# Maturity and Security: Static Analysis of Reverse-Engineered Android Applications

Daniel E. Krutz, Andrew Meneely, Casey Klimkowsky, and Samuel Malachowsky
Rochester Institute of Technology, Rochester, NY, USA
{dxkvse, axmvse, cek3403, samvse}@rit.edu

***Abstract*—Android is currently the most popular mobile operating system in the world, with a myriad of applications ("apps") freely available to users. As with most software, Android apps can put users at risk with inadvertent defects, vulnerabilities, and even intentionally placed malware. App stores typically mitigate these risks by moderating app reviews and managing the deployment of new releases. As a result, Android users may arrive at the belief that by upgrading their apps, they are installing a more stable and secure product. In this research, we investigate the maturity of these applications and their relationship with quality and security by statically analyzing Android applications. We collected and reverse-engineered 30,020 Android applications from the GooglePlay store and analyzed each app using six static analysis tools. We examined maturity (via release number), application size, rate of potential defects, adherence to coding standards, rate of overprivileged settings, privileges used, potential vulnerability level, and number of code clones. We also compared measurements between benign and known malware applications. Among our results were the conclusion that Android applications degrade over time in potential vulnerabilities and number of overprivileged permission settings. We have publicly released our result set for examination with a robust web-based tool.**

## I. Introduction

Android users download more than 1.5 billion applications ("apps") from GooglePlay every month [2]. Apps are a major part of mobile consumer technology and have changed the computing experience of our modern digital society, allowing users to perform a variety of tasks not previously possible in a portable environment.

Software, mobile or not, is in need of constant maintenance. Android apps often contain inadvertant defects and vulnerabilities that can seriously impact a mobile user. Even worse, malicious developers regularly create malware apps to attack trusting Android users. App stores such as GooglePlay mitigate these risks by managing the delivery of app updates and moderating review systems so that users can make safe choices in what they install on their devices. As a result, apps are routinely updated for bug fixes, vulnerability mitigations, support for new hardware, and feature additions.

With a constant stream of updates, however, users may develop the perception that their favorite apps are improving as the app is maturing. Between releases, developers have the opportunity to refactor, redesign, respond to reviews, and improve the product overall. However, as an app gains users, the size and complexity of the code base can also become unwieldy, leading to a regression of quality and security.

For developers, static analysis tools are one way of identifying potential risks of defects or vulnerabilities. Modern static analysis tools have been adapted to specific platforms, such as Android, to examine risks in areas such as overprivileges, coding standards, and potential defects. In recent academic studies [20, 24, 36], static analysis tools have also been used as one method of measuring the quality and security of mobile software. An empirical analysis of a large body of Android applications over time can therefore provide a broad view into what "improvements" users are seeing as their apps are continually updated.

*The goal of this work is to provide a overall understanding of the relationship between the maturity of mobile applications with potential quality and security defects.* We collected and reverse engineered 30,020 Android applications in 41 different genres from the GooglePlay store. We analyzed each of the apps using six static analysis tools: Stowaway [20], AndroRisk [1], CheckStyle [4], JLint [9], Simcad [35], and APKParser [3]. We examined maturity (via the release number), application size, rate of potential defects, adherence to coding standards, rate of overprivileged settings, potential vulnerability level, and number of code clones. We also compared our measurements between benign and known malware applications. Finally, we examined the relationship between our quality metrics and security metrics.

The contributions of this work are:

- An empirical analysis of how potential defects and vulnerabilities evolve over time in Android applications.
- A public data set of our results for future analysis by other researchers. This includes robust search and reporting mechanisms.

## II. Research Questions

This research is guided by the following questions. More details on how we collected our data can be found in Section IV, and details on how we analyzed these metrics can be found in Section IV-C.

**RQ1:** *Does the number of potential security risks increase over time?*
We explore if applications increase or decrease in their potential security risks. The security risks we examine are overprivileges and identification of vulnerabilities as identified by a static analysis tool. This analysis is discussed in Section V.

We perform this analysis with Stowaway to measure overprivileges and Androrisk to measure an application's risk. See

section IV-C for more details on these metrics.

**RQ2:** *What are the most pervasive overprivileges?* Defining overly broad permissions for an Android app is a simple mistake that can lead to security issues arising in the future. We measured the number and types of overprivileges across our data set to examine the most common mistakes.

**RQ3:** *What is the variation of risks across genres?* Different types of apps may lend themselves to different kinds of maturity, feature evolution, or security risks. We evaluate our results within GooglePlay genres.

**RQ4:** *Are benign apps measured as more secure when compared against known malware?* To investigate the soundness of the static analysis tools and our measurements, we compare our measurements against a population of known malware. We examine the same risks as we did in RQ1.

## III. Android Applications

The Android operating system is the most popular mobile platform in the world with apps being available on numerous types of devices from a variety of manufacturers [2]. This flexibly has allowed the Android operating system to flourish, but results in many different hardware platforms and OS versions for app developers to support.

### A. Android Application Structure

The Android application stack is comprised of four primary layers. The top layer is the Android application layer, which is followed by the the three application framework layers. The Android Software Development Kit (SDK) allows developers to create Android applications using the Java programming language. Isolation between Android applications is enforced through the use of the Android sandbox [12], which typically prevents applications from intruding upon one another.

*Intents* are a communication mechanism to exchange information between the components (*Activities*) of an Android application, and are reported to the user upon installation and use. Inter Process Communication (IPC) is the composition mechanism performed using Intents which is used to invoke another application component. Attacks that exploit Intents for malicious reasons include *permission collusion*, *confused deputy*, and *intent spoofing* [18, 22, 25, 30].

Android applications are packaged in APK files, which are compressed application files which includes the application's binaries and package metadata. Table I shows the breakdown of a typical APK file.

TABLE I: APK Contents

| File | Description |
|---|---|
| AndroidManifest.xml | Permissions & app information |
| Classes.dex | Binary Execution File |
| /res | Directory of resource files |
| /lib | Directory of compiled code |
| /META-INF | Application Certification |
| resources.arsc | Compiled resource file |

Android applications are available from a variety of different locations including AppksAPK[1], F-Droid[2], and the GooglePlay store[3]. These stores differ from the iOS app store, which forces all non-jailbroken devices to access applications through an Apple controlled store. GooglePlay provides verification of uploaded applications using a service called Bouncer which scans applications for malware [15]. In spite of these efforts, malicious apps are sometimes found on the GooglePlay store [41]. GooglePlay separates apps into *Genres* based on their realm of functionality, some of which are Action, Business, Entertainment, Productivity, and Tools. The *AndroidManifest.xml* file contains permissions and application information as defined by the developer.

### B. Android Permission Structure

Android developers operate under a permission-based system where apps must be granted access to various areas of functionality before they may be used. If an app attempts to perform an operation which it does not have permission, a *SecurityException* is thrown. When an Android app is created, developers must explicitly declare in advance which permissions the application will require [20], such as the ability to write to the calendar, send SMS messages, or access the GPS.

When installing the application, the user is asked to accept or reject these requested permissions. Once installed, the developer cannot remotely modify the permissions without releasing a new version of the application for installation [31], prompting the user if new permissions are required. These security settings are stored in the AndroidManifest.xml file and include a wide range of permissions, some of which are *INTERNET*, *READ_CONTACTS*, and *WRITE_SETTINGS*. Unfortunately, developers often request more permissions than they actually need, as there is no built in verification system to ensure that they are only requesting the permissions their application actually uses [20].

A basic principle of software security is the *principle of least privilege*, or the granting of the minimum number of privileges that an application needs to properly function [29]. Granting more privileges than the application needs creates security problems since vulnerabilities in other applications, or malware, could use these extra permissions for malicious reasons. Additionally, this limits potential issues due to non-malicious developer errors. Unfortunately, due to the lack of granularity of the permission spectrum used by Android, the developer must often grant more permissions to their application than it actually requires. For example, an application that needs to send information to one site on the internet will need to be given full permissions to the internet, meaning that it may communicate with with all websites [23].

In this study, we use the term *overprivilege* to describe a permission setting that grants more than what a developer needs for the task. Likewise, an *underprivilege* is a setting

[1] http://www.appsapk.com/
[2] https://f-droid.org/
[3] https://play.google.com/store

for which the app could fail because it was not given the proper permissions. Overprivileges are considered security risks, underprivileges are considered quality risks.

## IV. App Collection & Static Analysis

We analyzed 30,020 Android application files over a period of 3 months using a variety of different tools. The results of this analysis have been stored in a publicly accessible database located on our project website[4]. Our methodology is as follows:

1) Collect APK files
2) Reverse-engineer binaries
3) Execute static analysis tools
4) Complete evaluation (see Section V)

We created the *Darwin* tool that downloads Android Application (.apk) files and invokes various static analysis tools against these files.

### A. Step 1: Collect APK files

Android APK files were pulled from GooglePlay with a custom-built collector, which used *Scrapy* [5] as a foundation. We chose to pull from GooglePlay since it is the most popular source of Android applications [10] and was able to provide various application information such as the developer, version, genre, user rating, and number of downloads. To limit the impact of seldom-downloaded applications, we divided of our results into two groups: applications with at least 10,000 downloads, and those with less than 10,000 downloads. Of the 30,020 applications downloaded, 12,215 had at lt least 10,000 downloads, while 17,805 had fewer.

### B. Step 2: Reverse-engineer binaries

Some of our static analysis tools require source code instead of binary code, so we followed a reverse engineering process similar to that as proposed by previous research [24, 39]. For many of our static analysis tools, the downloaded APK files had to be decompiled to .java files. The first step was to unzip the .apk file using a simple unix command, which creates the files shown in Table I. Next, we used two open source tools to complete the reverse engineering process. These were:

- **dex2jar [6]:** Convert the .dex file into a .jar file. A java jar command is then used to convert this to .class files.
- **jd-cmd [8]:** A command line decompiler that converts .class files to .java.

Additionally, we recorded the number of extracted class and java files. The de-compilation process is shown in Figure **??**.

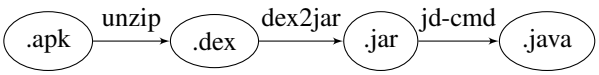


Fig. 1: APK Extraction Process

[4]http://darwin.rit.edu
[5]http://scrapy.org

### C. Step 3. Execute static analysis tools

The next phase was to analyze the extracted source code for a variety of metrics, including potential security risks, permissions issues, potential non-security defects, and misuse of coding standards. We also collected information about software clones, which are functionally equivalent portions of an application that may differ syntactically. A sign of poorly written software, clones may be detrimental to an application in a variety of ways, including increased maintenance costs and inconsistent bug fixes [27]. We used the following tools for our analysis:

**Stowaway[20]:** Reports the overprivileges and underprivledges of an application, which we recorded. Slight modifications were made to the existing version of Stowaway to accommodate our process and current Android applications with updated permissions. Permlyzer [38], a more modern permission detection tool, was not used since its authors have not made it available for download.

**AndroRisk[1]:** A component of the Androguard reverse engineering tool which reports the risk indicator of an application for potential malware. We recorded the reported risk level for each APK file.

**CheckStyle[4]:** A development tool to measure how well developers adhere to coding standards such as annotation usage, size violations, and empty block checks. We recorded the total number of violations of these standards. Default application settings were used for our analysis.

**Jlint[9]:** Examines java code to find bugs, inconsistencies, and synchronization problems by conducting a data flow analysis and building lock graphs. We recorded the total number of discovered bugs. This tool was selected over FindBugs [7] since it was able to analyze the applications much faster, while still providing accurate results [28].

**Simcad[35]:** A powerful software clone detection tool which we used to record the number of clones discovered for each target application.

**APKParser[3]:** A tool designed to read various information from Android APK files including the version, intents, and permissions. We used the output from this tool to determine the application version, minimum SDK, and target SDK.

We also recorded other metrics about each application including total lines of code, number of java files, application version, target SDK, and minimum SDK.

Stowaway and Androrisk were able to analyze the raw APK files, while CheckStyle, Jlint, and Nicad required the APK files to be decompiled. All results were recorded in an SQLite [6] database, which is publicly available on the project website. The full analysis process is shown in Figure **??**.

## V. Evaluation

We evaluated the reverse-engineered Android applications in a variety of ways including comparing the effects of the maturity level, comparing benign applications to malware, and comparing genres with one another.

[6]http://www.sqlite.org/

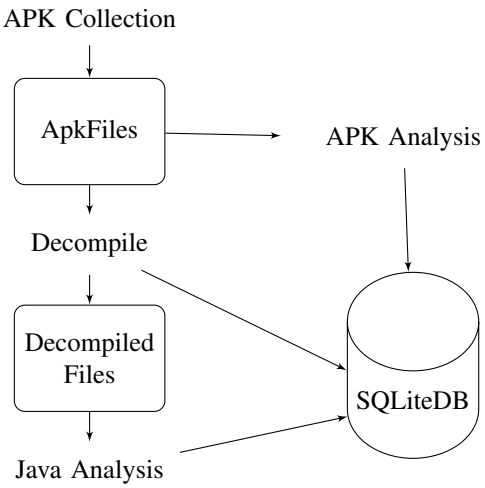


Fig. 2: APK Analysis Process

### A. RQ1: Does the number of potential security risks increase over time?

We examined the effect the version of the application had on the observed quality of that application. All applications downloaded from GooglePlay were grouped together according to their application version. We separated all applications into six distinct groups, with applications with a version number of 0 - .99 assigned to group 0, versions 1 - 1.99 as group 1, and so on. We had 1225 applications with indecipherable version information, so we ignored those in this analysis. We then averaged scores for the groups to determine their average AndroRisk score, number of overprivileges, discovered defects, and adherence to coding standards. Adherence to coding standards was determined by dividing the number of mistakes found for each application by the number of classes in it, with the same process being used to determine defects. The AndroRisk score, code clones, and overprivileges were found by finding the average values of each for all applications in each group. User ratings were not considered since the ranges between the various groups were statistically insignificant. Finally, to assist with the visual representation of the results, values were normalized in the chart to follow consistent values with one another while retaining their accuracy. The results are shown in Figure **??**.

These results indicate that **potentials risks increase with release number**. Not only did the risk grow over each version, but the number of overprivileges generally grew as well. This indicates that as applications evolve, the drive for features may be overcoming careful avoidance of vulnerabilities. This may come as a surprise to users who believe that with each update they receive better and better apps.

Furthermore, these results also indicate that, over time, developers may be paying more attention to small coding mistakes (that Jlint might find). Code clones, however, nearly triple in number over the first five releases, indicating the possibility that code or code design may be rotting and the *Don't Repeat Yourself* principle may be traded in for speed and new features. Interestingly, coding standards mistakes generally decreased in spite of the aforementioned issues. A correlation was also found between the discovered Jlint defects and coding standard deviance.

### B. RQ2: What are the most pervasive overprivileges?

A basic principle of security is the concept of granting of the least amount of privileges to an application that it needs to properly function. Granting extra privileges creates unnecessary security vulnerabilities. Previous research has found that while Android developers try to follow this principle, they often add extra privileges to make the app work properly or, due to confusion over the permission name, they add it unnecessarily believing its functionality sounds related to their app [20].

We analyzed the most overused permissions in 41 application genres ranging from Communication and Productivity to Sports, Puzzles, and Entertainment. We separated the applications into two groups, those with at least 10,000 downloads, and those with fewer. In Table II, we show all permissions types which had an overprivilege rate of at least 2.5% in either group of downloads. Complete results may be found on the project website.

TABLE II: Top Occurring Overprivileges

| | **%Apps** | |
|---|---|---|
| **Permission** | $\geq$ **10K** | $<$ **10K** |
| ACCESS_COARSE_LOCATION | 2.06 | 2.77 |
| ACCESS_FINE_LOCATION | 2.24 | 3.34 |
| ACCESS_LOCATION_EXTRA_COMMANDS | 2.69 | 5.34 |
| ACCESS_NETWORK_STATE | 3.02 | 3.04 |
| ACCESS_WIFI_STATE | 6.73 | 8.72 |
| CALL_PHONE | 7.02 | 13.56 |
| CAMERA | 2.76 | 0.01 |
| CHANGE_NETWORK_STATE | 3.84 | 1.4 |
| CHANGE_WIFI_STATE | 2.76 | 1.7 |
| FLASHLIGHT | 2.01 | 2.73 |
| GET_ACCOUNTS | 9.68 | 10.7 |
| GET_TASKS | 3.63 | 2.27 |
| MODIFY_AUDIO_SETTINGS | 2.42 | 2.73 |
| MOUNT_UNMOUNT_FILESYSTEMS | 2.91 | 1.99 |
| READ_EXTERNAL_STORAGE | 9.28 | 5.61 |
| READ_HISTORY_BOOKMARKS | 2.19 | 2.76 |
| READ_PHONE_STATE | 5.14 | 6.99 |
| READ_SMS | 3.08 | 1.2 |
| SEND_SMS | 1.21 | 4.48 |
| SYSTEM_ALERT_WINDOW | 5.96 | 4.03 |
| WAKE_LOCK | 2.87 | 2.49 |
| WRITE_CONTACTS | 4.04 | 4.22 |
| WRITE_EXTERNAL_STORAGE | 3.89 | 3.85 |
| WRITE_SETTINGS | 2.91 | 1.88 |

Some widely occurring overprivileges in both groups were *GET_ACCOUNTS*, which permits access to the list of accounts in the accounts service, *READ_EXTERNAL_STORAGE*, which allows the application to read from an external storage device, and *CALL_PHONE*, which allows an application to start a phone call without the user confirming it through the dialer interface [13]. The permission *GET_ACCOUNTS* is

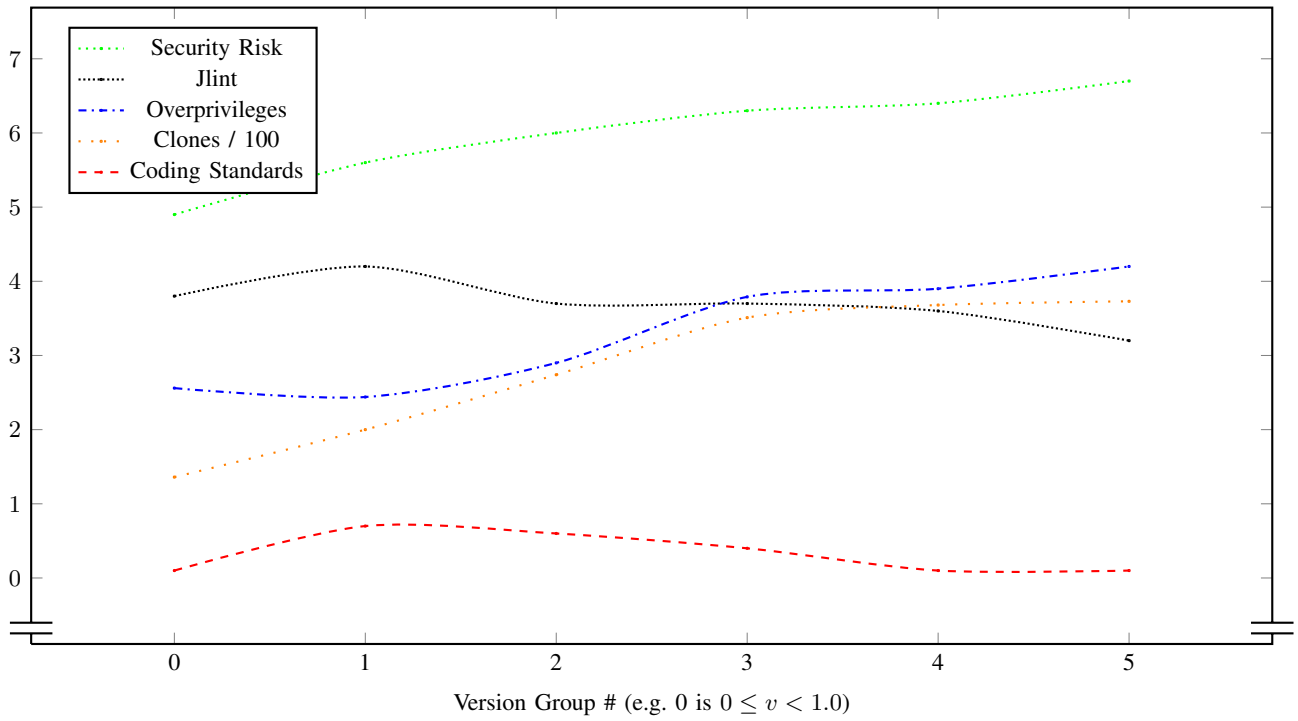


Fig. 3: Maturity and Metrics (RQ1)

potentially dangerous since a malicious application could gain access to all device accounts. *READ_EXTERNAL_STORAGE* is potentially dangerous since an application could be granted access to information which it may not need or should not see, while *CALL_PHONE* could allow a dangerous application to dial any phone number without the consent of the user. Previous research [20] has also found that many of our top identified overprivileges, including *ACCESS_NETWORK_STATE* *READ_PHONE_STATE*, and *WRITE_SETTINGS* are commonly found to be unnecessary permissions.

We next examined pairs of overpriviledges which frequently occurred together in each group. In Table III, we display all instances of permissions that occur together in at least 1.5% of applications in either group of at least 10,000 downloads, and that of less than 10,000 downloads. One commonly occurring overprivilege combination is CALL_PHONE and GET_ACCOUNTS, which is dangerous since it would allow a malicious application to gather private account information and make possibly expensive calls. READ_PHONE_STATE and ACCESS_WIFI_STATE or GET_ACCOUNTS and ACCESS_WIFI_STATE could potentially allow malicious software to use the open ACCESS_WIFI_STATE open to gather personally identifiable information [14].

Overall, we found that in applications with at least 10,000 downloads, 40.7% had at least one overpermission, 25.8% had at least 2, and 8.2% had 4 or more. Rates were generally higher for applications with less than 10,000 downloads. These results are shown in Table IV. In 2011, previous

TABLE IV: Applications With Overprivileges

| | **%Apps** | |
|---|---|---|
| **$\geq$ OverPrivs** | **$\geq$ 10K** | **$<$ 10K** |
| 1 | 40.7 | 47.7 |
| 2 | 25.8 | 29.6 |
| 3 | 16.6 | 19.4 |
| 4 | 11.7 | 14.4 |
| 5 | 8.2 | 9.3 |
| 6 | 6.9 | 6.9 |
| 7 | 5 | 5 |
| 8 | 4.3 | 2.8 |
| 9 | 39 | 2 |
| 10 | 2.7 | 1.7 |
| 10+ | 2.3 | 1.4 |

research found that approximately 33% of all applications were overprivileged with only about half of those containing more than one overprivilege and only 6% requesting more than four extra permissions [20]. This indicates that the rate of overprivileged applications is growing, along with the number of extra privileges for each of these applications. Although the number of available permissions has grown in that time the difference isn't significant enough to account for the change in overprivilege (134 available permissions in 2011 [20] vs. 146 in 2014 [11]).

### *C. RQ3: What is the variation of risks across genres?*

We next compared the various application genres to examine their rates of overprivileges, which we computed by dividing the number of overprivileges for all applications in each genre by the total number of applications in each genre. We

TABLE III: Top Rates Overprivileges Appear Together

| | | %Rate | |
|---|---|---|---|
| **Permission 1** | **Permission2** | **$\geq$ 10K** | **$<$ 10K** |
| ACCESS_FINE_LOCATION | ACCESS_COARSE_LOCATION | 1.56 | 2.48 |
| ACCESS_LOCATION_EXTRA_COMMANDS | CAMERA | 0 | 1.51 |
| CALL_PHONE | GET_ACCOUNTS | 1.35 | 4.41 |
| CALL_PHONE | SEND_SMS | 0.58 | 3.94 |
| CALL_PHONE | ACCESS_WIFI_STATE | 0.93 | 2.5 |
| CHANGE_CONFIGURATION | SYSTEM_ALERT_WINDOW | 0.57 | 1.75 |
| CHANGE_CONFIGURATION | READ_HISTORY_BOOKMARKS | 0.37 | 1.64 |
| CHANGE_CONFIGURATION | WRITE_HISTORY_BOOKMARKS | 0.32 | 1.63 |
| GET_ACCOUNTS | ACCESS_WIFI_STATE | 1.68 | 3.09 |
| GET_ACCOUNTS | SEND_SMS | 0.33 | 2.03 |
| MODIFY_AUDIO_SETTINGS | ACCESS_LOCATION_EXTRA_COMMANDS | 0.28 | 1.61 |
| READ_EXTERNAL_STORAGE | GET_ACCOUNTS | 1.68 | 1.2 |
| READ_HISTORY_BOOKMARKS | SYSTEM_ALERT_WINDOW | 0.87 | 1.88 |
| READ_HISTORY_BOOKMARKS | WRITE_HISTORY_BOOKMARKS | 0.91 | 1.84 |
| READ_PHONE_STATE | GET_ACCOUNTS | 1.35 | 1.92 |
| READ_PHONE_STATE | ACCESS_WIFI_STATE | 1.99 | 1.92 |
| READ_SMS | WRITE_SMS | 1.56 | 0.51 |
| WAKE_LOCK | ACCESS_WIFI_STATE | 1.29 | 1.51 |
| WRITE_CONTACTS | CALL_PHONE | 1.8 | 1.58 |
| WRITE_CONTACTS | CAMERA | 0 | 1.51 |
| WRITE_CONTACTS | ACCESS_LOCATION_EXTRA_COMMANDS | 0 | 1.5 |
| WRITE_HISTORY_BOOKMARKS | SYSTEM_ALERT_WINDOW | 0.56 | 1.69 |

TABLE V: Top Overprivileged Ratios Per Genre

| | Overprivilege Ratio | |
|---|---|---|
| **Genre** | **$\geq$ 10K** | **$<$ 10K** |
| Business | 2.66 | 2.21 |
| Communication | 4.29 | 2.73 |
| Lifestyle | 1.86 | 2.34 |
| Productivity | 3.08 | 1.54 |
| Role Playing | 2.05 | 1.12 |
| Shopping | 1.77 | 2.86 |
| Social | 2.8 | 2.47 |
| Tools | 2.01 | 1.21 |
| Travel & Local | 2.24 | 1.75 |

separated the results into applications with at least 10,000 total downloads and those with less than 10,000 downloads. The results of all genres with a rate of 2.0 or higher in either download group are shown in Table V.

Further analysis found the top 10 genres which had at least one overprivilege. For applications with at least 10,000 downloads, we found that Communication applications had a 77% likelihood of having at least one overprivilege, while Role Playing and Business were tied for second with an overprivilege rate of 68%. Complete results are shown in Table VI.

Next, we found the percentage of applications for each genre which contained at least one overprivilege, again separating them into groups of at least 10,000 downloads and less than 10,000 downloads. In Table VI, we show all genres which had at least 60% of applications containing at least one overprivilege in either download group.

We found that the Communication, Business, and Productivity genres frequently had at least one overprivilege (Table VI), but also contained a high ratio of overprivileges as well (Table V). The Education, Trivia, and Weather genres were least likely to contain overprivileges. Full results may be viewed on our project website.

TABLE VI: % of Genres With At Least 1 Overprivilege/App

| | %Apps | |
|---|---|---|
| **Genre** | **$\geq$ 10K** | **$<$ 10K** |
| Business | 68 | 70 |
| Casino | 65 | 36 |
| Communication | 77 | 69 |
| Family | 48 | 65 |
| Finance | 57 | 61 |
| Lifestyle | 59 | 65 |
| Productivity | 62 | 44 |
| Role Playing | 68 | 44 |
| Shopping | 64 | 73 |
| Simulation | 57 | 55 |
| Social | 71 | 67 |
| Transportation | 62 | 63 |
| Travel & Local | 67 | 62 |

Finally, we present overprivileges, AndroRisk score, and Jlint results across the top genres and version numbers as with RQ1 in Figures 4, 5, and 6. In each case, the top genres maintain an overall increase over time in keeping with the results of RQ1.

### *D. RQ4: Are benign apps measured as more secure when compared against known malware?*

We compared the results of the collected applications with at least 10,000 downloads from the GooglePlay store against 139 malware examples from the Contagio Mobile Mini Dump [5] and the Malware Genome Project [41] for the purpose of examining the differences between malicious apps and those which were considered benign. Since many of the malware examples represented only slight alterations from their counterparts, one result from each malware family was taken from

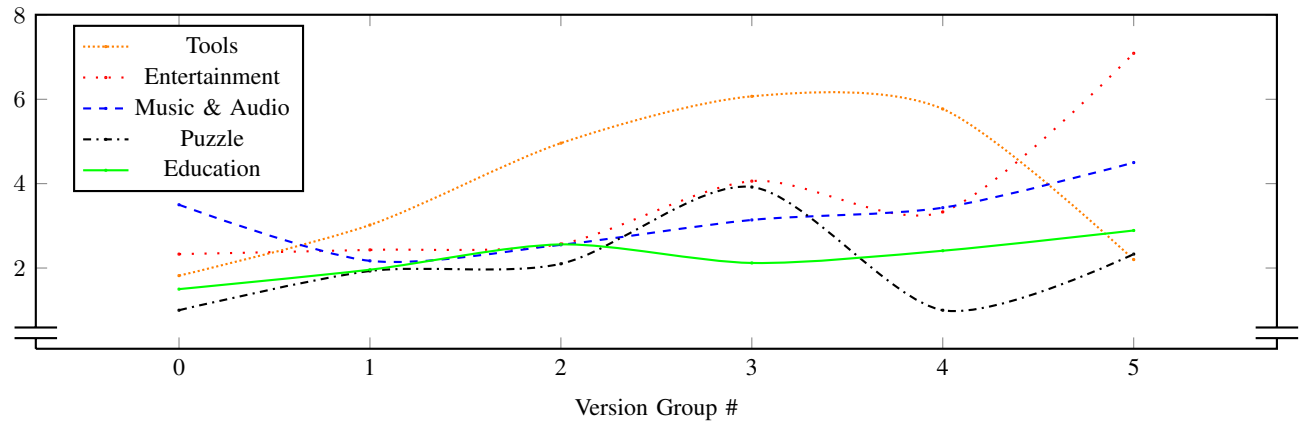


Fig. 4: Genres & Overprivileges

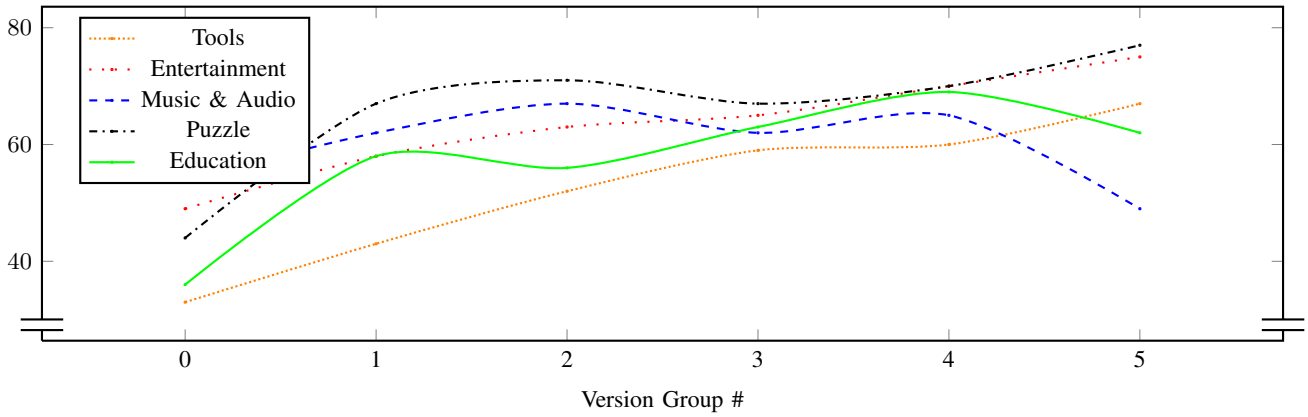


Fig. 5: Genres & AndroRisk Score

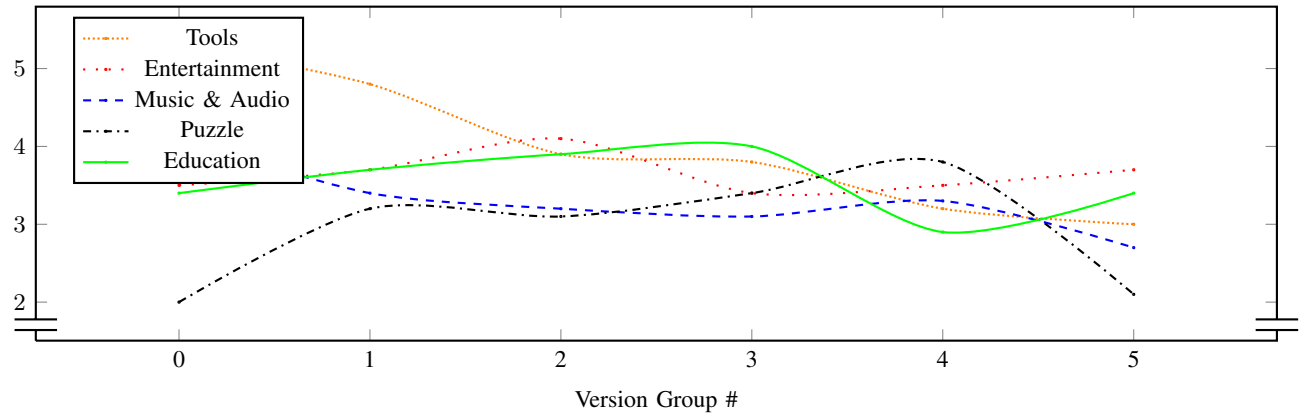


Fig. 6: Genres & Jlint Defects

TABLE VII: Top Occurring Overprivileges in Malware

| Permission | Malware | GooglePlay |
|---|---|---|
| WRITE_EXTERNAL_STORAGE | 10.7% | 3.9% |
| ACCESS_LOCATION_EXTRA_COMMANDS | 9.8 | 2.7 |
| CALL_PHONE | 9.8 | 7.0 |
| READ_HISTORY_BOOKMARKS | 9.8 | 0.1 |
| WRITE_HISTORY_BOOKMARKS | 8.9 | 0.1 |
| READ_SMS | 8.0 | 3.1 |
| SYSTEM_ALERT_WINDOW | 7.1 | 6.0 |
| WRITE_SMS | 7.1 | 1.8 |
| READ_CONTACTS | 6.3 | 0.7 |
| RECEIVE_SMS | 6.3 | 2.1 |

TABLE VIII: Malware vs. Non-Malicous

| Analysis Tool | Malicious | GooglePlay |
|---|---|---|
| AndroRisk | 46.25 | 53.72 |
| JLint/Class | .395 | .301 |
| Defect Count/Class | 11.07 | 1.87 |
| Overprivilege Ratio | 3.3 | 1.5 |

the Malware Genome Project to help limit the negative impact that families with many examples would have, creating biased results.

The compared areas included adherence to coding standards, discovered defects, permission gap, and the utilized permissions. This was accomplished by running the malicious applications through the same process as the applications collected from the GooglePlay store, described in Sections IV-B and IV-C. The results of this analysis are available on our website and are shown in Table VIII.

Most Android malware is piggybacked on existing applications whose malicious code is typically loosely coupled with the host application [19, 41]. We found that malicious applications had a much higher number of coding standards mistakes per class compared to their benign counterparts while also having a slightly higher number of defects per class. While we are not able to examine the development process of malware applications and interview its developers, we are able to draw several possible conclusions. Malware developers may be less likely to care about the actual user experience as opposed to developers of reputable applications, as they are probably less worried about things such as user reviews in the long run.

Not surprisingly, the malicious applications had a much higher number of overprivileges per application ($\frac{\#overprivileges}{\#apps}$) as compared to their benign counterparts. The top 10 most commonly occurring overprivileges, their rates, and how they differ from applications collected from the GooglePlay store, are shown in Table VII.

## VI. Publicly Available Dataset

Our data set is available from our publicly accessible GitHub repo[7] (linked from the Darwin website), which includes the scripts used for collecting apps and invoking the static analysis tools. The SQLite database with our complete results is updated on a regular basis from our scanning and analysis application. The goal of this dataset is to allow future researchers to both learn from and expand upon our work. This data includes the following fields for each app:

- Name
- Version
- Genre
- Number of downloads from GooglePlay
- Publication date on GooglePlay
- GooglePlay user rating
- Overprivileges
- Underprivileges
- Count of Jlint reported defects
- Count of CheckStyle coding standards mistakes
- Application size
- Count of .java files
- Vulnerability risk score from AndroRisk
- Code clone count from Simcad

Our project website (http://darwin.rit.edu) contains information about our project, links to our GitHub repository, and a robust reporting tool which will allow users to create their own data sets from over 30,000 analyzed applications. An example screenshot of the search and reporting functionality available for individual apps is shown in Figure 7.

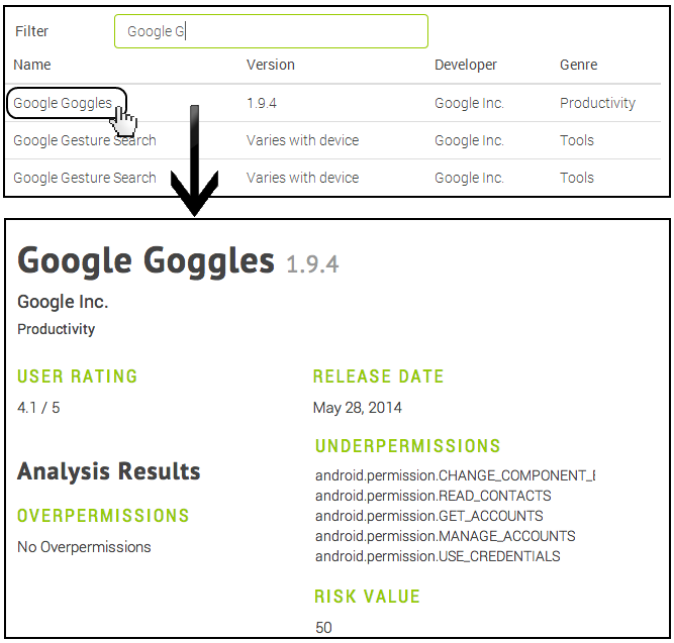


Fig. 7: darwin.rit.edu Website Reporting Tool

## VII. Limitations

Our maturity metric is based on the release version, and compares applications across release versions. Due to different preferences in versioning, developers may vary the way they iterate the version number (i.e. versions 1.0 to 2.0 or 1.0 to 1.1). Because GooglePlay reviews can be filtered by version, developers may have a low incentive to release frequently since that can effectively make positive reviews obsolete in the

[7]https://github.com/DroidDarwin

eyes of a potential customer. Because of this, we do expect some variation of maturity for the similar app version numbers, but the large number of applications processed may serve to mitigate this slightly.

While Stowaway is a powerful statical analysis tool which has been used in a substantial amount of previous research [23, 26, 33], it does suffer some drawbacks. Malicious code may be obfuscated and unnecessary API methods inserted into the application, rationalizing the permission [38]. Static analysis techniques can also be hindered by the Java reflection and may lead to inaccuracies [32, 34]. These types of limitations are inherent to all statical analysis tools.

We only analyzed applications from GooglePlay and not other sources such as AppksAPK or F-Droid, which would have led to more varied application origins. However, we feel the diversity of our applications was already quite robust since we collected 30,020 applications from 41 genres.

We also only examined free applications in our research due to cost constants. Thus, the measurements comparison of apps is not representative of the entire Android app market. Our results only apply as a comparison of free apps, not with paid apps.

## VIII. Related Work

Android applications have been extensively researched in numerous areas. The topic of reducing the permission gap in Android applications has received a considerable amount of attention recent years. Much of the existing work in this area has dealt with ways of reducing these unneeded permissions and the security vulnerabilities they may lead to. Jeon *et al.* introduced a framework for creating finer-grained permissions in Android. They believe that the course-grained permissions currently used by Android limit developers by forcing them to choose all of the permissions located in each "bucket" when they really only want to add a few of them. This leads to applications having many more permissions than they actually require. The authors believe that finer-grained permissions would lead to only having the needed permissions used by an application, and thus would lead to few vulnerability possibilities [23].

Wei *et al.* studied the evolution of Android to determine if the platform was allowing the system become more secure. They found that the privacy and security in the overall Android system is not improving over time and that the principle of least privilege is not being adequately addressed [37].

There have been innumerable studies analyzing mobile applications on a large scale. Sarma *et al.* evaluated several large datasets, including one with 158,062 Android applications in order to gauge the risk of installing the application, with some of the results broken down by genre. However, this work did not analyze the application using the range of static analysis tools which we used. Viennot *et al.* developed a tool called PlayDrode which they used to examine the source code of over 1,100,000 free Android applications. While the authors examined a very large number of applications, they largely only used existing information which could be gathered from GooglePlay and, while they carried out static analysis on the applications, they examined features such as library usage and duplicated code — not areas such as security vulnerability levels and overpriviledges, which was a part of our analysis.

While this work represents the largest known empirical analysis of developers allowing overprivileges to occur in Android applications, it is not the first research into developers not following the principle of least privilege. Felt *et al.* described some common developer errors found using their tool Stowaway including confusing permission names, the use of depreciated permissions, and errors due to copying and pasting existing code [20]. In another work, Felt *et al.* very briefly described some inclinations they had for why developers gave too many permissions to applications, but this was largely based on assumptions and not necessarily data [21].

Permlyzer is a tool which was built to determine where permissions are utilized in Android applications by using a mixture of static and runtime analysis [38]. This is a recently published tool, so it has not yet been discussed or used in a substantial amount of subsequent research. The authors were, however, able to achieve promising results and this may be a powerful tool for assisting in the permissions granting decision process for developers. *PScout* was another tool developed to extract permission specifications from Android applications using static analysis [16]. While the authors of this tool were able to achieve promising results, subsequent work has criticized this tool for not being accurate enough, since Android's permissions could be different at runtime — something the tool is not capable of discovering [40].

Bartel *et al.* and Wei *et al.* also discussed some basic, high level discoveries about why developers make these mistakes [17, 37]. While these works were beneficial for numerous reasons, no known works to date have explored the question of why developers do not adhere to the principle of least security as consistently as they should.

## IX. Conclusion

In this work, we demonstrated our technique of downloading and analyzing 30,020 Android applications in a variety of areas including security vulnerability level, overprivileges, code clones, potential defects, and coding standards mistakes. We found that, on average, Android applications increase in their potential security risks with each release while increase instances of code clones and decreasing potential defects. We conducted our analysis within genres and found that certain genres included more instances of overprivileges. As a control, we compared our analysis results against known malware and found that the static analysis tools we employed demonstrated high levels of security risk where malware was known. These results provide a broad overview of the state of the breadth of free apps in the GooglePlay store.

## Acknowledgment

This research would not have been possible without the hard work by two dedicated Software Engineering Students:

Shannon Trudeau and Adam Blaine. We would like to thank them for their dedication and the insights they have provided on this project.

## REFERENCES


[1] Androguard. https://code.google.com/p/androguard/.

[2] Android, the world's most popular mobile platform. http://developer.android.com/about/index.html.

[3] Apk parser. https://github.com/joakime/android-apk-parser.

[4] Checkstyle. http://checkstyle.sourceforge.net/.

[5] Contagio mobile. http://contagiominidump.blogspot.com.

[6] dex2jar. https://code.google.com/p/dex2jar/.

[7] Findbugs. http://findbugs.sourceforge.net/.

[8] jd-cmd. https://github.com/kwart/jd-cmd.

[9] Jlint. http://jlint.sourceforge.net/.

[10] List of android app stores. http://www.onepf.org/appstores/.

[11] Manifest.permission. http://developer.android.com/reference/android/Manifest.permission.html.

[12] Security tips. http://developer.android.com/training/articles/security-tips.html.

[13] Manifest.permission. http://developer.android.com/reference/android/Manifest.permission.html, July 2014.

[14] J. P. Achara, M. Cunche, V. Roca, and A. Francillon. Short paper: Wifileaks: Underestimated privacy implications of the access wifi state android permission. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless Mobile Networks*, WiSec '14, pages 231–236, New York, NY, USA, 2014. ACM.

[15] T. Armendariz. Virus and computer safety concerns. http://antivirus.about.com/od/wirelessthreats/a/Is-Google-Play-Safe.htm.

[16] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[17] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 274–277, New York, NY, USA, 2012. ACM.

[18] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[19] L. Deshotels, V. Notani, and A. Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, pages 3:1–3:12, New York, NY, USA, 2014. ACM.

[20] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[21] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[22] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[23] J. Jeon, K. K. Micinski, J. A. Vaughan, N. Reddy, Y. Zhu, J. S. Foster, and T. Millstein. Dr. android and mr. hide: Fine-grained security policies on unmodified android. 2011.

[24] S.-H. Lee and S.-H. Jin. Warning system for detecting malicious applications on android system. In *International Journal of Computer and Communication Engineering*, 2013.

[25] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 51–60, New York, NY, USA, 2012. ACM.

[26] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 71–72, New York, NY, USA, 2012. ACM.

[27] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[28] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256. IEEE, 2004.

[29] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[30] S. Salva and S. Zafimiharisoa. Data vulnerability detection by security testing for android applications. In *Information Security for South Africa, 2013*, pages 1–8, Aug 2013.

[31] K. Shaerpour, A. Dehghantanha, and R. Mahmod. Trends in android malware detection. *Journal of Digital Forensics, Security & Law*, 8(3), 2013.

[32] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41(6):387–400, June 2006.

[33] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries.

[34] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and

O. Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[35] M. Uddin, C. Roy, and K. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 236–238, May 2013.

[36] T. Vidas, N. Christin, and L. F. Cranor. Curbing android permission creep. In *In W2SP*, 2011.

[37] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 31–40, New York, NY, USA, 2012. ACM.

[38] W. Xu, F. Zhang, and S. Zhu. Permlyzer: Analyzing permission usage in android applications. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 400–410, 2013.

[39] S. Yerima, S. Sezer, and G. McWilliams. Analysis of bayesian classification-based approaches for android malware detection. *Information Security, IET*, 8(1):25–36, Jan 2014.

[40] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[41] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.