Rotten Fruit: Comparing Android Malware to Benign Apps

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

ABSTRACT

As a constant companion to its user, the smartphone provides a unique and powerful vector for vulnerability. A recent study found that nearly 1 in 5 Android applications ('apps') were actually malware and are capable of creating serious vulnerabilities ranging from sending unwanted SMS messages to stealing sensitive information on our phones.

We evaluated 1,417 Android malware samples and 31,234 benign apps collected from Google Play to better understand characteristics of malicious apps. We found that malicious apps typically request more permissions than those from Google Play (12.7 vs 7.7), have more coding standards defects and Jlint errors per LOC and have more than double the overprivileges per app (6.2 vs. 3). We also found that malware is growing in terms of size, more than quadrupling since 2012. This study will help researchers develop improved malware detection algorithms, understand how malware is evolving, and better understand some of the potential impacts of malware. In the following work we describe our collection and analysis process as well as interesting findings. We also present a publicly available data set for researchers to use in their own work.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection;

Keywords

Malware, Android, Static Analysis

1. INTRODUCTION

Smartphones are simply personal computers which happen to be capable of making phone calls. For many, they have grown to affect nearly every facet of daily life and have reached the point where a large portion of people actually become emotionally attached to their phones [24].

Unfortunately, this relationship comes with a steep risk. Malware is striving to attack phones in nearly every imaginable way and is a highly lucrative business, generating revenue in a multitude of ways including sale of user information, stolen credentials, and premium-rate calls and SMS messages [12]. Malware is a constant concern as recent studies have shown that nearly 20% of Android apps are malware [25]. To make matters worse, detecting malware is not easy. Even with a myriad of sophisticated detection systems, malware is often released to Google Play, and many detection techniques are built on manually created detection patterns which do not always work well for discovering new malware instances [28].

In order to better detect and defend against malware, we need to understand more about it. Knowing how it is created, evolves, and its characteristics are all valuable pieces of information in the battle against malware. For the following work we evaluated 1,417 Android malware samples and 31,234 benign apps from Google Play using several static analysis tools with the goal of gaining a better understanding of these malicious applications. Static analysis tools evaluate a system and its components without actually executing the software [3], and have been used in a variety of areas including defect detection [15] and security related activities [22]. Static analysis differs from dynamic analysis which actually executes the software.

The primary goal of this work is to better understand Android malware, its patterns, and evolution through the use of static analysis. Our research questions are:

RQ1: *How do benign & malicious Android apps compare in terms of quality?*
We compared malicious and benign apps using several static analysis tools and found that malware had a much higher rate of coding standards mistakes and Jlint errors per LOC, while also having a much higher rate of overprivileges per app (6.2 vs. 3). These quality metrics demonstrate that developers of malware are typically paying much less attention to quality than those who create legitimate apps.

RQ2: *What are the most common permissions in Android malware?*
We recorded the most common permissions in malware and compared their rate of occurrence against the rates of these permissions in apps collected from Google Play. We found that malicious apps requested a much higher rate of SMS-related permissions than those from Google Play. These extra permissions are likely used to send premium SMS messages, transfer user information, or even propagate malicious data across devices.

RQ3: *Is Android Malware Growing?*
We found that malware is growing in terms of LOC (17,693 prior to 2013 vs 109,947 in 2015). This is an indication that malware is becoming more complex on an ongoing basis.

The rest of the paper is organized as follows: Section 2 discusses related works, and Section 3 provides an overview of Android apps. Section 4 describes our app collection and static analysis processes, and Section 5 presents and discusses our findings. The location, format, and possible uses of our public data set are presented in Section 6. Limitations of our study and future work is discussed in Section 7. The paper concludes with final remarks in Section 8.

2. RELATED WORK

There have been several recent papers which present methods of detecting malware by analyzing an app's requested permissions. Lui et al. [17] developed a technique of detecting malware using pairs of requested permissions to identify malicious apps. The basic premise is that extracted features such as requested permissions, utilized permissions, and permission pairs of malicious and benign apps can be used to produce training models which can then be used to identify malicious apps. A primary benefit of this technique is that previous examples of specific types of malware do not necessarily need to be used as a training set.

Su et al. [23] proposed a similar malware detection technique based on combinations of requested permissions. They found that malware requests specific pairs of permissions at a much higher frequency compared to benign apps. Xiaoyan et al. [26] also proposed a permission-based malware detection system which used a Support Vector Machine algorithm for malware detection.

Sarma et al. [21] investigated the possibility of informing users of an app's potential risks based on its requested permissions, and category. They found that an app's category and the permissions the app requested could serve as a warning of a potentially dangerous app for users. Similar to our work, they compared requested permissions for malware and benign apps collected from Google Play. While this work was significant, it only analyzed 121 malicious apps and only examined the permissions used by benign and malicious apps, and did not use the variety of static analysis tools that we utilized.

Typical approaches for detecting malware include, among others, taint analyzers and signature-based detectors. Taint analyzers are typically capable of finding applications that leak private information while signature-based malware detectors use regular expressions to match already known malware. Unfortunately, many benign apps also access sensitive information and the "leaked" information may actually be required for legitimate use [13]. Program obfuscation can alter program signatures enough so signature-based systems will not detect malware [19].

While profound, none of the previous works took malware release date into consideration or implemented other static analysis results as we have done. Additionally, these studies were all conducted at a much smaller scale, when compared with ours.

3. ANDROID APPLICATIONS

Android apps are are packaged inside compressed APK files, which include the app's binaries and repackaged meta data. Table 1 shows the breakdown of a typical APK file.

Table 1: 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

A large reason for Android's success is that it is available from a wide variety of sources. Some of these locations include AppsAPK[1], F-Droid[2], and the Google Play store[3]. While sites such as Google Play have some protection mechanisms that scan uploaded apps for malware [6], less reputable sites offer no protection or assurance that an app is not malicious. Unfortunately, even Google Play often misses malware and allows malicious software to be hosted on their site [28].

During the installation process, a user is asked to accept or reject an app's requested permissions. These include options such as requesting access to the internet and the ability of the app to read SMS messages. These permissions, contained within the *AndroidManifest.xml* file, are intended to be used for legitimate purposes, but have the potential for abuse by malware.

An *overprivilege* is the granting of extra, unneeded permissions to an app. 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 [10]. 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. 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.

4. COLLECTION & STATIC ANALYSIS

We analyzed 31,234 Android application files over more than a year using a several different static analysis 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

4.1 Step 1: Collect APK files

Android APK files were collected from Google Play with a custom-built collector, which used *Scrapy*[5] as a foundation. We chose to pull from Google Play since it is the most popular source of Android applications [2] and was able to provide various application metadata such as the developer, version, genre, user rating, and number of downloads. We collected a total of 70,785 apps from Google Play, but ignored apps with less than 10,000 downloads to limit the impact of seldom used apps. This left us with a total of 31,234 apps.

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 been demonstrated to be effective in similar research [16, 27]. After unzipping the files, we used two open source tools to complete the reverse engineering process. These were:

- **dex2jar[6]:** Converts the .dex file into a .jar file. A Java jar command is then used to convert this to .class files.

[1]http://www.appsapk.com/
[2]https://f-droid.org/
[3]https://play.google.com/store
[4]http://hiddenToKeepAnonymous
[5]http://scrapy.org
[6]https://code.google.com/p/dex2jar/.

- **jd-cmd**[7]**:** A command line decompiler that converts .class files to .java.

The de-compilation process is shown in Figure 1. While no reverse engineering process can ever be considered perfect, this technique has been demonstrated to be highly effective in previous research [5, 8].

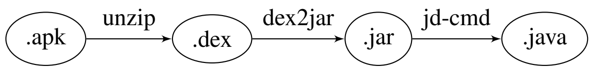

Figure 1: APK Extraction Process

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 non-security defects, and misuse of coding standards. The tools we use are:

Stowaway: This tool reports the overprivileges and underprivileges 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.

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 area is the presence of permissions which are deemed to be more dangerous, such as the ability to access the internet, manipulate SMS messages, or 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: A 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 settings for Android were used for our analysis.

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[8] since it was able to analyze the applications much faster, while still providing accurate results [20]. Jlint's default settings for Android were used for our analysis.

Custom APK Analysis Tool: We built a tool to extract the requested permissions, application version, minimum SDK and target SDK from the *AndroidManifest.xml* file. We've made the tool's source code available for public use on the project website.

We also recorded other metrics about each application including total lines of code and number of Java files in the reversed engineered app.

Stowaway and AndroRisk were able to analyze the raw APK files, while CheckStyle, and Jlint required the APK files to be decompiled. We also recorded the number of extracted Java files and number of lines of code (LOC). 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.

[7]ttps://github.com/kwart/jd-cmd.

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

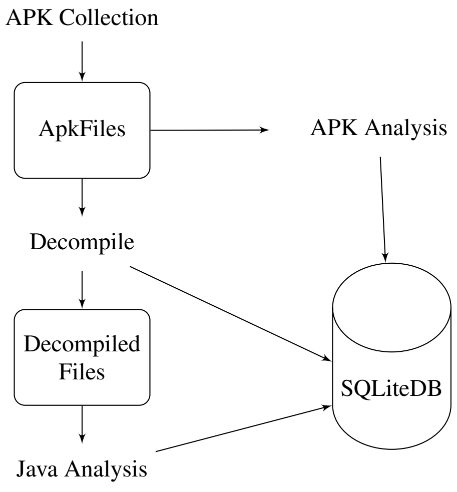

Figure 2: APK Analysis Process

5. EVALUATION

In the following sections, we describe our findings based on our analysis and answer our research questions.

5.1 RQ1: How do benign & malicious Android apps compare in terms of quality?

In order to better understand Android malware, we compared the static analysis results from apps collected from the Google Play store against 1,417 malware examples taken from the Contagio Mobile Mini Dump [1] and the Malware Genome Project [28]. The Contagio Mobile Mini Dump has been collecting malware affecting many platforms, including Android, for several years. In this study, 160 malware examples from the Contagio Mobile mini dump were used. The Malware Genome Project began in 2010 and has collected a substantial number of mobile malware. For our analysis, we used 1,257 examples from 49 malware families.

We chose to evaluate the malware samples against those collected from Google Play in a variety of areas including AndroRisk score, adherence to coding standards, discovered potential defects, over & underprivileges, and the requested app permissions. The results of this analysis are available on our website and are summarized in Table 2.

Table 2: Averages for Malware vs. Non-Malicious

Analysis Value	Malicious	Google Play
Jlint Errors/LOC	0.00298	0.00256
Coding Standard Mistakes/LOC	0.055	0.031
Overprivileges/App	6.2	3.0
Underprivileges/App	1.9	3.3
Permissions/App	12.7	7.7
AndroRisk	56.95	63.28

After collecting the above results we next tested the statistical significance of our findings using the one tailed Mann Whitney U (MWU) test for the hypothesis testing, since it is non-parametric and we can find out if the low-rated apps indeed have higher or lower values for each of the security metrics. 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. Conversely, 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. As shown Table 3, the results of this analysis further validate our findings displayed in Table 2.

Table 3: MWU Results for Malicious & Benign apps

	Greater In	
Area	**Malware**	**Google Play**
Jlint Errors/LOC	✓	
Coding Standard Mistakes/LOC	✓	
Overprivleges/App	✓	
Underpriviledges/App		✓
Permissions/App	✓	
AndroRisk		✓

Based on our comparison of malicious apps compared to those from Google Play, we made several primary discoveries. The first is that *Developers of Malware are not overly concerned with the quality of their apps:* Most Android malware is piggybacked on existing applications and malicious code is typically loosely coupled with the host application [11,28]. We found that malicious applications had a much higher number of coding standards mistakes per LOC compared to their benign counterparts while also having a higher number of possible bugs discovered by Jlint per LOC. Although we are not able to examine the development process of malware applications or interview its developers, we are able to draw several possible conclusions. Malware developers likely care less about the actual user experience as they are not overly concerned with app quality or coding standards. This lack of attention to quality is also demonstrated by the number of overprivileges in malware compared to apps from Google Play.

Underprivileges are a quality concern since an app will likely crash if it requests a permission which it was not granted. What is somewhat surprising is that malicious apps actually have fewer underprivileges per app compared to apps from Google Play. An explanation for this could merely be that the permissions in malicious apps are more often being used, unfortunately for disruptive activities.

Malicious apps request more permissions than benign apps: Malicious apps request an average of 12.7 permissions compared with 7.7 permissions for those collected Google Play. Malware is more likely to request these additional permissions to perform malicious activities [21].

We also found that *AndroRisk is not a good evaluator of determining if an app is malicious:* On average, malicious apps actually had a lower AndroRisk score as compared to Google Play apps. This is a representative of the difficulties that static analysis tools have in detecting malware [13, 18] and should not be construed as a statement that AndroRisk is any better or worse than any other static analysis tools.

5.2 RQ2: What are the most common permissions in Android malware?

We next compared the requested permissions for all malicious and benign apps using a custom-built permissions extraction tool. 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.

We compared the top ten rates of requested permissions for the malicious apps against those recorded from Google Play. The results of this comparison are shown in Table 4. The Android Malware Repository[9] used a different malware oracle and found similar permissions results to what we discovered, which provides confidence to our findings. Unfortunately, they do not appear to have analyzed any apps since 2012.

Table 4: Permissions Usage

Permission	**Malware %**	**G-Play %**
INTERNET	98	94
READ_PHONE_STATE	93	45
ACCESS_NETWORK_STATE	81	84
WRITE_EXTERNAL_STORAGE	68	62
ACCESS_WIFI_STATE	61	36
READ_SMS	60	4
RECEIVE_BOOT_COMPLETED	56	1
WRITE_SMS	49	2
SEND_SMS	45	4
RECEIVE_SMS	42	5

While most apps from each group request `INTERNET` and `ACCESS_NETWORK_STATE`, we found that malicious apps requested a disproportionately high number of other permissions. 93% of malicious apps requested `READ_PHONE_STATE`, which is more than twice the average for Google Play apps. This permission is particularly dangerous since it provides the app access to a wide variety of information and functionality including verifying the user's phone with IMEI information and gathering personal information such as your phone number.

Malicious apps also requested a much higher rate of SMS related permissions including `READ_SMS`, `WRITE_SMS`, `SEND_SMS` and `RECEIVE_SMS`. Malware often propagates to other devices through SMS messages. One such example of this is *Selfmite* [29], which is an Android worm that spreads through SMS messages. SMS systems may also be abused by sending premium rate SMS messages, which could likely go unnoticed until the user receives and examines their next bill [12].

`RECEIVE_BOOT_COMPLETED` is requested 56% of the time by malicious apps compared with only 1% for GoogePlay apps. This permission is often used by malware to notify it that the phone has been rebooted allowing it to begin installing malicious services [7].

Malicious apps request `ACCESS_WIFI_STATE` 61% of the time compared to only 36% for apps from Google Play. This permission allows apps to access several pieces of information related to the Wi-Fi interface including accessing a user's location. Unfortunately, malicious apps have used this permission to leak personal information about the user, including their location, which they often sell to advertisers [4].

The knowledge of what permissions have a higher prevalence in malware has several possible uses. Apps which request these permissions could be noted to have a higher probability that they are malicious. Since many benign apps have legitimate uses for these permissions, they cannot be the sole indicator, but can be an sign of a possibly malicious application. Understanding what permissions malware requests may be useful for understanding how current and future malware spreads and affects devices.

We next conducted a small case study on several malicious An-

[9]https://sites.google.com/site/androidmalrepo/permission-stats

droid apps collected from the Contagio Mobile Mini Dump. The first app we selected was a fake version of Netflix, which would collect the user's information when faking a login to the Netflix service. After reverse engineering this app, we found that some of its requested privileges included `ACCESS_NETWORK_STATE`, `READ_PHONE_STATE` and `ACCESS_WIFI_STATE`, which are all permissions which are much more likely to be requested by malware. We selected a sample known as 'LoveTrap' [14] which was originally available in a third-party app store. Some of the app's requested permissions included `READ_PHONE_STATE`, `RECEIVE_BOOT_COMPLETED`, `SEND_SMS` and `RECEIVE_SMS` which are all much more likely to appear in malware than in benign apps. The app used `RECEIVE_BOOT_COMPLETED` permission to execute itself once the infected Android device was rebooted, and used the SMS related permissions to subscribe to services which may lead to unwanted charges for the infected user

5.3 RQ3: Is Android Malware Growing?

We next compared how Android malware has grown over the years in comparison to benign apps collected from Google Play. We separated the apps collected from the Contagio Mobile Mini Dump into those created in or before 2012, 2013, 2014, and 2015. We did not use apps from the Malware Genome Project since we were unable to collect accurate date information for these apps. The results of this analysis are shown in Figure 3.

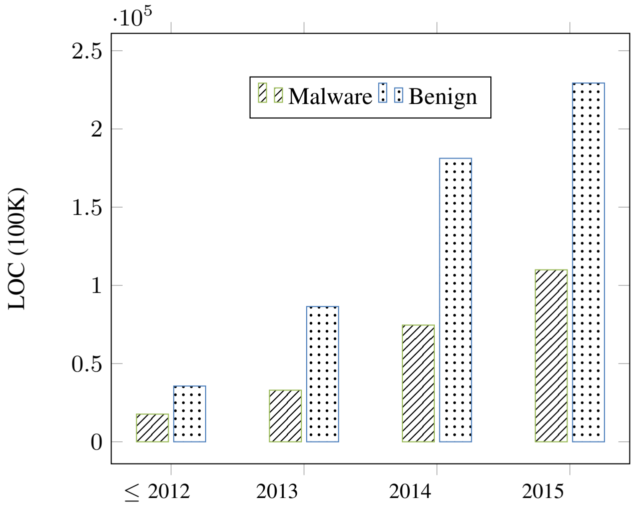

Figure 3: Evolution of Malware Size in LOC

This data indicates that both malware and benign apps are growing in terms of LOC on a yearly basis, although benign apps are typically larger.

6. PUBLIC DATASET

Our project website is located at **http://hiddenToKeepAnonymous** and we have made our data set available for future researchers to use in their own studies. The raw data used in our analysis is available in three SQLite databases, with one each for the malware data from Contagio and the Malware Genome projects, and a third for the collected information from Google Play.

The databases schemas are constructed in a similar format to one another and contain views that make data analysis easier. Further information regarding these views, along with ER diagrams to visually represent the dataset are available on the project website. The website also contains prebuilt reports with data available in a variety of formats including html, pdf, xls, and csv; users are also able to build their own reports. Unfortunately, the actual malicious APK files may only be obtained from the Contagio and Genome websites due to usage agreements.

7. LIMITATIONS & FUTURE WORK

While we have achieved interesting and profound results, our work is not without its limitations. We relied upon several static analysis tools for our results. While these tools have been substantially used in previous research, no static analysis tool is perfect and generally inherently contain limitations [9]. However, we believe that our results based on these static analysis tools are accurate, since other works have found them to produce accurate results [5, 8], and also due to the magnitude of the apps in our study.

We measured the adherence to coding standards using CheckStyle, a popular static analysis tool. We only evaluated apps against the default Android settings for this tool, which could create issues since groups rely upon a variety of different coding standards. This means that an app could perfectly adhere to a company's specific coding standards, but still have violations reported by the tool. While this will lead to inaccuracies using any single coding standard benchmark against a variety of apps, we feel that this issue does not significantly impact our results since we used the same benchmark for such a large number of apps at an aggregate level and thus our basic findings will be unchanged. When using Jlint, we may only assume that *potential* defects are found. Actual defects should only be identified through manual analysis, and no single static analysis tool should ever be solely relied upon to discover actual defects.

While we can reasonably assume that the vast majority of apps from Google Play are benign, it is likely that several malicious applications have successfully circumvented Google's protection mechanisms. However, even if this is the case, we believe that it would be a statistically insignificant portion of the apps.

Although we analyzed a substantial number of benign and malicious apps, we obviously only examined a very minor portion of all apps. We believe that our results accurately represent each group and that it is unreasonable to expect and study to examine all possible apps.

A large portion of malware is piggybacked on legitimate apps, so many of our results may be skewed by the fact that an overwhelming portion of a malicious app may have originally been legitimate. The significant differences discovered between malicious and benign apps help to alleviate this concern.

We have presented novel results, but there is still room for building on our work. A natural next step is to study more malicious and benign apps. Apps could also be broken down into groups, such as genres. Malware could also be examined at different levels of granularity. The primary purpose of our study was to understand malware at a higher, more aggregate level, but there is always a need to perform lower, more in depth studies to examine malware at a much more granular level. One example could be to study how malware uses various Android libraries and other components. Finally, our study could largely be applied to various other types of software including iOS apps.

We chose to use the Mann Whitney U analysis to add confidence in our comparison of malicious and benign apps. However other correlation metrics such as the Spearman, Kendall, or Pearson rank correlations may also be used in future research.

8. CONCLUSION

In order to better understand Android malware, we compared

1,417 malicious apps against 31,234 benign apps collected from Google Play using a variety of static analysis tools. Our primary findings were that malware developers typically pay far less attention to app quality as compared to benign apps. Several permissions, specifically SMS related privileges, are much more common in malware, and that malware is growing both in terms of app size and requested permissions. We have also made our data set public for future researchers.

Acknowledgements

Author and funding acknowledgments hidden for review anonymity.

9. REFERENCES

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

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

[3] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[4] J. Achara, M. Cunche, V. Roca, and A. Francillon. Wifileaks: Underestimated privacy implications of the access_wifi_state android permission. 2014.

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

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

[7] G. Canfora, F. Mercaldo, and C. A. Visaggio. A classifier of malicious android applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 607–614. IEEE, 2013.

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

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

[10] 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.

[11] 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.

[12] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[13] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[14] L. Gu. Love trap android malware found in third-party app stores. http://blog.trendmicro.com/trendlabs-security-intelligence/love-trap-android-malware-found-in-third-party-app-stores/.

[15] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

[16] 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.

[17] X. Liu and J. Liu. A two-layered permission-based android malware detection scheme. In *Proceedings of the 2014 2Nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, MOBILECLOUD '14, pages 142–148, Washington, DC, USA, 2014. IEEE Computer Society.

[18] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430, Dec 2007.

[19] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA, 2013. ACM.

[20] 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.

[21] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: A perspective combining risks and benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 13–22, New York, NY, USA, 2012. ACM.

[22] D. Song, J. Zhao, M. Burke, D. Sbîrlea, D. Wallach, and V. Sarkar. Finding tizen security bugs through whole-system static analysis. *arXiv preprint arXiv:1504.05967*, 2015.

[23] M.-Y. Su and W.-C. Chang. Permission-based malware detection mechanisms for smart phones. In *Information Networking (ICOIN), 2014 International Conference on*, pages 449–452, Feb 2014.

[24] G. Thorsteinsson and T. Page. User attachment to smartphones and design guidelines. *International Journal of Mobile Learning and Organisation*, 8(3-4):201–215, 2014.

[25] D. Tynan. Report: 1 in 5 Android Apps Is Malware. https://www.yahoo.com/tech/report-one-in-five-android-apps-is-malware-117202610899.html, Apr. 2015.

[26] Z. Xiaoyan, F. Juan, and W. Xiujuan. Android malware detection based on permissions. In *Information and Communications Technologies (ICT 2014), 2014 International Conference on*, pages 1–5. IET, 2014.

[27] 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.

[28] 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.

[29] Z. Zorz. Aggressive Selfmite SMS worm variant goes global. http://www.net-security.org/malware_news.php?id=2881, Sept. 2014.