The Pennsylvania State University
The Graduate School

ANDROID MARKET: LARGE SCALE RECONSTRUCTION AND ANALYSIS

A Thesis in
Computer Science and Engineering
by
Matthew Dering

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2014

The thesis of Matthew Dering was reviewed and approved* by the following:

Patrick McDaniel
Professor of Computer Science and Engineering
Thesis Advisor

Sencun Zhu
Associate Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Department Chair

*Signatures are on file in the Graduate School.

Abstract

Smartphones have experienced a boom over the last 10 years. Currently a user is left to trust their device manufacturer when installing applications. In this paper we outline our suite of applications that are designed to provide us and users information about applications leveraging a reversed API for the Google Play market, working in concert with several analysis tools, some of our own design. In order to gather as complete of a picture of the market as possible, we propose a method for recreating the market in its entirety. Using this procured data, we present an overview of how two popular Android features are used in the real world: Permissions and Libraries. We find that there is a large correlation between library use and Permission use, which represents a security concern.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank Bradley Reaves for first introducing me to the software upon which I built my vast app empire. Without his patient explanation of asynchronous communication and serialized data, none of this would have been possible. Additionally, without Steve McLaughlin's unparalleled ability to tell me what is and is not interesting, I do not know where I might have ended up, but I can be sure it would not have constituted real research. A big thank you as well to Will Enck, for providing me the benefit of his years of experience pioneering and shaping the face of android security research. Finally, my advisor, Dr. Patrick McDaniel, who finally taught me how to write where so many had tried and failed previously, for providing me these excellent opportunities. Thank you all.

Dedication

I dedicate this work to my family. My mother, Jackie, who listened patiently to my rants and raves on this topic and who had the grace to not stop me when I at times exceeded her depth, my father, Ed, for encouraging me to change advisors, and for galvanizing me to pursue my PhD, and to Harley and Luke, who are an inspiration to me each and every day. I would also like to dedicate this to Caroline Price, whose care and affection continue to guide me through the jungle that is graduate research.

Chapter 1

Introduction

1.1 Introduction

The rising popularity of smartphones has changed the way that people and enterprises use computers. This sort of access to computers has given rise to untold levels of productivity and convenience in every industry, government and for private citizens as well. It is estimated there are currently over 1 billion smartphones in use worldwide. Nearly all of these have gone into circulation in roughly the last 5-8 years, virtually assuring that the security behind the smartphones cannot keep up with the technology as it becomes available.

A major selling point for modern smartphones is their operating systems. In order to be considered a smart phone, a phone will generally contain a full featured operating system that features a web browser and email. Most smart phones will also include other functionality such as music and movie playback, and cameras. Some popular examples of this are made by RIM, Symbian, Google's android, and Apples iOS.

Over the past 5 years, smartphones have evolved to be more and more full featured. This has lead to the inclusion of app stores on the phone, and by extension apps created by 3rd party developers. Apps can be used to perform many functions. Some are as simple as a tip calculator. Others have more varied uses, including sound recorders, different keyboards to allow for alternative input methods, social networks like twitter and facebook, photo editing, sports scores and games. Apps can be of varying degrees of sensitivity. Many applications are banking applications or are related to confidential trade or sales information.

These applications can present appealing attack vectors, and are prone to security risks For example, this sound recorder or keyboard may be accessing the internet in the background and violating user privacy in a big way. In fact, previous attacks have been shown that can exfiltrate information from such applications using other covert channels, without the users knowledge, or consent to allowing an application permission to do something.

Our focus in this paper is applications on the google play app store. Since it comes installed with almost every handset and tablet, Google play is the most popular digital app distribution marketplace for android. This service allows users to download applications as well as music, movies, and books, directly to their device.

In order to fully understand the scope of this problem, it became incumbent on us to create a large dataset of android applications. To do this proved to be very nontrivial. In this paper we propose our large scale android application analysis pipeline that enables us to simultaneously build and analyze a large dataset, and provide easy to digest findings for many of the hundreds of thousands of applications in the Google Play store.

1.2 Google Play Market

In order to amass this large dataset, we decided to concentrate on the google play app store. However, it is by no means the only option. Apps for other app stores can be downloaded through google play. Additionally, apps can be installed directly from the web or from SD cards. However, the most popular method is through the google play market. The app store market has garnered praise from many circles for its ease of use and impressive infrastructure. Previous phones with application capability typically required the user to visit a website either on their phone or their computer and then transfer it. It also left the e-commerce portion of paid applications to the developer, something most of them were not equipped to handle. With all this work and infrastructure required, it's no wonder that the application model struggled on older platforms.

The app store model changed all of this. The ease of access for the users, and the tools available to the developers opened up a whole new world of software. The developer aspect of this transformation cannot be understated. Android comes with an SDK that makes it easier than ever to develop applications for it. The SDK is written in java and comes in the form of an eclipse plugin and several companion applications, including a dalvik translator (more on that later), an android emulator, and a tool to access the command line interface on an attached phone.

Once an application has been developed and is prepared for submission, the developer must go through the google vetting process. First the application must meet quality standards put forth by the market. Notably this includes including design guidelines such as UI patterns, icons, and navigation standards. It also requires certain functionality to be minimized except where relevant. For example, SMS access or music playback. It also sets forth Performance and Stability standards such as not crashing when possible, and offers onscreen feedback for operations taking longer than 2 seconds.

Finally, all code must be signed by the developer. Google does not require that the code be signed by any authority, so self-signing is sufficient. Once this is accomplished the developer can upload the apps to the store. These APK files must be smaller than 50 MBs in size, and must be given a unique package name. These generally follow regular java conventions, such as com.google.chat. It is also worth noting that developers may upload multiple APKs under the same package name and listing each targeted to different devices. The developer provides the store with several different pieces of listing details, including language, title, description, at least 2 screenshots and other metadata such as category and type.

After submission, the application must be approved by google. The application that attempts to accomplish this is known as the bouncer. Previous work has explored this in more detail, but from appearances it is a dynamic analysis tool running in a qemu environment that runs for a limited amount of time. Evading this bouncer was the focus of some of our work, which we discuss later on.

This process of coding with a full featured SDK, compiling, signing, uploading, and vetting, has made it so that it has never been easier for a developer to get applications into the mainstream. The Market's search and various web integrations make it easier than ever for developers to get started. As we will soon see, this is both a good thing and a bad thing, and many issues are still available in the wild today.

1.3 Privileges

An android APK is a zip archive, and contained with the archive are the important pieces of the application. Nearly everything needed to run is contained within the APK, including layouts, any images, other resources, such as graphical files. Most importantly, each archive contains a file called ”AndroidManifest.xml” (the manifest) and the ”classes.dex” (the binary)

The Android Manifest is a file that declares all the resources to be used by a file as well as all the screens/intents (an activity) that an application will open, send or receive. The Manifest contains all of the most important information

It also declares application permissions. In order for an application access a privileged portion of the device, or to perform a privileged operation. The contents of the permissions portion of this file are included in application descriptions on the market and users are asked to accept these permissions on installation. If a developer fails to disclose that an application will need a certain permission, say for bluetooth, and if that application attempts to invoke code that requires a permission that doesn't appear in their permissions database entry, then the application will crash.

1.4 Design

In this section, we provide a general description of the architecture of our application analysis engine. Note that each component operates more or less independently of the other. This will allow us to easily make alterations to each component with minimal interruption. This also allows for significant downtime for any given component without interruption to our other components. Finally, this will allow for upgrades to be put in place and results seen more or less immediately.

Any upgrade made will bubble up to the final product in at most a week. It is for this reason that we propose the following construction, each dependent on the last in some way, but with minimal interprocess dependency and using only persistent storage, we are able to ensure that if any component is interrupted or altered, the subsequent components will not be affected.

The Core elements of the system are as follows:

- Account spoofing. This allows us to impersonate an account and act as if we are the account, including browsing and downloading. This will pass an authorization token to the following component.
- The Market Crawler, nicknamed DORA. Using this authorization, we navigate the marketplace gathering IDs stored in a mySQL database. These IDs are passed to:
- The Market Downloader, nicknamed BACKPACK. Using the provided IDs, and Authorizations we are able to gather a large amount of binaries in the form of apk files. These are stored to disk.

From here, we are able to perform many operations on our dataset. At present, we have focused on work with the permissions needed by the application and the libraries included in each binary. To ensure that such analyses scale well, we constrained our work to only permissions provided and enforced by Android OS, and

Chapter 2

Market Crawler

2.1 Introduction

In our design section, we discussed the necessity for a market crawler as a component for our engine. We designed this to be as robust as possible, with minimal data loss, so that the crawler will not only serve as a means to acquire binaries and continue with our engine, but as a successful reconstruction of the metadata available for all applications in the app store.

The android market boasts over 800,000 applications in it. Included in each application entry is many important pieces of metadata, including description, creator, price, count of ratings, and permissions requested by the application. This information seemed important enough to merit further study, and so we decided to store this information along with the applicationID's that allow for download. The goals of this component application are twofold: metadata acquisition for analysis, and enabling app download.

Our API closely mirrors the front end provided by a stock vending.apk application, and this phase is no exception. Typically when a device intends to search for an android app, it does so in one of 2 ways: through a google provided list, or through the search function of the market app on the device. For example if a user opens up the market app, it will give the user a list of popular apps, categories, and a search field. Likewise, our API allowed for requests of lists of applications from these sort types (such as popular and new) and categories (such as finance and social). Additionally, these sort types are available for each of the 34 categories.

Additionally, the user will also see the search bar at the top, which is used for a more directed search, for example, for "Angry Birds". Similarly, our API provides functionality to search for specific applications. Google will return the results to us in list form, sorted by their relevance algorithm, which can be somewhat arbitrary, however.

With these two functionalities in mind, we developed a 2 pronged approach to marketplace reconstruction. Our first prong is the more straightfoward: provided queue monitoring. Recall that each category has 3 lists provided for it (new, popular and featured), our task was to repeatedly query them to completion. It isn't clear exactly how they work, but the new queue lists all applications in that category added recently, typically around 300 of them, the popular queue is 500 applications in size, sorted somewhat arbitrarily, and the featured queue is 10 applications in size. Using cron jobs, we scheduled a crawl of each of these every 4 hours, a process that took roughly 4 hours to complete, thus running more or less continuously. This also enabled us to make better guarantees about not allowing new applications into the market without our knowledge.

However, it should be clear from our description that this will not be sufficient for our purposes. Currently the android app store boasts more than 800,000 applications, whereas in the best case this method offers approximately 28,000 applications for discovery. While there may not be any overlap among these applications, this is still woefully inadequate for our purposes, necessitating further work on our part.

Our second prong is more complex: directed word searching. Recall that our API provided a way to directly query the app store. Under ideal conditions, a very common word such as "the" or "a" could be repeatedly queried until we were satisfied that we had obtained the vast majority of applications. Indeed, from our findings, we see that 81% of applications contain the word "the". Unfortunately, the query results are quite constrained.

The first way in which they are constrained has already been covered but bears repeating: they are sorted by relevance. This sorting is somewhat arbitrary and does not provide us with much information about how to best understand what we are seeing when we search for words.

The second constraint in our API is that we are only given 500 search results. The reasons for this are not clear, its is either to prevent attacks like the ones described in this paper, or to avoid excess processing power in indexing. This does however create a significant boundary against metadata accumulation. Another way of looking at this is as an easier form of the Minimal Vertex Cover problem of a bipartite graph. The minimal vertex cover problem is to find the minimal set of vertices in a graph such that each vertex in the graph is incidental to at least one of the vertices in the graph.

Imagine a bipartite graph with 2 columns of vertices. One column corresponds to each word in the english language. The other column corresponds to each android app, sorted by popularity. Finally, connect the vertices from the dictionary column to the first 500 entries application column whose description, title, or author name contains the word in question. Our challenge becomes to maximize the number of vertices covered in the application column with the minimal number of words from the dictionary column. We were unable to find what the optimal solution would be, since we were not made aware of the sorting criteria nor the sum total of the applications requiring discovery, we did not attempt to use any existing solutions to this class of problem. We discuss further how we approximated the solution to this problem in later sections.

2.2 Message Exchange

Our application was built around a set of messages and responses exchanged with the google servers. Sending a message of the correct format will result in receiving a response from the server. These formats are specified by a method known as protocol buffers. Protocol buffers are a method of serializing Structured data designed and developed by google consisting of two main components

The first part is the protobuf description language. This is the language in which the messages (or data) is defined. To many, it will resemble a C struct or something similar to this. It allows you to define messages, and the fields inside of them. Most datatypes are allowed, such as Signed integers, unsigned integers, Booleans, Strings, doubles, and floats. It also allowes for arbitrary byte strings, which can be used for things like images. It also defines C-like Enum fields where you can define your own datatype inside of the message definition.

The second part of this method is the protocol compiler. This is an application created by google that will compile the .proto files from the first portion into java, c++ or php code. The resulting code will contain classes for each of the message objects defined by your protocol buffer, all of which will implement a Message interface and extend a GeneratedMessageClass. Each class will have a way to create a new builder, which is how messages are created.

The compiler will create getters and setters for every field, however the setters can only be used by the Builder class. Using this generated class is how we were able to successfully communicate with Google Servers. Using protocol buffers reverse engineered, we were able to compile them to Java code and access them using these automatically generated methods.

With these serialized structures in place, the problem becomes exchanging messages (in our case, requests and responses) with the google Play API. We are provided with the following messages and data structures by our reverse engineered protobufs:

Table 2.1: Messages passed in the Discovery process

Functionality	Request	Response	Data Structure
Make Requests	RequestContext	ResponseContext	
Search	AppsRequest	AppsResponse	App
Browse Categories	CategoriesRequest	CategoriesResponse	Category
Browse SubCategories	SubCategoriesRequest	SubCategoriesResponse	Category
Read Comments	CommentsRequest	CommentsResponse	Comment
View Preview Images	GetImageRequest	GetImageResponse	ImageData

Constructing and sending these messages proved to be non-trivial. From start to finish, the process of metadata acquisition proceeds as follows:

- Recall that there are 2 ways to explore the Google play store, and we leverage both: search, and browse.
- Every message sent is in a protobuf known as a request. The request has 2 portions: the context and the specific request. It may contain only one of the types of requests named above.
- With this in mind, in order to browse the store, we must next find out the categories. To do this, we need to compile a request object, with the context, and a CategoriesRequest. This message has no fields
- The server will respond with a CategoriesResponse that consists of 2 categories: Applications and Games.
- The next step to successful browsing is to create a SubCategoriesRequest for each of the two categories. As before, this will also have the context. All requests will have the context included, so I will omit this going forward for brevity.
- The server will respond with a response of the subcategories of the requested category, in iteratable list format.
- Finally, we will issue an AppsRequest, for the correct subcategory, and for the requested OrderType, and free/paid. Additionally, you must also specify the number of entries you would like to view, and what number entry you would like to start with.
- The server will reply with a list of the requested number of applications matching the category criteria given.
- Alternatively in order to search for specific things, these CategoriesRequests and SubcategoryRequests do not need to be issued. Instead, only an AppsRequest needs to be sent.
- AppsRequest has a field called query. Set the query to the search term you are searching for, and send it. It's worth noting at this point that Google

does not currently support the drill down much on these types of requests. If a request is sent with a populated query field, the only other field that may be populated is the Free/Paid field. Populating any of the other fields will result in a null response, as google does not support these types of searches.

Each of these will return a protobuf with a de-serializable list of Apps that contains most of the metadata, including creator, ratings count, rating, permissions, price, and other fields. Some notable absences are category and a precise download count. Instead, the store has some approximate buckets that these fall under, the smallest being under 50 and largest being over 250,000. Most importantly though is the unique id field returned for each of these applications. This is discussed in more in following sections

2.3 App Store Reconstruction

The first step for library recreation was to discover as many applications as possible. Since there is no publicly available master list, the reconstruction needed to be done in a more clever fashion. The majority of this is accomplished using the search functionality. Going forward, whenever a search request results in an app that was not previously known, we refer to the app as being "discovered".

With all this in mind, we attempted to reconstruct the Google Play app store in an efficient and expeditious fashion by leveraging the API outlined above. To accomplish this we adopted two approaches: queue monitoring and targeted wordlist searching. As outlined above, the queues are specialized, per category lists available for perusal, typically used to find new apps or popular apps of a certain type. We implemented a scanner which would iterate over all available categories and queues provided and exhaustively search them, which would run every 4 hours. This way we were able to ensure that, among other things, new apps were not added to the store without our knowledge.

Searching was the focal point of our reconstruction efforts, and presented several unique challenges. Previously we mentioned that there are strict limitations placed on the search results. The reasons for this are unclear, but performance concerns or competitive advantages may best explain some of the choices made. One of the limitations was that the search results were returned sorted in an arbitrary way. It's possible they use the google pagerank algorithm, which is a mix of relevance and popularity, but there's no way to be sure, and as such we were forced to assume they were in some hidden total ordering maintained on the server, and not transparent to us.

The other challenge we faced concerned the results themselves. Each search query returned metadata for 10 apps at a time, and this could be repeated up to 50 times for a maximum of only 500 apps per word. Without these restrictions, reconstructing this database would be fairly trivial. For example, picking several very common words, would likely result in the majority of the applications. A search of our database reveals that in absence of these limitations, an exhaustive for 'the', 'be', and 'and' will match approximately 90% of apps based on Title and Description alone (reasonably, every english application will contain one of these 3 words). With this rate limiting, a search of these 3 words will result in a maximum of 1500 applications, and since their results will likely overlap heavily, probably significantly less than that.

Let us consider the example above as a starting point, but for brevity's sake, only the first 10 results of each search will be discussed. Searching for 'the' may yield the 10 most popular apps in the store containing the word 'the'. Since we are starting with an empty application database, all 10 of these will be added. This process may take approximately 2 seconds all told, for a rate of approximately 5 applications/second. Next we search for the word 'be', which will present us with the 10 most popular applications containing the word 'be'. Of these 10 applications, perhaps 6 may not have been included in the first search result, giving us 16 applications in our database. However, this search has taken another 2 seconds, which means our rate has fallen to 4 applications/second.

Finally we will search for the word 'and'. With these final results, we compare against the 16 already contained in our database. Since these are the 10 most popular applications containing the word ‘and‘, it’s not unreasonable to assume we may only find 2 new applications. After the 2 seconds required for this search, we have accumulated 18 applications in 6 seconds, for a rate of 3 applications/second. Each of the duplicate applications returned in search results represents a loss of efficiency. In fact, in our hypothetical example, the efficiency fell 40% in just a few short seconds because of these duplicate applications. It is this loss of efficiency that we sought to avoid.

This loss of efficiency can be very damaging when trying to recreate a database of over 1,000,000 applications. This can mean the difference between weeks and months of time taken to complete a reconstruction. Efficiency can be lost in other ways, as well. A request that is not replied to represents a loss of efficiency as well, as well as the sizable effort that goes into making each of these lists.

2.4 Word Selection

Currently the Google Play Market contains around 1.1 million applications. The lists outlined above would only encompass around 3% of this number, this left us with the other option of crawling the market manually using targeted word searches. This involved compiling around 50 different wordlists and exhaustively searching the words contained therein. Since each search incurred a not insignificant time delay, anywhere from 1 to 5 seconds, and each search was repeated up to 50 times per word, it was in our interest to minimize the amount of overlap between results from each word, as well as from one list to the next. Any previously discovered application encountered constitutes a loss of efficiency, one we could not afford given the scope of the task ahead, so our goal was to smartly target lists of words to satisfy 3 main intuitions. We constructed wordlists to be relevant, categorical, and with Low Redundancy.

2.4.1 Relevant

Relevance is an important part of selecting a good word for query. Since there is overhead incurred by issuing a query request, we want our queries to be productive. What this means is we would like to avoid issuing a query only to receive 0 results. It was important for us to not waste too much time searching for terms that were unlikely to bear any fruit at all. The API is structured in such a way so that even in the event of an unsuccessful search, it incurs the same wait time as a successful fetch of relevant results. Naturally, to launch an efficient attack, it was incumbent upon us not waste our time querying the store for things that did not seem likely to be related to applications. The converse of this is to try to emphasize words that seem likely to appear in applications. Categories such as ”fun leisure activities” and ”game genres” would satisfy this. On the other hand, a category such as ”list of named stars” would satisfy the other two requirements, but the number of apps containing the word ”alcor” is most likely very small, so it would not be a smart search term, and would thus result in lost efficiency.

2.4.2 Low Redundancy

This was the most challenging characteristic to maintain. Low redundancy means that a search is likely to result in a good amount of new applications. What defines a good amount can be hard to define, but a 10% return on new applications would not be unreasonable, though a higher number is expected early on in our reconstruction. Hence this became harder and harder to satisfy as our application ran. With this in mind, we attempted to identify things that would not likely return similar results to other search terms in the same list. This per-list level of granularity was all we could reasonably expect. Put another way, we would like to find words who are not liable to appear with other words in the same list in the same application.

This idea is derived directly from our two main constraints from the API, the arbitrary sorting, and the 500 result limit. Words that are in violation of this low redundancy requirement will result in very low efficiency. In fact, this is the largest source of inefficiency in our crawling procedure, with many sets of 10 results not containing . This characteristic was usually violated for one of two reasons: the words in the list were too common, or too specific. For example, a list of all prepositions is likely in violation of this since its likely that most app descriptions contain words such as ”as” or ”from” so these results will probably overlap, introducing a lot of lost time. Alternatively a list of too much specificity might also violate this. For example, a list of every part of a car, would likely only return car related apps, and since there would only be 500, and all sorted by something we do not understand, these would likely be redundant. In contrast, we had plenty of success searching for first names, last names, which are likely to give us application developer results, and city names, which were unlikely to appear. For example, an application is unlikely to contain both emily and katie, or new york and los angeles.

2.4.3 Categorical

The last criteria is categorical. What this means is we would like for simplicity's sake to have our word lists be related. This enables us to quickly and easily assemble good lists of words related to each other, without requiring manual labor, which is error prone and time consuming. Doing this keeping in mind the 2nd requirement will result in a large amount of applications with minimal loss of efficiency and without putting a huge requirement on the operator.

With these ideas in mind, we endeavored to build word lists of words that would be diverse enough to cover many things, but were not so vague as to appear in all of the same applications. This intuition leads to common words with some degree of specificity, so that applications are not liable to contain more than one of them. Some of the more successful lists we employed were: last names, colors, common typos, first names, adjectives, corporations, popular social networks, and top level domains.

Chapter 3

Market Downloader

3.1 Introduction

The next segment in our engine is the Market Downloader. In the previous section we detailed how we accumulated a large amount of market metadata in a short amount of time using our Crawler segment. Our next challenge encountered was how to leverage this information about the existence of apps into the corresponding binary. As in our previous section, this process mirrors a front end process. Normally the messages passed in this step are passed when the user consents to download and install an application in vending.apk. This allows an apk to be downloaded and is passed to a 2nd application for installation.

Our construction endeavors to duplicate this process without the benefit of a device or its associated software. In order to download the applications, we must have the appID. This is a unique Identifier for each package and version pair. In other words, each version of each app has a special appID. We have observed that at the outset of the store, each ID was a randon unsigned integer. However, at some point, possibly at the point of the google play migration from the old monicker of the android app store, the IDs became more meaningful. Currently, each appID follows the following pattern: ”v2:” then the packagename, then ”:1:” and finally the version code. With this in mind, it might be possible to automatically generate appIDs for older versions of applications. However, this is not within the scope of this work.

3.2 Message Passing

This step only made use of one message/response pair:

Table 3.1: Messages passed in the App Acquisition Process

Functionality	Request	Response	Data Structure
Download	GetAssetRequest	GetAssetResponse	App URL, Cookie

In order to acquire the apk file from the google server, first, the client device must determine the AppID of the app in question. This is done automatically by our market crawler. Recall that each request has a requestcontext appended to it. Previously our context has been obtained by logging in to the android service. However this is a second context obtained by logging in with the android-secure service. It's important to note that this is a second authorization token issued and the context carries a secure bit as well, meaning that the crawling and download process each carry different tokens and are for different services, bringing us to

The message exchange for the binary acquisition process is as follows:

- The first protobuf that needs to be sent is a GetAssetsRequest. This is sent to download a binary to the androidsecure service. It has just 1 field that needs to be set, assetId which is the ID obtained earlier. This field is the nexus of the metadata accumulation and the binary acquisition portions of our project. Once the assetID has been set, the protobuf can be sent

- At this point google will verify if the app download is authorized. The main reason this might fail is if the app is a paid app, in which case the response will be null. Google will respond with a protobuf called GetAssetResponse. This response includes some metadata, including a hash and a size. More importantly, It also contains a URL and a cookie.

- In order to download the binary, we must use these two fields. Posting to the URL with the cookie will result in the download of the app.

Additionally, we have implemented is support for old application downloads. In some events, if the app is out of date, the GET request to the URL above will not return the app, but rather will result in a redirect code. Parsing this redirect request results in a cross-protocol redirect to a google archive. Normally java will not allow this since it's a security risk. This allows us to download older apps, provided we have at some point discovered them.

3.3 Market Download Challenges

The process of app acquisition is itself nontrivial. We have previously detailed the message passing process that will result in app download. This exchange of messages will allow our application to download from the google app store directly to the computer in use. However, this only tells a partial story.

Previously we detailed how the application must spoof account credentials. The request context varies from account to account, based on the auth token obtained during the login process. In order to allow login, and provide an auth token, google does a number of things to validate credentials. The login process itself has 3 inputs: username, password, and androidID. The origins of the androidID are undetermined. We have observed that each device is assigned an androidID, meaning that 2 accounts registered on the same device will have the same androidID. Additionally, it is reassigned on reboot, so whenever one of our decoy devices runs out of batteries, it necessitated a reconfigure of our credentials. Finally, we believe it is assigned by google and not locally, since it can only be found in google talk and android market applications.

In order to obtain this, on the device in the phone application, we needed to access a debug mode using a USSD code. A USSD code is a special code entered into the dialer application that allows the user to access various internals. In this case, the USSD code we used was *#*#8255#*#*, which brings up the GTalk service monitor. This screen will let us know what our androidID is. Providing these androidIDs along with the accompanying username and password to the login process is how to login. The username and password will be validated and if the correct accounts are independently registered on this specific androidID, the login is successful.

We maintain a data structure of all of these context objects, for use at appropriate moments. The requests must contain a valid authtoken resulting from a successful login. This data structure also contains a bit indicating if the connection is secure or not. In order to successfully request a download, this bit must be set to 2. Additionally, the login request must be sent with the correct service specified in order to return an authtoken that allows downloads. This means in effect we maintain 2 data structures per account, both of the type context, 1 that can be used for metadata crawling and one for app acquisition.

Google imposes rate limiting on a per account basis for the download of apps. This is different from the metadata download, which seems to have no restriction. The rate limiting is actually quite low, so in order to increase our throughput, we implemented a round robin approach to application download. Each account downloads an application, then the context used will rotate to the next account.

We have a set of about 25 accounts each operating for the purpose of downloading. As discussed earlier, we spoof each accounts credentials, then use their context structure to download the applications. When one download concludes, the application switches to the next account, and the next application, and begins a new download. This is done to maximize the amount of data we can download per day, since each account added to this list allows us to download an additional allotment of data.

We've already touched upon the download limit, however, circumventing this and handling it appropriately has proven difficult. The failure to not adequately handle this limit results in an HTTP 509 (bandwidth exceeded) error. We have not been able to deduce the exact limit, but in the course of our experiments we have deduced a few other things:

- The rate limiting is imposed on a per byte limit, instead of a per app limit. We were able to deduce this through repetition. Under our construction, every account is used with approximately the same frequency, since account 2 follows account 1 and is followed by account 3 and so on every time. If google were limiting the application downloads on a per app per account basis, it would follow that each account would exceed its allotment within a relatively small window of each other. Since we did not observe this, indeed many accounts would run continuously for over a week, while others would receive errors within a day of starting, we concluded that each account is allowed a certain number of bytes per account per day.

- The rate limiting was done based on account, and not account/ip pair, or simply by IP. All of our access is done
- Requerying the store appears to reset the counter, or in some other way prevent it from ever coming unblocked. The result of this was our requery continued for several days, much longer than it should have.

As a result of these, we implemented 2 main measures to prevent loss of efficiency. First, we implemented a strict rate limit on our client side. We implemented a waiting system to enforce a waiting period between uses of the same account. Our application does not allow for a certain account to be used more than once every 5 minutes. This is to ensure that each application does not over-download during the allotted period. In other words we might spoof accounts 1-25, and the whole process might take 90 seconds. At this point the application will sleep for 3 and a half minutes. This also applies to any of the accounts apart from the 1st account.

The second step taken was to implement a blacklisting procedure, wherein that account would be shut down for a number of hours until we believe that the account could be used again without continuing to receive bandwidth exceeded errors. In our case we shut down the accounts for 12 hours.

This will allow us to run mostly uninterrupted. On average we experience about 70% of our accounts running at any given time. In the future we would like to implement some sort of math based timer to make sure that we do not try to download too many bytes. The intuition behind this is that on a per-byte bandwidth limit, waiting the same amount of time after downloading an application that is 500k in size as you would after downloading an application that is 2 GB in size would not help us avoid account blacklisting in this case. Rather in this case it would make more sense to wait a short amount of time for the small app, and a large amount of time for the large app.

Due to googles policy of secrecy, we are not exactly sure how to verify how many of the possible applications we were able to find, however, we believe using these techniques we were able to sustain a level of over 90% of the applications available in the store replicated in our database.

Chapter 4

Permissions

4.1 Background

The next phase of our analysis was to ascertain the permissions requested by each of the applications we obtained. Fortunately, Android's construction allows us to easily process what permissions an application requests. Using a tool called the Android Asset Packaging Tool (aapt) that comes included with the android SDK, we were able to easily dump each of the requested permissions by each of the applications, and insert them in a database.

4.2 Android Permission System

Android implements what is known as a permissions-based security model. A permission allows a holder to perform some set of privileged actions. Each permission is granted at install time. Common permissions allow access to things like a phone's camera and the Internet. Android applications are written in java using APIs provided and documented extensively by Google. When an application calls a portion of the API, it is handled by the provided library. This in turn will invoke a stub also in the library. Finally this stub will send a request to the system process asking it to perform some action, typically on its behalf. Some of these API calls are privileged and will therefore invoke a permissions check. This is done by consulting a database of all installed applications (which are also each separate users by design) and the permissions they hold. If the application requesting the action holds the appropriate permission, the call succeeds. Otherwise an exception is thrown, and if not handled, the application will terminate. Previous work has discovered that many popular ad libraries will often handle these exceptions in an effort to dynamically ascertain an applications permissions and with it protected user data without the users knowledge or consent.[1]

Other privileged operations are not so simple as direct method calls. Android also provides ways for applications to communicate both with each other and with system services. One such method is the Intent. The Intent is a message that can be broadcasted to some or all of the installed applications. Permissions may be set by the broadcaster dictating who may receive the broadcast. For example, an application may broadcast an Intent relating to an external bluetooth accessory, and may stipulate that only applications who hold the `BLUETOOTH` permission may receive it. Conversely, an application may register to receive Intents of a certain format as a part of IPC. For example, an application may allow an Intent to receive an image and then to post an image to a website. Android allows for this application to dictate that the sending application must hold the `INTERNET` permission. Similarly, applications may set up interfaces to be used by other applications using a Remote Procedure Call format known as a Binder which allows methods to be called by remote applications just like local methods, and can often require permissions as well. Finally the Android framework allows applications to access system databases using specific strings called URIs (Uniform Resource Identifiers) formatted as content://. This is how applications are able to query the contacts database, the calendar, browser bookmarks, and many other pieces of information. Many of these also require permissions as well.

Previous work has examined how permissions are used in real applications on the market [2]. Enck et al [3] proposed Kirin to allow for dangerous permission combinations to be specified and subsequently identified. Xmandroid [4] built on this idea to prevent two colluding applications from executing a privilege escalation attack based on the unions of their permissions using covert channels.

Permissions can be divided into several groups. In particular there are benign permissions, dangerous permissions and System Or Signature Permissions. System Or Signature Permissions are notable because they are not typically documented. Additionally access to them is restricted to either system or handset manufacturers. Developers are able to request permission for them in the manifest, however the system will not grant them, will not prompt the user to allow them, and any use of (undocumented) API calls will result in a security exception. These were also of interest to us because any use of them reflected a developer misunderstanding.

4.3 Permission usage

One of the goals of this study was to leverage our large database of applications to better understand how permissions are used. Previous experiments [2] have leveraged Neural Networks to view at a glance how permissions are related. They found that within some different application categories, there was very little variation in their permission requesting. They also found that between several other application categories there was a large correlation in how the categories used permissions. Other experiments of real world usage have focused on user comprehension [5] using surveys and using interviews [6] and have found low user comprehension. However, there have not been any large scale analysis of how developers use permissions.

4.4 Methodology

In order to get a better sense of how permissions are used, we turned to traditional data mining techniques. One of the things we wanted to examine was similarity among permission use. We felt this would give us the best general sense of how permissions are used. Typically this is accomplished by measuring the Jaccard of two sets. The Jaccard is defined as

$$J(A,B) = \frac{A \cap B}{A \cup B}$$

This measurement gives us a good indication of how closely related two sets are. For example $J(A,A) = \dfrac{A \cap A}{A \cup A} = 1$. Presenting each permission as a set of applications requesting it, we were able to calculate jaccards for each of the permissions in question. Making use of this common similarity index, we approached the problem by considering each pair of permissions.

Android defines 213 permissions, which means there are 45156 pairs of permissions to be analyzed. However, we took advantage of some common heuristics. To start with, 50 permissions are not used at all, which cuts the space roughly in half. We also did not consider any permission that did not exist in more than 10 applications, to avoid outliers. The analysis takes approximately 7 minutes to run. We make note of every pair with a jaccard of higher than .1, meaning that the permissions exist together in 10% of the places that either one appears. This initial threshold was artificially low in order to be inclusive for efficiency sake.

For each pair, we converted this two-way relationship into two one-way relationships by considering the conditional probabilities of each pair. In other words, we converted a $J(A, B) > 10\%$ into P(A—B) and P(B—A). The reason for this was to address some of the problems inherent in the Jaccard index. If A is a large subset of B, the Jaccard may be high, but the relationship is still inherently one way in nature. These changes allowed us to control for this situation. After this, we pruned our space down to only A,B such that $P(A|B), P(B|A) > .35$

We also leveraged more conventional techniques to ascertain which permissions are most common and how many permissions are typically requested. We also examined outliers, such as applications that request large numbers of permissions and applications that request

Chapter 5

Library Usage

5.1 Introduction

As is common on many platforms, libraries are an essential part of android programming. Hundreds of libraries are available for use by developers to achieve many ends. Popular library functionality includes allowing for easy information interchange, facebook, twitter and other 3rd party integration, simple functionality like base64 encoding, and most commonly, advertisement embedding. Most advertising networks provide a jar file for easy embedding of things like advertising windows. This is an especially critical piece of functionality because advertising represents a substantial source of income for many app developers. According to our findings, 77% of the applications in the store are free. As such, these applications typically rely on advertising or in-app purchases as their primary profit generation method.

5.2 Methodology

Previous work [1] has delved into how libraries are used, however their focus was mainly on advertising libraries and the practices employed by them. They combed 100,000 applications possessing internet access, and identified within them different ad libraries. From within these advertising libraries they identified risky API calls, and constructed control flow graphs and performed reachability analysis for these calls. Using this method, they identified many invasive practices that ad libraries will either engage in, or attempt to engage in.

Our analysis is somewhat higher level than this. Since Android applications are written in Java, they retain all of the namespace characteristics that regular java does upon compilation. Leveraging this, we implemented an altered version of a popular Dalvik disassembler called Baksmali [7]. Baksmali is a tool that takes as input a compiled APK file. Within each APK is a bytecode file called classes.dex, containing all of the code for a particular application. Baksmali will convert this file into a hierarchy of human-readable code written in a higher level language similar to that used by Jasmin [8]. This folder hierarchy will exactly mirror the namespaces of the various classes contained in the dex file. For example com.admob.* will appear in the folder com/admob/. Since we weren't interested in the contents of these disassembled smali files and the process of creating them is somewhat time consuming, we modified Baksmali to only output the directory structure.

In order to identify which libraries are run by which applications, we first used baksmali above to decompose an application into its component namespaces. Since compilation of a library along with application code maintains separation of their namespaces, we were able to compare the resulting list with a handmade list of namespaces for popular libraries, 100 in total. If a match is found, we make a note of it. Lastly, we hand coded each library by type, Advertising, Analytics, Tool, Service, Game and SDK.

By amassing such a large amount of library usage metadata, we were able to get a big picture view of how libraries are used in the real world. In particular, we were interested in seeing how many developers made use of libraries, what kinds of libraries did they use, and how many did they use. Additionally, our data helped illuminate which libraries were often used in conjunction, and what relationship they had with one another.

Lastly, we wished to better understand how libraries were used together. This included examining any interplays of libraries with one another and any otherwise unknown relationships, specifically keeping in mind the advertising libraries in particular. To accomplish this, we once again leaned on jaccard indices. This would give us an idea of which libraries frequently imply one another.

Chapter 6

Results

6.1 Market Recreation

Using the tools and techniques described above, we were able to quickly and efficiently amass a large catalog of application metadata and their corresponding binaries. Over the course of just a few months, we were able to assemble the majority of the Google Play market metadata using our targeted word lists. Subsequently most crawling has been maintenance related. Additionally, for the past 12 months we have been downloading binaries using our system and storing them in a RAID. Using these resulting applications, combined with the Binary analysis and maps from our design section, we launched what we believe to be the largest analysis of android applications of any type.

6.2 Wordlist Performance

In our design section we discussed the need to discover a large amount of apps quickly before the binary acquisition process could begin. Our construction used specially crafted wordlists to quickly gather this metadata. Efficiency was a necessity for our work because of the not insignificant delay incurred between each request. Unfortunately our efforts to reduce this delay were unsuccessful, as our traffic was filtered as spammy and dropped or replied to with error codes. We also laid out several criteria we believe a good word list should have, namely the list should be extensive and easily compiled, it should be contain words likely to appear in android applications, and the words contained within it should be unlikely to appear together.

To evaluate the quality of our word lists, we ran a full wordlist crawl for each list, starting with a new empty database. Any app that has already been discovered is discarded, otherwise they are added to the database and the catalog grows. Our primary goal here was to ensure that each request yielded a high number of new (or novel) apps. We note that each request may return up to 10 applications, but can return fewer than 10 if there are no more results in the Store. For experimental control we selected a list of the 1000 most popular english words. Our thinking for this was that since the requirement that words not often appear together was most paramount, this particular list was least likely to satisfy this requirement. Comparing our other lists against this list would therefore prove most instructive in quantifying the novelty afforded by enforcing this property.

In the course of our searches, apart from the word, we were only able to modulate whether or not we were searching for free or paid applications. We also noted that google only indexed the top 500 results for each, at that point the store stopped returning results, and our application would move on. This means that per word, the discoverer may send up to 100 requests. This may act as a good indicator of a lists relevance to the task at hand, though as we'll see, that did not necessarily correlate with higher efficiency.

Table 6.1 contains some of the results from performing this experiment on our lists.

From our observations and intuition, it was clear that as a word search continued, the novelty of its results would decrease. Since each search would be compared with the sum of every search before, it stood to reason that the novelty would approximate an exponential function. To test this we constructed graphs of the progress over time, which are shown in Figure 6.1. To make the graph more readable, we divided the efficiency measurements into 20 smaller parts, with each bar representing the average efficiency of that portion of the search. We also included a variance in the number of successful requests per word. Unsurprisingly the lowest variance came from the list of common words, as these would be most likely to appear in many many applications. A higher variance indicates that some

Table 6.1: Word Results

Description	Length	Reqs/word	Novel/Req	Coverage	Pen Var
Most Pop Words	1000	80.8	3.42	86810	387
Big Cities	334	24.3	6.66	54339	569
First Names	999	34.1	4.93	161742	908
2 letter combos	676	61.2	4.77	131238	992
Relaxation terms	360	71.9	4.76	122254	548
Social Networks	115	45	7.02	36342	1438
Chinese words	132	26.8	4.44	14880	2260
Colors	899	18.6	4.77	79691	747

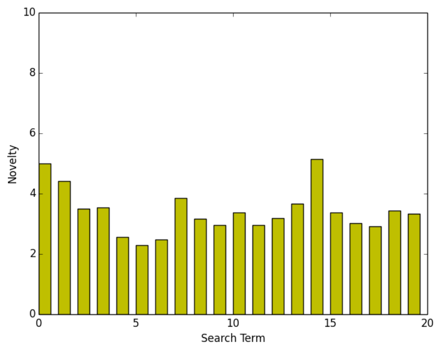
of the words in our wordlists could be left out entirely, in an effort to make it more efficient. This may be a topic for future work.

We also analyzed the efficiency of searching across search terms within iterations. In other words, we took a look at how the efficiency compared for a given wordlist the farther down in the search results a given search progressed. So a comparison of the averages for all requests for apps numbered 1-10, 11-20 and so on. Interestingly, for many lists, the efficiency seemed to increase as the search got deeper. Intuitively this makes sense as the deeper searches are likely to lead to more esoteric apps, and therefore greater divergence from the more universally popular apps displayed in the earlier results.

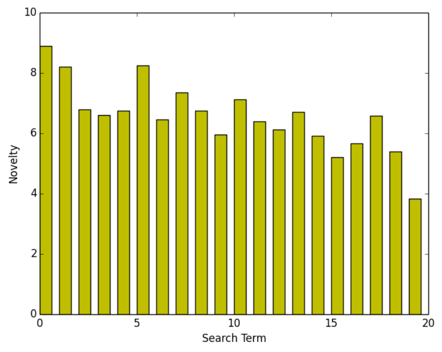
We also included in our table coverage data.

6.3 Application Acquisition

Using this method we were able to obtain metadata for 90% of the Google play market in a matter of weeks. Once we reached this point the application fell back into a maintenance mode, monitoring the lists of newly added apps, and checking

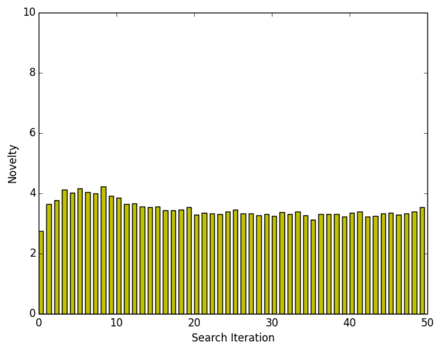


(a) Control (1000 Most Popular Words)

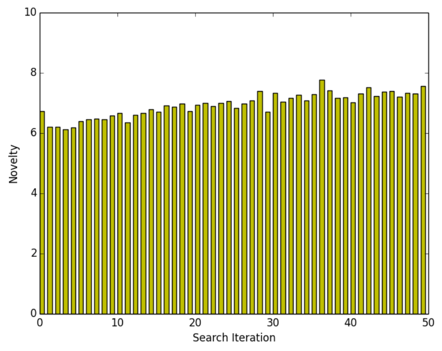


(b) Large Cities around the World

Figure 6.1: Note the decrease in return over time as a word search progressed.



(a) Control (1000 Most Popular Words)



(b) Large Cities around the World

Figure 6.2: Efficiency vs depth of search

for updates. At this point the Market boasted somewhere between 700 and 800,000 applications, and we have not dipped below 90% of the announced applications at any point since. Some statistics of the metadata we possess is pictured in table 6.2.

Table 6.2: App Statistics

	Versions	Distinct Apps
Metadata	1614334	1092790
Downloaded	703413	452445

With this metadata we outlined how it enabled us to in turn download the binaries in large amounts programmatically. This allowed us to assemble the largest library in existence outside of the market providers. Specifics about how many applications we were able to download are also in Table 6.2. This process was much more time consuming, and has been running for the better part of a year. Currently we have over 700,000 binaries taking 15 TB of space stored locally.

6.4 Permissions Usage

Using this design above we were able to obtain a large list of the permissions usage of all 700,000 thousand applications. Recall that each application includes an XML file called the manifest, containing many critical details and pieces of metadata required for successful OS integration of the application. Among the things included are information about each activity, advertising data, IPC controls, and permissions requested by the application. Each APK file obtained is a standard zip file containing this manifest, and a classes.dex file corresponding to the bytecode of the application, as well as graphical resources, screen layouts, and other information critical for the operation of the The Android SDK ships with a program called aapt (the Android Asset Packaging Tool), which allows us to view the details of the manifest from the command line. Using this tool we were able to quickly insert our permission data into a database of each application and which of the 213 permissions declared by past and present android versions its developer requests.

From our analysis we were able to find

The question then became how to best represent these relationships. A graph seemed a natural way to represent many data points sparsely connected like we have here. We first placed a point for each relevant permission, and connected each permission to its partner with a directed line, of a weight of P(A—B) That graph can be seen in figure 6.3. From the original 213 permissions, only 55 remained after this process. The others may be for our purposes considered noise. Of particular interest was how each of the permissions was grouped. The graph is mostly made up of pairs of permissions, joined to each other. These pairs are

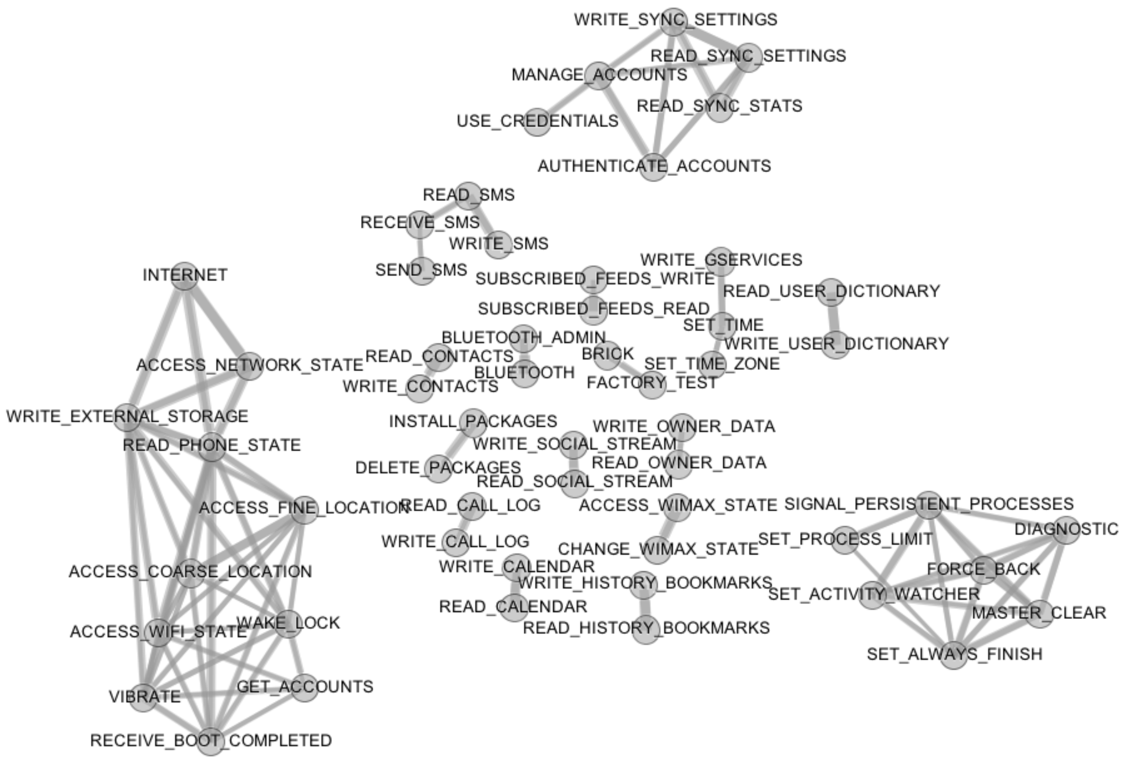


Figure 6.3: Permissions clustering

typically made up of closely related permissions. For example `READ_CALENDAR` and `WRITE_CALENDAR` or `BLUETOOTH` and `BLUETOOTH_ADMIN` or `READ_CONTACTS` or `WRITE_CONTACTS`. The larger clusters are for other similarly related pieces of functionality. Permissions related to SMS are grouped together, as are those related to managing account synchronization and account management, required for in-app purchases. The largest cluster, however, is for the most common functionality. Among them, `WRITE_EXTERNAL_STORAGE`, permissions to access phone location, `RECIEVE_BOOT_COMPLETED`, which is used to make an app start upon phone boot, and `INTERNET`.

These clusters have for the most part had some common thread of functionality. Each of their members is related in some way to help an application accomplish some goal. Calendar, Contacts, Internet, Phone, Location, SMS, Accounts, and Bluetooth are all clusters that are represented by this data. This common thread of functionality is telling about how permissions are used, as they are used mostly in response to functional needs, which is in keeping with the general goals of the permissions system as it exists.

The final cluster, however, is illuminating. Previously we discussed that there are dozens of permissions that are system or signature permissions, meaning that either the OS manufacturer or handset manufacturer must have signed the application requesting it in order for the permission to be granted. 3rd Party developers can request them without penalty, but the operating system will not consent to grant them and therefore any use of these permissions will result in a security exception. However, there is no warning or other issue with developers requesting them anyway, hence it is a relatively common occurrence. A count of our database finds over 40,000 applications requesting at least one System or Signature permission.

The last cluster consists entirely System or Signature permissions. This cluster is particularly interesting because the permissions included have do not share any common functionality, keeping in mind that even if they were related, the application could not run the privileged operations corresponding to these permissions. This represents an interesting trend in permissions use. Whereas with other clusters, they are grouped by a common operation or use case for a particular application, or set of applications, the common denominator here is more nebulous. The commonality of these applications seems to be that the developers are confused by the permissions system. Since the functionality cannot be executed, and there is no functional similarity among the permissions listed, the most likely explanation is that confusion is the primary motivating factor.

In other words, this cluster tells us that for the system and signature permissions, given that a developer has already mistakenly added a system or signature permission to their application, they are likely to request another. This is an important finding because it shows that there continues to be a large disconnect between developers and the permissions system, as it currently exists.

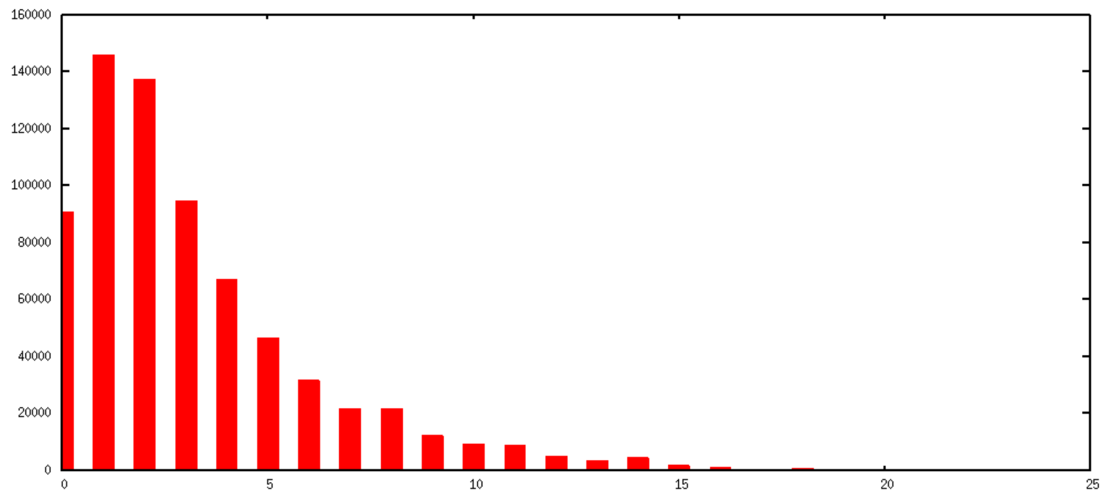


Figure 6.4: How many applications used X libraries

6.5 Library Analysis

In trying to determine how libraries were used, we used a handpicked list of 100 library namespaces derived in part by inspection of commonly used namespaces. Since most applications don't share the same namespaces, this was a matter of finding the most popular namespaces combined with manual inspection to verify that this was the top level namespace. We accumulated a list of the 100 most common 3rd party namespaces and categorized them based on whether they were for a service, an application tool, pertinent to an SDK, a library used for game development, used for analytics, or advertising.

We found that a plurality of applications make use of just 1 library, with 28% of applications using just 1. An additional 25% use 0 libraries that we identified. The remaining 47% use 2 or more libraries. 1.5% of all applications studied used 10 or more, with 9 applications each using 20 or more libraries. In our set of 181,398 a total of 396,001 libraries are used, an average of 2.2 libraries/application. The most popular libraries in use can be seen in table 6.3. In our sample, we found that 6 of the top 20 libraries were related to advertising, 9 were application tools such as support libraries or libraries that enable in app purchases. 1 was a popular analytics library that allows for a developer to track user application usage. 2 were related to an SDK, including one that allows for applications to be compiled from HTML. Lastly, 2 were related to popular services, twitter and a bug reporting service. Each of these libraries were included in over 2.5% of applications. The most popular library was the google ads library, com.google.ads, which appeared in 32% of applications.

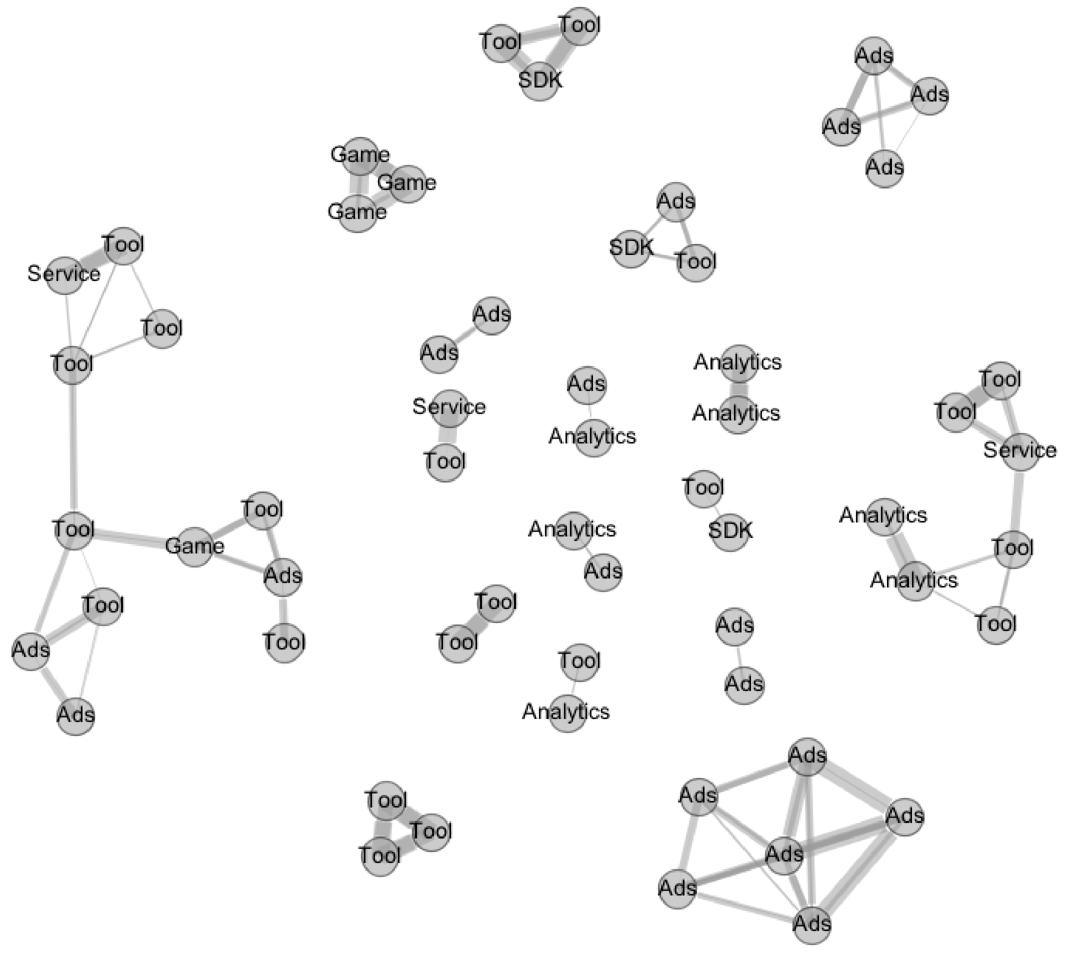


Figure 6.5: Libraries clustering

6.6 Library Usage

Using the outlined design above, we were able to scan all 700,000 of our downloaded applications for 100 common libraries by disassembling the applications and verifying namespaces. This took about a week to compile, using our modified disassembler. Using a similar methodology to how we approached the Permissions usage above, we decomposed each of our top 100 libraries into a set of applications. The goal in this section is to shed a brighter light on how developers make use of libraries.

Similar to above, we calculated the jaccards for each of the pairs of 100 libraries. For each of the pairs, if the jaccard was over .1, we set it aside. Again, for each pair with Jaccard over .1, we recalculated each of their conditional probabilities. For this experiment, we lowered our threshold to .25. So for each remaining pair, we ensure that both probabilities are above 25%. At this point we construct a graph similar to the one mentioned above, found in Figure 6.4.

As we can see in this graph, once again, the connections are primarily of the pair type. After this process, we were left of 57 of the possible 100 having some relationship with at least one other library. Some cases were of a set of 3 strongly connected nodes with high probabilities. It seems logical to conclude that these were likely included in one larger library, or family of libraries, and that our identification process in this case was overly specific. There were also some sets whose names were similar. Further inspection revealed that these were typically two version of the same library. For example, one popular analytics library, crittercism, appeared twice on our list, and was shown to have a high correlation, possibly due to incomplete upgrades, or versioning.

One of the most notable clusters is the strongly connected set located at the bottom of our figure. This one depicts 6 different advertising libraries, each of which was found to have a large similarity with every other advertising library. It's worth noting that none of these are explicitly related, (i.e. a dependency or a version). Rather, this indicates that several ad libraries, as many as 6 of these, appear together with a high frequency.

Other important observations include:

- The isolation of the gaming libraries. They imply each other with a high probability, indicating a propensity to leverage several gaming technologies at once. However, they do not relate to other technology such as social networks or advertising.

- Another smaller cloud of advertising libraries, not as strongly connected, but still notable for not being included with the others. This may indicate some underlying thread for the larger cluster that does not exist for these. Nevertheless, it seems that using more than one advertising library in conjunction is a common practice.

- In our entire graph, there is only a single linkage of an Advertising library and an analytics library. Since these are more generally "Goal-oriented" libraries, as opposed to support libraries, this is interesting that their underlying goals do not overlap in any specific way.

- Many services, SDKs, and advertising libraries correlate highly the support of tools. These can be anything from a popular 64 bit encoding library to the official library allowing in-app purchases, or googles protocol buffer library. This may imply that they are often required, or simply offer some functionality needed to assist in accomplishing a developers goal that a library does not.

We believe that displaying the interactions between libraries gives a greater insight into how developers create applications.

6.7 Library/Permission correlation

Lastly, we turned to a way to quantify how the two studies above were related. To do this we first joined our data for permissions use with our data for library use. This gave us one longer table where each application was listed by which permissions it requested and the libraries that it includes, based on the methodology we laid out so far. From here, we ran a similar jaccard-based study. First we turned to correlating individual permissions with libraries and vice versa. In other words, if a permission and a library had a high jaccard index, we would add it to our graph.

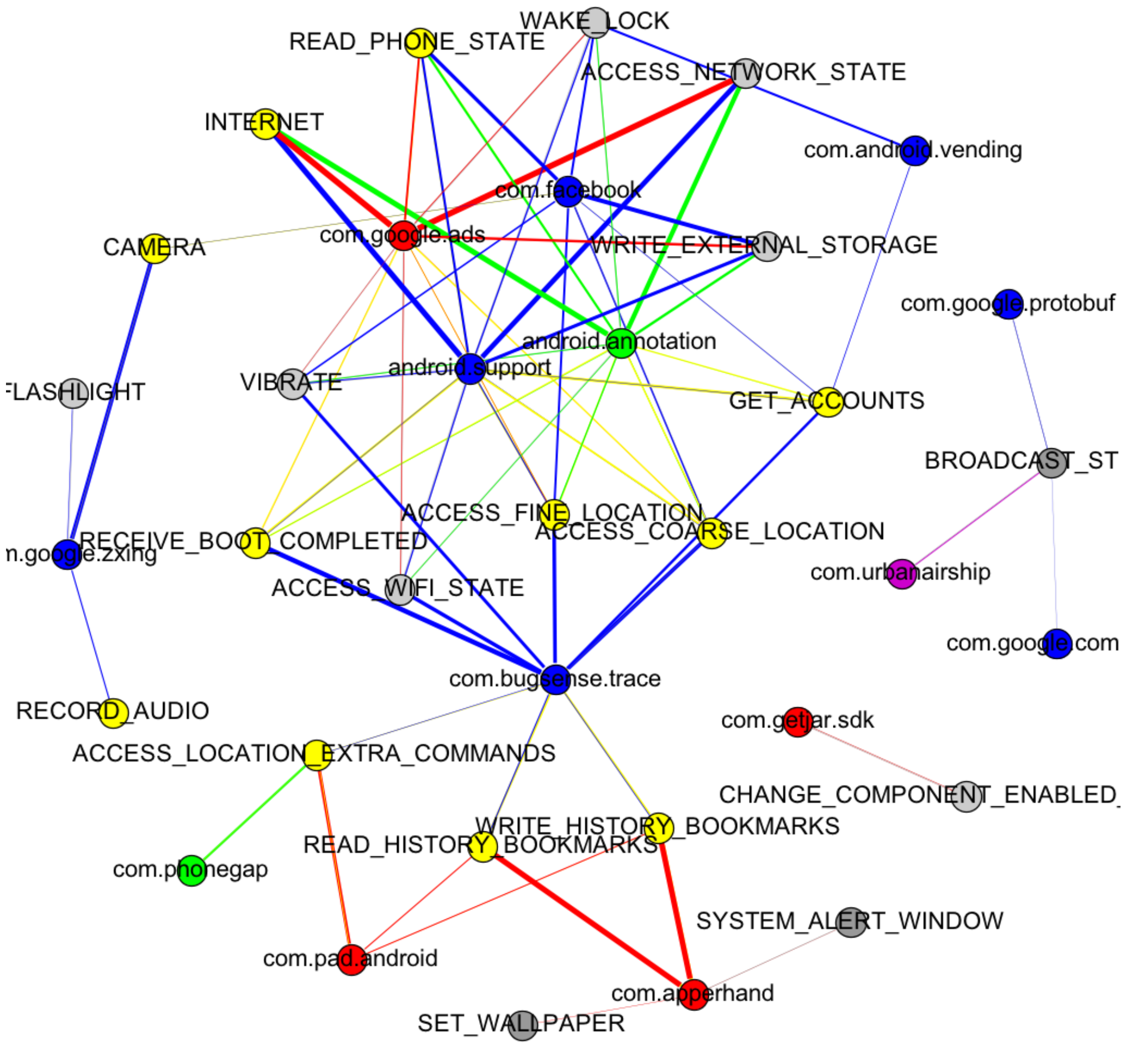


Figure 6.6: Libraries vs Permissions

The resulting graph was somewhat cluttered, so we colored it. Any privacy sensitive permission we colored yellow, and any advertising library we colored red. Blue libraries were ones we deemed typically benign, either related to a social network, or general support libraries. Any link between a red and a yellow node is considered a problem.

In particular some things to note:

- The Library com.apperhand is a known piece of malware. Previously it was known as counterclank, and it was deemed malware by symantec. Some known behaviors were inserting many icons on a users home screen as well as hijacking their search requests. Additionally, they would insert bookmarks into a users bookmarks at will. This is clear from our study as well, using 26,000 applications we had which contained this known piece of malware.

- There were links as well between the google ads library and many of our privacy sensitive permissions. This was probably due to the relative prevalence of each of the permissions and library in question.

- Additionally com.pad.android, which is a library provided by leadbolt, and also known as pubxapp, also was linked to history and bookmarks, which we believe is a finding.

Still, we felt this format gave us an incomplete picture of how libraries and permissions interplayed. Specifically from a privacy standpoint we were interested in how permissions were used with respect to libraries that had privacy sensitive functionality. To that end we turned back to a more simple methodology: correlation. For each application we considered whether it contained one of our 13 advertising libraries, and which permissions it requests.

From this point we were able to have a top down view of the interplay of advertising libraries and permissions. In particular we were curious what proportion of applications that request a particular permission contain advertising libraries. It's important to note that under our construction we can only establish a lower bound of how many applications contain advertising code. We tested for 13 common libraries in namespace. For example, the namespaces were obfuscated in any way, we would have a false negative. Additionally, it is clear that if an application had an advertising library that was not on our list, that would also be a false negative.

Nevertheless, our results were very convincing. By hand we coded each permission based on the perceived threat it posed to a user. Red corresponds to the highest stakes, orange medium, and yellow is the lowest threat. We found 14 per-

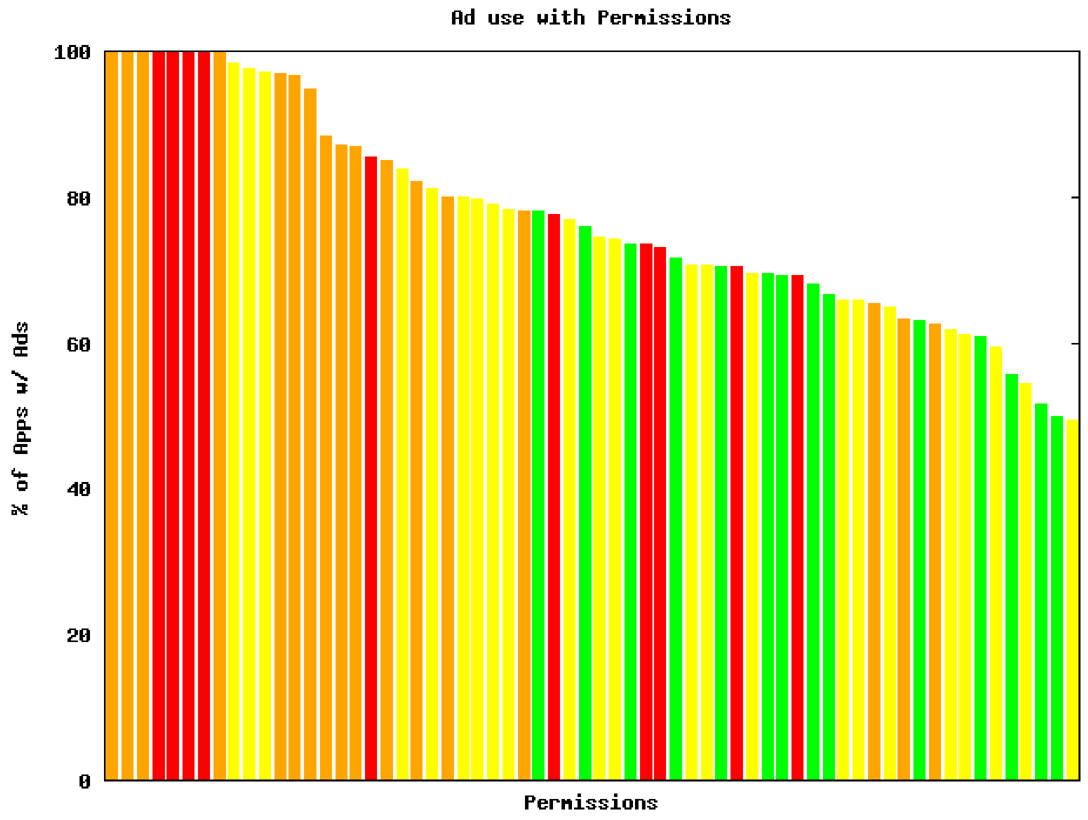


Figure 6.7: Libraries vs Permissions

missions which appeared in more than 1000 applications and of those applications, over 94% contained advertising libraries. Specifics can be found in table 6.3.

In particular several of these are very worrying.

Table 6.3: Permissions found in many Apps with Ads

Permission	% containing Ads
READ_PROFILE	100.00%
READ_USER_DICTIONARY	100.00%
WRITE_CALL_LOG	100.00%
READ_CALL_LOG	100.00%
READ_HISTORY_BOOKMARKS	100.00%
READ_PHONE_STATE	100.00%
READ_SMS	100.00%
WRITE_HISTORY_BOOKMARKS	99.94%
RECEIVE_MMS	98.33%
PROCESS_OUTGOING_CALLS	97.72%
RECEIVE_WAP_PUSH	97.22%
WRITE_SMS	96.93%
WRITE_USER_DICTIONARY	96.58%
USE_SIP	94.71%

Chapter 7

Related Work

The work presented thus far in this paper has been a study on the influence of Advertising libraries in applications offered on a mobile app store on the use of permissions. We believe this intersects with 4 major areas of current research

- Mobile security
- Application/Store security
- Permission systems
- Application Analysis

7.1 Mobile Security

First and foremost, our work concentrates on threats we feel are specific to a mobile environment. With the emergence of the smart phone, a phone running a fully featured commodity operating system, those existing threats have exploded to a point where the security community often struggles to keep pace. Mobile phones face a host of different threats that traditional desktops do not.

For one, cell phones often contain a more diverse set of devices, which have often times not been subject to scrutiny. In 2006, Bose et al [9] demonstrated common vulnerabilities of bluetooth. They proposed several example attacks, which have all been reported, including brute-force proximity scanning, address discovery, and common software errors. This same paper, along with others [10, 11] warn of MMS vulnerabilities as well. Mulliner et al in [10] made use of a fuzzer to test various aspects of the MMS protocol. In particular they fuzzed lengths of fields provided, modulating self-reported string lengths, and employing other techniques designed to falsify a bounds checker. They found several buffer overflow errors, some of which could be used to alter the return address and thus hijack the control flow. Racic et al in [11] also warned of an availability threat that is often overlooked in traditional computing: battery life. Their paper outlined how an attacker may exploit the MMS and PDP protocols to drain a phones battery extremely quickly. Their attack functioned in two stages, the first of which exploited the fact that MMS messages are unauthenticated and often will automatically download their payload. With this in mind they devised an attack to build a target list by sending MMS messages to a large list of known area codes and prefixes. The messages also contain a payload directed to their malicious server. When a handset sends requests, it will include their identifying information including IP, handset type, and other data. By modulating the URL slightly, an attacker could build a list that maps URLs to phone numbers. With this list in hand, the attacker can begin to send small 1-byte UDP packets to these numbers. Since having a phone in the ready state is much more battery intensive than in the standby state, this will drain the battery extremely quickly, and since there is no notification to the user that a phone is in ready vs in standby, it will happen without the users knowledge.

As far back as 2004, papers [12, 13, 14] have been published warning that as smartphones became smarter, they would become a more attractive target for malware. Smartphones are different from their predecessors in that they run operating systems much more like those we are used to seeing. Hence they are vulnerable to many of the same compromises as traditional PCs. Worms, viruses and trojans are all possible and have been documented as far back as 2004 [15] on smartphones. This warned of a virus that spread among symbian phones which would send text messages to a premium number at a substantial cost to the user. The virus would typically be installed by installing a cracked version of a game called mosquito. Since a phone is automatically linked to billing information, this also represents another threat not typically understood for traditional computers.

Since most smartphones have an always-on internet connection, they are an appealing target for botnets as well [16]. This paper presents a new botnet design oriented towards cell phones making use of bluetooth for command and control. In their design, only compromised devices who can communicate directly with the most other bots may contact the master directly. All other devices receive commands and code updates via bluetooth from these devices. This also serves the purpose of naturally spacing out bot activity, which can make detection and location of the network much more difficult. To test their model, they created a model of a mass transit system, saying that this would be an ideal scenario for such a setup. They go on to propose new methods of botnet detection and defense, such as the now implemented process of over the air security updates for phones, and desktop based virus detection during sync with a PC.

Keeping with botnets, they can have a devastating impact on other things as well such as the cellular network [17], which may not possess a direct security analog in the desktop world. This paper presented an attack that could be made against mobile network appliances accomplished with a relatively small number of compromised devices. At the core of every mobile network is an appliance known as a HLR or home location register. Whenever a certain phone numbers location is needed, the HLR is queried. Their attack focuses on a DDOS attack against this appliance, which in turn jeopardizes the availability of all the devices under its purview. In order to demonstrate this they relied on a simulation, which included modeling the impact of the 7 commands undertaken by an HLR and what proportion represents normal traffic. With this in mind they attempted to develop an efficient and successful DDOS attack by repeatedly issuing certain transactions which they found to be the highest impact.

Additionally, a mobile device presents other unique threats. For instance, mobile phones possess the ability to track a users location by design (ie, GPS, cellular location, nearby wifi networks). Exposure of such data to unauthorized sources would ideally not occur, however some work has identified such activity [18]. Additionally, phones can contain other unique sensitivities such as a microphone [19].

This paper described a side channel attack in which an application with a few benign permissions could steal a users information such as credit card number, and ATM PIN. This attacker application will then communicate with a either a legitimate application or a second colluding application which will communicate this information to the internet. This is just one of many examples of how the threats facing a mobile device may be different from traditional desktops.

Cellular phone security research is still a relatively new field, made more complicated by several factors. The continuing drive to innovate has traditionally favored all else at the expense of security, and with innovations like app stores, new security challenges are arising quicker than the community can tackle them.

7.2 Application/Store Security

One of the largest threat to mobile phone users stems from the unique threats presented by application stores. Every major smartphone operating system manufacturer has its own application store, which allows for the installation of third party applications. Developers can submit these applications to the store, where they will be listed for download. This allows developers to bypass setting up their own eCommerce and other overhead that used to be associated with downloadable software sales, as well as implementing DRM.

Android boasts the Google Play [20] store, however, there are other android app stores as well. For example, Amazon allows users to install their own app store [21], and take advantage of their catalogue as well. Other more niche alternative app stores exist as well, such as MiKandi [22], which caters to adult app developers, F-Droid [23], which offers all free and open source applications, and several alternative Chinese application stores.

These stores are an easy way for users to find Apps, then pay for, download, and install them. Android apps are lightweight applications, typically written in java, that users can download and install from these stores. However, having such ease leads to new security concerns. For one, applications in the store are viewed with more trust than they are due. Users often install apps without thoughts to security. However, as we will see, the security in the android store is often very lax. In contrast, the hand review done by the iOS app store, which results approval wait times of days [25], the google play store will typically list an application in a matter of minutes. In order to gain entry to the store, each application is vetted by a dynamic analysis system known as bouncer.

In 2012, Oberheide et al [26] submitted an application to the google play store which allowed them to explore the environment running bouncer. The application opened up a prompt and enabled the researchers to find out several important details that would enable an attacker to bypass the protection afforded by bouncer. In particular, they noted the ease at which an attacker could fingerprint the system. It's processor is listed as "goldfish" and it contained a qemu_trace folder in the root directory. An application would be able to pass a malware check by suppressing questionable activity if either of these things were detected. They also noted that each application was run for only a short amount of time, between 5 and 10 minutes. Any attacker wishing to bypass the malicious behavior test need only postpone malicious behavior beyond a certain time period. Since publishing this work, google presumably has made changes to Bouncer, but as it is still closed source, their efficacy cannot be fully judged yet.

There have been several instances of finding malware in the google play store. Some are used to steal contact information [27], others place costly phone calls to premium phone numbers [28], join a botnet [29] or even install other applications without a users knowledge by exploiting a root vulnerability [30]. A few works have surveyed the different types of malicious applications that managed to gain entrance to the Google Play store. In [31], the author gave an overview of existing mobile malware. While they considered several platforms, all of these were found in their various application stores. They categorized a corpus of 46 malicious application by behavior type. They found that 28 exfiltrated user information, 24 placed premium calls or SMS, 8 sent advertisements via SMS, 6 seemed to be constructed for novelty or amusement of the application author, 4 exfiltrated user credentials, 1 instance of hijacking of search results and even 1 instance of Ransom. This particular application would lock the phone entirely and demand a 5 euro ransom to unlock them. This categorization was done based on the incentive to the author of the malware.

In a similar study, focused on android[32], the authors attempted to come up with a scalable approach to detect the overall health of a market. In order to accomplish this, they relied on several heuristics. First of all they were only concerned with applications that were able to perform certain actions that would require corresponding permissions. Their second method was more involved, as they implemented a form of behavioral fingerprinting. Using known malware they would compare which API calls are run and in what order, to try to identify matches. This is similar to existing intrusion detection methods in that they build a somewhat flexible behavior profile. They also used a similar approach on string literals. They uncovered 211 malicious applications, mostly from alternative marketplaces, though 32 were from the google play market, including a zero day vulnerability. This vulnerability named Plankton would download a malicious jar remotely and execute it locally. These papers gave a helpful overview of the malware situation in the market, but their scope was limited to only malware.

More recently, more sophisticated attacks have been envisioned. In a pair of papers [4, 33], they proposed a system called XMandroid. In this system they implemented a reference monitor to prevent interprocess communication between applications per a certain policy. This reference monitor was novel since it policed both overt and covert channels. In other words, two applications would not be able to send intents, nor communicate via temporary files and sockets, if these two applications violated the policy. To do this, they implemented a runtime monitor as a policy enforcement mechanism. They also modified the application installer and implemented a policy installer to allow users to specify their own sets of concerning permissions, as well as a mechanism for storing previous decisions made about past ICC.

In a similar vein [34] they proposed a way to categorize applications into protection domains, using a coloring system. The scenario they lay out concerns an enterprise provided device, running enterprise provided applications. In order to prevent a user from attempting to circumvent the security policy, the enterprise wishes to continue to allow 3rd party applications on the device. However, they may be concerned with privacy leakage between the provided applications installed by the enterprise and the 3rd party applications installed by the user. Consequently, they implemented a reference monitor to prevent any explicit IPC between applications, if they were of different ”colors” (ie 2nd and 3rd party). However, neither of these works focused on applications that are currently available, rather these were defenses against specific types of attacks.

Yet another threat is more subtle. Our work found that paid applications were rarely popular. One direct consequence of this has been a rise in the use of advertising libraries. Ad libraries allow a developer to monetize their application, in exchange for displaying advertisements, either in a small banner or in a splash screen. However, substaintial work has been done that finds that advertising libraries engage in certain privacy violating behaviors.

Taintdroid [18] was one such paper. In this work they modified the dalvik VM, which is the VM in which the android bytecode runs to track taint from certain data sources. In particular they tracked location data and other privacy sensitive data sources. They found explicit data flows from user location providers to network sockets. Tracing the sources and destinations of these flows lead them to discover that advertisers were transmitting many pieces of private information in the clear to their servers.

Building on this, the authors subsequent work [35] made use of a retargeter. This was itself not trivial as plenty of information is lost during the translation from java bytecode to dalvik bytecode which must be inferred upon reversal. Using this retargeter they then decompiled the resulting java bytecode into java source code. From here they leveraged the large group of tools available for static java analysis, eventually settling on Fortify. Fortify is a rule based static analysis tool which allows for things like taint tracking. Using this capability they found something similar to the taintdroid work, namely that different types of private data were exfiltrated, including device ID, location information, and installed applications. They also checked for botnet behavior and eavesdropping behavior, but did not find any. These two studies were quite in depth but made use of a small amount of applications, 50 and 1100 respectively.

The broadest work done to date on advertising was done by Grace et al [36]. In this paper they identify the 100 most common advertising libraries, and look in depth at their behavior. They made use of a corpus of applications totaling approximately 100,000, disassembled, and analyzed a sample of each library identified. In their analysis they were looking for several types of risky behavior. Among the behavior looked for, they used reachability analysis to determine if a library will list accounts, place phone calls, read calendars, contacts or call logs, bookmarks, phone number, phone information, send text messages, and other privileged operations.

They also looked for more complex behavior including dynamic class loading capabilities, java reflection, obfuscation, javascript, and finally permissions probing. Permission probing is behavior in which an application attempts to execute a privileged operation, and in the event of a failure, rather than crashing on the resulting exception, they catch the exception and continue. This ensures that a library can dynamically request as much private information as possible, transparently to the user. However, its worth noting that this work does not explicitly explore the influence that the advertising libraries have on the application ecosystem on a whole.

7.3 Permissions

Android implements a permission system similar to traditional capability systems. The permissions are granted to an application on installation. This requires the developer to specify ahead of time which permissions their application requires. In the installation prompt, the user is presented with a list of the permissions the application requests. Installation of the application requires accepting these permissions, the only other option is to decline to install it. If an application attempts to execute a privileged operation, the operating system will verify that the application holds the required permission, if it does not, the application will throw an exception. If the exception is not handled (like above) the application will crash.

This leads to an interesting problem for the developer. In their short paper, Au et al [37] discussed the tradeoffs involved here. They first discussed the differences between the permissions systems of the most popular smartphone OSes. In particular they identified user control, system information, and interactivity. They then discussed the costs and benefits associated with developing under the permission model. They claim that requesting more permissions than is needed has several benefits to the developer. For one, it allows them to work quickly and prevents an unanticipated security exception without requiring much time consuming thought, they claim. On the other hand, requesting too many has drawbacks, such as violating the principle of least privilege and possibly discouraging users from installation. In other words, if an application requests too many permissions it exposes a user to unnecessary risk in the event of a compromise, additionally a user may choose not to install an application because they erroneously believe an application accesses some information they do not want it to.

This raises the question about flaws in the permission implementation. Since the approach requires a priori knowledge of the sum total of privileged operations that could ever be executed by this application, and since there are approximately 75 permissions available, it is prone to developer error. At a glance, it seems like this problem would lend itself well to static analysis to determine which operations are run, and compare to a map. However, generating such a map has proven to not be trivial. In early work, Felt et al [38] tried to make a map based on the published documentation. However, they ultimately found it lacking in both specificity and completeness.

In a followup work [39], the same authors took a new approach for deriving this map. Using a modified version of the android kernel (which logged each permission check), and an application fuzzer, they attempted to dynamically build a map between API calls and the permissions required. Doing so they believe they covered approximately 85% of the API calls specified. The others they were unable to successfully fuzz due to required inputs or instantiation of variables. They then implemented a tool to ascertain what privileged operations the applicastion invoked, including considering reflection and reachability. Using this list, they compared to a map to determine what permissions they believed an application would require, and contrasted that list with what developers actually requested. They found that roughly 1 out of 3 applications requested more permissions than was necessary. They also found that often times the unnecessary permissions appeared to be related to developer confusion. The authors noted

In contrast was a similar paper done by Au et al[40] which introduced PScout. Pscout is a static analysis tool that attempts to analyze the android source code to determine the permission mapping. To do this they constructed control flow graphs for the OS software. They identified each instance of a permission check, and traced backwards until they reached one of 3 conditions. The first condition is a call to change calling identity, since this will ensure that the check succeeds no matter what, and is thus not pertinent for their study. The second is access to a content provider, at this point an inference can be made between the content provider and the permission required. The last is a documented API call, if an API call is reached, they also abstract an association between the API and the permission required. In order to test their mapping, they used a User Interface fuzzer to dynamically check 1260 applications they obtained from the market. They found that approximately half of them were overprovisioned. This static derivation and dynamic testing of their map is in contrast to the dynamic derivation and static testing proposed in the Stowaway paper. However, neither of these papers are directly related to permissions usage.

The permissions system has not been studied as just a systems implementation. Felt et al [5] also studied user understanding in a laboratory setting. When discussing Stowaway we touched on developer confusion about the purpose of permissions. They noted that developers were likely to request permissions that sounded useful, rather than were required by their understanding of the documentation or operating system. However, they did not consider user comprehension of the permissions requested, erroneously or not. This paper took a laboratory approach and touched on several questions related to the psychology, usability and design of the system. They found that most participants in their study demonstrated low attention to permissions installed, low comprehension of the permissions and risks involved. However, they did find that most users had declined to install an application based on its permissions. This study does touch on the concerning aspects of permissions and whether or not users are mindful of permission behavior, but the findings are somewhat inconclusive.

We believe our work speaks to the amount of influence advertising libraries have on the use of permissions. To our knowledge no work has demonstrated such an effect. However, in Grace et al [36], they did touch on the interaction between libraries and permissions. They profiled behavior they called ”permissions probing”. It’s worth revisiting this since this has several ancillary effects that are not explored. For one, this type of behavior will cause problems for most static analysis tools, since the reachability of this cannot be determined at compile time even by more advanced techniques such as symbolic execution or other sensitivities. Enclosing a call that may fail inside of a catch exception statement may turn up as a false positive when constructing a list of sensitive API calls made by an application, depending on whether or not that call would fail. Considering that the permissions required will likely be in the library documentation, these API calls, which are not even required for proper operation, may be having a sizable impact on the ecosystem.

To our knowledge, the only direct study of how permissions are used came from Barrera et al [2]. In this paper they examined how the permissions were used with respect to the application category. They hypothesized that an application category, which is a very general proxy for the application functionality, would often dictate what permissions the application used. What they found was that for the most part, applications did not group well within their categories, with a few exceptions. They also found that between categories, often permissions are used very similarly. They concluded the paper by proposing some enhancements to android including allowing finer grained permissions which might communicate to the user more information about why an application is requesting a permission.

Our work builds on this idea by proposing that the single driving force for large swaths of the android permissions system as it currently exists is advertising libraries. To show this we draw on the largest corpus of applications used by any of the papers above.

7.4 Application Analysis

Several parts of the work we've profiled thus far have focused on application analysis as a methodology, some of which deserve more attention. The stowaway paper [39] made use of a static analyzer to determine which API calls and URI strings were used by a given application. In order to do this, the first used a disassembler called Dedexer which converted the bytecode into a human readable format. Their analyzer then would run rudimentary interpretation of this code. First it would parse the code for all standard API calls, including tracking inheritance of android classes by application classes. They then used heuristics to handle reflection and 2 edge cases. First they tracked the value of every String, StringBuilder, Class, Method, Constructor, Field, and Object to resolve any calls to the standard reflection invocations. Their heuristics also help resolve service resolution, as well as performing type tracking. They also noted that the Internet and External Storage permissions were special cases, and implemented ways to handle them. They handled Internet by mapping any call to a webview to the internet permission, and any addressing of file locations beginning with sdcard map to the External Storage permission.

Similarly, the ded paper [35], they also performed a static analysis of applications. However, since their goal of performing more in depth code analysis, they needed to take it a step further. To do this they first would translate dex bytecode back into java bytecode, which is not a trivial task. From here, they leveraged a decompiler to produce java source code. With this java source code in hand they were able to track different information flow patterns and view at a glance what issues an application might have. Their taint tracking rules revealed several instances of sensitive data being sent to advertisers or other outside sources, without the users knowledge. However, both of these methods were prone to the trappings of static analysis, namely overcompleteness, as both papers were careful to note. Static analysis is limited to saying what may happen during an execution, instead of what will happen. This approach is similar to the one used in this work, in that we accepted developer permission declaration as a possibly over-complete representation of what an application is able to do.

Not every paper took a static approach, however. In Taintdroid, [18] the authors made use of a modified runtime environment which would track information flow in real time, dynamically. They similarly found many instances of location data, and other identifying features being transmitted in the clear to 3rd parties without the users consent. This information also included a users phone number. They were able to make these modifications to the VM with only a small performance penalty.

Similarly the PScout paper [40] made use of dynamic analysis to test their proposed mapping. They made use of a UI Fuzzer to randomly navigate through an application and try to invoke as much of its code as possible. They used this method to supplement their static analysis of an application in response to the obfuscation provided by java reflection. This approach is not unlike the technique used by Stowaway to generate their map at the outset. They both made use of a fuzzer, which would help them to execute large portions of the target code, and both note the drawbacks of such an approach. These are not unlike the well known problems with dynamic analysis, which is that the resulting analysis is only as good as the fuzzer used. In the PScout UI Fuzzer, they were able to cover about 70% of the applications in question, while for Stowaway they boasted a 85% coverage rate. Since we didn't make use of either of these techniques, instead relying on a large amount of developer feedback, we feel that we have avoided the problems specific to these two approaches.

Bibliography

[1] Grace, M., Y. Zhou, Z. Wang, and X. Jiang (2012) "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *NDSS '12*.

[2] Barrera, D., H. G. Kayacik, P. C. van Oorshot, and A. Somayaji (2010) "A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android," in *Proceedings of the ACM Conference on Computer and Communications Security*.

[3] Enck, W., M. Ongtang, and P. McDaniel (2009) "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, pp. 235–245.

[4] Bugiel, S., L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi (2011) "Xmandroid: A new android evolution to mitigate privilege escalation attacks," *Technische Universität Darmstadt, Technical Report TR-2011-04*.

[5] Felt, A. P., E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner (2012) "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ACM, p. 3.

[6] Kelley, P. G., S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall (2012) "A conundrum of permissions: Installing applications on an android smartphone," in *Financial Cryptography and Data Security*, Springer, pp. 68–79.

[7] "Smali: An assembler/disassembler for Android's dex format," .

[8] "JASMIN HOME PAGE," .

[9] Bose, A. and K. G. Shin (2006) "On Mobile Viruses Exploiting Messaging and Bluetooth Services," in *Proceedings of the International Conference on Security and Privacy in Communication Networks (SecureComm)*, pp. 1–10.

[10] Mulliner, C. and G. Vigna (2006) "Vulnerability analysis of mms user agents," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, IEEE, pp. 77–88.

[11] Racic, R., D. Ma, and H. Chen (2006) "Exploiting mms vulnerabilities to stealthily exhaust mobile phone's battery," in *Securecomm and Workshops, 2006*, IEEE, pp. 1–10.

[12] Guo, C., H. J. Wang, and W. Zhu (2004) "Smart-Phone Attacks and Defenses," in *Proceedings of the 3rd Workshop on Hot Topics in Networks (HotNets).*

[13] Jamaluddin, J., N. Zotou, R. Edwards, and P. Coulton (2004) "Mobile Phone Vulnerabilities: A New Generation of Malware," in *Proceedings of the IEEE International Symposium on Consumer Electronics*, pp. 199–202.

[14] Dagon, D., T. Martin, and T. Starner (2004) "Mobile Phones as Computing Devices: The Viruses are Coming!" *IEEE Pervasive Computing*, **3**(4), pp. 11–15.

[15] Cyrus Peikari, J. R., Seth Fogie, "Summer Brings Mosquito-Borne Malware," .

[16] Singh, K., S. Sangal, N. Jain, P. Traynor, and W. Lee (2010) "Evaluating bluetooth as a medium for botnet command and control," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, pp. 61–80.

[17] Traynor, P., M. Lin, M. Ongtang, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta (2009) "On cellular botnets: measuring the impact of malicious devices on a cellular network core," in *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, pp. 223–234.

[18] Enck, W., P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth (2010) "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." in *OSDI*, vol. 10, pp. 255–270.

[19] Schlegel, R., K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang (2011) "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS).*

[20] (2013), "Google Play," .

[21] "Amazon Appstore for Android," .

[22] "MiKandi - World's Leading Adult App Store," .

[23] "F-Droid," .

[24] "Application Fundamentals," .

[25] "Average App Store Review Times," .

[26] Oberheide, J. and C. Miller (2012) "Dissecting the android bouncer," *SummerCon2012, New York.*

[27] "Android Malware Promises Video While Stealing Contacts," .

[28] "More malware found hosted in Google's official Android market," .

[29] "17 Bad Mobile Apps Still Up, 700,000+ Downloads So Far," .

[30] "Security Alert: DroidDream Malware Found in Official Android Market," .

[31] Felt, A. P., M. Finifter, E. Chin, S. Hanna, and D. Wagner (2011) "A Survey of Mobile Malware in the Wild," in *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM).*

[32] Zhou, Y., Z. Wang, W. Zhou, and X. Jiang (2012) "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium.*

[33] Bugiel, S., L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry (2012) "Towards Taming Privilege-Escalation Attacks on Android," in *Proceedings of the 19th Annual Network & Distributed System Security Symposium.*

[34] Bugiel, S., L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry (2011) "Practical and Lightweight Domain Isolation on Android," in *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM).*

[35] Enck, W., D. Octeau, P. McDaniel, and S. Chaudhuri (2011) "A Study of Android Application Security." in *USENIX security symposium.*

[36] Grace, M. C., W. Zhou, X. Jiang, and A.-R. Sadeghi (2012) "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, ACM.

[37] Au, K. W. Y., Y. F. Zhou, Z. Huang, P. Gill, and D. Lie (2011) "Short paper: a look at smartphone permission models," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, pp. 63–68.

[38] Felt, A. P., K. Greenwood, and D. Wagner (2011) "The Effectiveness of Application Permissions," in *Proc. of the USENIX Conference on Web Application Development (WebApps).*

[39] Felt, A. P., E. Chin, S. Hanna, D. Song, and D. Wagner (2011) "Android Permissions Demystified," in *Proc. of the ACM Conf. on Computer and Communications Security (CCS).*

[40] Au, K. W. Y., Y. F. Zhou, Z. Huang, and D. Lie (2012) "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, pp. 217–228.