Quality and Security: An Evaluation of 70,000 Reverse Engineered Android Applications

xxxxxxxxxxxxxxx
xxxxxxxxx
xxxxxxxxx
xxxxxxxxx
xxxxxxxxx, xx, xxx
xxxxxx@xxxxx.xxx

ABSTRACT

Android is the most popular mobile operating system in the world. Unfortunately, the quality of many of these applications ('apps') suffer for a variety of reasons including misuse of permissions, bugs, and security vulnerabilities. In order create better, more secure apps it is important to understand more about them.

Over the course of more than a year, we collected and reverse engineered 70,785 Android apps from the Google Play store and analyzed each app using several static analysis tools to record a variety of information about each of them. We found that 'Communication' apps suffer from the highest rate of overprivileges while requesting the most permissions, and that several security related permissions are the most common overprivileges. We also found that apps are growing on an annual basis in terms of size and requested permissions, and that apps with at least 10,000 downloads were typically larger and requested more permissions with more Jlint defects discovered per lines of code (LOC) as compared to their less downloaded counterparts. We also present an easy to use, robust website and dataset for others to use in their own research.

Categories and Subject Descriptors

D.2.3 [**Coding Tools and Techniques**]: Standards

Keywords

Android Applications, Mobile Development, Software Engineering

1. INTRODUCTION

Android apps often contain inadvertent defects and vulnerabilities that can seriously impact a mobile user. These apps are routinely updated for bug fixes, vulnerability repairs, support for new hardware, and feature additions. Static analysis tools are one way developers may identify potential 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. An overprivilege is a permission which has been granted to an app which the app does not actually need. Overprivileges are considered to be security concerns since they leave an app exposed to various bugs and vulnerabilities [10]. Conversely, an underprivilege is when an app does not request enough permissions.

In recent academic studies, static analysis tools have also been used as one method of measuring the quality and security of mobile software [10, 14, 23]. *An empirical analysis of a large body of Android applications over time can therefore provide a broad view into how individual apps and app genres relate to one another and provide more insight on these apps.*

The goal of this work is to better understand Android apps through the use of static analysis. We collected and reverse engineered 70,785 Android applications in 41 different genres from the Google Play store. Each of the apps were analyzed using six static analysis tools: Stowaway [10], AndroRisk[1], CheckStyle[2], Jlint[3], Simcad [22], and APKParser[4]. We examined application size, rate of potential defects, adherence to coding standards, rate of overprivileges, potential vulnerability level, and number of code clones. An additional benefit of our work is a publicly available dataset and robust website which may be used by researchers, developers, students, and general Android users to better understand these apps.

Our research is guided by the following questions:

RQ1: *How do app genres compare in terms of quality metrics?*
We found that 'Tools' apps suffer from the highest rate of coding standards defects per line of code (LOC), while 'Communication' apps have the highest rate of overprivileges.

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. We found that `GET_ACCOUNTS`, `READ_EXTERNAL_STORAGE`, `ACCESS_WIFI_STATE`, and `CALL_PHONE` were the most common overprivileges in apps with at least 10,000 downloads.

RQ3: *How are apps changing over time?*
We separated apps into different groups based upon the year each version was published to the Google Play store. We found that apps are not only becoming larger, but are requesting more permissions and are becoming more over & under privileged.

RQ4: *Do quality metrics change for apps with more or less than 10K downloads?*
We compared the static analysis results for apps with less than 10,000 downloads against apps with more than 10,000 downloads to understand if there was a difference between more popular apps compared with those which were more seldom used. We found that

[1]https://code.google.com/p/androguard/
[2]http://checkstyle.sourceforge.net/
[3]http://jlint.sourceforge.net/
[4]https://github.com/joakime/android-apk-parser.

apps with less than 10,000 downloads have a higher rate of coding standards mistakes per LOC, and have more overprivileges. Apps with at least 10,000 downloads are larger in terms of LOC, have a higher vulnerability score reported by AndroRisk, have more Jlint defects per LOC, posses more underprivileges, and request more permissions.

The rest of the paper is organized as follows: Section 2 discusses related works, while section 3 describes the structure of Android apps. Section 4 provides details of how we collected the apps and conducted our static analysis on them. Section 5 discusses our results and Section 6 presents information regarding our public dataset. Section 7 discusses limitations of our research and future work to be conducted. Section 8 concludes our study.

2. RELATED WORK

There have been many studies which analyzed mobile apps on a large scale. Sarma *et al.* evaluated several large data sets, including one with 158,062 Android apps in order to gauge the risk of installing the app, with some of the results broken down by genre. However, this work did not analyze the apps using the range of static analysis tools which we used. Viennot *et al.* developed a tool called 'PlayDrone' which they used to examine the source code of over 1,100,000 free Android apps. While the authors studied a very large number of apps, they largely only used existing information which could be gathered from Google Play and only examined features such as library usage and duplicated code. The did not study areas such as security vulnerability levels and overprivileges, which were a part of our analysis.

Khalid et al.[12] examined Android apps to determine the relationship between user ratings and FindBugs warnings. They randomly collected 10,000 free apps from the Google Play market and covered a diverse set of categories and ratings. The study found that lower-rated apps had higher FindBugs warnings and proposed that developers could benefit from static analysis tools such as FindBugs to create apps with higher ratings.

Krutz et al.[13] created a public dataset of over 1,100 Android apps from the F-Droid[5] repository. This research analyzed a much smaller number of apps than our study and focused more on the lifecycle of the apps and how each iteration of the app evolved with every version control commit.

There are several other websites which gather metrics about Android apps. One of the most popular is AppAnnie[6] which collects Android apps and performs several types of analysis on each of them including downloads of the app over time and advertising analytics. However, no known services perform the same types of static analysis and comparisons on apps that we do.

3. ANDROID APPLICATIONS

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 apps using the Java programming language. Isolation between Android apps is enforced through the use of the Android sandbox [2], which typically prevents apps from intruding upon one another. Android applications are packaged in APK files, which are compressed application files which includes the application's binaries and package metadata.

Android applications are available from a variety of different locations including AppksAPK[7], F-Droid, and the Google Play store[8]. Google Play provides verification of uploaded applications using a service called Bouncer which scans apps for malware [5]. In spite of these efforts, malicious apps are sometimes found on the Google Play store [25]. Google Play 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.

3.1 Android Permissions

The *principle of least privilege* is the concept of granting of the least amount of privileges to an application that it needs to properly function [20]. Granting extra privileges creates unnecessary security vulnerabilities by allowing malware to abuse these unused permissions, even in benign apps. These extra privileges also increase the app's attack surface [6, 9]. Previous research has found that Android developers often mistakenly add unnecessary privileges in a counterproductive and futile attempt to make the app work properly, or due to confusion over the permission name they add it incorrectly believing its functionality is necessary for their app [10].

When installing the application, the user is asked to accept or reject these requested permissions. 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 [10]. 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 for which the app could fail because it was not given the proper permissions. Overprivileges are considered security risks, underprivileges are considered quality risks. The primary difference between requested permissions and overprivileges is that requested permissions are merely those that the app asks to use, and does not take into consideration if the app actually needs them or not.

4. COLLECTION & STATIC ANALYSIS

We analyzed 70,785 Android application files over a one year period using a variety of different tools. The results of this analysis have been stored in a publicly accessible database located on our project website[9]. Our methodology is as follows:

1. Collect APK files
2. Reverse-engineer binaries
3. Execute static analysis tools
4. Complete evaluation

4.1 Step 1: Collect APK files

Android APK files were pulled from Google Play with a custom-built collector, which uses *Scrapy*[10] as a foundation. We chose to pull from Google Play since it is the most popular source of Android applications [1] 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

[5]https://f-droid.org/
[6]https://www.appannie.com
[7]http://www.appsapk.com/
[8]https://play.google.com/store
[9]http://xxx.hidden.edu
[10]http://scrapy.org

than 10,000 downloads. Of the 70,785 apps downloaded, 31,234 had at least 10,000 downloads.

4.2 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 that has already demonstrated itself to be effective in similar research [14, 24]. 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. We then used two open source tools to complete the reverse engineering process:

- **dex2jar**[11]**:** Convert the .dex file into a .jar file. A java jar command is then used to convert this to .class files.
- **jd-cmd**[12]**:** Converts .class files to .java.

Additionally, we recorded the number of extracted class and java files. The de-compilation process is shown in Figure 1.

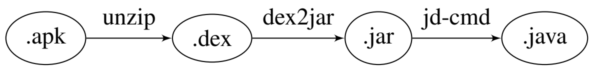


Figure 1: APK Extraction Process

While no reverse engineering process can ever be considered perfect, our technique has been demonstrated to be highly effective in previous research [4, 7].

4.3 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 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 [18]. We used the following tools for our analysis:

Stowaway: Reports the overprivileges and underprivileges of an application, which we recorded. Modifications were made to the existing version of Stowaway to accommodate our process and stay current with updated Android permissions.

AndroRisk: 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. AndroRisk determines the security risk level of an application by examining several criteria. The first is the presence of permissions which are deemed to be more dangerous. These include the ability to access the internet, manipulate SMS messages, or the rights to make a payment. The second is the presence of more dangerous sets of functionality in the app including a shared library, use of cryptographic functions, and the presence of the reflection API.

CheckStyle: Measures 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 for Android were used in our analysis. While adherence to coding standards may seem to be a trite thing to measure, compliance to coding standards in software development can enhance team communication, reduce program errors and improve code quality [15, 16].

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

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

Jlint: 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[13] since it was able to analyze the applications much faster, while still providing accurate results [19].

Simcad: A powerful software clone detection tool which we used to record the number of discovered code clones.

APKParser: 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 database, which is publicly available on the project website. The full analysis process is shown in Figure 2.

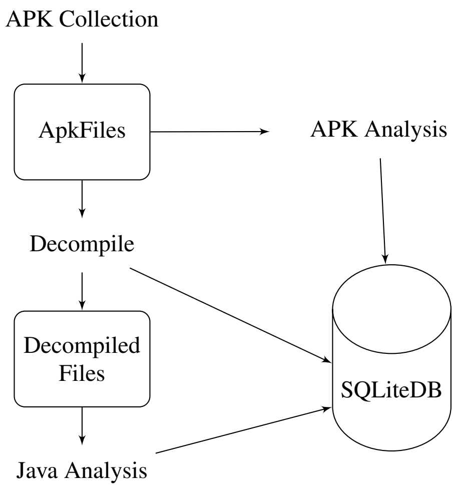


Figure 2: APK Analysis Process

5. EVALUATION

We will answer our research questions in the following sections.

5.1 RQ1: How do app genres compare in terms of quality metrics?

We sought to better understand how apps from different genres compare against each other based on several metrics including coding standards issues, use of permissions, rate of over & under privileges, and app size. We evaluated the top ten collected genres and display the results in Table 1.

Adhering to proper coding standards is important for software development since it has been shown to have a variety of benefits including aiding team communication, increasing code quality and elevating overall application quality [16]. We measured the adherence to coding standards for each genre by dividing the number of coding standards mistakes discovered by CheckStyle by the LOC in the app. We found that there was a reasonably large variation of the

[13]http://findbugs.sourceforge.net/.

Table 1: Comparison of Genres

Genre	LOC	CS Mistakes/LOC	OverPrivileges	UnderPrivileges	Requested Permissions
Arcade	152,563	.01048	2.3	3.8	6.6
Books & Reference	123,600	.00675	2.6	2.7	5.4
Casual	150,396	.0103	2.2	4	6.5
Communication	166,083	.06824	5.2	3.6	14.6
Education	131,615	.0234	2.2	3.3	5.8
Entertainment	138,229	.01367	2.6	2.8	6.3
Lifestyle	166,114	.02639	2.9	3	8.2
Music & Audio	146,211	.00985	2.5	2.7	6.7
Personalization	113,539	.06075	3.8	2.5	8.3
Puzzle	142,998	.01025	2	3.8	5.6
Tools	106,131	.07202	4.1	2.9	8.4

amount of coding standards mistakes for genres with 'Tools' having the most mistakes with .072 mistakes/LOC and 'Books & Reference' apps having the least with .00675 mistakes/LOC. While this may not seem like a large variation, the average LOC for all collected apps was 156,515, meaning that 'Tools' apps would have an average of 10,212 more coding standards defects per app. Across the genres, we did not find a significant difference in the number of discovered possible defects identified by Jlint.

We did find a large variation in the rate of overprivileges for each genre with 'Casual' apps having the fewest with 2.2 per app and 'Communication' having the largest with 5.2 for each app. 'Personalization' apps had the fewest number of underprivileges (2.5), while 'Casual' apps had the highest rate with 4 per app. 'Lifestyle' apps were narrowly the largest with an average 166,114 LOC per app, while 'Tools' apps were the smallest with 106,131 LOC.

5.2 RQ2: What are the most pervasive overprivileges?

A basic security principle 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 by allowing malware to abuse these unused permissions, even in benign apps. These extra privileges also increase the app's attack surface [6, 9]. Previous research has found that while Android developers 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 [10]. We analyzed the most overused permissions for all apps and present the ten most commonly occurring overprivileges in Table 2.

Table 2: Top Occurring Overprivileges

Permission	Rate %
GET_ACCOUNTS	10
READ_EXTERNAL_STORAGE	9
CALL_PHONE	7
ACCESS_WIFI_STATE	7
SYSTEM_ALERT_WINDOW	6
READ_PHONE_STATE	6
WRITE_EXTERNAL_STORAGE	4
WRITE_CONTACTS	4
GET_TASKS	4
CHANGE_NETWORK_STATE	4

These top occurring overprivileges can be dangerous for a variety of reasons. For example, `GET_ACCOUNTS`, permits access to the list of accounts in the accounts service, `READ_EXTERNAL_STORAGE`, allows the application to read from an external storage device and `CALL_PHONE`, allows an app to start a phone call without the user confirming it through the dialer interface [3].

5.3 RQ3: How are apps changing over time?

Our next research question was to understand how apps, and their characteristics are evolving over time. Software is constantly changing and new libraries, APIs, and development techniques are always being used. Our goal was to understand how apps, and their static analysis metrics change on a yearly basis. In order to do this, we grouped apps into the year they were created by using the *Date Published* field for the app, which was collected from Google Play. The results are shown in Table 3. We found that apps are growing in terms of size (LOC), and in the number of requested permissions. Unfortunately, the amount of over & under privileges are growing as well.

Table 3: Evolution of Apps By Year

Year	Collected App Count	LOC	Opriv	UPriv	Requested Permissions
2011	2,898	25,549	2.1	2.3	3.9
2012	6,020	38,183	2.3	2.4	4.8
2013	17,466	83,445	2.3	2.9	5.9
2014	32,351	206,903	3.3	3.3	9.2
2015	5,382	248,802	3.6	5.3	10

5.4 RQ4: Do quality metrics change for apps with more or less than 10K downloads?

We next sought to determine if there were differences between apps with less than 10K downloads, and apps with $\geq 10K$ downloads in terms of app size (LOC), AndroRisk score, adherence to coding standards per LOC, Jlint potential defects per LOC, under & over privilege rate, and number of requested permissions. In order to compare the two groups, we used the one tailed Mann Whitney U (MWU) test for the hypothesis testing since it is non-parametric and we can find out if the two groups differ in each of the evaluated areas. The MWU test compares two population means that originate from the same population set and is used to determine if two population means are equal. A p value which is smaller than the significance level implies that the null hypothesis can be rejected.

A p value which is equal to or greater signifies that the null hypothesis cannot be rejected. In our analysis, we used a significance level of .05 to determine if we have enough data to make a decision if the null hypothesis should be rejected. In our analysis, each of the p-values were greater than .05, and the results are shown in Table 4.

Table 4: MWU Results for Download Count

	Greater In	
Value	**<10K**	**$\geq$ 10K**
Lines of Code (LOC)		✓
AndroRisk Score		✓
Coding Standards/LOC	✓	
Jlint Defects/LOC		✓
Overprivileges	✓	
Underprivileges		✓
Permission Count		✓

Apps with less than 10K downloads have a higher rate of coding standards defects per LOC, along with more overprivleges. Apps with at least 10K downloads were larger (in terms of LOC), have a higher AndroRisk score, have more Jlint defects per LOC, and have more Underpriviledges and requested permissions. The fact that apps with $\geq 10K$ downloads requested more permissions is not surprising since these apps were found to be larger, and would thus likely also ask for more permissions.

6. PUBLIC DATASET

Our dataset is available from our publicly accessible GitHub repo[14], 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 collection and analysis software. The goal of this dataset is to allow future researchers to both learn from and expand upon our work. Some of the types of data are shown in Table 5. This dataset contains the information collected from Google Play, and the results of our static analysis tools including 115,313 total overprivileges, 207,179 total underprivileges and 531,426 permissions for all of the apps.

Table 5: Dataset Information

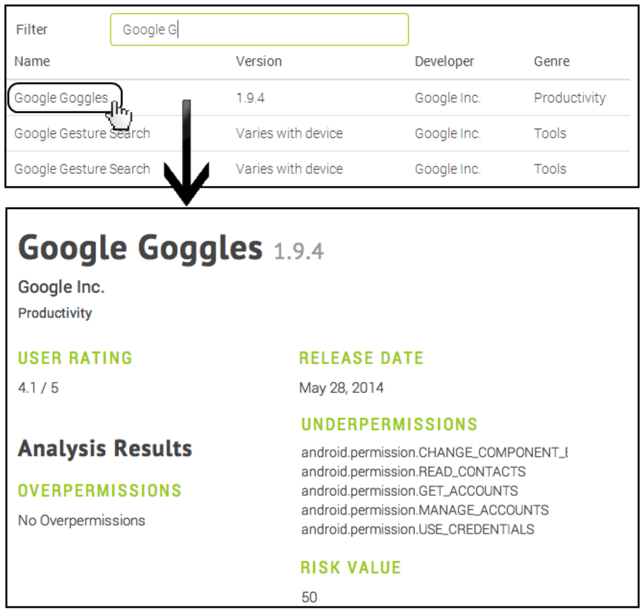
Version	Genre
GP Downloads	Publication Date
GP User Rating	App Size (LOC)
Overprivilege count	Underprivilege count
Jlint defects	Checkstyle defects
Requested Permissions	AndroRisk Score
App Signing Info	Code Clone Count

6.1 Website

Our project website (**http://hiddenToKeepAnonymous**) contains information about our project, links to our GitHub repository, and a robust reporting tool which will allow users to create their own datasets from over 70,000 analyzed applications. New apps will be added on a regular basis as they are pulled and analyzed from Google Play. As shown in Figure 3, users may view assorted information about specific versions of apps.

The website also includes several data driven result sets about apps at a more aggregate level. One example is shown in Figure 4,

[14]https://github.com/-Hidden-


Filter: Google G

Name	Version	Developer	Genre
Google Goggles	1.9.4	Google Inc.	Productivity
Google Gesture Search	Varies with device	Google Inc.	Tools
Google Gesture Search	Varies with device	Google Inc.	Tools

Google Goggles 1.9.4
Google Inc.
Productivity

USER RATING
4.1 / 5

RELEASE DATE
May 28, 2014

Analysis Results

OVERPERMISSIONS
No Overpermissions

UNDERPERMISSIONS
android.permission.CHANGE_COMPONENT_E
android.permission.READ_CONTACTS
android.permission.GET_ACCOUNTS
android.permission.MANAGE_ACCOUNTS
android.permission.USE_CREDENTIALS

RISK VALUE
50

Figure 3: xxx.hidden.edu Website Reporting Tool

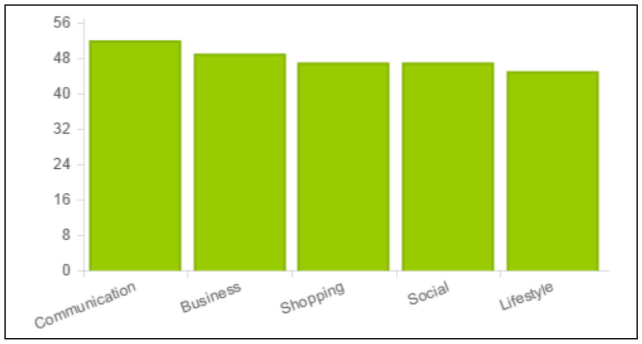


Figure 4: Overprivileged Apps By Genre from Project Website

which displays an example graph with the percentage of apps in the top 5 genres which contain at least 1 overprivilege.

16 prebuilt reports in .csv format are available in several areas including the rate at which two overprivileges appear together in all applications, the percentage of times each overprivilege occurs in all applications, and the percentage of times each overprivilege occurs in all applications.

7. LIMITATIONS & FUTURE WORK

While static analysis tools have demonstrated their value in numerous previous works [10, 17], it is unreasonable to expect that any tool will ever be flawless and that no static analysis tool is perfect and generally inherently contains limitations [8]. Although Stowaway is a powerful static analysis tool which has been used in previous research [11, 17, 21], it does suffer from drawbacks. Stowaway's own authors state that the tool only achieves 85% code coverage [10], so the over & under privileges reported by this tool are imperfect. Additionally, any reported vulnerabilities or defects by a static analysis tool should be deemed as *possible* vulnerabilities or defects, not necessarily actual ones.

Identifying possible vulnerabilities or security risks is extremely difficult, and like any static analysis tool AndroRisk is only capable of making educated observations about the risk level of an app and that more substantial risk assessments will require a far more substantial level of analysis, which will likely include a manual investigation of the app. Due to the large number of examined apps in our study, this thorough level of analysis was not practical. Even with almost certain imperfections, we believe that AndroRisk was a good choice due to its ability to quickly analyze apps and its use in existing research [13].

We compiled much of our data through reverse engineering APK files from the Google Play store. While similar reverse engineering techniques have been successfully used in previous works [14, 24], no reverse engineering process can ever be expected to be totally accurate. However, based on manually verifying a small subset of our results and previous research, we have a high confidence in our reverse engineering process.

While we have demonstrated profound results through the collection of over 70,000 Android apps, future work may be conducted in several key areas. We only analyzed free apps, and an interesting study would be to compare the free and paid apps using a similar process as ours. Future studies could also analyze how apps evolve over time through the examination of numerous released versions of the same app. Google's new API "M", received a massive permissions overhaul and work may be done to see how this new release affects how developers use permissions. Naturally more apps can always be examined, and with new apps being released on a daily basis the process is never ending. Additional research may also be done to examine apps collected from other sources, such as AppksAPK or F-Droid.

8. CONCLUSION

In this work, we described our process of gathering and analyzing 70,785 Android applications in a variety of areas including security vulnerability level, overprivileges, code clones, potential defects, and coding standards mistakes. We also provide a robust dataset and website which may be used by future researchers in their work.

Acknowledgements

Author and funding acknowledgments hidden for review anonymity.

References

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

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

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

[4] A. Apvrille. Android reverse engineering tools, 2012.

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

[6] 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.

[7] T. K. Chawla and A. Kajala. Transfiguring of an android app using reverse engineering. 2014.

[8] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, (6):76–79, 2004.

[9] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] 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.

[11] 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.

[12] H. Khalid, M. Nagappan, and A. Hassan. Examining the relationship between findbugs warnings and end user ratings: A case study on 10,000 android apps. *Software, IEEE*, PP(99):1–1, 2015.

[13] D. E. Krutz, M. Mirakhorli, M. S. A., A. Ruiz, J. Peterson, A. Filipski, and J. Smith. A dataset of open-source android applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. ACM, 2015.

[14] 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.

[15] X. Li. Using peer review to assess coding standards-a case study. In *Frontiers in education conference, 36th annual*, pages 9–14. IEEE, 2006.

[16] X. Li and C. Prasad. Effectively teaching coding standards in programming. In *Proceedings of the 6th Conference on Information Technology Education*, SIGITE '05, pages 239–244, New York, NY, USA, 2005. ACM.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] 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.

[25] 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.