A Large Analysis of Android Applications

Daniel E. Krutz and xxxx
Affiliation
xxx1 Lomb Memorial Drivexxx
xxxRocheser, NY, USAxxx
{xxxxdxkvse, xxxx}@rit.eduxxxx

ABSTRACT

Abstract

- What the problem is - What our research is - What were some of our initial findings -

[Update these]

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Keywords

xxxxxxxx

1. INTRODUCTION

There are over 675,000 Android applications with over 12,000 being added every month. **[find citations and update values]**. Mobile applications (apps) are typically updated more frequently than traditional desktop applications**[cite]**, with many apps being updated several times a month. These updates occur for a variety of reasons including bug fixes, required updates for new phone hardware, or feature additions. While the intention is for each release is to make the software better, unfortunately they often cause more harm than good. These range from minor coding issues and maintenance difficulties for the development team, to significant security vulnerabilities which may expose the user to serious malicious actions. Additionally, when the end user installs an updated versions of an app, they typically believe that they are getting an enhanced version of the software, whether it be from the security, stability or functionality perspective.

The goal of this research is to understand how Android apps evolve over time from the perspective of security vulnerabilities, defects, maintainability, adherence to coding standards, and application size.*[Dan says: We are collecting much data, should we list it all?]* Specifically, we will address the following research questions:

RQ1: *How did the security vulnerability of Android applications change over time*
Did apps generally become more secure over time? How did the permissions gap evolve over time, and what were some of the common over and under privileges which occurred?

RQ2: *How do apps compare against known malware?*
How do the ratings of applications which are not known to be malicious compare against applications which are know to be malicious in a variety of metics *[Dan says: This really needs to be reworded]*

RQ3: *Can defect count and other metrics predict the number of downloads and user ratings for the* current *release of an app?*

RQ4: *Can poor coding standards and clones predict metrics about* future *releases of an app?*
Can the use of poor coding standards predict that future releases of an application will contain more defects. Also, can they predict the user ratings & number of downloads for the future releases of the app?

RQ5: *What correlations exist between metrics such as defects, coding standards and code clones with the vulnerability of the software?*
Blah Blah blah

[Dan says: This will need to be tweaked, these are just some I am starting with]

[Modify the format of this] This work is important because *[Dan says: really work on this]*

- Provide insight on the lifecycle of mobile applications - Can help improve how apps are written in the future. What types of correlations occur
- Provide insight if newer applications are generally better
- Provide a data set for future developers and researchers

The contributions of this paper are:
The rest of the paper is organized as follows. Section **??**

2. ANDROID APPLICATIONS

The Android operating system has emerged as a market leader in mobile computing, comprising over **[xxx]** of the mobile market.

Each application is grouped based upon its purpose into separate *Genres*. Some of which include Action, Business, Entertainment, Productivity, and Tools.

2.1 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 [1], which prevents applications from intruding upon one another. Application developers are required to explicitly declare the required application permissions in the *AndroidManifest.xml* file. This will allow the application to request privileges to specific functionality such as the ability to write to the calendar, or access the GPS. Before 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 [26]. This differs from iOS **[- How does this differ from iOS]**

Intents are a communication mechanism which use *Activitys* to exchange information between the components of an Android application. Inter Process Communication (IPC) is the composition mechanism performed using intents which is used to invoke another application component. There are three types of attacks which exploit Intents for malicious reasons. These include *permission collusion*, *confused deputy* and *Intent Spoofing* [25].**[Add to this]**

An Android Application Package File(apk) contains all of the files necessary to install an Android application and is essentially just a compressed file of the application's source code. There are several key components of an apk file, which are shown in Table 1.

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

Android applications may be released to the user in a variety of methods, with the most popular being through the GooglePlay store[2], while many are downloaded through alternative sites such as AppksAPK[3] and F-Droid[4]. This differs from iOS applications which are all forced to go through an Apple controlled central App Store. While the exact steps taken by Apple to ensure the quality of their applications is unknown, it is believed that they do check all apps submitted to the app store for various security standards [13]. Android users enjoy more freedom to download from a wider range of sources however, they are not provided with any of the verification which the Apple App Store states to provided.

- What are some problems seen?

2.2 Android Permission Structure

[Dan says: cut this down a bit?] The Android security model is a permission-based system where applications need to be granted access to various areas of functionality before they may be used. If an application attempts to perform an operation which it does not have permission, a *SecurityException* is thrown. When an Android application is created, its developer must declare in advance what permissions the application will require [17]. These security settings are stored in the `AndroidManifest.xml` **[redundant]** file and include a ride range of permissions including the ability to transmit data, access personal information, and charge subscription fees [11]. A few of these are *INTERNET*, *READ_CONTACTS*, and *WRITE_SETTINGS*. When an application is invoked, this manifest file is examined to determine the appropriate permissions the application should possess.

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 [24]. Granting more privileges than the application actually 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, developers are often forced to grant more permissions to their application than they actually need. Due to the 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 which 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 [19].

Each Android permission is assigned under one of several protection levels which helps to define the risk level associate with each permission [1]. These levels are:

Normal Default value with minimal risk. Grants the application access to isolated, application level features. Minimal risk is incurred by other applications and the end user.

Dangerous An elevated permission risk which provides the application access to private user information or control over the device should could negatively impact the end user. Upon the instillation of an application, the user needs to explicitly grant these permissions.

Signature Highest level of permission. Granted by the system only if the application is signed by the same certificate as was declared in the permission. The user is not notified that this permission is granted if the certificates match, and the application will automatically possess these privileges.

signatureOrSystem Application must be in the Android system image or signed with the same certificate as the application which declared the permission. The majority of developers should avoid using this protection level, as it was intended for use by applications created by multiple vendors who needs to have them built into the system image.

The number of Android permissions has risen from 103 in API Level 3 (Cupcake), in 2009 to over xxx in API 19 (KITKAT)**[find number and citation]**. This list is expected to continue to grow, with the permissions in the *Dangerous* group the largest, and fastest growing. Permissions are added over time as device functionality grows and changes [28].

- What are some ways that permissions are mis-used?

3. DATA COLLECTION & ANALYSIS

We analyzed over xxxx**[update number]** Android application files over a period of xxxx **[update]** using a variety of different tools. The results were stored in a publicly accessible database located on our project website. **[Add link to website]**

3.1 Collection Process

The Android apk files were pulled from GooglePlay using a custom built collector using *Scrapy* [8] as a foundation. We chose to

[1]http://developer.android.com/training/articles/security-tips.html
[2]https://play.google.com/store
[3]http://www.appsapk.com/
[4]https://f-droid.org/

pull from GooglePlay since it is the most popular source of Android applications and was able to provide various application information such as the developer, version, genre, number of downloads, user rating and number of downloads. Our scraper was initially run to randomly collect [xxxx] apk files, which were then decompiled and further analyzed.

3.2 Decompliation

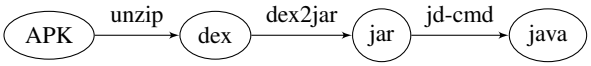
The downloaded .apk files had to be decompiled to .class and .java files to be further analyzed. The first step was to unzip the .apk file using a simple unix command. However, two open source tools were required to complete the decompilation process. These were:

dex2jar: Converts the .dex file into a .jar file. A java jar command is then used to convert this to .class files [4].

jd-cmd: A command line decompliler which converts .class files to .java [6].

The number of extracted class and java files are recorded. The de-compilation process is shown in Figure 1.

Figure 1: APK Extraction Process



3.3 Analysis Process

The next phase was to analyze the extracted source code for a variety of metrics in order to more appropriately understand how they evolve. Some of which included the vulnerability of the application from a security perspective, permissions gap, detected bugs, misuse of coding standards and software clones. These are functionally equivalent portions of an application which may differ syntactically and are often a sign of poorly written software and may be detrimental to an application in a variety of ways. Some of which include increased maintenance costs since changes will need to be done several times and inconsistent bug fixes [23].

This analysis was accomplished using a variety of existing tools including:

Stowaway[17]: Reports the overpermissions and underpermissions of an application. We recorded the overpermissions of the application, as well as all areas the application was underprivledged.

AndroRisk[15]:A component of the Androguard reverse engineering tool, this tool reports the risk indicator of an application for potential malware. We recorded the report risk level for the apk file.

CheckStyle[2]:A developent tool to measure how well developers adhere to coding standards. Some of which include naming conventions, annotation usage, size violations and empty block checks. We recorded the total number of violations of these standards. Default values were used in our analysis.

Jlint[7]: Examines java code to find bugs, inconsistences and synchronization problems by conducting a data flow analysis and building lock graphs. We recorded the total number discovered bugs. This tool was selected over FindBugs [5] since it was able to analyze the applications much faster. **[say something about findbugs??]**

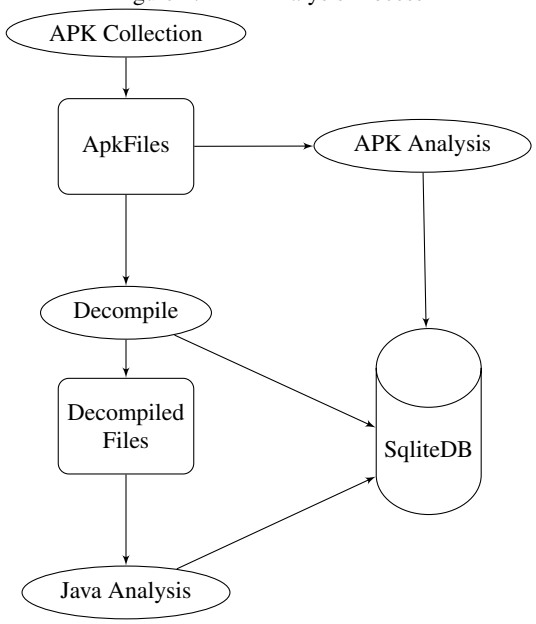
Simcad[27]: A powerful software clone detection tool. This tool was selected over other clone detection tools such as Nicad [14], CCCD [21] and MeCC [20] due to its ability to quickly detect a wide variety of clones. We recorded the number of clones discovered by Simcad for each target application.

APKParser [16]: A tool to read various information from an Android apk file including the version, intents, and permissions. We used the output from this tool to determine the application version, minimum sdk, target sdk, required intents, and requested permissions.

[Make sure to provide good justification for the use of selected tools. IE why was one tool chosen over another...]

Stowaway and Androrisk were able to analyze the raw apk files. CheckStyle, Jlint and Nicad required the apk files to be decompiled. All results were recorded in a sqliteDB, which is publicly available on the project website. The full analysis process is shown in Figure 2.

Figure 2: APK Analysis Process



[Other items recorded] - Lines of code - Java files - Class files - APK FileSize - APKParser - Version - MinSDK - Interprocess communications -

4. EVALUATION

Blah

- Discuss results - How long did we let the scanner run - Implications - Interesting findings - Answer research questions - What are our main findings? - Break the privileges down into threat levels (see pg 3)[18]

- Which over and under privs ocurred most often - How many apps were over and under privs

4.1 Comparison to Malware

[Dan says: This section will need to be cleaned up] **[Make sure all values are up to date]**

We next compared our findings against 139**[update?]** malware examples from the Contagio Mobile Mini Dump [3] and the Malware Genome Project [31] for the purpose of examining the differences between malicious and benign Android applications. Since many of the malware examples represented only slight alterations from their counterparts, one result from each malware family was taken from the Malware Genome project to help limit the negative impact that familes with many examples would have in skewing the results.

The compared areas included the adherence to coding standards, discovered defects, permission gap and the utilized intents & permissions. This was accomplished by running the malicious applications through the same process as the benign application, as was described in Sections 3.2 and 3.3. The results of this analysis are available in our public GitHub repository. Since we were unable to unable to analyze the lifecycle of the malware examples, we took the average values for the first 10,000*[Dan says: update]* collected benign Android applications and compared them to the average values for the malicious applications. The number number of defects found by jLint and coding standards mistakes were divided by the number of classes to help normalize these results by the size of the application. The results are shown in Table 2.

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. **[What this means]**

Not surprisingly, the malicious applications had an average of more than twice the number of over privileges as compared to their counterparts.

[Dan says: What are most of the over privileges] **[What this means]**

Table 2: Malware vs. Non-Malicous

Test	Malicious	Regular
Fuzzy Risk	46.25	53.72
JLint/Class	.395	.301
Defect Count/Class	11.07	1.87
Overpriviledges	3.3	1.34
Underpriviedges	1.07	1.96
Intent Count	2.05	2.58

[Make sure to compare this to the overall averages of all "regular" applications] [For the genome project, break each set of malware into seperate groups. This is because each group is a family of malware, and using all and aggregating the data could create problems]

We next compared fake, malicious copies of Netflix and Player as identified by the Malware Genome Project. A single fake copy of Netflix was included, with six fake copies of Player. We ran these fake copies of the applications against legitimate versions attained from the GooglePlay store. In both instances, the fake versions of the applications were significantly smaller than their real counterparts. We chose to only report on the Fuzzy Risk, over privledge count, and number of Java files since many of the other results of the analysis were not deemed useful due to the small size of the fake applications. The Fuzzy risk value was similar for both versions of the Netflix applications, but the real version of the Player application had a significantly higher value than its fake counterpart. The number of overprivliledges was much higher for the Fake Netflix application compared

- How did Netflix compare? Fake = 9 over priv = total 10 Real = 2 over priv = total = 14

Fake wanted Dump, inject events, read logs, read phone state. It did not request likely none malicious actions such as vibrate, expand status bar and system_alert_window

See pg 75 in "Contrasting Permission Patterns beteen Clean and Malcious Android applications"

- Look at where things are over privileged

Full results are shown in Table 3.

Table 3: Fakes vs. Regular

Application	Fuzzy Risk	Over Privileges	Java Files
Fake Netflix	51	9	11
Netflix	50	2	2429
Fake Player (Avg)	50	0	14
Player	92	0	813

- Fakes are smaller - No discernable difference in Fuzzy Risk

- What were the fakes?

- ******** Separated by each Genre

- Correlations for jLint & ratings - At least 10,000 downloads & 5 apps - People cared more about ratings for business, Communication, Personalization, productivity - No affect for games, Entertainment, Music and Audio, - Inversely proportional for lifestyle,

4.2 Overused Permissions

A basic principle of security is the granting of the least amount of privileges to an application which it needs to properly function. Granting extra privileges creates unnecessary security vulnerabilities *[Dan says: cite]*. We analyzed the most overused permissions in 41 application genres ranging from Communication and Productivity, to Sports, Puzzles and Entertainment. In 12 of the genres, the most over used permission was *android.permission.GET_ACCOUNTS*, which permits access to the list of accounts in the accounts service [9]. This was followed by *android.permission.CALL_PHONE* which allows an application to start a phone call without the user confirming it through the dialer interface. *GET_ACCOUNTS* is potentially dangerous since a malicious application could gain access to all device accounts, while *CALL_PHONE* could allow a dangerous application to dial any phone number without the consent of the user. In the observed applications, the combination of these over permissions occurred 291**[update]** times. When occurring in unison, these over permissions are dangerous since a malicious application could again access to a user's contact list, and place calls to each. Both of these permissions are in the API level 1 interface and are potentially dangerous over permissions which are frequently used in malware and have been found to be commonly misused in previous work[**?**]. These were also found to be the most commonly occurring over permissions, with the ten most occurring over permissions are shown in Table 4. Overall, we found that 46% of applications had at least one over permission, with 27% having more than one.

Which ones occur most often in unison - Which percentage of applications had over permissions

[update this list with up to date data]

Table 4: Most Occurring OverPermissions

Permission	% Occurrence
CALL_PHONE	10.06
GET_ACCOUNTS	7.93
ACCESS_WIFI_STATE	7.07
READ_PHONE_STATE	5.87
SYSTEM_ALERT_WINDOW	5.45
READ_EXTERNAL_STORAGE	5.44
WRITE_EXTERNAL_STORAGE	3.81
WRITE_CONTACTS	3.76
ACCESS_LOCATION_EXTRA_COMMANDS	3.71
CAMERA	3.18

4.3 Defects & User Ratings

- How to defects affect ratings

- User ratings go up when a file is downloaded more (not surprsing), was very slight. - Less coding standards issues for apps which were downloaded more

4.4 Application Lifecyle

We examined the effect the version of the application had on the observed quality and user rating for the application. All applications downloaded from GooglePlay were grouped together according to their application version. We separated all applications into nine distinct groups based on their version, where applications with a version number of 0 - .99 being assigned to group 0, applications with the version of 1 - 1.99 went to group 1 and so on. Based on our collected applications, there was a need of 8 groups, with 342 applications not being unable to be grouped, and were therefore ignored in this analysis. We then averaged the already determined scores for the groups to determine their average fuzzRisk, number of over-permissions, 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. FuzzRisk, code clones and over permissions were found by finding the average values of each for all applications in each group. User ratings were not considered since they were virtually identical, ranging from 3.9 - 4.2. 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 **[reword]**. The results are shown in shown in Figure **??**, with further data provided on our project webpage.

Some of our findings are:

Applications become less secure over time: Not only did the FuzzRisk grow over each version, but the number of over-permissions generally grew as well. This indicates that as applications evolve, they become less secure and that developers are not using proper security standards and are not checking for over permissions. While each of these values grew, there is not a direct correlation between them.

Fuzz Risk & Coding Standards are inversely related: The fuzz risk for each application version progressively became larger from version group 0 to version group 6, with a slight drop off for the version 7 group while the adherence to coding standards progressively became better, until there was a slight rise in group #7. There does not appear to be a strong correlation between adherence to coding standards and the number of application over-permissions. *[Dan says: talk about why this is important?]*

Over time, developers pay more attention to general quality than security: The number of discovered clone clones, number of class files, and discovered of an application generally decrease as the application becomes more mature. This indicates that developers are refactoring and eliminating issues during the application's life cycle. Unfortunately, it does not appear as though they paying enough attention to security vulnerabilities in the application.

-

[Remove values on left]

[Break down some of the stats for each genre]

- Show a couple, smaller charts

[Answer the research questions which I am asking]

4.5 Effects of Genre

Blah Blah Blah

5. PUBLICLY AVAILABLE DATASET

Our data set is available from our publicly accessible GitHub repo [5] . The SQLite database is automatically updated on a nightly basis from our scanning and analysis application. All previous versions of the database are also available to the researcher. The goal of this dataset is to allow future researchers to both learn and extend upon our work.

[Dan says: Add more examples about or dataset - IE how the data is formatted and so forth?]

6. THREATS TO VALIDITY

Blah

- Only analyzed data from GoogePlay - Could only analyze a subset of all applications - Only analyzed for an X period of time - Only looked at free apps - Relied on existing tools - Relied upon existing coding standards - Android runs code on the Dalvik Virtual Machine(DVM)[22], instead of the JVM. Could some of the differences have affected our results? - For the lifecyle of the application, - Could mean that applications became better over time or that only "Good" applications made it that far.

7. FUTURE WORK

Future Work

- Analyze apks from more sources
- Use more tools to analyze
- Analyze existing VCS to understand why the application was improved or made worse

8. RELATED WORK

[This section needs to be changed to not just address the permissions gap]

The topic of reducing the permission gap in Android applications has received a considerable amount of attention recent years. Much of the existing work on 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 b̈ucketẅhen 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 lead to few vulnerability possibilities [19].

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

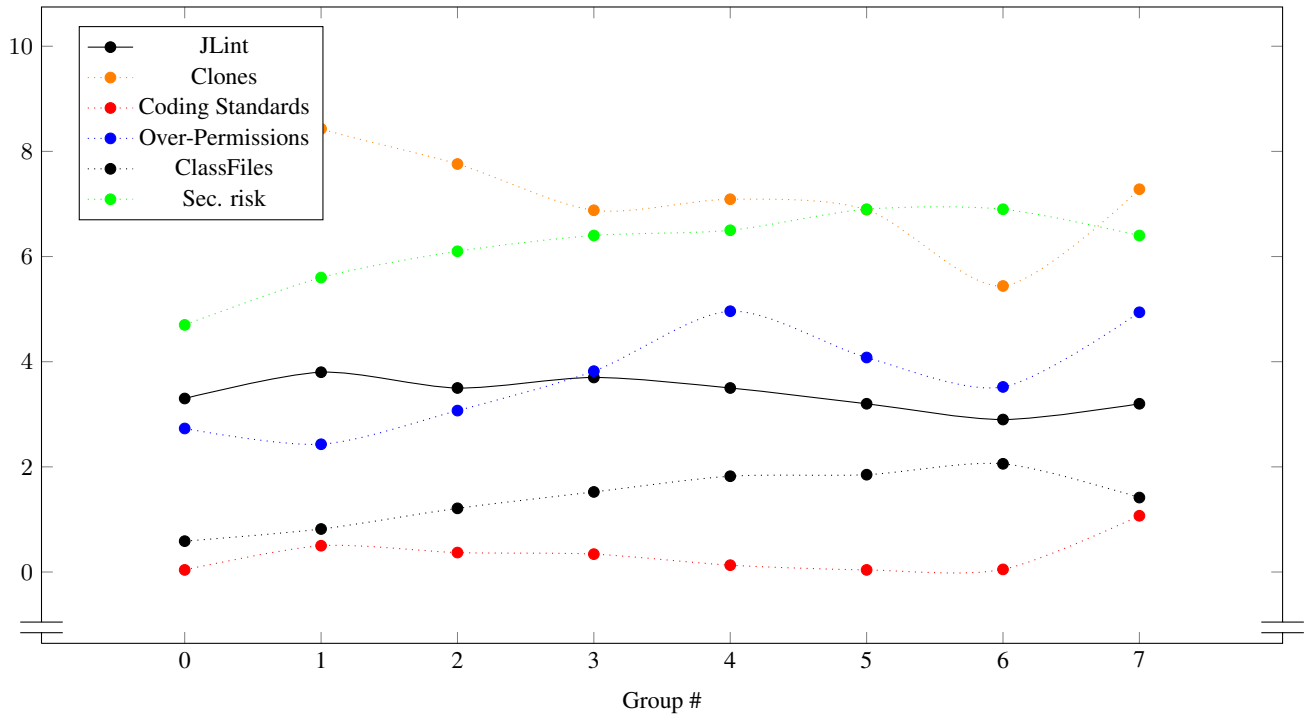


Figure 3: Effects on Version

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 overall security in the overall Android system is not improving over time and that the principle of least privilege is not being adequately addressed [28].

There have been several tools which have been developed to assist in the decision making, permission process for developers. Felt *et al.* created a tool known as *Stowaway* which uses a permissions-to-API calls maps in order to statically analyze request permissions in Android applications [17]. This tool notes the extra, unneeded permissions requested by the application, along with permissions that should have been requested, but were not. One criticism of this tool is that it may be difficult to determine if a permission is actually used through the use of static analysis.

Permlyzer is another tool which was built to determine where permissions are utilized in Android applications by using a mixture of static and runtime analysis [29]. This is a recently published tool, so it has not yet been discussed or used in a substantial amount of subsequent research. However, the authors were 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 [10]. 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, which is something the tool is not capable of discovering [30].

While this work represents the largest known, empirical analysis of developers allowing over privilege 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 they found using their tool Stowaway, including confusing permission names, the use of depreciated permissions and errors due to copy and pasting existing code [17]. In another work, Felt *et al.* very briefly described some inclinations they had for developers gave too many permissions to applications, but this was largely based on assumptions and not data [18].

Bartel *et al.* and Wei *et al.* also discussed some basic, high level discoveries about why developers make these mistakes [12] [28]. While these works were beneficial for numerous reasons, none to date have explored the question of why developers do not adhere to the principle of least security nearly as much as they should. **[Make sure to tweak this section with our actual findings]**

9. CONCLUSION

Conclusion

10. ACKNOWLEDGMENTS

This research would not have been possible without the hand-work by two dedicated Software Engineering Students. We would like to acknowledge the assistance received from Shannon Trudeau and Adam Blaine and thank them for their dedication.

References

[1] Android developers.

[2] Checkstyle.

[3] Contagio mobile mini dump.

[4] dex2jar.

[5] Findbugs.

[6] jd-cmd.

[7] Jlint.

[8] Scrapy.

[9] Manifest.permission, July 2014.

[10] 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.

[11] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[12] 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.

[13] E. Chin, A. P. Felt, V. Sekar, and D. Wagner. Measuring user confidence in smartphone security and privacy. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 1:1–1:16, New York, NY, USA, 2012. ACM.

[14] J. R. Cordy and C. K. Roy. The nicad clone detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 219–220, Washington, DC, USA, 2011. IEEE Computer Society.

[15] A. Desnos. Androguard.

[16] J. Erdfelt. Apk parser, June 2014.

[17] 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.

[18] 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.

[19] 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.

[20] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.

[21] D. Krutz and E. Shihab. Cccd: Concolic code clone detection. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 489–490, Oct 2013.

[22] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.