Android Market Reconstruction and Analysis

Matthew L Dering
School of Computer Science
and Engineering
Penn State University
State College, PA 16802
Email: dering@cse.psu.edu

Patrick McDaniel
School of Computer Science
and Engineering
Penn State University
State College, PA 16802
Email: mcdaniel@cse.psu.edu

***Abstract*—With the rise in both personal and official smartphone use by the military, large scale application store analysis is an appealing idea, but until now there has been no real method of quickly accumulating a large library for analysis. In this paper we present a population study of permission and library use in Android apps, obtained from the Google Play Store. To accomplish this, we present a novel method for quickly reconnoitering a large database using a novel wordlist approach. Employing this method we compiled a library of over 700,000 applications. By leveraging program analysis and data mining techniques, we analyzed how permissions and libraries are used in real world settings on a population-wide level. From this we were able to make several claims about the health and apparently direction of the Android ecosystem.**

I. Introduction

Android is currently the worlds most popular mobile operating system. It can be found in many classes of devices from set top boxes to tablets to gaming consoles. However, most of the hardware running Android OS comes in the form of smartphones. One of the most appealing aspects of the Android operating system is its included store to download small self contained applications. Currently the market boasts over 1,500,000 apps, most of which are free for users to download and install. Such a large catalog comes with many questions of it's own, such as:

- How are different operating system features used? How might they be improved from a security standpoint? Given the low barrier of entry, how are developers choosing to use readily available code?
- Is there a way to efficiently at a glance obtain metrics and other information about the apps listed in the catalog? Could this method be improved?

To this end, we embarked on constructing a system that would be able to amass and analyze a large amount of applications quickly. Since Google does not freely and easily distribute its market, it fell to us to attempt to gather these applications for further reconnaissance of the security impact made by these public markets. In their work [1] Bugiel et al make the case for increased protection between applications when the device is provisioned for use by a party such as the military. They argue for allowing personal use of devices to avoid encouraging users to circumvent security in place through rooting or other methods as this would jeopardize any assets residing on the device. However, the larger scale real world impact of this idea has yet to be investigated. With this in mind, this paper makes the following contributions:

- We have accumulated what we believe to be the largest collection of Android metadata outside of store operators, currently over 1.5 million app:version pairs. Similarly, we have assembled over 700,000 apps binaries saved locally spanning over 450,000 apps, many with several versions saved.
- We outline how we were able to accumulate this large corpus with increased efficiency. Using our heuristics we were able to greatly increase our searching efficiency.
- We analyze how permissions are used, and how use of each permission interacts with use of others, and discuss the guiding principles of permission use.
- Similarly, we investigate how do developers make use of many popular Android libraries.

II. Background

Android is a popular phone operating system, implemented as middleware on top of a Linux based Kernel. It provides an extensive API written in Java which allows for developers to access many different services provided by the OS, such as a camera, location, network access and external storage. It has seen widespread adoption in the armed forces. There is a special application marketplace for apps related to the Army available for installation on personal phones [2], developed in response to an app-building competition. DARPA also has several Android related programs currently in progress focusing on growing this store with more helpful software [3] as well as in depth analysis of apps in their own store [4]. However, so far there has not been much large-scale work on public markets, which represent a serious security impact.

The Google Play market is currently the most popular Android app store, and hence it was our focus. In order to be listed in the market, a developer must submit a compiled application to the store using a developer account. The application must also contain other important information in a file called the manifest, such as requested permissions. Finally it must pass the Bouncer, a proprietary dynamic analysis tool, which has been scrutinized in the past[5]. When a user finds the app, they will be prompted to accept the permissions it requests in the developer defined manifest file. Once a user accepts the

978-1-4799-6770-4/14 $31.00 © 2014 IEEE
DOI 10.1109/MILCOM.2014.52

proposed set of permissions, the application will download and install.

A. Android Permissions

Android implements a permissions-based security model. A permission allows a holder to perform some set of privileged actions. Each permission is granted at install time by a user, when they choose to install the applications. When an application calls a portion of the API, it may invoke a permissions check, failure of this check will throw an exception, 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 probe an applications permissions without the users knowledge.[6]

Previous work has examined how permissions are used in real applications on the market [7]. Enck et al [8] proposed Kirin to allow for dangerous permission combinations to be specified and subsequently identified. Xmandroid [9] 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.

III. App Store Reconstruction

To systematically build a large corpus of apps, we found that the problem naturally decomposed into two smaller problems: App Discovery and App Acquisition. In order to interface with the Google Play Market, we employed an existing third party reversed API [10], and augmented it with some improvements of our own, which have since been updated in the third party API as well. The API can be used to download a specific app, once that app has been discovered. Using these two pieces of functionality, we constructed 2 corresponding applications.

Broadly, our construction consists of an application that queries the Google Play store for different words, and using these search results to download many applications. Our goal for using this construction was to obtain the largest number of unique apps possible. In this section we will discuss this process in more detail and our novel method of solving the problems encountered.

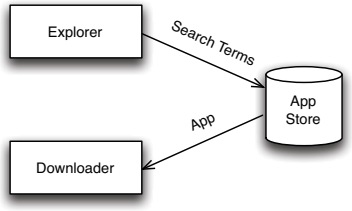


Fig. 1: General Construction

A. App Discovery

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. 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. The main 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 search 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). These 3 words would result in a maximum of 1,500 apps being discovered, however, in practice, due to result overlap, it will be significantly less than that.

Currently the Google Play Market contains around 1.5 million applications. Our search based approach 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, so our goal was to smartly target lists of words to satisfy 3 main intuitions. We constructed wordlists to be relevant, discriminatory and categorical.

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. The converse of this is to try to emphasize words that seem likely to appear in applications. For example lists of gaming terms and Relaxing activities were successful.

2) Discriminatory: This was the most challenging characteristic to maintain. Discriminatory means that a search is likely to result in a good amount of new applications. This idea is a direct consequence from our two main constraints from the API, the arbitrary sorting, and the 500 result limit. Words that are in violation of this discriminatory requirement

will result in very low efficiency. In fact, this is the largest source of inefficiency in our crawling procedure, with many sets not containing any novel applications. 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. Alternatively a list of every part of a car, would likely only return car related apps, which 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 together. For example, an application is unlikely to contain both emily and katie, or New York and Los Angeles.

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. Having something that lends itself to listing can also assist with the other criteria as well. Successfully doing so 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 discriminatory power. Some of the more successful lists we employed were: last names, colors, common typos, first names, last names, common Chinese, Japanese, Russian, Arabic words and names, adjectives, corporations, popular social networks, and top level domains.

B. Market Downloader

The metadata acquired in the previous process is fairly exhaustive. Nearly every field depicted on the details page for an app in the Google Play app on an Android device is included. This includes basic information such as title, category, price, developer email, and description. It also includes more advanced data such as rating, number of times it has been rated, installation size, and permissions requested by the app, which will be displayed at install time, as well as a proprietary application ID.

With this Application ID in hand, we turn to the Application Acquisition portion of the API. First, using our API, a message requesting permission to download the app. To do this, a request called a GetAssetRequest containing the App ID must be sent. If the download is authorized, the server will reply with a GetAssetResponse. A valid GetAssetResponse contains several pieces of information, including a URL and a cookie which acts as a token authorizing the download.

For security and commercial purposes, Google employs a fairly aggressive bandwidth limiter on its app store. The bandwidth is limited on a per account basis. Each account is allowed to download a certain amount of data, approximately 700 MB, per day. In order to scale this project well, we registered around 25 accounts, and utilized them in a round robin format.

Each request to Google is accompanied by a ”Request Context” object. This context contains many pieces of identifying information that is validated by the server. Among them a device identifier, and an Authorization Token. This token is obtained on login and must be kept for the duration of a session and presented with every request. Since this context is the only identifying piece of information provided to a server, the process switching accounts is a matter of changing which context is provided with a given request. We maintain a list of each context, and enforce a policy of not using any account more than every 5 minutes.

IV. Results

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.

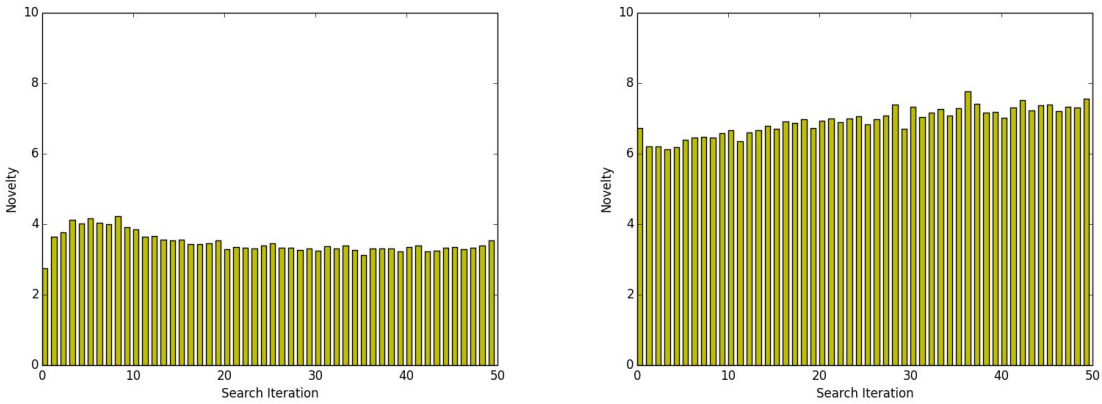
A. Wordlist Performance

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. We also laid out several criteria we believe a good word list should have.

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. 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 elected a naive approach: a list of the 1000 most popular english words. Comparing our constructed lists against this list would therefore prove instructive in quantifying the novelty afforded by enforcing these properties.

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. This means that per word, the discoverer may send up to 100 requests, 50 each for free and paid. 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 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



(a) Control (1000 Most Popular Words)

(b) Large Cities around the World

Fig. 2: Efficiency vs depth of search

TABLE I: Word Results

Desc	Len	Reqs/word	Novel/Req	Novel Apps
Pop words	1000	80.8	3.42	86810
Cities	334	24.3	6.66	54339
Names	999	34.1	4.93	161742
2 letters	676	61.2	4.77	131238
Rlx terms	360	71.9	4.76	122254
Soc Nets	115	45	7.02	36342
Chinese	132	26.8	4.44	14880
Colors	899	18.6	4.77	79691

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 2. To make the graph more readable, we divided the efficiency measurements into 20 buckets of 5% of the search terms, with each bar representing the average efficiency of that portion of the search.

B. Application Acquisition

Using this method we were able to obtain metadata for 90% of the Google play market in a matter of weeks. Some statistics of the metadata we possess is pictured in Table 2.

TABLE II: 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. Specifics about how many applications we were able to download are also in Table II. 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.

C. Permission 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. From here 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.

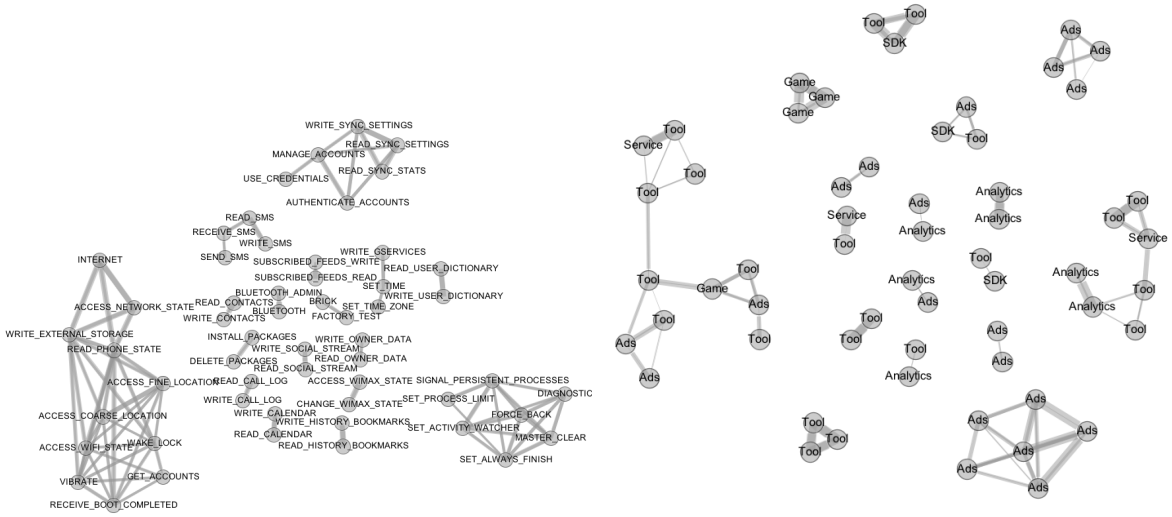
In order to get a better sense of how permissions are used, we turned to traditional data mining techniques. One of the things we wanted to examine was similarity among permission use. We felt this would give us the best general sense of how permissions are used. Typically this is accomplished by measuring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

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.



(a) Permissions

(b) Libraries

Fig. 3: Usage

We first placed a point for each relevant permission, and connected each permission to its partner with a line, of a weight of P(A|B). That graph can be seen in figure 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 connected permissions. These pairs are typically made up of closely related permissions. For example `READ_CALENDAR` and `WRITE_CALENDAR` or `BLUETOOTH` and `BLUETOOTH_ADMIN`. 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 such as `WRITE_EXTERNAL_STORAGE`, permissions to access phone location, 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 consistent with the general goals of the permissions system.

The final cluster, however, is illuminating. Android provides 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. We note that over 40,000 applications requesting at least one of these permissions. There are 2 possibilities for why these permissions appear in such a way: developer misunderstanding or permissions probing.

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. Alternatively, if these are applications that are indulging in permission probing, this might represent a security finding as well, and would merit further study.

D. 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. Currently, we have analyzed 181,398 applications.

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 sample, we found that 6 of the top 20 libraries were related to advertising with the most popular library being the Google ads library, com.google.ads, which appeared in 32% of applications.

E. Library Usage

Using this design, 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 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. On further inspection, we found that these libraries were actually part of one larger library.

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:

- Gaming library isolation. This indicates 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 indicating 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, this is interesting that their underlying goals do not overlap in any specific way.

We believe that displaying the interactions between libraries gives a greater insight into how developers create applications, as well as how the different types of functionality provided by these libraries interact.

V. Conclusion

In this paper we presented a new system enabling large scale app analysis. We proposed a method to recreate a large database under some constraints, and evaluated its efficacy in recreating the Google Play Market. Using our database of over 1 million applications, we amassed over 700,000 Android applications locally. This is an important step in launching continuing large scale app analysis. By using existing data mining techniques and metrics we analyzed both Library use and Permission use in our app population. We were able to find correlations and co-residency of both permissions and Libraries. By understanding these findings we were able to observe that the permissions choosing process is still somewhat a mystery to most developers. Finally, we observed that advertising libraries are used in conjunction very often, with some applications boasting as many as 6 different advertising libraries.

References

[1] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 51–62.

[2] (2012) Army launches apps marketplace prototype. [Online]. Available: http://www.army.mil/article/75966/Army_launches_apps_marketplace_prototype/

[3] Transformative apps. [Online]. Available: http://www.darpa.mil/Our_Work/I2O/Programs/Transformative_Apps.aspx

[4] Automated program analysis for cybersecurity. [Online]. Available: http://www.darpa.mil/Our_Work/I2O/Programs/Automated_Program_Analysis_for_Cybersecurity_(APAC).aspx

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

[6] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12. New York, NY, USA: ACM, 2012, pp. 101–112. [Online]. Available: http://doi.acm.org/10.1145/2185448.2185464

[7] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 73–84.

[8] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 235–245. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653691

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

[10] An open-source api for the android market. [Online]. Available: https://code.google.com/p/android-market-api/