 2008, no. 167, Mar. 2008

[ENCK 11] Enck, W.; Ocateu, D.; McDaniel, P.; Chaudhuri, S., “A Study of Android Application Security”, *Proceedings of the 20th USENIX Conference on Security*, pp. 21 – 37, 2011.

[FELT 11a] Felt, A.; Chin, E.; Hanna, S.; Song, D.; Wagner, D., “Android Permissions Demystified”, *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 627 – 638, 2011

[FELT 11b] Felt, A.; Greenwood, K.; Wagner, D., “The Effectiveness of Application Permissions”, *Proceedings of the 2nd USENIX Conference on Web Application Development*, pp. 75 – 86, June 2011.

[FREE 12] Freeware Lovers - The Best Thing in Life are Free <http://www.freewarelovers.com/>

[GARTNER 11] Gartner News Room Press Releases – Gartner Says Worldwide Smartphone Sales Soared in fourth Quarter of 2011 With 47 Percent Growth.
<http://www.gartner.com/it/page.jsp?id=1924314>

[GOLD 12] Gold, S., “'App napping'?: Will holey handsets get caught,” *Engineering & Technology*, vol.6, no.8, pp.78-81, September 2011

[GOOGLE 11] A Closer Look at 10 Billion Downloads <http://android-developers.blogspot.com/2011/12/closer-look-at-10-billion-downloads.html>

[GOOGLE 12a] Publishing on Google Play
<http://developer.android.com/guide/publishing/publishing.html>

- [GOOGLE 12b] Google Play Developer Content Policies
<http://support.google.com/googleplay/android-developer/bin/answer.py?hl=en&answer=113474>
- [GOOGLE 12c] Google Android Developer's Guide
<http://developer.android.com/guide/index.html>
- [GOOGLE 12d] Google Play Market Features <https://play.google.com/about/features/>
- [GOOGLE 12e] Google Play Market Listing
<https://play.google.com/store/apps/details?id=com.zeptolab.ctr.paid&feature=top-paid#?t=W251bGwsMSwxLDIwNiwiY29tLnplcHRvbGFiLmN0ci5wYWlkII0>.
- [GOOGLE 12f] Google Terms of Service <http://www.google.com/intl/en/policies/terms/>
- [GOOGLE 12g] The Android Manifest File
<http://developer.android.com/guide/topics/manifest/manifest-intro.html>
- [GOOGLE 12h] Android Apps Break the 50MB Barrier <http://android-developers.blogspot.com/2012/03/android-apps-break-50mb-barrier.html>
- [GOOGLE 12i] Android Developers Guide – Platform Versions
<http://developer.android.com/resources/dashboard/platform-versions.html>
- [GOOGLE 12j] Android Developer's Guide – Canvas and Drawables
<http://developer.android.com/guide/topics/graphics/2d-graphics.html#drawables>
- [GRACE 12] Grace, M.; Zhou, W.; Jiang, X.; Sadeghi, A.; “Unsafe Exposure Analysis of Mobile In-App Advertisements”, *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 101 -112, 2012.
- [GRUVER 12] Gruver, B.; et al.;, “Smali: An Assembler/Disassembler for Android's DEX Format”, <http://code.google.com/p/smali/> , May 2012.
- [GUILFOYLE 12] Guilfoyle, J.;, “Android – Random Collection of Extended Examples for Android Developers” <http://code.google.com/p/android-random/> May 2012.
- [HALL 09] Hall, S.; Anderson, E.;, “Operating Systems for Mobile Computing,” *Journal of Computing Sciences in Colleges*, vol. 25, no. 2, pp 64 -71 Dec. 2009
- [HECKMAN 11] Heckman, S.; Horton, T.B.; Sherriff, M.; , "Teaching second-level Java and software engineering with Android," *Software Engineering Education and Training (CSEE&T), 2011 24th IEEE-CS Conference on* , vol., no., pp.540-542, 22-24 May 2011
doi: 10.1109/CSEET.2011.5876144
- [HENZE 11] Henze, N.; Rukzio, E.; Boll, S.;, “100,000,000 Taps: Analysis and Improvement of Touch Performance in the Large”, *MOBILE HCI '11 Proceeding of the 13th International*

Conference on Human Computer Interaction with Mobile Devices and Services, pp.133 – 142, 2011 doi: 10.1145/2037373.2037395

[HIGA 08] Higa, D.; , "Walled Gardens versus the Wild West," *Computer* , vol.41, no.10, pp.102-105, Oct. 2008 doi: 10.1109/MC.2008.439
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4640677&isnumber=4640644>

[HOLZER 11] Holzer A., Ondrus J. Mobile application market: A developer's perspective. *Telematics and Informatics* vol. 28, pp. 22–31, 2011

[HUSTED 11] Husted, N.; Saidi, H; Gehani, A., "Smartphone Security Limitations: Conflicting Traditions", *Proceedings of the 2011 Workshop on Governance of Technology, Information, and Policies, ACM*, pp. 5 -12

[LIB 12] Apps Lib - The Application Market Place for Android Smart Phones <http://appslib.com/>

[ITU 09] The World in 2009: ICT Facts and Figures. <http://www.itu.int/ITU-D/ict/facts/2011/material/ICTFactsFigures2009.pdf>

[ITU 11] The World in 2011: ICT Facts and Figures. <http://www.itu.int/ITU-D/ict/facts/2011/material/ICTFactsFigures2011.pdf>

[LOCHMANN 11] Lochnamm, K.; Goeb, A., "A Unifyng Model for Software Quality", *Proceedings of the 8th International Workshop on Software Quality*, ACM, pp. 3 – 10

[MCDANIEL 10] McDaniel, P.; Enck, W.; , "Not So Great Expectations: Why Application Markets Haven't Failed Security," *Security & Privacy, IEEE* , vol.8, no.5, pp.76-78, Sept.-Oct. 2010 doi: 10.1109/MSP.2010.159
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5601494&isnumber=5601474>

[MYSQL 12] Why MySQL? <http://www.mysql.com/why-mysql/>

[MYSQLDB 12] MySQLdb User's Guide <http://mysql-python.sourceforge.net/MySQLdb.html>

[NETANEL 07] Netanel, L.; "Temptations of the Walled Garden: Digital Rights Management and Mobile Phone Carriers," *Journal on Telecommunications and High Technology Law*, vol. 6, pp. 77 – 100.

[NOKIA 12] Nokia Developer – Global Reach Statistics
<http://www.developer.nokia.com/Distribute/Statistics.xhtml>

[OCTEAU 10] Oceau, D.; Enck, W.;, "The ded Decompiler" *Technical Report NAS-TR-0140-2010*, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, Sept. 2010.

[PAN 12] Pan, X.;, "Dex2jar Tools to Work with Android .dex and Java .class files"
<http://code.google.com/p/dex2jar/> May 2012.

[PCW 12] Android Market Hits 450K Apps, Challengers Abound
http://www.pcworld.com/article/250765/android_market_hits_450k_apps_challengers_abound.html

[POSTOLACHE 11] Postolache, O.; Girao, P.S.; Postolache, G.; Gabriel, J.; , "Cardio-respiratory and daily activity monitor based on FMCW Doppler radar embedded in a wheelchair," *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE* , vol., no., pp.1917-1920, Aug. 30 2011-Sept. 3 2011
doi: 10.1109/IEMBS.2011.6090542

[ROVIO 11] Rovio Entertainment Reports 2011 Financial Results
<http://www.rovio.com/en/news/press-releases/161/rovio-entertainment-reports-2011-financial-results/>

[SKIBA 12] Skiba, D.;, "Android4Me J2ME port of Google's Android"
<http://code.google.com/p/android4me/> May 2012.

[SMALI 12] Smali Hello World Example
<http://code.google.com/p/smali/source/browse/examples/HelloWorld/HelloWorld.smali>

[SYER 11] Syer, M.D.; Adams, B.; Ying Zou; Hassan, A.E.; , "Exploring the Development of Micro-apps: A Case Study on the BlackBerry and Android Platforms," *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on* , vol., no., pp.55-64, 25-26 Sept. 2011 doi: 10.1109/SCAM.2011.25

[VENNON 10] Vennon, T.; Stroop, D.;, "Android Market: Threat Analysis of the Android Market", *Tech Republic*, SMobile Systems, 2010.

[WISNIEWSKI 12] Winsniewski, R.;, "Android – Apktool: A Tool for Reverse Engineering Android apk Files" <http://code.google.com/p/android-apktool/> May 2012.

[ZHOU 12] Zhou, W.; Zhou, Y.; Jiang, X.; Ning, P.;. "Detecting Repackaged Smartphone Applications in Third-Party Android Marketplace", *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, pp. 317 – 326, 2012

Appendix 1 – Source Code

Apps For Adam Targeted Crawler

```
'''
Created on Jan 30, 2012

@author: Billy Symon
'''

import urllib
import re

#TODO
def getPageCount(url):
    numPagesExp = "class=\"pagination_last\">(\d+)</a>"
    site = urllib.urlopen(url)
    html = site.read()
    site.close()

    numPages = int(re.findall(numPagesExp, html)[0])
    return numPages

def getHTML(url):
    site = urllib.urlopen(url)
    html = site.read()
    site.close()
    return html

def getDownloadLinksAndNames(html):
    downloadLinkExp = "<a href=\"(http://appsforadam.tk/apps/(.*?))\"
target=\"_blank\">Download</a></span></span><br />"
    return re.findall(downloadLinkExp, html)

def getAppPageLinks(html):
    applinkexp = "App: <a href=\"(.*)\">"
    return re.findall(applinkexp, html)

def downloadApp(url, path, appName):
    f = open(path+"appsforadam_"+appName, "wb")
    app = urllib.urlopen(url)
    f.write(app.read())
    f.close()
    app.close()

#####

domain =
"http://appsforadam.tk/search.php?action=results&sid=2d988417050b102d5c3a54d3028e7b61&
sortBy=dateline&order=desc&uid="
outputPath = "E:\\Graduate Thesis\\apk files\\appsforadam\\"
appsDownloaded = 0

#this is where we will any errors we experience while downloading apps such as
#a problem with the formatting or if the app cost money to buy
errorFile = open(outputPath+"appsforadam_errors.txt", "w")

numPages = getPageCount(domain)
```



```

print "Downloading apps from "+str(numOfPages)+" pages . . ."
print "Scanning approximately "+str(numOfPages*20)+" apps for canidates . . ."

##for i in range(numOfPages+1):
for i in range(numOfPages+1):
    #set up the page url
    print "!!!!!!begining page #"+str(i)+" !!!!!!"
    currentPage = domain+"&page="+str(i)

    #get the html code
    try:
        html = getHTML(currentPage)
        appPages = getAppPageLinks(html)
        try:
            for page in appPages:
                downloadLinks = getDownloadLinksAndNames(html)
                for link in downloadLinks:
                    try:
                        downloadApp(link[0], outputPath, link[1])
                        appsDownloaded += 1
                        if appsDownloaded % 5 == 0:
                            print "***** "+str(appsDownloaded) + " Apps downloaded
*****"
                    except:
                        errorFile.write("Problem downloadig app at "+link[0]+"\n")
                except:
                    errorFile.write("Problem getting download links from "+page+"\n")
            except:
                errorFile.write("Problem geting html from "+currentPage+"\n")

errorFile.close()

print str(appsDownloaded)+" apps downloaded"

```

App Town Targeted Crawler

```
'''
Created on Jan 29, 2012

@author: Billy Symon
'''
import urllib
import re

#TODO
def getPageCount(url):
    numOfPagesExp = "&nbsp;<span class=\"nav-dots right-delimiter\">...</span>&nbsp;<a
class=\"nav-page right-delimiter\"
href=\"http://www.apptown.com/Android/?page=(\d+)\"\"
    site = urllib.urlopen(url)
    html = site.read()
    site.close()

    numOfPages = int(re.findall(numOfPagesExp, html)[0])
    return numOfPages

def getHTML(url):
    site = urllib.urlopen(url)
    html = site.read()
    site.close()
    return html

def getDownloadLinks(html):
    downloadLinkExp = "<a href=\"./download_free_products.php?id=(.*?)\">Download
Now</a>"
    return re.findall(downloadLinkExp, html)

def downloadApp(url, path, appName):
    f = open(path+"apptown_"+appName+".apk", "wb")
    app = urllib.urlopen(url)
    f.write(app.read())
    f.close()
    app.close()

#####

domain = "http://www.apptown.com/android"
outputPath = "E:\Graduate Thesis\apk files\apptown\"
appsDownloaded = 0

#this is where we will any errors we experience while downloading apps such as
#a problem with the formatting or if the app cost money to buy
errorFile = open(outputPath+"apptown_errors.txt", "w")

numOfPages = getPageCount(domain)

print "Downloading apps from "+str(numOfPages)+" pages . . ."
print "Scanning approximately "+str(numOfPages*13)+" apps for candidates . . ."

##for i in range(numOfPages+1):
for i in range(numOfPages+1):
    #set up the page url
    print "!!!!!!begining page #"+str(i)+" !!!!!!"
```

```

currentPage = domain+"?page="+str(i)

#get the html code
try:
    html = getHTML(currentPage)

    downloadLinks = getDownloadLinks(html)
    for link in downloadLinks:
        try:

downloadApp("http://www.apptown.com/download_free_products.php?id="+link, outputPath,
link)
            appsDownloaded += 1
            if appsDownloaded % 5 == 0:
                print "***** "+str(appsDownloaded) + " Apps downloaded *****"
            except:
                errorFile.write("Problem downloadig app at
http://www.apptown.com/download_free_products.php?id="+link+"\n")
            except:
                errorFile.write("Problem geting html from "+currentPage+"\n")

errorFile.close()

print str(appsDownloaded)+" apps downloaded"

```

And App Online Targeted Crawler

```
'''
Created on Jan 26, 2012

@author: symonwi
'''
import urllib
import re

def getPageCount(url):
    applicationsPage = url+"/applications"
    numOfPagesExp = "<li class=\"pager-last last\"><a
href=\"/applications?page=(\d+)\"

    site = urllib.urlopen(applicationsPage)
    html = site.read()
    site.close()

    numOfPages = int(re.findall(numOfPagesExp, html)[0])
    return numOfPages

def getHTML(url):
    site = urllib.urlopen(url)
    html = site.read()
    site.close()
    return html

def getApplicationPageLinks(html):
    appPageExp = "<h2 class=\"title\"><a href=\"(/application/.*)\"

    return re.findall(appPageExp, html)

def getDownloadLinks(html):
    downloadLinkExp = "<div class=\"download-button\"><a href=\"(.*)\"

    return re.findall(downloadLinkExp, html)

def downloadApp(url, path, appName):
    f = open(path+"slideme_"+appName+".apk", "wb")
    app = urllib.urlopen(url)
    f.write(app.read())
    f.close()
    app.close()

def getAppName(html):
    appNameExp = "<div class=\"download-button\"><a href=\".*?\" title=\"(.+?)\"

    return re.findall(appNameExp, html)[0].translate(None, "<:\/\|?*")

#####

domain = "http://www.slideme.org"
outputPath = "E:\\Graduate Thesis\\apk files\\slideme\\"
appsExamined = 0
appsDownloaded = 0

#this is where we will any errors we experience while downloading apps such as
#a problem with the formatting or if the app cost money to buy
errorFile = open(outputPath+"slideme_errors.txt", "w")

numOfPages = getPageCount(domain)
```

```

print "Downloading apps from "+str(numOfPages)+" pages . . ."
print "Scanning approximately "+str(numOfPages*10)+" apps for canidates . . ."

##for i in range(numOfPages+1):
for i in range(numOfPages+1):
    #set up the page url
    print "!!!!!!begining page #"+str(i)+" !!!!!!"
    currentPage = domain+"/applications?page="+str(i)

    #get the html code
    html = getHTML(currentPage)

    #extract the links to the actual application pages
    applicationPages = getApplicationPageLinks(html)

    #for each page parse to see if it is a free app and if so download
    for page in applicationPages:
        #get the html code
        html = getHTML(domain+page)
        #extract the links
        downloadLinks = getDownloadLinks(html)
        appsExamined += 1
        #give status update for each 100 downloaded
        if appsExamined % 5 == 0:
            print "***** "+str(appsExamined) + " Apps examined *****"

        #if no links there is a formatting error with the page
        if len(downloadLinks) > 0:
            downloadLink = downloadLinks[0]
            #if it doesn't end with .apk it's a pay app
            if downloadLink[-4:] == ".apk":
                #print domain+downloadLink
                downloadApp(domain+downloadLink,outputPath,getAppName(html))
                appsDownloaded += 1
                if appsDownloaded % 5 == 0:
                    print "***** "+str(appsDownloaded) + " Apps downloaded *****"
            else:
                errorFile.write("$$$$$$ Pay app found at "+domain+page+" $$$$$$\n")
        else:
            errorFile.write("XXXXXX Formating Error with page "+domain+page+"
XXXXXXXX\n")

errorFile.close()
print str(appsExamined)+" apps examined"
print str(appsDownloaded)+" apps downloaded"

```

Slide Me Targeted Crawler

```
'''
Created on Jan 26, 2012

@author: symonwi
'''
import urllib
import re

def getPageCount(url):
    applicationsPage = url+"/applications"
    numOfPagesExp = "<li class=\"pager-last last\"><a
href=\"/applications?page=(\d+)\"

    site = urllib.urlopen(applicationsPage)
    html = site.read()
    site.close()

    numOfPages = int(re.findall(numOfPagesExp, html)[0])
    return numOfPages

def getHTML(url):
    site = urllib.urlopen(url)
    html = site.read()
    site.close()
    return html

def getApplicationPageLinks(html):
    appPageExp = "<h2 class=\"title\"><a href=\"(/application/.*)\"
    return re.findall(appPageExp, html)

def getDownloadLinks(html):
    downloadLinkExp = "<div class=\"download-button\"><a href=\"(.*)\"
    return re.findall(downloadLinkExp, html)

def downloadApp(url, path, appName):
    f = open(path+"slideme_"+appName+".apk", "wb")
    app = urllib.urlopen(url)
    f.write(app.read())
    f.close()
    app.close()

def getAppName(html):
    appNameExp = "<div class=\"download-button\"><a href=\".*?\" title=\"(.+?)\"
    return re.findall(appNameExp, html)[0].translate(None, "<:\/\|?*")

#####

domain = "http://www.slideme.org"
outputPath = "E:\\Graduate Thesis\\apk files\\slideme\\"
appsExamined = 0
appsDownloaded = 0

#this is where we will any errors we experience while downloading apps such as
#a problem with the formatting or if the app cost money to buy
errorFile = open(outputPath+"slideme_errors.txt", "w")

numOfPages = getPageCount(domain)
```

```

print "Downloading apps from "+str(numOfPages)+" pages . . ."
print "Scanning approximately "+str(numOfPages*10)+" apps for canidates . . ."

##for i in range(numOfPages+1):
for i in range(numOfPages+1):
    #set up the page url
    print "!!!!!!begining page #"+str(i)+" !!!!!!"
    currentPage = domain+"/applications?page="+str(i)

    #get the html code
    html = getHTML(currentPage)

    #extract the links to the actual application pages
    applicationPages = getApplicationPageLinks(html)

    #for each page parse to see if it is a free app and if so download
    for page in applicationPages:
        #get the html code
        html = getHTML(domain+page)
        #extract the links
        downloadLinks = getDownloadLinks(html)
        appsExamined += 1
        #give status update for each 100 downloaded
        if appsExamined % 5 == 0:
            print "***** "+str(appsExamined) + " Apps examined *****"

        #if no links there is a formatting error with the page
        if len(downloadLinks) > 0:
            downloadLink = downloadLinks[0]
            #if it doesn't end with .apk it's a pay app
            if downloadLink[-4:] == ".apk":
                #print domain+downloadLink
                downloadApp(domain+downloadLink,outputPath,getAppName(html))
                appsDownloaded += 1
                if appsDownloaded % 5 == 0:
                    print "***** "+str(appsDownloaded) + " Apps downloaded *****"
            else:
                errorFile.write("$$$$$$ Pay app found at "+domain+page+" $$$$$$\n")
        else:
            errorFile.write("XXXXXX Formating Error with page "+domain+page+"
XXXXXXXX\n")

errorFile.close()
print str(appsExamined)+" apps examined"
print str(appsDownloaded)+" apps downloaded"

```

Decompiler

```
'''
Created on Jan 31, 2012

@author: Billy Symon
'''
import os
import os.path

count = 0
for root, dirs, files in os.walk("C:\\Users\\Billy Symon\\Desktop\\Grad Research\\GRAD THESIS [BACKUP_DO_NOT_USE]\\apk files\\slideme\\5"):
    for file in files:
        print "*****"
        os.system("apktool -v d -f \""+os.path.join(root, file)+"\" \"C:\\Users\\Billy Symon\\Desktop\\Grad Research\\GRAD THESIS [BACKUP_DO_NOT_USE]\\Decompiled Files\\slideme\\5\\\""+file[0:-4]+"\"")
        count += 1
    print "*****"
    print "~~~~~"
    print "          "+str(count)+" files decompiled"
    print "~~~~~"
```


Main Data Extraction

```
import os
import xml.dom.minidom as minidom
import DBWriter

dataDirectory = "C:\\Users\\Billy Symon\\Desktop\\Grad Research\\GRAD THESIS
[BACKUP_DO_NOT_USE]\\Decompiled Files\\"
markets = ["appsforadam", "andapponline", "apptown"]

slideMeDirectory = "C:\\Users\\Billy Symon\\Desktop\\Grad Research\\GRAD THESIS
[BACKUP_DO_NOT_USE]\\Decompiled Files\\slideme\\"
slideMeDirs = ["1", "2", "3", "4", "5"]

def extractData(location, market):
    path = location+market
    files = os.listdir(path)
    directorySize = len(files)

    print "!"*150
    print "NOW EXTRACTING DATA FROM " + path
    print str(directorySize)+" APPS FOUND"
    print "!"*150

    i = 1
    for f in files:

        filePath = path + "\\ "+ f
        #PRINTING HEADER
        print "*"*50
        print "EXTRACTING DATA FROM FILE "+ str(i) + " OF "+ str(directorySize)
        print "*"*50
        i += 1

        if validateManifest(filePath):
            #EXTRACTING FILENAME
            fileName = f + ".apk"
            print "File Name -> " + fileName

            #EXTRACTING MARKET
            #market = "slideme"
            print "Market -> " + market

            #EXTRACTING FILE SIZE
            try:
                fileSize = getFileSize(filePath)
            except:
                fileSize = 0
                writeError("?????" + filePath, "Error when gathering file size data")
            print "File Size -> " + str(fileSize) + " bytes"

            #####THE REMAINING EXTRACTED INFORMATION COMES FROM THE MANIFEST.XML
FILE#####

            if os.path.exists(filePath+"\\AndroidManifest.xml"):
                manifest = minidom.parse(filePath+"\\AndroidManifest.xml")

            #EXTRACTING APP LABEL
            appLabel = extractAppLabel(manifest, filePath)
            try:
```

```

    print "App Label -> " + appLabel
except:
    appLabel = "DATA NOT FOUND - INVALID ENCODING"
    print "App Label -> " + appLabel

#EXTRACTING FULLY QUALIFIED APP NAME
appFQName = extractFQName(manifest)
print "Fully Qualified Name -> " + appFQName

#EXTRACTING MIN SDK LEVEL
minSDKLevel = extractMinSDKLevel(manifest)
print "Minimum SDK Level -> " + minSDKLevel

#EXTRACTING PERMISSIONS USED
permissionsUsed = extractPermissionsUsed(manifest)
numPermissionsUsed = len(permissionsUsed)
print "Number of Permissions Requested -> " + str(numPermissionsUsed)
if numPermissionsUsed > 0:
    for permission in permissionsUsed:
        try:
            print "Permission Requested -> " + permission
        except:
            pass
else:
    print "No Permissions Requested"

#EXTRACTING PERMISSIONS SET UP BY THE APP
permissionsSetUp = extractPermissionsSetUp(manifest)
numPermissionsSetUp = len(permissionsSetUp)
print "Number of Permissions Set Up -> " + str(numPermissionsSetUp)
if numPermissionsSetUp > 0:
    for permission in permissionsSetUp:
        print "Permission Set Up By App -> " + permission
else:
    print "No Permissions Set Up By App"

#EXTRACTING PERMISSIONS REQUIRED TO INTERACT
permissionsRequired = extractPermissionsRequired(manifest)
numPermissionsRequired = len(permissionsRequired)
print "Number of Permissions Required -> " + str(numPermissionsRequired)
if numPermissionsRequired > 0:
    for permission in permissionsRequired:
        print "Permission Required to Interact with App -> " + permission
else:
    print "No Permissions Required to Interact with App"

#EXTRACTING ACTIVITIES
activities = extractActivities(manifest)
numActivities = len(activities)
print "Number of Activities -> " + str(numActivities)
if numActivities > 0:
    for activity in activities:
        print "Activity -> " + activity
else:
    print "No Activities Declared"

#EXTRACTING SERVICES
services = extractServices(manifest)
numServices = len(services)
print "Number of Services -> " + str(numServices)
if numServices > 0:
    for service in services:
        print "Service -> " + service

```

```

else:
    print "No Services Declared"

#EXTRACTING RECEIVERS
receivers = extractReceivers(manifest)
numReceivers = len(receivers)
print "Number of Receivers -> " + str(numReceivers)
if numReceivers > 0:
    for receiver in receivers:
        print "Receiver -> " + receiver
else:
    print "No Receivers Declared"

#EXTRACTING PROVIDERS
providers = extractProviders(manifest)
numProviders = len(providers)
print "Number of Providers -> " + str(numProviders)
if numProviders > 0:
    for provider in providers:
        print "Provider -> " + provider
else:
    print "No Providers Declared"

#EXTRACTING LIBRARIES USED
libraries = extractLibraries(manifest)
numLibraries = len(libraries)
print "Number of Libraries Used -> " + str(numLibraries)
if numLibraries > 0:
    for library in libraries:
        print "Library -> " + library
else:
    print "No External Libraries Used"

#EXTRACTING NUMBER OF LAYOUTS
numLayouts = extractNumLayouts(filePath)
print "Number of Layouts -> " + str(numLayouts)
if numLayouts != 0:
    altLayouts = checkAltLayouts(filePath)
else:
    altLayouts = "No"
print "Alternative Layouts Provided -> " + altLayouts

#EXTRACTING NUMBER OF STRINGS IN STRING.XML
numStrings = extractNumStrings(filePath)
print "Number of Strings -> " + str(numStrings)
if numStrings > 0:
    altStrings = checkAltStrings(filePath)
else:
    altStrings = "No"
print "Alternative Strings Provided -> " + altStrings

#WRITE DATA GATHERED OUT TO DATABASE
#####
#####VARIABLES AVAILABLE#####
#####
#fileName -> name of the file including .apk extension [string]
#market -> the name of the market from which the app came [string]
#fileSize -> the size of the file in bytes [int]
#appLabel -> the label displayed on the device for the application
[string]
#appFQName -> the fully qualified name of the app [string]
#minSDKLevel -> the minimum sdk level needed to run the app [string]

```

```

        #numPermissionsUsed -> the number of permissions requested by the app
[int]
        #permissionsUsed -> a list containing the names of the permissions used
[list of strings]
        #numPermissionsSetUp -> the number of permissions set up by the app [int]
        #permissionsSetUp -> a list containing the names of the permissions set up
by the app [list of strings]
        #numPermissionsRequired -> the number of permissions required to interact
with the app [int]
        #permissionsRequired -> a list containing the names of the permissions
required to interact app [list of strings]
        #numActivities -> the number of activities set up by the app [int]
        #activities -> a list containing all of the activities set up by the app
[list of strings]
        #numServices -> the number of services set up by the app [int]
        #services -> a list containing all of the services set up by the app [list
of strings]
        #numReceivers -> the number of receivers set up by the app [int]
        #receivers -> a list containing all of the receivers set up by the app
[list of strings]
        #numProviders -> the number of providers set up by the app [int]
        #providers -> a list containing all of the providers set up by the app
[list of strings]
        #numLibraries -> the number of external libraries used by the app [int]
        #libraries -> a list containing all of the libraries set up by the app
[list of strings]
        #numLayouts -> the number of layouts present in the directory /res/layout
[int]
        #altLayouts -> a string stating if alternative layouts are provided for
the app [string (yes|no)]
        #numStrings -> the number of strings present in the file
/res/value/strings.xml [int]
        #altStrings -> a string stating if alternative strings are provided for
the app [string (yes|no)]

```

```

        #Printing DB Write Header
        print "\n~~~~~"
        print "WRITING GATHERED DATA TO DATABASE..."
        print "~~~~~\n"

        DBWriter.writeAppInfoTableEntry(fileName, market, fileSize, appLabel,
appFQName)
        DBWriter.writePermissionsInfoTableEntry(fileName, numPermissionsUsed,
numPermissionsSetUp, numPermissionsRequired)
        DBWriter.writePermissionsRequestedTableEntry(fileName, permissionsUsed)
        DBWriter.writePermissionsSetUpTableEntry(fileName, permissionsSetUp)
        DBWriter.writePermissionsRequiredTableEntry(fileName, permissionsRequired)
        DBWriter.writeIntentsTableEntry(fileName, numActivities, numServices,
numReceivers, numProviders)
        DBWriter.writeActivitiesTableEntry(fileName, activities)
        DBWriter.writeServicesTableEntry(fileName, services)
        DBWriter.writeReceiversTableEntry(fileName, receivers)
        DBWriter.writeProvidersTableEntry(fileName, providers)
        DBWriter.writeAdditionalInfoTableEntry(fileName, numLibraries, numLayouts,
numStrings, minSDKLevel, altLayouts, altStrings)
        DBWriter.writeLibrariesTableEntry(fileName, libraries)
        DBWriter.writeMasterEntry(fileName, market, fileSize, appLabel, appFQName,
numPermissionsUsed, numPermissionsSetUp, numPermissionsRequired, numActivities,
numServices, numReceivers, numProviders, numLibraries, numLayouts, numStrings,
minSDKLevel, altLayouts, altStrings)

        print ""

```

```

    else:
        print "DATA NOT FOUND - ERROR FOUND WITH FILE AndroidManifest.xml"
        print ""

def getFileSize(start_path):
    total_size = 0
    for dirpath, dirnames, filenames in os.walk(start_path):
        for f in filenames:
            fp = os.path.join(dirpath, f)
            total_size += os.path.getsize(fp)
    return total_size

def extractNumStrings(path):
    if os.path.exists(path+"\\res\\values\\strings.xml"):
        try:
            xmlFile = minidom.parse(path+"\\res\\values\\strings.xml")
            strings = xmlFile.getElementsByTagName("string")
            return len(strings)
        except:
            writeError("OOOOO"+path+"\\res\\values\\strings.xml", "Error parsing
strings.xml in resources")
            return 0
    else:
        return 0

def checkAltStrings(path):
    if os.path.exists(path+"\\res"):
        contents = os.listdir(path+"\\res")
        count = 0;
        for content in contents:
            if content[0:6] == "values":
                dirContents = os.listdir(path+"\\res\\"+content)
                if dirContents.count("strings.xml") > 0:
                    count += 1
        if count > 1:
            return "Yes"
    return "No"

def extractNumLayouts(path):
    if os.path.exists(path+"\\res\\layout"):
        return len(os.listdir(path+"\\res\\layout"))
    else:
        return 0

def checkAltLayouts(path):
    if os.path.exists(path+"\\res"):
        contents = os.listdir(path+"\\res")
        count = 0;
        for content in contents:
            if content[0:6] == "layout":
                count += 1
        if count > 1:
            return "Yes"
    return "No"

def extractLibraries(manifest):
    libraries = []
    tags = manifest.getElementsByTagName("uses-library")
    for tag in tags:
        library = tag.getAttribute("android:name")
        libraries.append(library)
    return libraries

```

```

def extractProviders(manifest):
    providers = []
    tags = manifest.getElementsByTagName("provider")
    for tag in tags:
        provider = tag.getAttribute("android:name")
        providers.append(provider)
    return providers

def extractReceivers(manifest):
    receivers = []
    tags = manifest.getElementsByTagName("receiver")
    for tag in tags:
        receiver = tag.getAttribute("android:name")
        receivers.append(receiver)
    return receivers

def extractServices(manifest):
    services = []
    tags = manifest.getElementsByTagName("service")
    for tag in tags:
        service = tag.getAttribute("android:name")
        services.append(service)
    return services

def extractActivities(manifest):
    activities = []
    tags = manifest.getElementsByTagName("activity")
    for tag in tags:
        activity = tag.getAttribute("android:name")
        activities.append(activity)
    return activities

def extractPermissionsRequired(manifest):
    permissionsRequired = []
    tags = manifest.getElementsByTagName("application")
    for tag in tags:
        permission = tag.getAttribute("android:permission")
        if permission != "":
            permissionsRequired.append(permission)
    return permissionsRequired

def extractPermissionsSetUp(manifest):
    permissionsSetUp = []
    tags = manifest.getElementsByTagName("permission")
    for tag in tags:
        permission = tag.getAttribute("android:name")
        permissionsSetUp.append(permission)
    return permissionsSetUp

def extractPermissionsUsed(manifest):
    permissionsUsed = []
    tags = manifest.getElementsByTagName("uses-permission")
    for tag in tags:
        permission = tag.getAttribute("android:name")
        permissionsUsed.append(permission)
    return permissionsUsed

def extractMinSDKLevel(manifest):
    tag = manifest.getElementsByTagName("uses-sdk")
    if len(tag) > 0:
        for item in tag:
            sdkLevel = item.getAttribute("android:minSdkVersion")

```

```

        return sdkLevel
    else:
        return "UNKNOWN"

def extractFQName(manifest):
    tag = manifest.getElementsByTagName("manifest")
    for item in tag:
        fqName = item.getAttribute("package")
        return fqName

def extractAppLabel(manifest,path):
    tag = manifest.getElementsByTagName("application")
    for item in tag:
        appName = item.getAttribute("android:label")
        if len(appName) > 0:
            if appName[0] == "@":
                return getResource(path, appName)
            else:
                return appName
    else:
        return "DATA NOT FOUND"

def validateManifest(filePath):
    if os.path.exists(filePath+"\\AndroidManifest.xml"):
        return True
    else:
        writeError("XXXXX"+filePath, "Unable to find AndroidManifest.xml")

def writeError(filePath, message):
    f = open("Extraction_error_log.txt", 'a')
    f.write(filePath + " -> " + message+"\n")
    f.close()

def getResource(path, appName):
    if os.path.exists(path+"\\res\\values\\strings.xml"):
        try:
            strings = minidom.parse(path+"\\res\\values\\strings.xml")
        except:
            return "DATA NOT FOUND - ERROR PARSING XML FILE"
        elements = strings.getElementsByTagName("string")
        for tag in elements:
            if tag.getAttribute("name") == appName[8:]:
                return tag.firstChild.nodeValue
        return "DATA NOT FOUND - No STRINGS.XML"
    else:
        return "DATA NOT FOUND - NO STRINGS.XML"

#extractData(dataDirectory,markets[0])
#extractData(dataDirectory,markets[1])
#extractData(dataDirectory,markets[2])
#extractData(slideMeDirectory, slideMeDirs[0])
#extractData(slideMeDirectory, slideMeDirs[1])
#extractData(slideMeDirectory, slideMeDirs[2])
#extractData(slideMeDirectory, slideMeDirs[3])
#extractData(slideMeDirectory, slideMeDirs[4])

```

DBWriter

```
'''
Created on Mar 29, 2012

@author: Billy Symon
'''
import MySQLdb as mdb
import sys

def connect():
    return mdb.connect('localhost', 'root', '', 'thesis');

def writeError(message):
    f = open("DB_write_error_log.txt", 'a')
    f.write(message+"\n")
    f.close()

def executeDBCommand(command):
    con = connect()
    try:
        with con:
            cur = con.cursor()
            cur.execute(command)
    except mdb.Error, e:
        writeError(command + "Resulted in Error %d: %s" % (e.args[0],e.args[1]))
    except:
        writeError ("!!!!!! Unexpected error:" + str(sys.exc_info()[0]) + " When
Executing Command "+command)
    finally:
        if con:
            con.close()

def writeAppInfoTableEntry(fileName, market, fileSize, appLabel, appFQName):

    command = "INSERT INTO appinfo VALUES(\""+fileName+"\", \" "+market+"\",
\""+str(fileSize)+"\", \" "+appLabel+"\", \" "+appFQName+"\")"
    executeDBCommand(command);

def writePermissionsInfoTableEntry(fileName, numPermissionsUsed, numPermissionsSetUp,
numPermissionsRequired):
    command = "INSERT INTO permissionsinfo VALUES(\""+fileName+"\", \"
"+str(numPermissionsUsed)+"\",
\""+str(numPermissionsSetUp)+"\", \" "+str(numPermissionsRequired)+"\")"
    executeDBCommand(command);

def writePermissionsRequestedTableEntry(fileName, permissionsUsed):
    for permission in permissionsUsed:
        command = "INSERT INTO permissions_requested VALUES(\""+fileName+"\", \"
"+permission+"\")"
        executeDBCommand(command);

def writePermissionsSetUpTableEntry(fileName, permissionsSetUp):
    for permission in permissionsSetUp:
        command = "INSERT INTO permissions_setup VALUES(\""+fileName+"\", \"
"+permission+"\")"
        executeDBCommand(command);

def writePermissionsRequiredTableEntry(fileName, permissionsRequired):
    for permission in permissionsRequired:
```



```

        command = "INSERT INTO permissions_required VALUES(\""+fileName+"\", \"
"+permission+"\") "
        executeDBCommand(command);

def writeIntentsTableEntry(fileName, numActivities, numServices, numReceivers,
numProviders):
    command = "INSERT INTO intents VALUES(\""+fileName+"\", \"
"+str(numActivities)+"\", \""+str(numServices)+"\", \""+str(numReceivers)+"\",
\""+str(numProviders)+"\") "
    executeDBCommand(command);

def writeActivitiesTableEntry(fileName, activities):
    for activity in activities:
        command = "INSERT INTO activities VALUES(\""+fileName+"\", \" "+activity+"\") "
        executeDBCommand(command);

def writeServicesTableEntry(fileName, services):
    for service in services:
        command = "INSERT INTO services VALUES(\""+fileName+"\", \" "+service+"\") "
        executeDBCommand(command);

def writeReceiversTableEntry(fileName, receivers):
    for receiver in receivers:
        command = "INSERT INTO receivers VALUES(\""+fileName+"\", \" "+receiver+"\") "
        executeDBCommand(command);

def writeProvidersTableEntry(fileName, providers):
    for provider in providers:
        command = "INSERT INTO providers VALUES(\""+fileName+"\", \" "+provider+"\") "
        executeDBCommand(command);

def writeAdditionalInfoTableEntry(fileName, numLibraries, numLayouts, numStrings,
minSDKLevel, altLayouts, altStrings):
    command = "INSERT INTO additional_info VALUES(\""+fileName+"\", \"
"+str(numLibraries)+"\", \""+str(numLayouts)+"\", \""+str(numStrings)+"\",
\""+minSDKLevel+"\", \""+altLayouts+"\", \""+altStrings+"\") "
    executeDBCommand(command);

def writeLibrariesTableEntry(fileName, libraries):
    for library in libraries:
        command = "INSERT INTO libraries VALUES(\""+fileName+"\", \" "+library+"\") "
        executeDBCommand(command);

def writeMasterEntry(fileName, market, fileSize, appLabel, appFQName,
numPermissionsUsed, numPermissionsSetUp, numPermissionsRequired, numActivities,
numServices, numReceivers, numProviders, numLibraries, numLayouts, numStrings,
minSDKLevel, altLayouts, altStrings):
    command = "INSERT INTO master VALUES(\""+fileName+"\", \" "+market+"\",
\""+str(fileSize)+"\", \""+ appLabel+"\", \""+appFQName+"\", \"
"+str(numPermissionsUsed)+"\",
\""+str(numPermissionsSetUp)+"\", \""+str(numPermissionsRequired)+"\", \"
"+str(numActivities)+"\", \""+str(numServices)+"\", \""+str(numReceivers)+"\",
\""+str(numProviders)+"\", \" "+str(numLibraries)+"\", \""+str(numLayouts)+"\",
\""+str(numStrings)+"\", \""+minSDKLevel+"\", \""+altLayouts+"\", \""+altStrings+"\") "
    executeDBCommand(command);

def writeLanguageLocDataEntry(fileName, locData):
    command = "INSERT INTO language_loc_data VALUES(\""+fileName+"\", \"
"+locData+"\") "
    executeDBCommand(command);

def writeLayoutLocDataEntry(fileName, locData):
    command = "INSERT INTO layout_loc_data VALUES(\""+fileName+"\", \" "+locData+"\") "

```

```
executeDBCommand(command);

def writeDrawableLocDataEntry(fileName, locData):
    command = "INSERT INTO drawable_loc_data VALUES(\""+fileName+"\",\""+locData+"\")"
    executeDBCommand(command);
```

Language Localization Extraction

```
'''
Created on Apr 14, 2012

@author: Billy Symon
'''
import os
import DBWriter
import re

dataDirectory = "C:\\Users\\Billy Symon\\Desktop\\Grad Research\\GRAD THESIS
[BACKUP_DO_NOT_USE]\\Decompiled Files\\\"
markets = ["appsforadam", "andapponline", "apptown"]

slideMeDirectory = "C:\\Users\\Billy Symon\\Desktop\\Grad Research\\GRAD THESIS
[BACKUP_DO_NOT_USE]\\Decompiled Files\\slideme\\\"
slideMeDirs = ["1", "2", "3", "4", "5"]

def extractLanguageLocData(location, market):
    path = location + market
    files = os.listdir(path)
    directorySize = len(files)

    print "!"*150
    print "NOW EXTRACTING DATA FROM " + path
    print str(directorySize)+" APPS FOUND"
    print "!"*150

    i = 1
    #languageData = []
    for f in files:
        languages = []
        resFilePath = path + "\\\"+ f + '\\res'
        #PRINTING HEADER
        print "**"*50
        print "EXTRACTING DATA FROM FILE "+ str(i) + " OF "+ str(directorySize)
        print "**"*50
        i += 1

        if os.path.exists(resFilePath):
            resContents = os.listdir(resFilePath)
            for content in resContents:
                #if content[0:7]=="values-":
                if (re.match("values-..$",content) or re.match("values-..-r.",
content)):
                    languages.append(content[7:])
            filename = f+".apk"
            if len(languages) > 0:
                for language in languages:
                    DBWriter.writeLanguageLocDataEntry(filename, language)
                print filename
                print languages

extractLanguageLocData(slideMeDirectory, slideMeDirs[4])
```

Drawable and Layout Localization Extraction

```
'''
Created on Apr 14, 2012

@author: Billy Symon
'''
import os
import DBWriter
import re

dataDirectory = "C:\\Users\\Billy Symon\\Desktop\\Grad Research\\GRAD THESIS
[BACKUP_DO_NOT_USE]\\Decompiled Files\\\"
markets = ["appsforadam", "andapponline", "apptown"]

slideMeDirectory = "C:\\Users\\Billy Symon\\Desktop\\Grad Research\\GRAD THESIS
[BACKUP_DO_NOT_USE]\\Decompiled Files\\slideme\\\"
slideMeDirs = ["1", "2", "3", "4", "5"]

def extractLanguageLocData(location, market):
    path = location + market
    files = os.listdir(path)
    directorySize = len(files)

    print "!*"150
    print "NOW EXTRACTING DATA FROM " + path
    print str(directorySize)+" APPS FOUND"
    print "!*"150

    i = 1
    for f in files:
        layouts = []
        drawable = []
        resFilePath = path + "\\\"+ f + '\\res'
        #PRINTING HEADER
        print "**"50
        print "EXTRACTING DATA FROM FILE " + str(i) + " OF " + str(directorySize)
        print "**"50
        i += 1

        if os.path.exists(resFilePath):
            resContents = os.listdir(resFilePath)
            for content in resContents:
                #if content[0:7]=="values-":
                if re.match("layout-", content):
                    layouts.append(content[7:])
                if re.match("drawable-", content):
                    drawable.append(content[9:])
            filename = f+".apk"
            print filename
            if len(layouts) > 0:
                for layout in layouts:
                    DBWriter.writeLayoutLocDataEntry(filename, layout)
                print "layouts"
                print layouts
            if len(drawable) > 0:
                for draw in drawable:
                    DBWriter.writeDrawableLocDataEntry(filename, draw)
                print "drawable"
                print drawable
```

```
extractLanguageLocData(slideMeDirectory, slideMeDirs[4])
```

Permission Extraction

```
'''
Created on Mar 31, 2012

@author: Billy Symon
'''
import MySQLdb as mdb
import sys

markets = ["appsforadam", "andapponline", "apptown", "slideme"]

permissions = ["ACCESS_CHECKIN_PROPERTIES",
               "ACCESS_COARSE_LOCATION",
               "ACCESS_FINE_LOCATION",
               "ACCESS_LOCATION_EXTRA_COMMANDS",
               "ACCESS_MOCK_LOCATION",
               "ACCESS_NETWORK_STATE",
               "ACCESS_SURFACE_FLINGER",
               "ACCESS_WIFI_STATE",
               "ACCOUNT_MANAGER",
               "ADD_VOICEMAIL",
               "AUTHENTICATE_ACCOUNTS",
               "BATTERY_STATS",
               "BIND_APPWIDGET",
               "BIND_DEVICE_ADMIN",
               "BIND_INPUT_METHOD",
               "BIND_REMOTE_VIEWS",
               "BIND_TEXT_SERVICE",
               "BIND_VPN_SERVICE",
               "BIND_WALLPAPER",
               "BLUETOOTH",
               "BLUETOOTH_ADMIN",
               "BRICK",
               "BROADCAST_PACKAGE_REMOVED",
               "BROADCAST_SMS",
               "BROADCAST_STICKY",
               "BROADCAST_WAP_PUSH",
               "CALL_PHONE",
               "CALL_PRIVILEGED",
               "CAMERA",
               "CHANGE_COMPONENT_ENABLED_STATE",
               "CHANGE_CONFIGURATION",
               "CHANGE_NETWORK_STATE",
               "CHANGE_WIFI_MULTICAST_STATE",
               "CHANGE_WIFI_STATE",
               "CLEAR_APP_CACHE",
               "CLEAR_APP_USER_DATA",
               "CONTROL_LOCATION_UPDATES",
               "DELETE_CACHE_FILES",
               "DELETE_PACKAGES",
               "DEVICE_POWER",
               "DIAGNOSTIC",
               "DISABLE_KEYGUARD",
               "DUMP",
               "EXPAND_STATUS_BAR",
               "FACTORY_TEST",
               "FLASHLIGHT",
               "FORCE_BACK",
               "GET_ACCOUNTS",
```

"GET_PACKAGE_SIZE",
"GET_TASKS",
"GLOBAL_SEARCH",
"HARDWARE_TEST",
"INJECT_EVENTS",
"INSTALL_LOCATION_PROVIDER",
"INSTALL_PACAKGES",
"INTERNAL_SYSTEM_WINDOW",
"INTERNET",
"KILL_BACKGROUND_PROCESSES",
"MANAGE_ACCOUNTS",
"MANAGE_APP_TOKENS",
"MASTER_CLEAR",
"MODIFY_AUDIO_SETTINGS",
"MODIFY_PHONE_STATE",
"MOUNT_FORMAT_FILESYSTEMS",
"MOUNT_UNMOUNT_FILESYSTEMS",
"NFC",
"PERSISTENT_ACTIVITY",
"PROCESS_OUTGOING_CALLS",
"READ_CALENDAR",
"READ_CONTACTS",
"READ_FRAME_BUFFER",
"READ_HISTORY_BOOKMARKS",
"READ_INPUT_STATE",
"READ_LOGS",
"READ_PHONE_STATE",
"READ_PROFILE",
"READ_SMS",
"READ_SOCIAL_STREAM",
"READ_SYNC_SETTINGS",
"READ_SYNC_STATS",
"REBOOT",
"RECEIVE_BOOT_COMPLETED",
"RECEIVE_MMS",
"RECEIVE_SMS",
"RECEIVE_WAP_PUSH",
"RECORD_AUDIO",
"REORDER_TASKS",
"RESTART_PACKAGES",
"SEND_SMS",
"SET_ACTIVITY_WATCHER",
"SET_ALARM",
"SET_ALWAYS_FINISH",
"SET_ANIMATION_SCALE",
"SET_DEBUG_APP",
"SET_ORIENTATION",
"SET_POINTER_SPEED",
"SET_PREFERRED_APPLICATIONS",
"SET_PROCESS_LIMIT",
"SET_TIME",
"SET_TIME_ZONE",
"SET_WALLPAPER",
"SET_WALLPAPER_HINTS",
"SIGNAL_PERSISTENT_PROCESSES",
"STATUS_BAR",
"SUBSCRIBED_FEEDS_READ",
"SUBSCRIBED_FEEDS_WRITE",
"SYSTEM_ALERT_WINDOW",
"UPDATE_DEVICE_STATUS",
"USE_CREDENTIALS",
"USE_SIP",
"VIBRATE",

```

"WAKE_LOCK",
"WRITE_APN_SETTINGS",
"WRITE_CALENDAR",
"WRITE_CONTACTS",
"WRITE_EXTERNAL_STORAGE",
"WRITE_GSERVICES",
"WRITE_HISTORY_BOOKMARKS",
"WRITE_PROFILE",
"WRITE_SECURE_SETTINGS",
"WRITE_SETTINGS",
"WRITE_SMS",
"WRITE_SOCIAL_STREAM",
"WRITE_SYNC_SETTINGS"]

zeroValuePermissions = ["ADD_VOICEMAIL",
                        "BIND_REMOTEVIEWS",
                        "BIND_TEXT_SERVICE",
                        "BIND_VPN_SERVICE",
                        "BRICK",
                        "BROADCAST_PACKAGE_REMOVED",
                        "DIAGNOSTIC",
                        "DUMP",
                        "FACTORY_TEST",
                        "FORCE_BACK",
                        "INSTALL_PACKAGES",
                        "READ_HISTORY_BOOKMARKS",
                        "READ_PROFILE",
                        "READ_SOCIAL_STREAM",
                        "REBOOT",
                        "SET_ALWAYS_FINISH",
                        "SET_DEBUG_APP",
                        "SET_POINTER_SPEED",
                        "SET_PROCESS_LIMIT",
                        "SIGNAL_PERSISTENT_PROCESSES",
                        "UPDATE_DEVICE_STATS",
                        "WRITE_APN_SETTINGS",
                        "WRITE_HISTORY_BOOKMARKS",
                        "WRITE_PROFILE",
                        "WRITE_SOCIAL_STREAM"]

def connect():
    return mdb.connect('localhost', 'root', '', 'thesis');

def executeDBCommand(command):
    con = connect()
    try:
        with con:
            cur = con.cursor()
            cur.execute(command)
            value = cur.fetchone()
            return value[0]
    except mdb.Error, e:
        writeError(command + "Resulted in Error %d: %s" % (e.args[0],e.args[1]))
    except:
        writeError ("!!!! Unexpected error:" + str(sys.exc_info()[0]) + " When
Executing Command "+command)
    finally:
        if con:
            con.close()

def writeError(message):
    f = open("Data_Mining_log.txt", 'a')
    f.write(message+"\n")

```



```

f.close()

def getRequestedPermissionCountByMarket(permission, market):
    return executeDBCommand("SELECT COUNT(*) FROM appinfo, permissions_requested WHERE
appinfo.filename = permissions_requested.filename AND appinfo.market = \" \" + market + \" \"
AND permissions_requested.permission = \" android.permission.\" + permission + \" \")

def getTotalRequestedPermissionCount(permission):
    return executeDBCommand("SELECT COUNT(*) FROM permissions_requested WHERE
permission = \" android.permission.\" + permission + \" \")

for permission in zeroValuePermissions:
    print "~"*50
    print "Permission -> " + permission
    print "~"*50
    for market in markets:
        print market + " -> " + str(getRequestedPermissionCountByMarket(permission,
market))
    print "TOTAL -> " + str(getTotalRequestedPermissionCount(permission))
    print "*" * 50

```

File Information Extraction

```
'''
Created on Mar 21, 2012

@author: Billy Symon
'''
import os
import re

appNameExp = "<string name=\"app_name\">(.*?)</string>"
manifestNameExp = "<application android:label=\"(.*?)\">"

filepath = "C:\\Users\\Billy Symon\\Desktop\\Grad Research\\GRAD THESIS
[BACKUP_DO_NOT_USE]\\Decompiled Files\"
market = "appsforadam\"

dirs = os.listdir(filepath+market)
numFiles = len(dirs)

def getSize(start_path):
    total_size = 0
    for dirpath, dirnames, filenames in os.walk(start_path):
        for f in filenames:
            fp = os.path.join(dirpath, f)
            total_size += os.path.getsize(fp)
    return total_size

def getAppName(appDirectory):
    #print filepath+market+dir+"\\res\\values\\strings.xml"
    if os.path.exists(filepath+market+dir+"\\res\\values\\strings.xml"):
        stringsXML = open(filepath+market+dir+"\\res\\values\\strings.xml")
        XML = stringsXML.read()
        stringsXML.close()
        appName = re.findall(appNameExp, XML)
        if len(appName)>0:
            return appName[0]
    if os.path.exists(filepath+market+dir+"\\AndroidManifest.xml"):
        manifestFile = open(filepath+market+dir+"\\AndroidManifest.xml")
        manifest = manifestFile.read()
        manifestFile.close()
        appName = re.findall(manifestNameExp,manifest)
        if len(appName)>0:
            return appName[0]
        else:
            return "NO NAME MENTION IN MANIFEST FILE"
    return "NAME NOT FOUND"

i = 1
for dir in dirs:
    print "*****"
    print "-----App "+str(i)+" of "+str(numFiles)+"-----"
    marketName = market[:-1]
    print "Market Name -> " + marketName
    fileName = dir
    print "File Name/Key -> " + fileName
    try:
        size = getSize(filepath+market+dir)
    except:
        size = "UNKNOWN"
```

```
print "Size in bytes -> " + str(size)
appName = getAppName(filepath+market+dir)
print "App name -> "+ appName
i += 1
print "*****"
```