Understanding the Responsiveness of Mobile App Developers to Software Library Updates

Tatsuhiko Yasumatsu
Waseda University
Tokyo, Japan
ty@nsl.cs.waseda.ac.jp

Takuya Watanabe
NTT Secure Platform Labs
& Waseda University
Tokyo, Japan
watanabe.takuya@lab.ntt.co.jp

Fumihiro Kanei
NTT Secure Platform Labs
Tokyo, Japan
kanei.fumihiro@lab.ntt.co.jp

Eitaro Shioji
NTT Secure Platform Labs
Tokyo, Japan
shioji.eitaro@lab.ntt.co.jp

Mitsuaki Akiyama
NTT Secure Platform Labs
Tokyo, Japan
akiyama@ieee.com

Tatsuya Mori
Waseda University
& RIKEN AIP
Tokyo, Japan
mori@nsl.cs.waseda.ac.jp

ABSTRACT

This paper reports a longitudinal measurement study aiming to understand how mobile app developers are *responsive* to updates of software libraries over time. To quantify their responsiveness to library updates, we collected 21,046 Android apps, which equated 142,611 unique application package kit (APK) files, each corresponding to a different version of an app. The release dates of these APK files spanned across 9 years. The key findings we derived from our analysis are as follows. (1) We observed an undesirable level of responsiveness of app developers; 50% of library update adoptions by app developers were performed for more than 3 months after the release date of the library, and 50% of outdated libraries used in apps were retained for over 10 months. (2) Deploying a security fix campaign in the app distribution market effectively reduced the number of apps with unfixed vulnerabilities; however, CVE-numbered vulnerabilities (without a campaign) were prone to remain unfixed. (3) The responsiveness of app developers varied and depended on multiple factors, for example, popular apps with a high number of installations had a better response to library updates and, while it took 77 days on average for app developers to adopt version updates for advertising libraries, it took 237 days for updates of utility libraries to be adopted. We discuss practical ways to eliminate libraries with vulnerabilities and to improve the responsiveness of app developers to library updates.

CCS CONCEPTS

• **Security and privacy → Software security engineering**;

KEYWORDS

Android Security; Mobile Apps Measurement; Software Library; Mobile App Developers

ACM Reference Format:
Tatsuhiko Yasumatsu, Takuya Watanabe, Fumihiro Kanei, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. 2019. Understanding the Responsiveness of Mobile App Developers to Software Library Updates. In *Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19), March 25–27, 2019, Richardson, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3292006.3300020

CODASPY '19, March 25–27, 2019, Richardson, TX, USA

ACM ISBN 978-1-4503-6099-9/19/03…$15.00
https://doi.org/10.1145/3292006.3300020

1 INTRODUCTION

The rise of mobile app marketplaces has attracted a large number of app developers who wish to publish or monetize their apps on mobile app distribution platforms. The number of applications on these platforms has dramatically increased since the first iPhone /Android was released way back in 2007/2008. As of September 2018, there are over 2.8 million apps available on the official app distribution platform for Android–Google Play [3]. Published Android apps have been installed in over 2 billion Android devices worldwide.

Modern mobile app development is highly dependent on software libraries, which enable developers to incorporate rich functionalities into their apps without requiring large coding effort, for example, installing advertisements to monetize apps, providing access to social media services such as Twitter or Facebook, and providing QR code reader functionalities. In general, mobile app developers use third-party libraries to ease the burden of the app development process.

The widespread use of software libraries has resulted in high productivity in the development of apps, however some software libraries can be piggybacked with vulnerabilities, thereby putting end users of the app at a security risk. In fact, previous studies [8, 24, 37, 43] have reported that many app vulnerabilities originate from vulnerabilities in third-party libraries. The problem is that, even if a developer of a software library has fixed a vulnerability, many app developers may not notice the update or may not adopt it immediately, leaving the vulnerable version of the library in their apps. In addition, defects found in a third-party library may lower

the quality of the apps that use that library, consequently degrading the quality of the end users' experience [5, 35].

Therefore, being responsive to updates of software libraries plays a key role in maintaining the security and/or quality of mobile apps. However, it is not clear how responsive app developers are to software library updates. In particular, we are interested in the following research questions.

- *How long does it take for an app developer to adopt an update of a software library?*
- *What are the primary factors that determine their responsiveness to updates of software libraries?*

To address these research questions, we performed a longitudinal, large-scale measurement study of Android apps. We first collected 21,046 unique Android apps. For each app, we collected its previous versions, resulting in 142,611 unique application package kit (APK) files, each corresponding to a different version of an app. Note that these APK files were released between November 2008 and August 2017, that is, our measurement time window is approximately 9 years. By performing a statistical analysis on these APK files, we detected 152 different software libraries, or 1078 unique versions. For the detected libraries, we also noted their release dates. By correlating the release dates of the libraries with the detected libraries in the different versions of the apps, we can measure the time taken for an app developer to adopt an update of a library.

Our key findings are summarized as follows:

- We found an undesirable level of responsiveness by the app developers: 50% of library update adoptions were performed for more than 3 months after the release date of the library, whereas 50% of outdated libraries used in apps were retained for over 10 months.
- Security fix campaigns conducted by the app market provider can effectively reduce the number of apps with unfixed vulnerabilities; however, CVE-numbered vulnerabilities (without a security fix campaign) were prone to remain unfixed.
- The responsiveness of the app developers varied and depended on several different factors, for example, popular apps with a high number of installations had a better response to library updates. They took 77 days, on average, to adopt library version updates for advertising libraries and took 237 days to update utility libraries.

Given these observations, we discuss possible ways to eliminate vulnerable libraries and improve the responsiveness of mobile app developers.

2 DATA

In this section, we present two key data acquisition methodologies we used for our analysis: (i) extracting the version histories of apps and (ii) software library detection. Then, we provide an overview of the data we collected. Note that we adopted Android OS as our analysis target because, of the official and third-party mobile app marketplaces, it has the largest number of users and available apps.

2.1 Data Acquisition Methodologies

2.1.1 *Collecting previous versions of apps.*

In the Android marketplace (Google Play), a mobile device will only see the latest version of an app. However, if one specifies a previous version, it is possible to retrieve this version by sending a request via Google Play API [23]. The version that needs to be specified is called the *versionCode*, which is expressed as an integer value assigned by the app developer. App developers can version apps arbitrarily; however, *versionCode* takes any value between 1 and 2,100,000,000[1].

We collected the version histories of apps using the following procedure. First, we checked the latest version of the app. Then, we made iterative requests for previous versions by decrementing the version requested. Note that downloading one version of an app corresponds to downloading a single APK file. APK is a file format used to install Android application software. It contains the execution codes and resources required to execute the app on an Android device. To limit any harmful effects to the service, we limited our rate of requests and spent 2 months on data collection. Other ethical considerations that we took when collecting the data are discussed in Section 5.

2.1.2 *Library Detection.*

To analyze the responsiveness of app developers when adopting library updates, we need to know which version of a library is included in the APK. There have been several studies that have proposed methods to detect libraries included in an APK [4, 11, 36, 43]. We adopted a tool known as LibScout [4], which can detect libraries and their versions for a given APK file. LibScout constructs fingerprints from class profiles of both apps and libraries and establishes an obfuscation-resilient similarity scoring.

By detecting the library and its version from each APK in the version history of an app, we could identify the timings at which an app adopted each library update. We used this timing information to study the responsiveness of the app developers. Because there are diverse types of libraries, we manually investigated the functionalities of the detected libraries and categorized them into 10 groups, as shown in Table 1. The compiled rule of categorization is available at https://github.com/Library-Responsiveness/category.

2.2 Data Overview

App version histories were carefully collected from Google Play from July 2017 to August 2017, applying the methodologies covered in Section 2.1. Apps on Google Play were classified into 34 categories, for example, Social, News, or Games. We compiled a list of the top apps in each category. In addition, for each app listed, we checked the supplementary apps developed by the same developer and added them to the list.

As a result, we collected the version histories of 21,046 apps and, because each app history may consist of multiple APKs, our data set included 142,611 distinct APKs. Of these apps, 12,790 (134,355 APKs) had more than two versions, indicating that they had been updated at least once. We succeeded in downloading the version history of the apps in the following top-five categories: Games (2137 apps), Lifestyle (1978 apps), Personalization (1786 apps), Entertainment (1376 apps), and Education (1088 apps). The measurement window of the collected APKs spans 9 years, from November 2008 to August 2017.

[1]The greatest value allowed for versionCode is set to 2,100,000,000 on Google Play.

Table 1: Library categories.

Category	Description
Advertising	Ad-libraries.
Analytics	Libraries for analytics services.
AndroidDev	Android support or Google Play Services libraries.
AndroidUtil	Utility libraries that are specific to Android apps.
Extension	Extensions of Java core functions, such as IO improvement.
External	Libraries that provide access to external services other than SocialMedia.
Image	Libraries that support image processing.
Network	Libraries that support networking, such as HTTP or TLS client.
SocialMedia	Libraries that provide access to social media services.
Util	Generic utility libraries and other libraries.

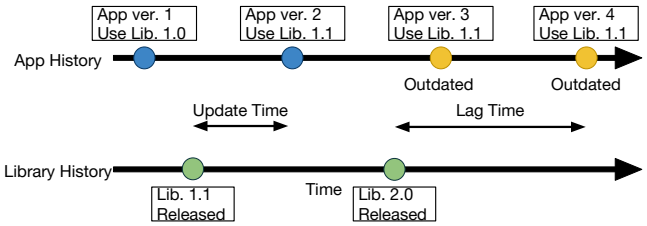

Figure 1: Example of measuring the update time and the lag time.

Applying library detection to the collected apps, we detected 152 different libraries with 1078 library versions. Apart from core Android development libraries, the most often-detected software library was Gson, which is an open-source JSON parser. Gson was used in 21,290 different APKs (12,896 apps).

3 ANALYSIS OF RESPONSIVENESS

In this section, we introduce two metrics, *update time* and *lag time*, aiming to measure the responsiveness of mobile app developers to software library updates. Using these two metrics, we then analyze the responsiveness of the mobile app developers. Finally, we perform an in-depth analysis of their responsiveness by looking at the differences between the libraries.

3.1 Metrics

To quantify the responsiveness of mobile app developers to library updates, we defined two key metrics, update time and lag time. Update time is defined as the time difference (in days) from the day the new version of the library was released to the day the app adopted the newly updated version of the library. Figure 1 illustrates an example of an update time measurement where the developer of an app adopted a version update of a library from version 1.0 to 1.1. The update time is the number of days between the release date of the library version 1.1 and the day that the next version of the app (version 2) was compiled and built. The update time measures the time needed for the app developer to adopt an update of a software library. App developers may build an APK multiple times before releasing it to Google Play. In our study, we used the build date of the APK that was actually released to Google Play and available to end users.

The lag time is defined as the time (in days) the app used an outdated (lagged) version of the library. Library usage is considered to be lagged when the latest version of the library is not used at the time when a version of an app is built. In other words, the lag time represents the period when the app is using an outdated version of the library. Figure 1 illustrates an example of a lag time measurement where versions 3 and 4 of an app used an outdated version of the library. Note that, if the next version, which is likely version 5, updates the library to 2.0, the lagged period will not be extended. The lag time measures the time a developer continues using an outdated version of a software library. In other words, it represents the length of time for which an app developer has ignored a software library update.

We opted to use the build date of the app instead of the release date in the marketplace because we were interested in how developers respond to library updates at the time of app development. Here, the build date can be extracted from the timestamp of a file called "AndroidManifest.xml," which is given in every APK file. Leveraging the build date gives us more accurate insights into the responsiveness of the app developers. Note that the build dates and the release dates of the apps in the marketplace are fairly close. We analyzed 17,000 randomly sampled apps and found that, for more than 80%, the time differences between the build date and the release date were within 5 days, which indicates that the majority of app developers release an app to the marketplace right after they built the latest version.

Note that, while both metrics look similar, they have clear differences, that is, while the update time characterizes app developers who are responsive to software library updates, the lag time characterizes developers who are less responsive and do not maintain their libraries. More specifically, the update time measures the time taken to adopt a library update since its release, and the lag time measures how long the library used in an app was left outdated. For example, when an app developer adopts the latest library update (i.e., App ver. 2 in Figure 1), this adoption cannot be measured with the lag time because the corresponding library was not outdated (lagged) at all. Therefore, examining both the update time and the lag time will provide better insights into the responsiveness of mobile app developers.

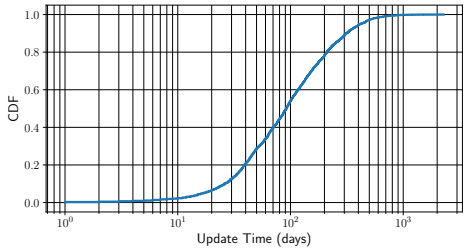

Figure 2: CDF of the update time.

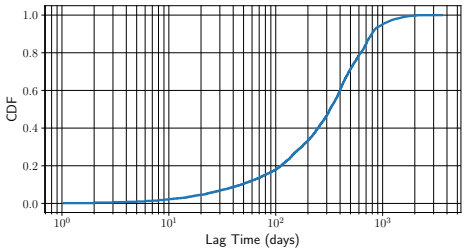

Figure 3: CDF of the lag time.

3.2 Analysis of the update time and the lag time

3.2.1 Measuring the update time and the lag time.
Measuring the update time and the lag time requires the release dates for each version of a library and the build dates for each version of an app throughout its version history. For the release dates of the libraries, we used the information included in LibScout [4]. For the build dates of the apps, we used the timestamp for "AndroidManifest.xml", a file every APK must contain. This timestamp records the time the APK was built.

Limitation: Note that recent versions of Gradle, which is the default build system for Android Studio (the IDE for developing an Android app), zero out the timestamps of files in APKs. Therefore, for APKs developed with recent versions of Gradle, we could not extract the time information. We disregarded such APKs in our analysis (except when counting the number of vulnerabilities). We observed update times that had negative values. These cases were caused primarily because of incomplete library-release-date data. Libraries with incomplete release date data were removed from our analysis. As the fraction of removed apps was 15% in total, we believe that this data cleansing will not affect the subsequent analyses.

3.2.2 Results.
Figure 2 shows the cumulative distribution of the update time analysis applied to our data. We found that approximately 50% of the library version update adoptions were performed for more than 90 days after the release of the library, that is, it took approximately 3 months for app developers to adopt library updates to succeeding app versions. Note that libraries used in apps that have never been updated by app developers will not appear in this figure. Figure 3 shows the cumulative distribution of the lag times measured from the version histories of our app data set. We found that approximately 50% of outdated usages of libraries remain for more than 300 days (10 months). The majority of this long lag time corresponds to cases where library updates were not adopted during the measurement period.

To our surprise, we found that 58.7% of the adoptions of library updates in apps did not adopt the newest versions. This indicates that app developers adopted outdated versions of the library update instead of the latest version at the time the APK was built. As Derr et al. [11] reported, Android app developers tend to search for library updates on the Internet (e.g., official web sites, blogs, or Stack Overflow) because the Android app platform does not provide a central library marketplace such as CocoaPods for iOS. Because the information available on such web sites can become obsolete, developers are prone to adopting outdated versions of libraries.

3.3 In-depth analysis of responsiveness

Next, we studied how the responsiveness of mobile developers differs between the library categories shown in Table 1. Table 2 summarizes the results. Overall, library categories that access external services (e.g., Advertising, Social Media, and External) adopt library updates quicker than libraries that do not. Note that Network libraries are those that provide networking functionalities to clients but do not provide access to external services by themselves. Because libraries empowered with external services are prone to changes in services and/or APIs, it is natural that app developers attend to such updates so that the apps can continue to work with the external services. Specifically, advertising libraries had the shortest average update time and lag time, indicating developers' incentive to keep such libraries updated so as not to lose revenue from advertisements.

Conversely, libraries such as Extension or Util had the worst update/lag statistics. Because these libraries may not change their specifications drastically, it is natural that app developers do not adopt updates of these libraries even when newer versions become available.

To look into the responsiveness of mobile developers at a fine-grain scale, we sampled the three specific libraries as shown in Table 3. These libraries will be used for further analyses in Section 4. Figures 4 and 5 present the distributions of the average update time and lag time per app for the three libraries, respectively. The panels are sorted according to their median values. Note that Gson had the longest update/lag time. This agrees with the results shown above that the update/lag time of utility libraries tend to be longer than those of other libraries. This observation agrees with the results of Derr et al. [11], who reported that the primary motivation for app developers to adopt library updates is to make use of a new functionality. Because Gson is a parsing library, the frequency of new functionalities is less. In addition, we observed that the update/lag time for AdColony was relatively short. Again, this observation agrees with the results shown in Table 2, that is, app developers tend to adopt library updates more quickly for the advertising libraries than for other libraries.

Table 2: Statistics of the detected updates/lags for each library category.

Category	# detected updates	# detected lags	average update time (days)	average lagged time (days)
Advertising	768	2,017	77	223
SocialMedia	1,198	2,667	103	485
External	49	238	105	334
Network	681	2,937	123	487
AndroidDev	21,020	86,301	138	364
AndroidUtil	285	1582	145	482
Image	284	1,834	166	587
Analytics	50	284	199	416
Extension	295	2,276	218	566
Util	700	3,795	237	666

Table 3: The three specific libraries chosen for further analysis.

Name	Category	Description
Gson [22]	Utility	The most used library across all the categories except AndroidDev.
Facebook [13]	SocialMedia	The most used library in the SocialMedia category.
AdColony [1]	Advertising	The most used library in the Advertising category.

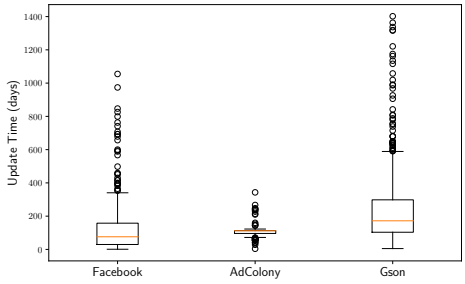

Figure 4: Distributions of the update time (box plots).

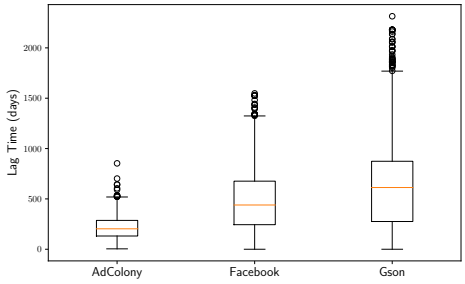

Figure 5: Distributions of the lag time (box plots).

Table 4: Statistics of vulnerable libraries.

Library	Vulnerable Versions	Vulnerable apps	Unfixed	CVE	CVSS	ASI Program
MoPub	3.10-4.3	119	8 (6.72%)	-	5.9	✓
Supersonic	5.14-6.3.4	71	10 (14.1%)	-	4.8	✓
Vungle	3.0.6-3.3.0	28	4 (14.3%)	-	9.0	✓
Facebook	3.15	54	23 (42.6%)	-	6.6	–
ApacheCC	3.2.1 / 4.0.0	34	18 (52.9%)	-	9.0	–
Dropbox	1.5.4-1.6.1	18	10 (55.6%)	CVE-2014-8889	5.3	–
OkHttp	2.1.0-2.7.4 / 3.0.0-3.1.2	1,031	653 (63.3%)	CVE-2016-2402	5.9	–
Total		1,355	726 (53.6%)			

4 FACTORS ASSOCIATED WITH THE RESPONSE TIME

In this section, we examine several factors that are likely associated with the responsiveness of developers. In particular, we study the vulnerabilities included in a library, app properties (popularity and category), library properties, developer properties, and software development conventions (semantic versioning).

4.1 Reaction to Vulnerabilities

4.1.1 Detecting Apps with Vulnerable Versions of Libraries. We were able to decide whether a library was vulnerable by parsing the library version detected from each APK and comparing that version to a vulnerable version of the library. Table 4 summarizes the vulnerabilities [18, 21, 34, 39, 40] detected for the seven libraries found in our data set and whether they were fixed. We sampled

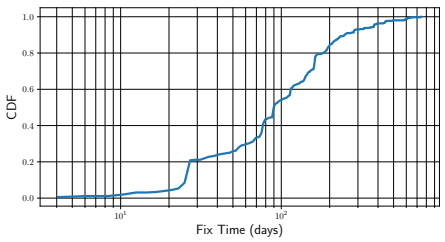

Figure 6: CDF of the fix time.

these seven libraries due to their popularity; in fact, we found that 28.3% of APK files we collected made use of one of these libraries. An app was deemed vulnerable if at least one vulnerable version of a library was featured in its history. Then each vulnerable app was classified into one of two groups: fixed apps and unfixed apps. An app was considered fixed if it is no longer vulnerable, either via updating the library or via removal of that library. If an app still had the vulnerability in its most recent version, it is considered to be unfixed.

For each vulnerability, we obtained the average number of installations for the fixed and unfixed apps. Contrary to our expectation that the fixed apps would have a larger number of installations than unfixed apps, there were three vulnerabilities (MoPub, Dropbox, and Apache CC) for which fixed apps had a lower number of installations. This result shows that even popular apps may leave vulnerabilities unfixed and expose end users to security risks. Because Google Play is known to be a supermarket [42], where a small percentage of apps contribute to most of the app downloads, the fact that even a popular app may not fix its vulnerabilities needs to be addressed.

4.1.2 *Time taken to fix the vulnerabilities.*

Having detected the vulnerable libraries in the apps, we can measure how long it takes for the app developers to fix a vulnerability. For apps that fixed their vulnerabilities, we measured the fix time, that is, the time difference in days from the day the very first safe version of the library was released to the day when the app that adopted a safe version of the library was built. The definition of the fix time is similar to the update time. The difference is that the measurement period starts only if the library update includes a vulnerability fix and ends when the app has adopted the fixed version of the library.

Figure 6 shows the cumulative distribution of the fix time analysis applied to the vulnerable libraries listed in Table 4. For this analysis, we excluded libraries with inaccurate release dates. The distribution shows that, for 50% of the vulnerable libraries in apps, approximately 90 days were required for the app developer to adopt library updates fixing a vulnerability. We also notice that there is a spike between 20 and 30 days. The spike corresponds to the apps developed by a single prolific developer. The developer has fixed many apps published by him/her in a short period of time. In Section 4.6, we will look into the effect of such prolific app developers who have published a large number of apps in the marketplace.

4.1.3 *CVE ID.*

We also studied whether the detected library vulnerabilities were assigned IDs of CVEs [33], which is a list of common identifiers for publicly known vulnerabilities. Of the libraries we investigated, Dropbox and OkHttp had vulnerabilities with CVE IDs, indicating that these vulnerabilities had been widely announced. For each library, we measured the *unfixed rate*, which is the fraction of unfixed apps of the apps that we marked vulnerable, that is, the apps that contained the vulnerable libraries. The fourth column in Table 4 shows the results. It is surprising to see that the unfixed rates of both vulnerabilities with CVE IDs are high (Dropbox with 55.6% and OkHttp with 63.3%) and that they are the highest unfixed rates out of all the vulnerabilities we analyzed; this indicates the absence of knowledge about CVE IDs among app developers.

4.1.4 *Severity of the Vulnerabilities.*

Next, we studied whether the severity of the vulnerabilities prompted app developers to fix the library vulnerabilities. To quantify the severity of each vulnerability, we used the Common Vulnerability Scoring System v3.0 (CVSS) [16], which is an open framework that generates a numerical score reflecting the severity of a given vulnerability; the scores range from 0 to 10, and the vulnerability is considered more severe when the CVSS score is high. The sixth column in Table 4 presents the CVSS score for each vulnerability. For vulnerabilities with a corresponding CVE ID, we made use of a CVE database [41] to obtain the CVSS scores. For the vulnerabilities without an assigned CVE ID, we manually calculated the CVSS score, following the CVSS specifications [17].

The result indicates that there is no clear correlation between the severity of the vulnerability and the unfixed ratio. For example, both the libraries Vungle and Apache CC had severe vulnerabilities, that is, CVSS scores of 9.0. While Vungle had a low unfixed rate of 14.4%, Apache CC had a high unfixed rate of 52.9%. Therefore, the high severity of a vulnerability does not always prompt app developers to fix it, implying that even severe vulnerabilities could be left unfixed in the app market. We believe that the primary reason why many app developers fixed the vulnerability found in Vungle was because the vulnerability had been targeted by Google's App Security Improvement Program (the ASI Program).

4.1.5 *App Security Improvement Campaign.*

A security fix campaign conducted as a part of the ASI Program [21] has targeted several vulnerabilities found in Android apps. When a vulnerability is targeted by an ASI Program campaign, app developers that have published an app with the targeted vulnerability receive a message via email that informs them of how to fix the vulnerability. Each ASI Program campaign sets a deadline to fix the vulnerability, and after the deadline, any app with the vulnerability will not be able to be uploaded to Google Play; therefore, app developers are forced to remove such vulnerabilities. This indicates that Google Play will scan the APK file for vulnerabilities before publishing it in the market. Note that apps with targeted vulnerabilities will not be deleted from Google Play even after the deadline. The only penalty is that the developers are prohibited from uploading a new version of an app, unless that new version has fixed the vulnerability.

Of the vulnerabilities listed in Table 4, MoPub, Supersonic, and Vungle were targeted by the ASI program. We can clearly see the effectiveness of the ASI Program campaign because apps with these targeted libraries have lower unfixed rates compared to those of other vulnerabilities. For these targeted vulnerabilities, more than 85% of apps fixed the vulnerabilities during the measurement period.

4.2 App Popularity

Next, we study the relationship between the app popularity and the responsiveness of developers to library updates using the number of installations per app for the three specific libraries shown in Table 3. It is possible for a version history of an app to have more than one update time or lag time for a single library, that is, an app using a software library can update/lag it more than once, resulting in multiple update/lag times. Therefore, we calculated the mean values of the update/lag times per app. Owing to space constraints, in the following analysis, we omit the results for the update time and only show the results for the lag time.

To make the analysis clear, we also classified the apps according to the number of installations, i_a, which can be obtained from metadata provided in the marketplace. Using the 33$^{\text{rd}}$ percentile, I_1 and the 67$^{\text{th}}$ percentile, I_2 of the apps' number of installations of an app subject to our analysis, we classified the apps into the following three groups:

(1) Unpopular: $i_a < I_1$;
(2) Normal: $I_1 \leq i_a < I_2$; and
(3) Popular: $I_2 \leq i_a$;

where $I_1 = 1000$ and $I_2 = 10,000$ in the data set. Note that the number of installations published in Google Play is discretized, for example, 100, 500, and 1000.

Figure 7 shows the CDFs of the lag time for the three libraries. For each library, we analyzed how the difference in the app popularity affected the responsiveness via the lag time. First, in all three libraries, no significant difference was found between the normal and unpopular apps. For Gson and Facebook, we can see that the distributions of the lag times for popular apps are skewed toward short lag times, that is, popular apps tend to adopt a library update more quickly than normal and unpopular apps. For Gson, for example, the average lag time for popular apps with more than 10,000 downloads is 597 days, which is long but still shorter than that of apps with normal popularity, which is 684 days. Mann–Whitney U-tests between the popular and normal app lag times revealed statistically significant differences for Gson ($p < 10^{-4}$) and Facebook ($p < 0.01$). On the contrary, for AdColony, we did not find any statistically significant differences between the three popularity classes. This observation implies that even apps with few installations and low prevalence are as sensitive as popular apps when it comes to advertisement libraries.

Finally, to verify the generality of the observed tendency, we extended the study by analyzing 10 additional libraries taken from the most popular libraries. For each library, we first classified the apps into three groups—unpopular, normal, and popular—following our definition shown above. We then analyzed the lag times of the apps in each category. We aimed to test whether there were statistically significant differences in the lag times for the different popularity classes. Accordingly, we used the Mann–Whitney U-test. Table 5 summarizes the results. Of the 13 libraries, including the 3 reference libraries, 10 libraries exhibited the same tendency, that is, popular apps had shorter lag times than normal/unpopular apps with a statistically significant difference. This result positively supports our observation that popular apps tend to adopt library updates more quickly than less popular apps.

Table 5: Mann–Whitney U-tests of the lag times for apps from different popularity classes: U, unpopular; N, normal; and P, popular. The libraries presented with bold fonts had statistically significant differences between popularity classes. (top) The three reference libraries and (bottom) additional libraries.

Library	U vs. N	N vs. P
	p-value	
Gson	0.0516	$\mathbf{< 10^{-4}}$
Facebook	0.0604	**< 0.01**
AdColony	0.839	0.261
OkHttp	0.233	$\mathbf{< 10^{-5}}$
Nine-Old-Androids	1.00	0.33
universal-image-loader	0.696	$\mathbf{< 10^{-10}}$
Picasso	**< 0.02**	0.499
Retrofit	0.119	$\mathbf{< 10^{-4}}$
Bolts	0.434	**< 0.02**
okio	0.624	$\mathbf{< 10^{-6}}$
Twitter4J	**< 0.03**	**< 0.02**
Apache-Commons-Lang	0.506	**< 0.01**
Apache-Commons-IO	0.454	0.148

Our findings suggest that app popularity does affect the motivation of app developers to adopt library updates. More popular apps react to new library releases more sensitively than do less popular apps.

4.3 Number of App Versions

To determine how the number of versions released for an app affects the responsiveness of the app developers, we classified the apps into three groups depending on the number of versions released. According to the number of released versions of an app, v_a, we grouped the apps as follows:

(1) Few versions: $v_a < V_1$;
(2) Medium versions: $V_1 \leq v_a < V_2$; and
(3) Many versions: $V_2 \leq v_a$;

where V_1 and V_2 are the 33$^{\text{rd}}$ and 67$^{\text{th}}$ percentiles, respectively. In our data set, the percentiles were $V_1 = 3$ and $V_2 = 8$.

In this analysis, we ignored apps that have released only one version. Figure 8 shows the distributions of the lag time for the three groups defined above. Overall, we do not see a clear unified relationship between the number of released versions for an app and the lag time. For Gson, apps with more versions had shorter lag times. For Facebook and AdColony, the relationship was opposite, that is, apps with fewer versions had shorter lag times. Therefore, we believe that the number of released app versions is not a primary factor determining the responsiveness of a developer.

4.4 App Category

To study the relationship between the app category and the responsiveness of the developers, we measured the average lag time per app for the top-five app categories in our data set, that is, Games, Education, Lifestyle, Personalization, and Entertainment. Table 6 summarizes the results using the three case study libraries shown

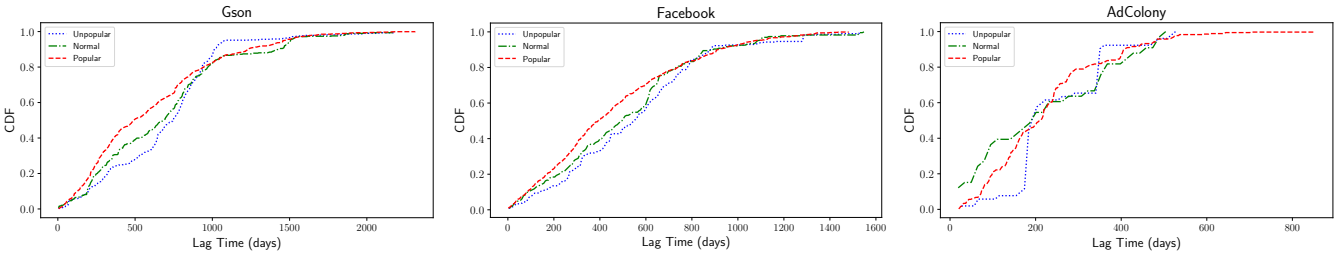

Figure 7: Popularity of apps versus the lag time per app for three libraries: Gson, Facebook, and AdColony.

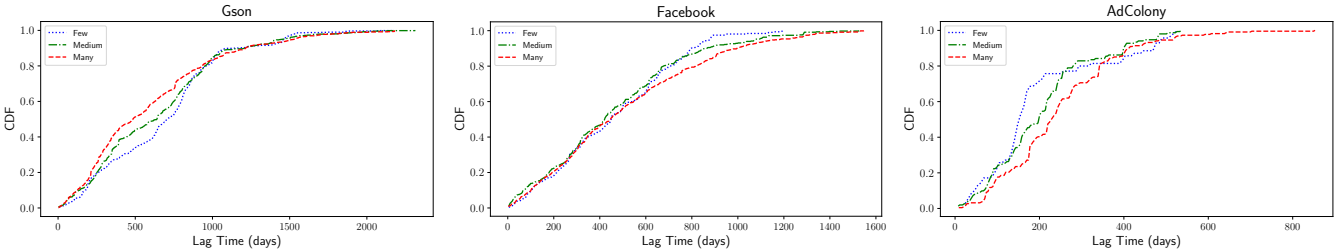

Figure 8: Number of released versions versus the lag time per app for three libraries: Gson, Facebook, and AdColony.

Table 6: Average lag time (days) for the top-five app categories in the data set.

	Gson	Facebook	AdColony
Game	772	449	259
Education	551	434	177
Lifestyle	735	560	190
Personalization	383	414	143
Entertainment	818	372	237

in Table 3. While we cannot observe any unifying rule of responsiveness in the categories, we found that the responsiveness to each library is different for each app category. For example, Gson lagged for 818 days on average for Entertainment apps, whereas the lag time for the same library was much shorter for Personalization apps, with an average lag time of 383 days. As expected, these are statistically different with a significance level of $p < 10^{-9}$.

If we focus on the app categories, we see that four out of the five app categories have AdColony as the library with the fastest responsiveness. Again, this advertising library exhibited the fastest responsiveness across all the app categories. For other libraries, we do not see a clear rule with which to interpret the results, for example, in Personalization apps, Gson had the next shortest average lag time of 383 days, whereas in Entertainment apps, Facebook had the next shortest average lag time of 372 days.

4.5 Frequency of Library Updates

To understand the relationship between the library evolution speed and the responsiveness of developers to library updates, we analyzed the correlation between the average update interval of the libraries and the responsiveness of app developers to library updates. For this, we adopted the Spearman rank correlation coefficient. For each library we detected, we first measured the average length of the update intervals. Next, for each library, we analyzed apps

Table 7: Spearman's rank correlation coefficients, ρ, for the pairs (T_i, T_u) and (T_i, T_l), where T_i is the average of update interval of a library and T_u and T_l are the averages of the update and lag times for a library, respectively.

pair	ρ	p-value
(T_i, T_u)	0.465	$< 10^{-5}$
(T_i, T_l)	0.317	$< 10^{-3}$

that were using that library, extracted their update/lag times, and took their average values. Finally, we computed the Spearman rank correlation coefficient for pairs of the average update interval of a library (T_i) and the average of the update/lag times for a library (T_u/T_l). The results are shown in Table 7. For both cases, we can see a low positive correlation. These positive correlations imply that either app developers had a tendency to keep up with a quickly evolving library or that they had a tendency to lose interest in adopting updates for slowly evolving libraries. However, because the correlations were weak, we could not derive a conclusive interpretation.

4.6 Prolific Developers

Throughout our analysis, we observed that there are a non-negligible number of prolific developers who have published many apps in the marketplace. Given this observation, we studied whether prolific developers are responsive to library updates. Figure 9 shows the CDF of the number of apps published by a developer. Our hypothesis was that prolific developers would adopt library updates more quickly than other developers. Similar to the previous analyses, we classified the apps into two groups depending on whether the developer of the app had developed five or more apps. We set the threshold, 5, as the 67^{th} percentile of the number of apps published by a developer. If the developer of an app had developed five or

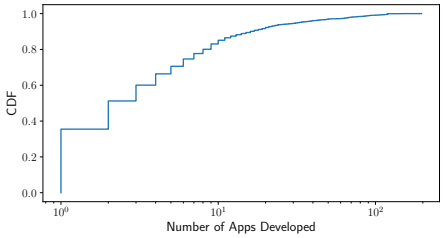

Figure 9: CDF of the number of apps published by a developer.

Table 8: Average lag time (days) for the apps developed by normal and prolific developers.

Library	Normal	Prolific	p-value
Gson	578	**652**	$< 10^{-5}$
Facebook	442	**498**	$< 10^{-3}$
AdColony	215	225	0.315

more apps, we categorized the app as an app developed by a prolific developer. If the developer of an app had developed four or fewer apps, we categorized the app as an app developed by a normal developer.

Table 8 summarizes the results. These results contradict our hypothesis, that is, for Gson and Facebook, apps that were developed by prolific developers tended to have longer lag times, indicating that prolific developers were not responsive to library updates. The average number of installations for apps developed by prolific developers was 30.7×10^4, whereas that for apps developed by normal developers was 13.2×10^5. This observation agrees with the result shown in Section 4.2 that popular apps adopt libraries more quickly than less popular apps.

4.7 Semantic Versioning

A previous study [11] reported that failures in semantic versioning are the primary *root cause* of the poor adoption of third-party library updates. Given this background, we studied how the semantic versioning strategy used by a library affects the responsiveness of developers to that library. Semantic versioning [38] is a versioning strategy for software development that consists of a set of simple rules. By following these rules, the versioning on a software update can inform software users of what type of update it was. The semantic versioning rule guides software developers to adopt the version format X.Y.Z and increment each digit following the next rules: increment X on backward-incompatible changes (MAJOR updates); increment Y if a new backward-compatible functionality is introduced (MINOR updates); and increment Z if only backward-compatible bug fixes are introduced (PATCH updates).

For the 40 libraries that we collected from the Maven repository, we used the LibScout API analysis functionality to measure how each library's updates followed or violated the semantic versioning strategy. In addition, we calculated the averages of the update time and the lag time as responsiveness measures for each library. Then, we measured the Spearman rank correlation (ρ) between the percentage of updates that followed semantic versioning and the responsiveness measures for each library. We found no correlation between the correctness of the semantic versioning and the update time ($\rho = -0.269$). Nor did we detect any correlation between the correctness and the lag time ($\rho = -0.279$).

We further investigated problematic violations of the semantic versioning rule where the versioning of the update is indicated as MINOR or PATCH, even though the update was actually backward incompatible. This violation is problematic because the versioning tells the library user that the update does not break any backward compatibility, even though the actual change in the library breaks this compatibility, which may lead the library user to experience difficulties. For each library we measured the percentage of problematic updates and measured the Spearman correlations with the library adoption measures. Again, no correlation was found between the rate of problematic updates and the update time ($\rho = 0.190$) or the lag time ($\rho = 0.107$).

Even though further study might reveal some relationship between the semantic versioning and the responsiveness of app developers to library updates, we could not find any clear correlations between the two. Even though failures in semantic versioning could contribute to the poor responsiveness of app developers, it may not be a primary *root cause*, seeing how we found many other factors that are associated with the responsiveness of mobile app developers.

4.8 Summary

We found that the responsiveness of mobile app developers to library updates is based on several different factors. Of the possible factors we analyzed, we observed clear correlations for the following three factors: awareness of the vulnerability, app popularity, and library category. In the following, we highlight these three factors.

First, we found that the presence of CVE ID and the high severity of a vulnerability had a small impact on prompting app developers to fix them. We also found that the deployment of Google's ASI Program campaign was effective in reducing unfixed vulnerabilities; more than 85% of apps with the three libraries targeted by the ASI program fixed such the vulnerabilities. These two observations imply that explicitly letting developers be aware of the presence of vulnerabilities in their apps is a key success factor in reducing the number of unfixed vulnerabilities.

Second, popular apps were more sensitive to adopting new versions of libraries than are unpopular apps. Of the 13 libraries we studied, 10 libraries indicated that popular apps had shorter lag times than normal/unpopular apps with a statistically significant difference.

Finally, via several avenues of research, we found that advertising libraries tend to elicit better responsiveness than those in other categories of libraries. Because developers monetize their apps using advertising libraries, it is natural that they will be sensitive to changes in such libraries, which represent the source of their revenue from these apps.

5 DISCUSSION

In this section, we discuss the limitations of our work. In addition, we discuss the effective methods to eliminate vulnerable libraries

and improve the responsiveness of mobile app developers. Finally, we discuss the ethical considerations we took when we performed our measurements.

5.1 Limitations

This study is a first step toward understanding the responsiveness of mobile developers in adopting updates of libraries. Our study has several limitations. One is a possible bias in our data set. Because our app data set consists of free apps from Google Play, our analysis result may not apply to paid apps or apps from other app markets such as the Apple App Store or other third-party marketplaces [27]. We note that more than 95% of Android apps published on Google Play are free apps [3]. In addition, we collected the app history, as explained in Section 2.1.1; this app collection method can collect the version history of apps but fails to collect the apps whose versionCode exceeds 40,374.

Because our library detection is based on an OSS tool LibScout, the accuracy of the study is limited by the precision of the tool. In addition, for some analyses, we based our app analysis on several case study libraries. A natural avenue for future work is to extend our analysis to libraries that we have not studied here. Now that we have undertaken basic case studies for several libraries and have obtained interesting insights into the responsiveness of app developers to library updates, it will be easier for us to extend our measurement to many other libraries that are being used in apps.

We performed in-depth app analyses in Section 3 and Section 4, using the 13 case study libraries we adopted. We note that although the number of libraries used for the analysis was limited, we have focused our case study analyses on popular libraries as explained in Table 3 and Section 4.2. Because popular libraries should have stronger effect on the entire app market, compared to unpopular libraries, analyzing popular libraries is a reasonable sampling way to understand the responsiveness of app developers in the app market.

5.2 Elimination of Vulnerable Libraries

To the best of our knowledge, we are the first to quantify the effectiveness of a market-wide security campaign (Google ASI). Vulnerabilities targeted by the ASI Program had a low unfixed rate with a maximum value of 14.3%. In addition, we demonstrated that the method of assigning CVE IDs was less effective in eliminating vulnerable libraries. Vulnerabilities with CVE IDs have a high unfixed rate with a minimum value of 55.6%, which is among the highest rates of the vulnerabilities we tested.

Given these observations, we suggest several ideas to reduce unfixed vulnerabilities in apps. One straightforward approach is to enhance a vulnerability fixing campaign similar to the ASI Program, which will encourage developers to fix vulnerabilities in their apps. Of the vulnerabilities we analyzed, four were not targeted by the ASI Program campaign. It is reasonable for a market provider (not only Google Play but also other marketplaces for other mobile platforms) to target these and other untargeted vulnerabilities to encourage app developers to fix vulnerabilities in their apps.

Another natural solution for encouraging developers to fix vulnerabilities would be to take actions to make the idea of CVE more widely understood by common Android app developers. As mentioned above, our finding imply that many app developers pay little attention to CVE; communicating the idea of CVE will help developers become aware of the existence of vulnerabilities.

Even for the severe vulnerability we found in the Facebook library, there was no official documents commenting on the vulnerability. This lack of documentation is problematic because Facebook is the most frequently used library that accesses external services. The reduced interest of library developers in announcing vulnerabilities will lessen opportunities for app developers to notice such vulnerability. Library developers must be more sensitive to security issues and let the app developers know about the vulnerabilities.

5.3 Improving Responsiveness to Library Updates

Our analysis revealed Android app developers' poor responsiveness to the updates of software libraries; the continued use of 50% of outdated libraries in apps, for more than 300 days. These poor and slow trends of library adoption may lead to various types of negative impacts on the Android ecosystem, such as security vulnerabilities or degradation in app quality.

In Section 4.2 we categorized the studied apps into three groups, popular, normal, and unpopular, depending on the number of installations. We found that popular apps adopt library updates quicker than normal or unpopular apps. Even though apps with reduced popularity are less dominant on Google Play, the apps in our data set have been installed more than 10 million times and their impact on end users is not negligible. Such observations suggest that enhancing the library responsiveness of these apps may greatly contribute to increasing the quality of the entire app market.

References [12, 15] highlight the increase and prevailing trend of citizen developers, that is, developers with lower technical skills. It is important to work toward a more sophisticated and easy-to-adopt library management system, such as an automatic library updating system, or semantic versioning with comprehensive information about the library version, to avoid confusion for such developers. However, because our analysis in Section 4.7 reveals no correlation between the semantic versioning and the library update adoption responses, in order for the semantic versioning to affect the responsiveness of app developers to library updates, the idea of semantic versioning must first be more widely recognized by developers.

A promising solution to improve the responsiveness of app developers is to set up a central library marketplace and to develop a package management tool for Android development. We observed that, when apps adopt library updates, 58.7% of apps do not adopt the latest version of the library. Because there is no central library marketplace, Android app developers tend to rely on information from the Web. However, obsolete information on web sites makes app developers prone to adopting outdated versions of libraries. As such, setting up a central market of libraries will help developers access fresh information and enhance their responsiveness to library updates. It is noteworthy that introducing a plugin to IDE such as Gradle Versions Plugin [6] could be another possible solution. Such IDE plugin is expected to automatically resolve the dependency of library updates.

5.4 Ethical considerations

Finally, we discuss the ethical considerations we took in our measurement study. We acquired all the APK files carefully so as not to violate the Acceptable Use Policy or to cause any harmful effects to the service. Accordingly, we slowly crawled the APK files and spent 2 months on the entire data collection. Our motive for of crawling the APKs was purely for research. For bona fide researchers who are interested in analyzing the data we collected, we are ready to share the data on request. Because our study exposed unfixed vulnerabilities in apps and libraries, we are following the principle of responsible disclosure and are now in the process of reporting our results to the corresponding app/library developers. The disclosures will include the app/library names and the categories of the vulnerability.

6 RELATED WORK

Mobile App Analysis. There have been several studies that have performed large-scale mobile app analyses [20, 25, 26, 32, 42, 44]. Viennot et al. [42] developed a system called PlayDrone, which efficiently crawls the official market (i.e., Google Play) and stores APK files. By collecting apps from Google Play for 47 days, McIlroy et al. [32] were able to analyze how frequently apps are updated. Ishi et al. [26] analyzed over 1.3 million Android apps collected from official and third-party marketplaces, and revealed that in the third-party marketplaces, 76% of the *cloned apps* that were originally published in the official market, were repackaged malware. Gonzalez et al. [20], examined the code reuse in both legitimate and malicious Android apps collected from various app marketplaces. Wu et al. [44] examined the relationship between the declared Android SDK versions in apps and the API calls used in the apps. They revealed that there were 1.8K Android apps that had the inconsistency between the SDK versions and API usage. Huang et al. [25] demonstrated that online malware scanning services are being used by not only legitimate users, but also by Android malware developers. Aldini et al. [2] proposed a method to detect repackaged Android apps by collecting execution traces of both genuine and repackaged apps from users.

There have been prior studies that have attempted longitudinal measurement studies of apps [9, 31, 37]. Carbundar et al. [9] crawled Google Play for 6 months and revealed that at most, 50% of apps are updated in all app categories.Taylor et al. [37] have taken snapshots of Google Play over time and investigated changes in permission and security. McDonnell et al. [31] analyzed the relationship between the core Android API stability and the API adoptions of app developers. Their results suggest that developers use fast-evolving APIs more than slow-evolving APIs.

Software Library Analysis. Software libraries play an essential role on modern software development in mobile platforms to reduce development costs and accelerate the release cycles of apps. The mainstream method of assessing mobile apps is first to detect the software libraries. There are a number of state-of-the-art library detection tools [4, 10, 29, 30, 43, 45]. LibD [29], LibRader [30], and LibScout [4] are publicly available. In particular, LibScout is the most frequently updated of these tools as of August 2018. The aim of library detection in our study was to analyze the responsiveness of app developers via changes in the libraries; therefore, we selected LibScout which can detect both the library and its version from a provided APK, as well as its release date.

Use of harmful or low quality libraries, can involve innocent host apps in security threats, or degrade the user experience of an app [5, 7, 28, 35, 43]. Li et al. [28] indicated that piggybacked apps with libraries containing malicious code could mislead security analyses. Bhoraskar et al. [7] also notes that a host app as a whole could become vulnerable if there were bugs in its libraries. Watanabe et al. [43] revealed that the majority of app vulnerabilities were attributed to third-party libraries. Bavota et al. [5] further revealed that high-rated apps are more likely to use stable APIs, whereas, Mojica et al. [35] demonstrated that the use of certain advertising libraries will lower the reputation of an app using those libraries.

Studies on App Developers. Derr et al. [11] conducted a human study on Android app developers via an online survey. They reported that many third-party libraries did not follow semantic versioning strategies, confusing app developers. However, they did not verify their finding via actual measurements over the app histories. On the contrary, we investigated the app developers' responsiveness to library updates by analyzing the version histories of the apps. Even though a further analysis may reveal the effect of semantic versioning, in our study, we did not find any correlation between the semantic versioning and the app developers' responsiveness to library updates.

Fischer et al. [14] examined if Android app developers copy security related code snippets from online discussion platform. They analyzed 1.3 million apps and revealed that 15.4% contained security related code snippets from online discussion platform.

Coarse-grained permission system may make developers prone to writing over privileged codes, resulting in insecure apps. Fratantonio et al. [19] demonstrated that providing *finer-grained* Android Internet permission management is a practical and desirable solution for the better management permission without sacrificing the usability. Also, app developers often make use of third-party services in their apps. It is known that some of such apps contain *developer credential* to authenticate themselves to the third-party server. Zhou et al. [46] analyzed Android apps collected from various markets and revealed that many developers do not protect their credentials at all, therefore it will be easy for an attacker to extract them from the apps.

7 CONCLUSION

In this paper, we report a longitudinal measurement study on the responsiveness of Android app developers to library updates. The study was based on app version histories collected from Google Play, and a static code analysis was applied to the app histories to detect which libraries were used in the apps. By comparing the app build dates and the library release dates, we were able to quantify the responsiveness of the mobile app developers.

Our findings revealed that the majority of developers were less responsive to library updates; 50% of library updates were adopted for more than 3 months after the library release date. We also studied factors that were likely associated with the responsiveness of the developers. We found that security fix campaign was effective in encouraging app developers to adopt library version updates, whereas several vulnerable libraries without such a campaign were

prone to being unfixed even if the vulnerabilities were severe or had CVE IDs. We also found that unpopular apps tended to respond to library updates slowly compared to popular apps and that it is important to enhance the responsiveness, to improve the overall quality of the app market. We hope our insights in this study will motivate the market, library developers, and the app developers to improve the status quo for responses to library updates.

ACKNOWLEDGMENT

A part of this work was supported by JSPS Grant-in-Aid for Scientific Research B, Grant Number 16H02832.

REFERENCES

[1] AdColony, Inc. [n. d.]. AdColony - Elevating mobile advertising across today's hottest apps. Retrieved September 22, 2018 from https://www.adcolony.com/

[2] Alessandro Aldini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. [n. d.]. Detection of repackaged mobile applications through a collaborative approach. *Concurrency and Computation: Practice and Experience* 27, 11 ([n. d.]), 2818–2838.

[3] AppBrain. 2018. Google Play stats. Retrieved September 22, 2018 from http://www.appbrain.com/stats/

[4] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *Proc. of ACM CCS, 2016*. 356–367.

[5] Gabriele Bavota, Mario Linares Vásquez, Carlos Eduardo Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2015. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *IEEE Transactions on Software Engineering* 41, 4 (April 2015), 384–407.

[6] Ben Manes. 2018. GitHub - Gradle Versions Plugin. Retrieved September 24, 2018 from https://github.com/ben-manes/gradle-versions-plugin

[7] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-Party Components.. In *Proc. of USENIX Security, 2014*. 1021–1036.

[8] Theodore Book, Adam Pridgen, and Dan S. Wallach. 2013. Longitudinal Analysis of Android Ad Library Permissions. *CoRR* abs/1303.0857 (2013). arXiv:1303.0857 http://arxiv.org/abs/1303.0857

[9] Bogdan Carbunar and Rahul Potharaju. 2015. A longitudinal study of the Google app market. In *Proc of IEEE/ACM ASONAM, 2015*. 242–249.

[10] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *Proc. of the IEEE SP, 2016*. 357–376.

[11] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proc. of ACM CCS, 2017*. 2187–2200.

[12] Dion Hinchcliffe. 2017. The advent of the citizen developer. Retrieved September 22, 2018 from https://www.zdnet.com/article/the-advent-of-the-citizen-developer/

[13] Facebook, Inc. [n. d.]. Android SDK - Facebook for Developers. Retrieved September 22, 2018 from https://developers.facebook.com/docs/android/

[14] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy amp;Paste on Android Application Security. In *Proc. of the IEEE SP, 2017*. 121–136.

[15] Fisher, Anne. 2017. How Companies Are Developing More Apps With Fewer Developers. Retrieved September 22, 2018 from http://fortune.com/2016/08/30/quickbase-coding-apps-developers/

[16] Forum of Incident Response and Security Teams. [n. d.]. Common Vulnerability Scoring System SIG. Retrieved September 22, 2018 from https://www.first.org/cvss/

[17] Forum of Incident Response and Security Teams. [n. d.]. Common Vulnerability Scoring System v3.0: Specification Document. Retrieved September 22, 2018 from https://www.first.org/cvss/specification-document

[18] The Apache Software Foundation. [n. d.]. Apache Commons Collections Security Vulnerabilities. Retrieved September 22, 2018 from https://commons.apache.org/proper/commons-collections/security-reports.html

[19] Yanick Fratantonio, Antonio Bianchi, William Robertson, Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2015. On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users. In *Proc. of DIMVA, 2015*. 282–303.

[20] Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. 2016. Measuring code reuse in Android apps. In *Proc. of PST, 2016*. 187–195.

[21] Google Inc. 2018. App Security Improvement Program. https://developer.android.com/google/play/asi.html

[22] Google, Inc. 2018. GitHub - google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back. Retrieved September 22, 2018 from https://sites.google.com/site/gson/

[23] Google Play API 2012. Google Play API. Retrieved September 22, 2018 from https://github.com/egirault/googleplay-api

[24] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Proc. of ACM WISEC, 2012 (WISEC '12)*. 101–112.

[25] Heqing Huang, Cong Zheng, Junyuan Zeng, Wu Zhou, Sencun Zhu, Peng Liu, Suresh Chari, and Ce Zhang. 2016. Android malware development on public malware scanning platforms: A large-scale data-driven study. In *Proc. of IEEE Big Data, 2016*. 1090–1099.

[26] Yuta Ishii, Takuya Watanabe, Mitsuaki Akiyama, and Tatsuya Mori. 2016. Clone or Relative?: Understanding the Origins of Similar Android Apps. In *Proc. of ACM IWSPA, 2016*. 25–32.

[27] Yuta Ishii, Takuya Watanabe, Fumihiro Kanei, Yuta Takata, Eitaro Shioji, Mitsuaki Akiyama, Takeshi Yagi, Bo Sun, and Tatsuya Mori. 2017. Understanding the security management of global third-party Android marketplaces. In *Proc. of ACM WAMA, 2017*. 12–18.

[28] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *Proc. of SANER, 2016*.

[29] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. Libd: Scalable and precise third-party library detection in Android markets. In *Proc. of ICSE, 2017*. 335–346.

[30] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proc. of IEEE/ACM ICSE, 2016*. 653–656.

[31] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proc. of IEEE ICSME, 2013*. 70–79.

[32] Stuart McIlroy, Nasir Ali, and Ahmed E Hassan. 2016. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering* 21, 3 (2016), 1346–1370.

[33] MITRE Corporation. 2018. CVE - Common Vulnerabilities and Exposures (CVE). Retrieved September 22, 2018 from https://cve.mitre.org/

[34] The Hacker News. 2014. Facebook SDK Vulnerability Puts Millions of Smart- phone Users' Accounts at Risk. https://thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html

[35] I. J. Mojica Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan. 2014. Impact of Ad Libraries on Ratings of Android Mobile Apps. *IEEE Software* 31, 6 (Nov 2014), 86–92.

[36] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, Thorsten Berger, Steffen Dienst, and Ahmed E Hassan. 2016. Analyzing ad library updates in android apps. *IEEE Software* 33, 2 (2016), 74–80.

[37] Vincent F. Taylor and Ivan Martinovic. 2017. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In *Proc. of ASIA CCS, 2017*. 45–57.

[38] Tom Preston-Werner. [n. d.]. Semantic Versioning 2.0.0. Retrieved September 22, 2018 from https://semver.org

[39] U.S. National Institute of Standards and Technology. 2017. CVE-2014-8889 Detail. Retrieved September 22, 2018 from https://nvd.nist.gov/vuln/detail/CVE-2014-8889

[40] U.S. National Institute of Standards and Technology. 2017. CVE-2016-2402 Detail. Retrieved September 22, 2018 from https://nvd.nist.gov/vuln/detail/CVE-2016-2402

[41] U.S. National Institute of Standards and Technology. 2018. National Vulnerability Database. Retrieved September 22, 2018 from https://nvd.nist.gov/

[42] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. In *Proc. of SIGMETRICS, 2014 (SIGMETRICS '14)*. 221–233.

[43] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishi, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. 2017. Understanding the Origins of Mobile App Vulnerabilities: A Large-scale Measurement Study of Free and Paid Apps. In *Proc. of MSR, 2017*. 14–24.

[44] Daoyuan Wu, Ximing Liu, Jiayun Xu, David Lo, and Debin Gao. 2017. Measuring the Declared SDK Versions and Their Consistency with API Calls in Android Apps. In *Proc. of WASA, 2017*. 678–690.

[45] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *Proc. of SANER, 2018*. 141–152.

[46] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. 2015. Harvesting Developer Credentials in Android Apps. In *Proc. of ACM WiSec, 2015*. Article 23, 23:1–23:12 pages.