To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution

Vincent F. Taylor
vincent.taylor@cs.ox.ac.uk

Ivan Martinovic
ivan.martinovic@cs.ox.ac.uk

Department of Computer Science
University of Oxford
Oxford, United Kingdom

ABSTRACT

Although there are over 1,900,000 third-party Android apps in the Google Play Store, little is understood about how their security and privacy characteristics, such as dangerous permission usage and the vulnerabilities they contain, have evolved over time. Our research is two-fold: we take quarterly snapshots of the Google Play Store over a two-year period to understand how permission usage by apps has changed; and we analyse 30,000 apps to understand how their security and privacy characteristics have changed over the same two-year period. Extrapolating our findings, we estimate that over 35,000 apps in the Google Play Store ask for additional dangerous permissions every three months. Our statistically significant observations suggest that free apps and popular apps are more likely to ask for additional dangerous permissions when they are updated. Worryingly, we discover that Android apps are not getting safer as they are updated. In many cases, app updates serve to increase the number of distinct vulnerabilities contained within apps, especially for popular apps. We conclude with recommendations to stakeholders for improving the security of the Android ecosystem.

1. INTRODUCTION

Android is the most popular mobile operating system with 87.6% market share as of 2016 Q2, outpacing its nearest rival, iOS, at 11.7% [9]. This domination is fuelled by myriad app developers, devices and consumers existing in a symbiotic relationship known as the Android ecosystem. Nielson reports that the average consumer uses over 26 different apps per month, spending more than one hour per day interacting with their smartphone [24]. This explosion in smartphone usage has been driven, in part, by the ease with which end-users can obtain third-party apps to extend the functionality of their devices. The availability of Android integrated development environments (IDEs) makes writing apps a simple task for even novice programmers. Moreover, the existence of app generators and myriad libraries providing out-of-the-box functionality further lowers the barrier to entry for anyone wanting to create their own app.

There is strong competition in the app ecosystem, with time-to-market for both new apps and app updates playing a critical role in user engagement, retention and the overall profitability of an app. Apps are scanned for malicious behaviour before being published to the official Google Play Store [18], but it remains unclear whether apps are thoroughly tested for vulnerabilities by their developers before the publishing process is initiated. Additionally, it is not well understood how user privacy erodes as apps are updated to provide additional features, in many cases necessitating the use of new dangerous permissions. Our work fills these gaps in the literature by answering the following research question:

How have Android apps been evolving with respect to their permission usage and the vulnerabilities they contain?

To investigate this important issue, we took two main research steps. First, we took snapshots of the Google Play Store every three months over a two-year period to capture app metadata such as frequency of app updates and permission changes that come about from app updates. In doing this, we aimed to uncover micro- and macro-trends in the official Android app store, which would motivate and inform our research. Prior work (as detailed in Section 7) has looked at permission evolution on the Android platform itself [32], or the evolution of permission usage in Android ad libraries [13], without looking at changes in permission usage at the app level or indeed across the Google Play Store. Other work has looked at the Google Play Store to understand developer behaviour and pricing [14] but neither at the breadth of our work nor to understand micro- or macro-trends in app permission usage. We consider this a critical gap in the literature and the first of two underlying motivations for our work.

Second, we analyse a set of 15,000 apps for which we have versions that are two years apart, for a total of 30,000 apps. In doing this, we gain an understanding of how app vulnerabilities and other app characteristics have changed over the period. Several classes of Android app vulnerabilities (and tools to identify them) have been exemplified by the literature (as detailed in Section 7). Prior research has looked at vulnerabilities within apps, such as

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates

ISBN 978-1-4503-4944-4/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3052973.3052990

improperly handled SSL [16] and confused deputies [22], but whether developers currently follow best practices or write safe apps remains unclear. Additionally, it is not known whether apps are getting more or less secure as they are updated. Especially considering that many apps are seeking ever-increasing access to users' devices in the form of dangerous permissions, understanding the evolution of the security posture of apps becomes critical. This is a gap in the literature and the second underlying motivation for our work. Using existing and custom-written tools, we decompile and analyse our sample of apps to identify the vulnerabilities they contain and understand whether apps are becoming more or less secure over time.

The contributions made by our paper are two-fold:

1. Permission usage evolution in the Google Play Store:
   - We present the first evaluation of permission usage evolution across the Google Play Store over a two-year period.
   - We report on the number and types of permissions that are being added to (or removed from) apps, and how app attributes such as cost or popularity affect changes in permission usage.
2. Security evolution of Android apps:
   - We present the first large-scale report on how Android apps been evolving with respect to the vulnerabilities they contain.
   - We report on how popular apps compare to random apps in terms of vulnerability evolution. We further analyse vulnerabilities to understand whether they stem from developer-written code or library code.

Roadmap. Section 2 satisfies our first research step by presenting our analysis of permission usage evolution across the Google Play Store over the last two years; Section 3 begins our second research step by overviewing the vulnerabilities that we focus on and the tools used to audit our dataset of apps; Section 4 describes the dataset of apps and how it was assembled; Section 5 presents and analyses the results of our app audits; Section 6 makes recommendations and discusses the limitations of our work; Section 7 surveys related work; and finally Section 8 concludes the paper.

2. PERMISSION USAGE EVOLUTION

The first aspect of our analysis is concerned with understanding how dangerous permission usage by apps has evolved over the last two years. Specifically, the two-year period we study is `OCT-2014` to `SEP-2016`. We focus only on the so-called dangerous permissions as designated by Android, i.e., the 24 system permissions that guard access to sensitive user data [10]. For this reason we refer to dangerous permissions as simply permissions for the rest of the paper.

Our long-term analysis of permission evolution requires data on apps in the Google Play Store. Collecting longitudinal data from a large source, such as the Google Play Store, is an arduous process. This is perhaps the reason that very few studies to date have focused on app permission usage, either at the magnitude or duration that this one has. The Google Play Store Crawler (GPSC) project [23] is concerned with collecting app metadata, but, until recently, did not collect data on permission usage. Thus, we built our own crawler[1] that retrieved complete app metadata from the Google Play Store, by leveraging the list of apps from the GPSC project.

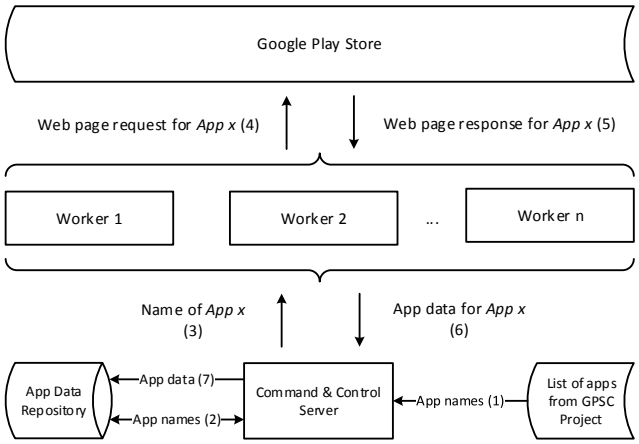

Figure 1: Highly-scalable cloud-based crawler architecture. The Google Play Store Crawler (GPSC) Project [23] and Google Play Store [19] itself were used as external sources of data. Numbers in brackets indicate the sequence of events.

Our first snapshot of the Google Play Store using our crawler was taken in March 2015 with subsequent snapshots taken at the same time of the month every three months after the initial snapshot. Additionally, we extracted permission usage information from a corpus of apps [11] obtained using the PlayDrone tool [31], to obtain another (earlier) snapshot of permission usage in the Google Play Store as at October 2014. The most recent snapshot used in our analysis is that of September 2016, for a total of eight snapshots (`OCT-2014`, `MAR-2015`, `JUN-2015`, `SEP-2015`, `DEC-2015`, `MAR-2016`, `JUN-2016`, `SEP-2016`) covering a two-year period[2]. All snapshots are three months apart except the first two which are five months apart because we appended the `OCT-2014` snapshot generated from the PlayDrone dataset [11].

2.1 Data Collection

In taking snapshots, our intention is to have the entire app store[3] crawled as quickly as possible. To this end, we developed a cloud-based crawler, with geographically distributed worker nodes fetching app data and returning it to a command and control server. Our architecture is shown in Fig. 1. Worker nodes make app store queries with random, valid *User-Agent* strings and are rate limited to *3 requests/second* each. Using a small-scale deployment with 3-4 workers, we can retrieve complete app metadata from the

[1]It is possible to obtain the data we require from the Google Play Store website without crawling any directories that are disallowed by the site's `robots.txt` file.

[2]For the remainder of the paper, any reference to a two-year period (or a studied period) is describing `OCT-2014` to `SEP-2016`.

[3]For brevity, we use the terms *app store* and *Google Play Store* interchangeably.

Table 1: Mean permission usage (and percentage change) across apps over the two-year period.

Downloads	OCT-2014	SEP-2016	Change
1-1K	3.13	3.16	+0.96%
1K-1M	2.37	2.45	+3.38%
1M-5B	3.40	3.58	+5.29%

app store in less than 48 hours. Our most recent snapshot[4] of the app store, at the time of writing, contains 31.2GiB of data on 1,918,833 apps available for download, and 813,421 apps that are no longer present in the app store.

With each new snapshot of the app store that we prepare to take, we carry over the entire list of apps from the previous snapshot, and append any new apps that have been added to the store. Our system is informed of new additions to the app store from the GPSC project. Apps that have been removed from the app store remain in our database, with an indicator that they are no longer available. Thus the number of apps in each of our snapshots monotonically increases over time.

2.2 Changes in Permission Usage

We analysed permission usage based on the number of downloads of an app. Apps were divided into three categories based on their total number of downloads: 1-1K (*low downloads*), 1K-1M (*medium downloads*), and 1M-5B (*high downloads*). Apps are frequently added to and removed from the app store, so we considered only those apps that were in all snapshots for the entire period ($n = 380,843$).

Table 1 shows how permission usage changed across the app store over the two-year period. Across all snapshots, every category of app had an increase in the mean number of permissions used. Overall, we found that apps with 1M-5B downloads had the highest increase in mean permission usage and percentage change, going from 3.40 to 3.58 permissions, an increase of 5.29%. Apps with 1K-1M downloads went from using 2.37 to 2.45 permissions on average. Apps with 1-1K downloads had the lowest increase in permission usage, going from 3.13 to 3.16 over the studied period. It is interesting to note that apps with 1-1K downloads used more permissions on average than apps with 1K-1M downloads. Two possible reasons are that apps with low downloads are using permissions unnecessarily, i.e., they are overprivileged, or they are trying to provide additional features (necessitating additional permissions) to gain market share. While the overall change in mean permission usage across our sample is small, we remind the reader that these numbers reflect the aggregate permission change across a large number of apps, including many apps that were not updated at all. Indeed, at the granularity of individual apps, the addition of several permissions between snapshots is not uncommon, as we show later in this section.

We analysed permission increase/decrease at the app level to understand how many permissions individual apps were adding or removing over each quarter. Table 2 shows this breakdown of permission change. In the table, each date shows how the number of permissions used per app changed between that snapshot and the previous snapshot. Note that we omit the OCT-2014 snapshot in this analysis because there was a five month gap between it and its subsequent snapshot.

[4]Our Google Play Store data is available upon request.

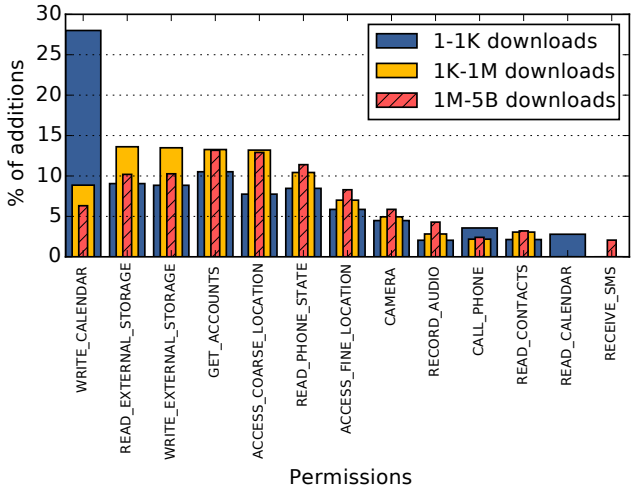

Figure 2: Breakdown of the most commonly added permissions. For clarity, we omit permissions that accounted for less than 2% of additions.

From the table, the majority of apps (on average 96.72%) did not have any change in the number of permissions used between snapshots. For apps that did have changes, the most likely change (1.15% of apps on average) was to add one new permission (+1). Averaging quarterly permission changes across our dataset, 1.87% of apps added one or more new permissions every three months, while only 1.42% of apps removed one or more permissions. In an app store of 1,900,000 apps, 1.87% would correspond to 35,530 apps adding one or more new permissions every three months.

2.3 Which Permissions Changed

We analysed the permissions that were added to apps when they were updated, to understand the potential erosion in privacy caused. Fig. 2 presents the most commonly added permissions, broken down by the popularity of the app adding the permission. From the figure, the Top 5 permissions (by aggregate total) that were added were WRITE_CALENDAR, WRITE_EXTERNAL_STORAGE, READ_EXTERNAL_STORAGE, GET_ACCOUNTS and ACCESS_COARSE_LOCATION. The WRITE_CALENDAR permission stands out for apps with 1-1K downloads. Manual analysis suggests that this permission is predominantly added by apps automatically generated using the Appcelerator [5] and PhoneGap [1] frameworks.

Fig. 3 shows which permissions were removed the most. The most commonly removed permission (by aggregate total) was WRITE_CALENDAR, covering approximately 40% of the incidents of permission removal from apps. Note that WRITE_CALENDAR was by far the most frequently added and removed permission for apps with 1-1K downloads. We postulate that inexperienced Android app developers are more likely to use app generators and app building frameworks. We further postulate that these inexperienced developers usually build apps with low numbers of downloads. Thus, the rapid addition and removal of permissions, such as WRITE_CALENDAR, may be explained by developer confusion or the ease with which these fledgling developers can turn on and off app features when using app generators. We

Table 2: How permission usage changed between quarters for the apps in our dataset.

	JUN-2015	SEP-2015	DEC-2015	MAR-2016	JUN-2016	SEP-2016
Total Increase	**2.15%**	**2.67%**	**1.86%**	**1.34%**	**2.03%**	**1.15%**
+3 or More	0.29%	0.43%	0.27%	0.20%	0.21%	0.23%
+2	0.44%	0.86%	0.51%	0.26%	0.34%	0.27%
+1	1.42%	1.38%	1.08%	0.88%	1.48%	0.65%
No Change	**96.81%**	**96.07%**	**96.69%**	**97.15%**	**96.50%**	**97.08%**
-1	0.63%	0.88%	1.08%	1.03%	0.92%	1.33%
-2	0.25%	0.20%	0.19%	0.30%	0.34%	0.23%
-3 or Less	0.16%	0.18%	0.18%	0.18%	0.21%	0.21%
Total Decrease	**1.04%**	**1.26%**	**1.45%**	**1.51%**	**1.47%**	**1.77%**

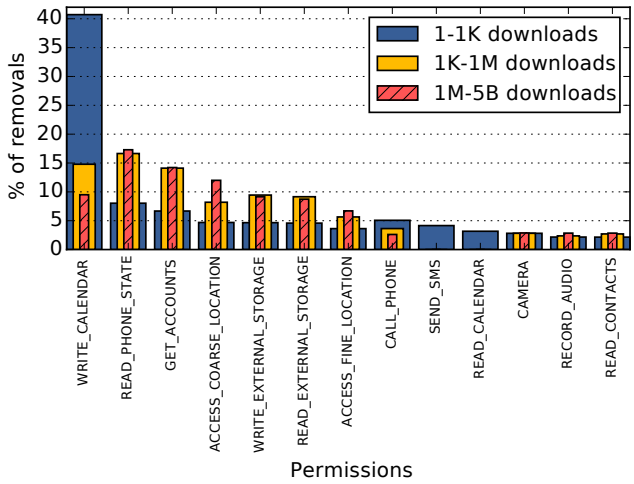

Figure 3: Breakdown of the most commonly removed permissions. For clarity, we omit permissions that accounted for less than 2% of removals.

leave further investigation into the privacy and security implications of using mobile app generator frameworks as an interesting area of future work.

2.4 Hypothesis Testing

On one hand, apps adding new permissions opens the door to added or improved functionality for users. On the other hand, new permissions may put user security and privacy at risk if they are abused, whether intentionally or unintentionally. To understand the phenomena of permission additions, we conducted hypothesis testing to understand what types of apps were adding permissions.

2.4.1 *Hypothesis 1: Free apps are more likely to add new permissions than paid apps.*

Free apps typically earn revenue for their developers through embedded ad libraries which display advertisements. Libraries, as we show in Section 4, leverage available permissions, ostensibly for better advertisement targeting through user profiling. This typically leads to undesirable privacy erosion since permissions granted to an app are leveraged by their bundled libraries. We want to determine whether there is a statistically significant difference between free apps and paid apps in terms of their likelihood of adding new permissions. We evaluated our hypothesis using a two-proportion z-test with a sample of 20,000 apps that added permissions. Specifically, it is our null hypothesis that the cost (free or paid) of an app is independent of its likelihood to add permissions. Our test returns $p < 0.01$, rejecting the null hypothesis at the 99% significance level. Thus free apps were statistically significantly more likely than paid apps to add new permissions.

Given that free apps enjoy much more[5] downloads than paid apps, this is cause for concern since a large proportion of smartphone users are affected. We add our voice to the call for privilege separation between apps and their libraries [29].

2.4.2 *Hypothesis 2: Popular apps are more likely to add new permissions than unpopular apps.*

Popular apps are installed by a large majority of smartphone users. Thus, popular apps adding new permissions leads to greater access to user data across a large cross-section of the smartphone ecosystem. As in Section 2.2, we consider apps having `1M-5B` downloads as being in the *high downloads*, i.e., popular apps category. We evaluate our hypothesis that popular apps are more likely than unpopular apps to add new permissions using a 2-proportion z-test with a sample size of 20,000. Our result showed $p < 0.01$, thus rejecting the null hypothesis that permission additions are equally distributed among popular and unpopular apps. That is, we confirmed that popular apps are indeed more likely to add permissions than unpopular apps.

It is somewhat concerning that popular apps are adding new permissions to a greater extent than less popular apps. Popular apps becoming more highly-privileged make them even more attractive targets to attackers. Additionally, if popular apps are the ones that are most likely to add new permissions, a large cross-section of users face the risk of being conditioned to automatically accept new permission requests.

2.5 Summary

Our analysis of permission changes across the Google Play Store over the last two years has finally confirmed several widely held beliefs:

1. Apps are becoming more permission-hungry over time.
2. Free apps and popular apps are more likely (than their respective counterparts) to add new permissions over time.

The addition of new permissions magnifies the risk of using apps if these apps contain vulnerabilities. This moti-

[5]At the time of writing, no paid app in the Google Play Store has more than five million downloads.

vates the second step of our research, where we study how the vulnerabilities contained within apps have changed over time and whether popular apps contain more vulnerabilities than unpopular apps. If popular apps contain more vulnerabilities, they are very attractive targets for attackers since they are more highly-privileged and have a larger userbase. If less popular apps contain more vulnerabilities, it suggests that less experienced developers are making mistakes that increase the attack surface of their app. Along similar lines, if the number of vulnerabilities within apps increases as time progresses, it highlights a worrying trend, in that app developers are not paying enough attention to the quality of their product as it relates to security.

3. VULNERABILITY ANALYSIS

In the previous section, we showed that apps are getting more permission-hungry over time. However, added access to user data amplifies the risks coming from vulnerabilities contained within apps. Thus, the second aspect of our analysis is concerned with understanding how user security and privacy is impacted by the evolution of app vulnerabilities as apps are updated. To understand this, we selected several classes of vulnerabilities and audited a dataset of apps containing current versions of apps and their corresponding versions from two years ago. We call these the `NEW` versions and `OLD` versions of apps, respectively. In comparing `NEW` and `OLD` versions of apps, we achieve a greater understanding of how vulnerabilities contained within apps have changed over time, i.e., whether vulnerabilities are still present, fixed, or new vulnerabilities have arisen from updates.

The Open Web Application Security Project (OWASP) systematised a Top 10 list of vulnerabilities affecting the mobile app ecosystem [26]. The vulnerabilities relevant to our study are those that come from improper or inadequate code implementation within the binary of the (client-side) app itself, i.e., vulnerabilities contained within the `.apk` file[6] of an app. Please see Appendix A for the complete OWASP Top 10. Vulnerabilities that do not arise from insecure coding practices on the client-side are outside of the scope of this study.

In the remainder of this section, we detail the vulnerabilities that we analyse and outline the tools (open-source and custom-written) that are used to scan our dataset of apps for these vulnerabilities. We elaborate on how we assemble our dataset of apps in Section 4.

3.1 Vulnerabilities Considered

There are a variety of vulnerabilities that affect Android apps. In doing our vulnerability analysis, we focus mainly on those vulnerabilities that stem from poor app coding practices in general, rather than those caused by idiosyncrasies of specific Android versions. Table 3 summarises the vulnerabilities we consider and the tools we use to check for each. Note that OWASP M2/M4 and M7/M8 are combined due to similarity, while M1, M5 and M9 are out of scope for this analysis.

3.1.1 Information disclosure (M2/M4)

Some apps contain weaknesses in program logic or access control that may allow the undesired leakage of sensitive information. We focus on apps creating world readable/writeable files (`INF-DISC-WRLRD`), ContentProviders that are exported but not properly secured (`INF-DISC-PRVDR`) and keystores that are not protected by passwords (`INF-DISC-KSNPW`).

3.1.2 Insecure network communication (M3)

Many apps rely on SSL/TLS to provide a secure communication channel when sending data to web services and/or fetching resources. Fahl et al. [16] observed that many apps fail to properly validate SSL certificates and thus are vulnerable to man-in-the-middle (MITM) attacks. We focus on apps that fail to use transport layer security at all, i.e., communicate over HTTP (`SSL-TLSX-PLAIN`), verify SSL certificates in an insecure way (`SSL-TLSX-INVLD`) and also improper certificate validation in WebViews (`SSL-TLSX-WVIEW`).

3.1.3 Broken cryptography (M6)

Some apps use available cryptographic protocols in a weak or insecure way. We check for apps using the weak ECB mode of encryption (`BRK-CRYP-ECBMD`) since ECB is known to not be semantically secure. Additionally, we check for apps using insecure random number generators (`BRK-CRYP-RANDG`). Insecure random number generators (or pseudo-random number generators) are not resilient against cryptographic attacks since they produce predictable values.

3.1.4 Miscellaneous (M7/M8)

Android allows apps to communicate with each other using components such as Intents, BroadcastReceivers and ContentProviders. Vulnerabilities may arise from apps improperly handling untrusted input that may allow an attacker to read/write data from/to a target app or trick the target app into performing tasks or behaving in ways that are unexpected. We focus on apps starting services with implicit Intents (`OTH-MISC-INTNT`) and apps that are debuggable (`OTH-MISC-DEBUG`).

3.1.5 Binary Protection (M10)

Apps that lack binary protection are susceptible to modification (typically repackaging and redistributing) by an adversary. No binary protection mechanism is perfect, thus best efforts only serve to slow an adversary down. Common binary protection methods include rooted device detection, validating app checksums and debug detection. We focus on whether apps contain root detection mechanisms (`BIN-ROOT-DTECT`).

3.2 Library Empowerment

Third-party libraries embedded within apps enjoy the privileges that have been granted to the host app. As apps evolve to request more permissions from users, bundled libraries now have access to sensitive data previously unavailable to them. We call this *library empowerment*. While not a vulnerability in strict terms, we consider library empowerment to be an interesting property to capture during our analysis. This is because many libraries are delivered as black-boxes, and since they inherit the privileges of the host app, it is important to understand the (potentially malicious) access that they have to sensitive user data.

[6]The `.apk` format is the file format used by Android to package and distribute apps.

Table 3: List of vulnerabilities that are considered.

OWASP ID	Type	Identifier	Description	Tool Used
M2/M4	Information disclosure	INF-DISC-WRLRD	App leverages world readable/writeable files	AndroBugs
		INF-DISC-PRVDR	ContentProvider exported but not secured	
		INF-DISC-KSNPW	Keystores not protected by a password	
M3	Insecure network communication	SSL-TLSX-PLAIN	Sending data over plain HTTP	AndroBugs
		SSL-TLSX-INVLD	Invalid certificate verification	
		SSL-TLSX-WVIEW	Improper WebView certificate validation	
M6	Cryptography	BRK-CRYP-ECBMD	Use of the ECB cryptographic mode	MobSF
		BRK-CRYP-RANDG	Use of insecure random number generators	
M7/M8	Miscellaneous	OTH-MISC-INTNT	Starting services with implicit Intents	AndroBugs
		OTH-MISC-DEBUG	App is debuggable	
M10	Lack of Binary Protections	BIN-ROOT-DTECT	App does not seem to have root detection	MobSF

We define two different forms of library empowerment as follows:

1. `OLD-LIBRARY-EMPOWERMENT` - This form of library empowerment happens when the older version of an app contains a library that has code that calls a permission-protected API. However, the library cannot access this permission-protected API because the host app itself has not declared the relevant permission in its manifest. However, the newer version of the app now declares additional permissions which enable the library to access the APIs it did not have access to before.

2. `NEW-LIBRARY-EMPOWERMENT` - This form of empowerment is similar to the previous one, except it allows for the library to have been updated between older and newer versions of the app. That is, the older version of the app contained the library, but the library itself did not contain calls to unpermitted APIs. However, the newer version of the app contains an updated version of the library that accesses new APIs, and the app itself now declares additional permissions which enable the library to access these additional permission-protected APIs.

3.3 Tools

To perform the vulnerability analysis of the apps in our dataset, we use a combination of open-source vulnerability detection tools and custom-written tools. We now describe the tools we use and give a brief description of their features.

3.3.1 AndroGuard

AndroGuard [3] is a powerful open-source app analysis framework that boasts a variety of features such as APK disassembly/decompilation and analysis. AndroGuard does static code analysis of APKs and can be easily extended to do custom app analysis.

3.3.2 AndroBugs

The AndroBugs Framework [2], hereafter called AndroBugs, is an open-source vulnerability scanning suite that leverages AndroGuard. AndroBugs is lightweight and highly-scalable and performs scanning for a variety of known Android vulnerabilities. AndroBugs categorises its results into severity levels. We focus on those vulnerabilities that have a corresponding mapping in the OWASP Top 10.

3.3.3 Mobile Security Framework

The Mobile Security Framework [7], MobSF, is a penetration testing framework that can perform static or dynamic analysis of APK files. Like AndroBugs, it is lightweight and highly-scalable. We use the additional vulnerabilities detected by MobSF to supplement AndroBugs.

3.3.4 QARK

QARK [8] is a vulnerability scanning tool that scans Android apps for several vulnerabilities. In addition to detecting vulnerabilities, QARK is able to generate "proof-of-concept" APKs or ADB commands that exploit the vulnerabilities it finds. We use QUARK to validate the results of AndroBugs and MobSF.

3.3.5 Mallodroid

Mallodroid [6] is a tool built on the AndroGuard framework and focuses specifically on detecting broken SSL validation in Android apps. We use Mallodroid to validate the relevant results of AndroBugs and MobSF.

3.3.6 PermFinder

PermFinder is a tool written by the authors that identifies method calls that are guarded by permissions. PermFinder first decompiles `.apk` files to `smali` code and extracts the APIs that are called. Using PScout [12] permission mappings (improvements over Stowaway [17]), PermFinder derives the permissions needed to access these APIs. PermFinder is used to measure library empowerment.

3.4 Limitations

We do not aim to analyse an exhaustive list of Android app vulnerabilities. Rather, we focus on several popular categories of app vulnerabilities that are well-known and have received treatment in the literature. We assume that these types of vulnerabilities will be the easiest to fix since they are widely known and thus more documented. Thus, we believe that our observations will represent a reasonable estimate of the upper bound of the likelihood that other (less studied) vulnerabilities are fixed.

For scalability, we use several lightweight static analysis tools to analyse apps. Static analysis tools suffer from their inherent reduced ability to handle dynamic programming artefacts such as dynamic code loading and execution of native code. Thus, the output from static analysis tools may be incomplete. However, barring false positives, the vulnerabilities detected by static analysis tools will represent a

Table 4: Datasets used in the analysis.

Dataset	# of APKs	Source	Date
TOP-OLD	5,000	PlayDrone	Oct-2014
TOP-NEW	5,000	Google Play	Sep-2016
RANDOM-OLD	10,000	PlayDrone	Oct-2014
RANDOM-NEW	10,000	Google Play	Sep-2016

lower bound of the set of all vulnerabilities present within an app.

To mitigate false positives, we validate the output of the tools used using a combination of manual analysis and cross-referencing their output with output from the other tools that detect the same vulnerability. We emphasise that our aim is not to extend the state-of-the-art relating to static analysis techniques. Rather, we use existing tools (except in the case of library empowerment where we write our own) to aid app analysis. We leave additional static and dynamic analysis as an interesting area of future work.

4. DATASET OVERVIEW

In this section, we describe how our datasets were assembled and report on additional non-vulnerability characteristics such as app update frequency, library usage and library empowerment.

4.1 Data description

Our main analysis is concerned with seeing how apps have evolved over the two-year period. Thus we needed to collect NEW versions of apps and their corresponding OLD versions from two years ago. To get OLD versions of apps, we leveraged the archive that was built from the PlayDrone project [31]. To obtain the NEW versions of apps, we used actual Android devices to download apps through the Google Play Store app, i.e., without violating the Google Play Store's Terms of Service. In addition to measuring how apps changed over time, we wanted to do a comparative analysis of the changes between TOP apps (the most popular apps in the Google Play Store) and RANDOM apps (apps randomly chosen from the Google Play Store).

We assembled a dataset of apps for which we have OLD and NEW versions of their .apk files. Specifically, we downloaded the current versions of 5,000 TOP apps, hereafter called TOP-NEW and their corresponding older versions, hereafter called TOP-OLD. We also downloaded the current versions of 10,000 random apps, hereafter called RANDOM-NEW and their corresponding older versions, hereafter called RANDOM-OLD. These four datasets contain a total of 30,000 apps and are summarised in Table 4.

Our dataset of TOP apps contains the 5,000 most popular apps (by number of downloads at the time of writing) in the Google Play Store for which there exists an older version of the .apk in the PlayDrone dataset. The Google Play Store has a long tail of apps with fewer than 100 downloads. To prevent bias towards these unrepresentative[7] apps, our dataset of RANDOM apps was assembled by selecting apps at random from a list of apps in the Google Play Store with more than 100 downloads. Overall, the dataset of TOP apps gives insight on the characteristics of the most common apps in the Android ecosystem, while the dataset of RANDOM apps is a useful representation of the ecosystem as a whole.

[7]Unrepresentative insofar as they would not be installed on many devices.

Table 5: File size statistics across datasets.

	TOP-OLD	TOP-NEW	RANDOM-OLD	RANDOM-NEW
mean	12.9MB	17.5MB	6.4MB	7.4MB
std	13.0MB	16.9MB	8.7MB	9.8MB
med	8.0MB	11.7MB	3.1MB	3.8MB
min	21.2KB	22.2KB	15.6KB	15.6KB
max	52.4MB	104.6MB	52.4MB	86.0MB

Table 5 shows statistics of file sizes within our four datasets. In general, TOP apps were larger than RANDOM apps. This confirms intuitive expectations, since TOP apps are more popular and thus may contain more functionality, better user-interface design, and the like. Also, newer versions of apps were bigger than older versions. This is also expected, since one would suspect that newer versions of apps would contain "improvements" and additional features, necessitating larger file sizes.

4.2 Update frequency

The update frequency of an app is a double-edged sword as it concerns app vulnerabilities. On one hand, a high update frequency may mean that vulnerabilities in an app get fixed more quickly. On the other hand, new updates may introduce new vulnerabilities into the app without necessarily fixing old ones. Conversely, an app with a low update frequency may contain vulnerabilities for longer, or they may go for longer without containing vulnerabilities.

We leverage an app's *updated* date, version number and file size from our snapshots of the Google Play Store (see Section 2) to understand how TOP apps compared to RANDOM apps in terms of their update frequency. Because our snapshots are three months apart, we may miss multiple updates that happen between snapshots. Thus, the number of updates we measure is a lower bound on the total number of updates that apps received. Using our snapshots of the Google Play Store, we observed that 45% of TOP apps had four or more updates over the two year period, while only 5% of RANDOM apps had four or more updates. This is in line with our expectations, since TOP apps have a larger userbase than RANDOM apps and, consequently, developers of TOP apps have a financial interest in ensuring that their apps are continuously being improved. From a security perspective, however, more frequent updates are not necessarily better unless vulnerabilities are actually patched by an update. Worryingly, as we discuss in Section 5, app updates tend to introduce, as opposed to rectify, app vulnerabilities.

4.3 Library usage

Understanding library usage as Android apps evolve is important, since libraries are able to access sensitive device resources that are granted to the host app. Thus libraries contribute privacy concerns in and of themselves. Moreover, libraries may contain vulnerabilities and thus contribute their own security concerns. Given that many apps may use the same library, vulnerable libraries, if popular, are a significant concern to the security of the Android ecosystem. We statically analysed the .apk files in our dataset to understand how the number of libraries used by apps changed as apps were updated. We decompiled .apk files

Table 6: Number of libraries used in apps across the datasets.

	TOP-OLD	TOP-NEW	RANDOM-OLD	RANDOM-NEW
mean	6.6	6.4	4.1	4.1
std	5.2	4.6	4.9	4.6
med	6	6	2	3
min	0	0	0	0
max	31	30	34	34

Table 7: Prevalence of library empowerment.

	TOP	RANDOM
OLD-LIBRARY-EMPOWERMENT	9.8%	2.8%
NEW-LIBRARY-EMPOWERMENT	14.3%	4.5%

using `apktool` [4] to convert the `.dex` files they contain to `smali` code. Libraries were identified using a whitelist of library signatures provided by the authors of FlexDroid [28]. In case of `.apk` obfuscation, library identification may fail and thus our statistics of library usage should be considered a lower bound on the actual total.

The library analysis results are shown in Table 6. From the table, there is no significant change in the number of libraries used between `OLD` and `NEW` versions of apps. A noteworthy result (which is perhaps expected) is that `TOP` apps use more libraries than `RANDOM` apps. `TOP` apps, on average, used approximately 6.5 libraries while `RANDOM` apps used approximately 4.1 libraries. Interestingly, a `RANDOM` app had the highest total number of detected libraries at 34 while the highest number of libraries for a `TOP` app was 31.

4.4 Library Empowerment

Given that apps use several libraries, understanding the prevalence of library empowerment is critical. As elaborated in Section 3.2, library empowerment occurs when apps begin to use new permissions that their bundled libraries are now able to use. Library empowerment is an important characteristic to capture, since library developers can gratuitously include permission-protected method calls in libraries, and later take advantage of them surreptitiously once the app has been granted the relevant permission. Table 7 shows the prevalence of library empowerment across our datasets. Some 9.8% of `TOP` apps suffered from `OLD-LIBRARY-EMPOWERMENT`. A somewhat higher 14.3% of `TOP` apps suffered from `NEW-LIBRARY-EMPOWERMENT`. `RANDOM` apps had lower rates of library empowerment, with 2.8% and 4.5% respectively. This finding may seem unsurprising since `TOP` apps contain more libraries to begin with. However, after correcting for the number of libraries per type of app, `TOP` apps were approximately twice as likely as `RANDOM` apps to suffer from library empowerment. Worryingly, this suggests that `TOP` apps typically use libraries and add permissions in such a way that fosters library empowerment.

Fig. 4 shows the new permissions that libraries were empowered to use. For `TOP` apps, in approximately 50% of cases, libraries were now able to read from a device's external storage. To a lesser extent, libraries in `TOP` apps were empowered to get the device's location, the list of accounts on the device, use the camera, read the contacts list and record audio. `RANDOM` apps were also most frequently empowered to read a device's external storage, but in less cases.

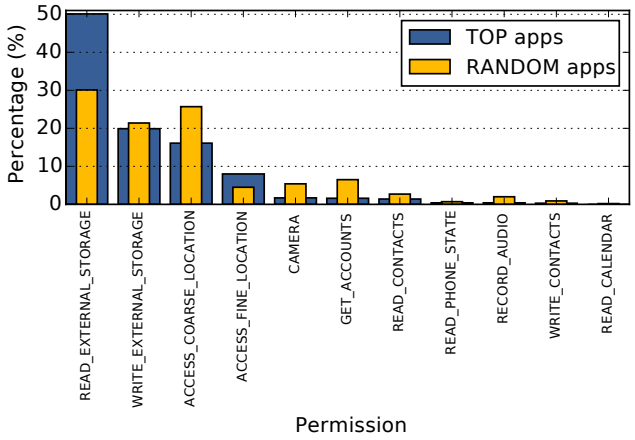

Figure 4: Additional permissions that libraries were empowered to use.

5. EVALUATION

Table 8 shows the fraction of apps within each dataset that contained one or more of the studied vulnerabilities[8]. The two most common vulnerabilities were `SSL-TLSX-PLAIN` (where apps sent data in plaintext using HTTP) and `BRK-CRYP-RANDG` (where apps used insecure random number generators). Almost all `TOP` apps sent some data in plaintext, while approximately 80% of `RANDOM` apps did so. Other common vulnerabilities are more serious and include failing to validate SSL certificates properly (`SSL-TLSX-VERIF` and `SSL-TLSX-WVIEW`), using ECB mode for encryption (`BRK-CRYP-ECBMD`) and starting services using implicit Intents (`OTH-MISC-INTNT`).

In all cases, except when an app was debuggable (`OTH-MISC-DEBUG`) or did not contain checks for rooted devices (`BIN-ROOT-DTECT`), a greater proportion of `TOP` apps were vulnerable than `RANDOM` apps. One explanation for this is the fact that `TOP` apps contain more functionality and thus have a larger codebase where vulnerabilities can be present. Regardless of the exact reason, this is alarming, since `TOP` apps are most popular with the general public and are thus the most attractive targets for adversaries in the first place.

Worryingly, for almost all the considered vulnerabilities, the fraction of apps containing each vulnerability increased between `OLD` and `NEW` versions. To make matters worse, for some vulnerabilities (`INF-DISC-WRLRD`, `INF-DISC-PRVDR` and `OTH-MISC-INTNT`), newer versions of apps were approximately twice as likely to be vulnerable as older apps. These substantial increases are highlighted in bold in Table 8. This suggests that app developers are not getting better at writing safer apps, and in many cases, leave users at even more risk as they update their apps.

The vulnerabilities `OTH-MISC-DEBUG` and `BIN-ROOT-DTECT`) were the only ones where `TOP` apps were less vulnerable than `RANDOM` apps. These two vulnerabilities are also among the few vulnerabilities that seem to be improving over time. We suspect that this is because these two vulnerabilities are more easily fixed. Turning off app debugging is a simple configuration change, while checking

[8]In line with vulnerability disclosure best practices, we are in the process of reporting the identified vulnerabilities to affected developers.

Table 8: Percentage of apps within each dataset containing one or more of each studied vulnerability. Numbers in brackets are the results when considering only those apps that were updated between OLD and NEW datasets.

Vulnerability	TOP-OLD (%)	TOP-NEW (%)	RANDOM-OLD (%)	RANDOM-NEW (%)
INF-DISC-WRLRD	**32.7 (34.1)**	**62.6 (69.1)**	16.4 (19.6)	24.8 (42.6)
INF-DISC-PRVDR	**5.02 (5.44)**	**11.2 (12.6)**	2.36 (2.87)	3.14 (5.01)
INF-DISC-KSNPW	3.06 (3.42)	2.90 (3.24)	2.27 (3.33)	2.25 (3.28)
SSL-TLSX-PLAIN	94.2 (95.8)	95.3 (97.0)	79.4 (87.1)	80.3 (89.5)
SSL-TLSX-VERIF	30.1 (31.4)	31.7 (33.2)	14.5 (20.0)	14.3 (19.3)
SSL-TLSX-WVIEW	18.4 (20.4)	20.7 (23.0)	9.87 (13.3)	9.35 (11.9)
BRK-CRYP-ECBMD	30.2 (27.5)	29.2 (26.3)	12.3 (6.61)	12.5 (6.61)
BRK-CRYP-RANDG	83.7 (73.7)	91.1 (80.7)	59.1 (26.3)	63.6 (30.6)
OTH-MISC-INTNT	**12.0 (13.1)**	**22.3 (25.0)**	3.07 (3.76)	5.02 (9.04)
OTH-MISC-DEBUG	0.46 (0.19)	0.30 (0.33)	2.21 (0.76)	1.93 (n/a)
BIN-ROOT-DTECT*	83.7 (84.4)	70.6 (71.5)	95.6 (97.3)	93.3 (97.3)

*An app may implement this in many ways, thus our results may contain false positives, so we consider this an upper bound.
Numbers in **bold** represent those percentages that approximately doubled between OLD and NEW versions of apps.

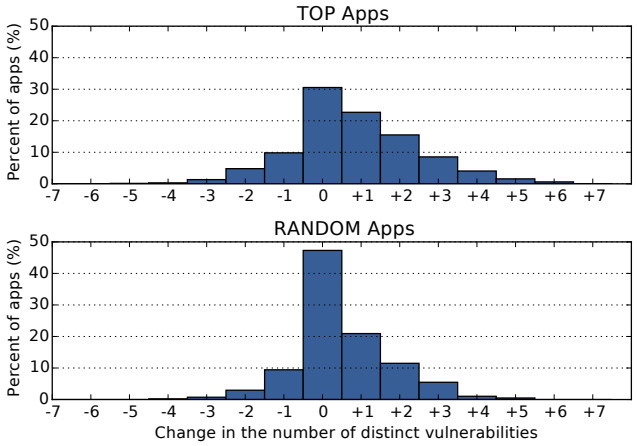

Figure 5: Histograms showing how the number of distinct vulnerabilities in apps changed between versions of apps. A larger percentage of TOP apps had an increase in the number of distinct vulnerabilities between OLD and NEW versions.

if a device is rooted can be as simple as checking for the presence of the `su` binary.

To better understand the impact of app updates, we examined how the number of distinct vulnerabilities contained within apps changed between older and newer versions of apps (for those apps that had an update). More specifically, for each app, we counted the number of distinct vulnerabilities in the new version of the app and subtracted the number of distinct vulnerabilities in the old version of the app. The result of this measurement is shown in Fig. 5. Worryingly, both TOP apps and RANDOM apps had an increase in the number of distinct vulnerabilities they contained as they got updated. Comparatively, RANDOM apps were more likely than TOP apps to have no change in the number of distinct vulnerabilities contained. We remind the reader here that Fig. 5 only summarises the *change in the number of distinct vulnerabilities* contained within each app. Thus, apps with no change, represented by zero, may still contain one or more vulnerabilities; this was usually the case. As an interesting observation from our dataset, 7.6% of TOP apps gained four or more new distinct vulnerabilities across versions. In contrast, only 3.3% of RANDOM apps gained four or more new distinct vulnerabilities across versions.

5.1 Origin of Vulnerabilities

The location of a vulnerability in an app will help to apportion liability to the appropriate party. We divided the location of vulnerabilities in apps into two parts: *library code* (LIB) and *developer-written code* (NON-LIB). We identified libraries as described in Section 4.3. If vulnerabilities are introduced by NON-LIB code, that points to errors in writing secure code on the part of the app developer. These vulnerabilities can be addressed with developer education and releasing updated apps. On the other hand, if vulnerabilities are introduced by LIB code, the responsibility lies with the library authors to fix their code. However, there is no straightforward way to automatically update libraries bundled within apps. Moreover, many app developers unknowingly use outdated libraries in their apps, and unwittingly introduce additional vulnerabilities to or fail to remove existing vulnerabilities from their apps.

Fig. 6 presents the results of our analysis of whether LIB or NON-LIB code was responsible for the introduction of vulnerabilities. In the figure, textured areas represent the contribution of LIB code and non-textured areas represent the contribution of NON-LIB code. Note that the vulnerabilities OTH-MISC-DEBUG and BIN-ROOT-DTECT are omitted since they are typically found in NON-LIB code.

Several vulnerabilities, such as INF-DISC-PRVDR, SSL-TLSX-VERIF and SSL-TLSX-WVIEW are more common in LIB code than in NON-LIB code. Other vulnerabilities, such as INF-DISC-WRLRD and OTH-MISC-INTNT, are almost exclusive to NON-LIB code. Other vulnerabilities, such as SSL-TLSX-PLAIN and BRK-CRYP-ECBMD, were more pronounced in TOP apps than in RANDOM apps. Along similar lines, the most "stable" vulnerability was INF-DISC-KSNPW that had approximately the same prevalence between OLD and NEW apps as well as TOP and RANDOM apps.

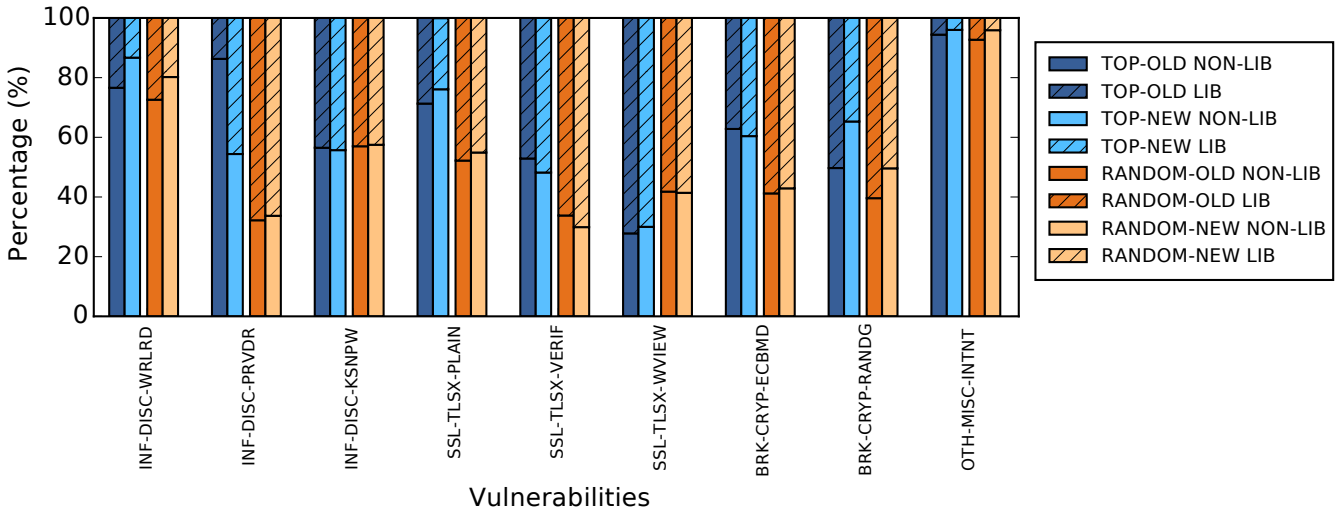

Figure 6: How vulnerabilities changed over the two-year period for the four datasets. Areas with texture indicate the fraction of vulnerabilities that come from library code (LIB) and vice-versa.

5.1.1 Older App Versions vs. Newer App Versions

Apps in the `TOP-OLD` dataset had more instances of the vulnerability `INF-DISC-PRVDR` within `NON-LIB` code than apps in the `TOP-NEW` dataset. Manual analysis reveals that the increase in this vulnerability coming from `LIB` code stems from updates to a handful of third-party libraries. This highlights the fact that libraries have the potential to greatly impact the attack surface of many apps, and that care must be taken by library developers when implementing libraries and by app developers in ensuring that libraries are updated with more secure versions.

5.1.2 Top Apps vs. Random Apps

For the vulnerability `BRK-CRYP-ECBMD`, `NON-LIB` code was a greater contributor in `TOP` apps than in `RANDOM` apps, when compared to other vulnerabilities. Further analysis indicates that `TOP` apps have more instances of this vulnerability per app than for `RANDOM` apps. This is presumably because `TOP` apps typically contain more functionality (and thus code), which naturally enlarges the attack surface, especially if developers are unaware of how to write code that mitigates the vulnerability in the first place.

5.1.3 LIB Code vs. NON-LIB Code

For the vulnerability `OTH-MISC-INTNT`, `NON-LIB` code was the main culprit, accounting for 94.8% of the occurrences on average. This vulnerability has to do with the insecure use of Intents to start services. On Android, an Intent is an abstract entity that passes messages describing some operation to be performed. Typically, libraries (i.e. `LIB` code), provide helper functions to apps and do not themselves interact with low-level entities such as services. This is one explanation as to why the majority of this vulnerability exists in `NON-LIB` code.

6. DISCUSSION

We direct our recommendations to those stakeholders that we deem to be able to make the greatest impact in fixing the problems identified.

Both app developers and library developers should familiarise themselves with the OWASP Top 10 [26] and Android security in general and strictly adhere to their best practices. Android IDE developers can reduce the likelihood of app vulnerabilities by triggering warnings during app development. Many app vulnerabilities stem from lack of developer understanding and the subsequent misconfigurations that are made at development-time. IDEs can easily warn (or prevent compilation) if insecure practices are detected, such as using pseudo-random number generators or choosing insecure options when declaring app components in an app's manifest. Android IDE developers can separate themselves from their competition by offering the feature of comprehensive security checking for apps at development time.

App stores have great power over the app ecosystem, since they can incentivise the building of safer apps by making app security a search ranking signal. Apps are already vetted for malicious activity before being admitted to the Google Play Store, thus it would be straightforward to also analyse apps using any of the static analysis tools we use, and give feedback to developers at the time of app submission. Better yet, well-funded app stores, such as the Google Play Store, could develop and contribute more robust static/dynamic vulnerability analysis tools to the community. This would be useful to security researchers and app developers alike, since existing static analysis tools sometimes yield false positive results. App stores employing vulnerability scanning stand to improve their reputation and market share, as consumers would be more confident about the security of the apps they download.

6.1 Limitations

6.1.1 Permission Usage Evolution

Our first aim was to understand how permission usage by apps in the Android ecosystem evolves over time. Due to storage limitations, snapshots of the Google Play Store were taken at three-month intervals. While this is useful for getting an overall picture of the Store, it is not granular enough to capture short-term phenomena when they happen.

We focused only on the so-called *dangerous permissions*, since *normal permissions*, if abused, only cause minor annoyance to a user, as opposed to putting their personal data at risk. We note that the addition of new permissions is not inherently bad, as many apps have legitimate reasons to request additional access to user data. However, many apps and libraries are also known to abuse their granted permissions for the purposes of profiling users or stealing their data. Thus apps becoming more permission-hungry remains an important phenomenon to understand.

6.1.2 App Vulnerability Evolution

Our second aim was to understand how the vulnerabilities contained within apps changed as apps were updated. We leveraged static analysis tools to perform vulnerability checking. As alluded to in Section 3.4, static analysis tools fail to understand aspects of code that are determined at run-time. Thus static analysis tools may yield output that does not paint the full picture. This is a limitation of all approaches that rely on static analysis. To mitigate the effect of false positives, the output from one tool was validated with that of other tools that scanned for the same vulnerability. In cases where there was a single tool checking for a particular vulnerability, manual analysis was used to validate the results. For future work, we plan to bolster our vulnerability analysis by leveraging dynamic analysis.

Old versions of `.apk` files were obtained from a repository built using app store crawling techniques described by the authors of PlayDrone [31]. The original tool appeared as a T-Mobile Galaxy Nexus device to the Google Play Store and thus apps were limited to those that would be accessible on such a device. In downloading new versions of apps, we use real devices (to comply with the Terms of Service) and thus are unable to guarantee that the `.apk` files obtained would perfectly match[9] what the PlayDrone T-Mobile Galaxy Nexus would have gotten at the time of fetching. From observation however, few apps in our dataset maintain multiple versions of apps for different devices, so we consider this threat to validity minimal.

7. RELATED WORK

7.1 Permission Usage

Viennot et al. performed the first large scale analysis of the Google Play Store using a tool they call PlayDrone to index and analyse over 1.1 million apps [31]. Our work is similar in that we take snapshots of the entire Google Play store as well, but differs in that our analysis is longitudinal and is concerned with gaining a greater understanding of how app permission usage and the vulnerabilities apps contain evolve over time and the potential impact of these phenomena on smartphone privacy and security.

Book et al. do a longitudinal analysis of Android ad library permissions [13]. The authors investigate a sample of 114,000 apps to build a chronological map of permission usage in Android ad libraries. This work is a step in our direction, but since ad libraries may only leverage a subset of the permissions used by an app, it fails to capture the full picture of the risk to devices that comes from app permission usage evolution as a whole. We complement this work by measuring the increased access obtained by libraries, a phenomenon we call library empowerment.

Wei et al. go in a tangential direction and characterise permission evolution on the Android platform itself [32]. They look at changes in the Android permission model since its first commercial release for smartphones in 2008. They find that permission growth is aimed towards offering access to new hardware features and not towards offering more fine-grained control to the user. The authors focus mainly on permission evolution within the Android platform itself, while our work focuses on looking at permission evolution across third-party apps in the official Android app market. Along similar lines, Vidas et al. [30] propose a tool to help mitigate permission creep by assisting developers to enforce the principle of least privilege [27]. Building on this, our work takes a look at whether there is a systematic permission-creep across apps in the Google Play Store.

Carbunar and Potharaju [14] analyse the Google Play Store to understand developer publishing and pricing behaviour. They found that developers are more likely to increase the price of apps when they are updated. This important work captures developer behaviour at a high level, i.e., publishing approaches, pricing, and app popularity. Complementary to this, we examine developer behaviour at a technical level, by analysing the security/privacy impact of app updates on the app ecosystem once they do happen.

7.2 Vulnerability Checking

Many authors have identified vulnerabilities and/or proposed tools to scan for vulnerabilities in Android apps. For brevity, we list a few of the most related ones. Fahl et al. [16] develop a tool called MalloDroid and use it to identify apps that are vulnerable to MITM attacks. Other authors present approaches that check for unprotected components [20–22]. Octeau and McDaniel [25] provide an approach to test the security characteristics of the interfaces exported by apps. Egele et al. [15] develop a program analysis approach to check whether apps use cryptographic APIs securely. Our work complements the efforts of these authors by assessing whether app developers are now writing safe code, how vulnerabilities in apps have evolved over time, and identifying the offending parties (whether library developers or app developers) when vulnerabilities are found.

8. CONCLUSION

In this paper, we performed a two-year study of Android app evolution in terms of permission usage and app vulnerabilities. Our analysis showed that apps are getting more permission hungry over time, with free apps and popular apps having greater increases. We measured the increase in access that libraries obtain by virtue of increased permission usage, a phenomenon we call library empowerment. We used static analysis techniques to identify how vulnerabilities contained within apps changed over the studied period. Worryingly, apps are seen to become more vulnerable over time, with popular apps more likely than random apps to become more vulnerable. We uncovered that some vulnerabilities predominantly come from different types of code, i.e., developer code or library code. By drawing these trends to the attention of the research community, we hope to generate additional interest, so that appropriate strategies can be developed to keep sensitive user data safe, as smartphones continue their explosive growth to ubiquity.

[9]Developers can target different versions of the same app to different devices.

Acknowledgement

Vincent F. Taylor is supported by a Rhodes Scholarship and the UK EPSRC.

9. REFERENCES

[1] Adobe PhoneGap. http://phonegap.com/.
[2] AndroBugs Framework. https://github.com/AndroBugs/AndroBugs_Framework.
[3] Androguard. https://github.com/androguard/androguard.
[4] Apktool - A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/.
[5] Appcelerator Open Source. http://www.appcelerator.org/.
[6] Mallodroid: Find broken SSL certificate validation in Android Apps. https://github.com/sfahl/mallodroid.
[7] Mobile Security Framework. https://github.com/ajinabraham/Mobile-Security-Framework-MobSF.
[8] QARK: Tool to look for several security related Android application vulnerabilities. https://github.com/linkedin/qark.
[9] Smartphone OS Market Share, 2016 Q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.
[10] System Permissions. http://developer.android.com/guide/topics/security/permissions.html.
[11] Archive.org. Android Apps. https://archive.org/details/android_apps.
[12] 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.
[13] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of Android ad library permissions. *arXiv preprint arXiv:1303.0857*, 2013.
[14] B. Carbunar and R. Potharaju. A Longitudinal Study of the Google App Market. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, ASONAM '15, pages 242–249, New York, NY, USA, 2015. ACM.
[15] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.
[16] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.
[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] Google Inc. Android and Security. http://googlemobile.blogspot.co.uk/2012/02/android-and-security.html.
[19] Google Inc. Android Apps on Google Play. https://play.google.com/store/apps.
[20] Y. Jiang and Z. Xuxian. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS)*, 2013.
[21] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.
[22] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.
[23] Marcello Lins. GooglePlayAppsCrawler https://github.com/MarcelloLins/GooglePlayAppsCrawler.
[24] Nielson. Smartphones: So Many Apps, So Much Time. http://www.nielsen.com/us/en/insights/news/2014/smartphones-so-many-apps–so-much-time.html, July 2014.
[25] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22nd USENIX Security Symposium*, pages 543–558, Washington, D.C., 2013. USENIX.
[26] OWASP. Projects/OWASP Mobile Security Project - Top Ten Mobile Risks. https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks.
[27] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
[28] J. Seo, D. Kim, D. Cho, T. Kim, and I. Shin. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.
[29] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st USENIX Security Symposium*, pages 553–567. USENIX, 2012.
[30] T. Vidas, N. Christin, and L. Cranor. Curbing Android permission creep. In *Proceedings of Web 2.0 Security & Privacy*, volume 2, 2011.
[31] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 221–233, New York, NY, USA, 2014. ACM.
[32] 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*, pages 31–40. ACM, 2012.

APPENDIX

A. OWASP TOP 10 MOBILE RISKS

Table 9 details the Top 10 Mobile Risks according to OWASP [26]. The vulnerabilities M1, M5 and M9 do not stem directly from inadequate or improper code implementation in the mobile app and are thus considered out of the scope of our analysis.

Table 9: OWASP Top 10 Mobile Risks

ID	Description	Relevant?
M1	Weak Server Side Controls	No
M2	Insecure Data Storage	Yes
M3	Insufficient Transport Layer Protection	Yes
M4	Unintended Data Leakage	Yes
M5	Poor Authorization and Authentication	No
M6	Broken Cryptography	Yes
M7	Client Side Injection	Yes
M8	Security Decisions Via Untrusted Inputs	Yes
M9	Improper Session Handling	No
M10	Lack of Binary Protections	Yes