Do Developers Update Third-Party Libraries in Mobile Apps?

Pasquale Salza
USI Università della Svizzera italiana
Switzerland
pasquale.salza@usi.ch

Fabio Palomba
University of Zurich
Switzerland
palomba@ifi.uzh.ch

Dario Di Nucci
Vrije Universiteit Brussel
Belgium
ddinucci@vub.ac.be

Cosmo D'Uva
University of Salerno
Italy

Andrea De Lucia
University of Salerno
Italy
adelucia@unisa.it

Filomena Ferrucci
University of Salerno
Italy
fferrucci@unisa.it

ABSTRACT

One of the most common strategies to develop new software is to take advantage of existing source code, which is available in comprehensive packages called third-party libraries. As for all software systems, even these libraries change to offer new functionalities and fix bugs or security issues. The way the changes are propagated has been studied by researchers, interested in understanding their impact on the non-functional attributes of the systems source code. While the research community mainly focused on the change propagation phenomenon in the context of traditional applications, only little is known regarding the mobile context. In this paper, we aim at bridging this gap by conducting an empirical study on the evolution history of 291 mobile apps, by investigating (i) whether mobile developers actually update third-party libraries, (ii) which are the categories of libraries with respect to the developers' proneness to update their apps, (iii) what are the common patterns followed by developers when updating a software library, and (iv) whether high- and low-rated apps present peculiar update patterns. The results of the study showed that mobile developers rarely update their apps with respect to the used libraries, and when they do, they mainly tend to update the libraries related to the Graphical User Interface, with the aim of keeping the mobile apps updated with the latest design tendencies. In some cases developers ignore updates because of a poor awareness of the benefits, or a too high cost/benefit ratio. Finally, high- and low-rated apps present strong differences.

CCS CONCEPTS

• **Software and its engineering → Software evolution**;

KEYWORDS

Third-Party Libraries, API Usage, Empirical Study, Mining Software Repository

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196341

ACM Reference Format:
Pasquale Salza, Fabio Palomba, Dario Di Nucci, Cosmo D'Uva, Andrea De Lucia, and Filomena Ferrucci. 2018. Do Developers Update Third-Party Libraries in Mobile Apps?. In *ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196341

1 INTRODUCTION

Software reuse is a common policy in modern development practices, because it avoids the costs related to the implementation of complex functions and modules as well as guarantees the usage of source code previously tested and validated [39]. Nowadays, even more companies develop software by means of *Application Programming Interfaces* (APIs), i.e., a set of subroutines and functionalities made available in the form of comprehensive packages, called *third-party libraries*, to allow other software systems to evolve by re-using such components. As an example, some of the largest software factories such as Google or Apple provide hundreds of APIs that allow software houses and newcomers to build upon these APIs their own software and re-distribute it into the market.

However, similarly to all the other software systems, even libraries need to change to be adapted to new market requirements and/or to be fixed with regard to bugs experienced by clients. Therefore, every *update* issued by the providers contains improvements aimed at making more stable and reliable the external APIs on which other systems are build. As soon as a new update is made available, the developer of an app that was using a previous version of the library may decide to upgrade it to inherit all the improvements implemented. Nevertheless, it can happen that a new version may require too much effort to be included in a project or simply be buggy, thus not being considered for an upgrade of its version. For the same reason, the developer may consider a downgrade to a previous version instead of an upgrade, in order to guarantee the stability of the app.

The way the changes made to libraries are propagated through the clients has been studied by researchers in the last years, which were interested in understanding the dynamics behind the update strategies as well as the effects of such changes to client systems. Most of them have devoted effort in studying the APIs usage in traditional applications from different perspectives, such as (i) the reasons pushing developers in using a specific version of an API [25], (ii) the mechanisms adopted to guarantee backward compatibility [4, 35], and (iii) the impact of API deprecations on the source code of client systems [6, 13, 36, 43].

Despite the important effort conducted by the research community in the context of traditional applications, only a little knowledge on how third-party libraries are treated in mobile apps is available so far. Indeed, while existing studies have been conducted to evaluate the diffuseness of third-party libraries in mobile apps [23, 30, 42], the impact of their non-functional attributes on the commercial success of mobile apps [5, 9], and their visualization [26, 27], up to date there is still lack of knowledge about the extent to which the phenomenon of *change propagation* is present in mobile applications. Investigating such a phenomenon is of a paramount importance since (i) the continuous release cycle [32], (ii) the vital importance that non-functional requirements have for mobile applications [10, 12], and (iii) the presence of active communities continuously requesting changes through user reviews [8, 34] make mobile apps deeply different from traditional applications, possibly leading developers to follow development practices which are unusual in other contexts [15]. As a consequence, understanding the mobile developers' behavior with respect to the management of libraries becomes a major challenge to face toward the definition of techniques and tools supporting them during their daily activities.

Thus, in this paper we aim at providing a large-scale empirical investigation on how mobile developers perform updates of used version of libraries in their code. Specifically, we mined the evolution history of the 291 Android apps in order to study the change propagation problem under four different perspectives:

(1) we studied *whether* mobile developers update the used version of external libraries;
(2) we performed a *fine-grained* investigation of the categories of libraries for which developers are more prone to update the used versions, shedding lights on the likely reasons pushing developers in having more care of them;
(3) we extracted the common patterns followed by mobile developers to update the use of libraries by means of an open coding procedure;
(4) we verified whether high- and low-rated apps present peculiar trends in the way developers update third-party libraries.

In this paper we refer to *version change* to indicate every type of change performed by developers of a mobile app in the usage of a third-party library, i.e., a version change can be an upgrade toward a newer version of a library or a downgrade toward a lower one. In the first place, the results of the study highlight that only 38 % of the external libraries in our dataset have been subject to at least one version change during the evolution history of the analyzed apps. Moreover, most of the updates are focused on third-party libraries related to the GUI of the app (≈50 %) or tools aimed at supporting development activities (27 %). By studying more in depth the likely causes behind the higher number of version changes for these categories, we discovered that developers aim at keeping the graphical user interface up to date with the latest tendencies, or updating Android support tools in order to develop for the latest Android versions. On the other hand, the results show that the main causes for the 62 % of libraries, whose version is not changed, are the carelessness of developers or a high cost/benefit ratio. Finally, we found that only 15 % of the library uses have been updated constantly during the evolution of the apps, and that most of them are related to successful apps.

Replication package. Besides the contributions reported above, we provide a comprehensive replication package containing the raw data and scripts used to carry out the empirical study [37].

Structure of the paper. The paper is organized as follows. Section 2 describes the design of the empirical study, while Section 3 reports and discusses the obtained results. Section 4 analyzes the threats that could affect the validity of the results of the study. Section 5 overviews the related literature on third-party libraries usage in traditional and mobile applications, and their effects, while Section 6 concludes the paper.

2 EMPIRICAL STUDY DESIGN

The *goal* of the empirical study is to analyze (i) whether mobile developers update the version of third-party libraries in their apps, (ii) which libraries developers are more and less prone to update, (iii) how developers react when new updates of libraries are available, and (iv) whether high- and low-rated apps present different trends with respect to the update of third-party libraries. The *quality focus* is improving software maintainability, by understanding how the phenomenon of libraries update is carried out. The *perspective* is of both researchers and practitioners: the former are interested in understanding the strategies adopted by mobile developers in the update of libraries, the latter in analyzing if successful apps are different from the others in terms of update patterns.

In Section 2.1 we present the research questions of the study. Section 2.2 describes the context of the study, in particular the Android app projects we selected and related libraries. The data mining process we devised is illustrated in Section 2.3 whereas Section 2.4 describes the applied analysis method.

2.1 Research Questions

In the context of the study we formulated the following research questions:

RQ$_1$ *To what extent mobile developers update the version of used third-party libraries?*
RQ$_2$ *Which types of third-party library uses are more prone to be updated?*
RQ$_3$ *Which types of third-party library uses are generally not updated?*
RQ$_4$ *What types of update patterns developers follow when updating the third-party libraries?*
RQ$_5$ *Are the update patterns of high-rated and low-rated apps different?*

The update of libraries is an essential part of software development, since it allows programmers to take advantage of the new functionalities as well as the improvements (e.g., bug fixing) over older versions of libraries. The first research question of the study aims at investigating *whether* mobile developers actually update libraries in mobile apps. Thus, **RQ$_1$** can be considered as a preliminary *coarse-grained* analysis that allows to quantify the developers' activities in updating the external libraries.

With **RQ$_2$** and **RQ$_3$** we further analyzed the phenomenon by conducting a *fine-grained* exploration into the types of libraries whose uses are more likely updated, aiming at understanding *what* categories (e.g., security) developers are more and less sensitive to, and *what* are the possible reasons behind their behavior.

RQ$_4$ aims at analyzing *how* developers update the used libraries. Specifically, we are interested in observing and possibly delineating a trend in the developers' reactions when an update of a library is available, with the goal of understanding whether they steadily update the libraries, or if the updates are rarely done. In the following, we will mention such trends as "update patterns".

Finally, **RQ$_5$** characterizes the update patterns of high- and low-rated mobile apps, with the aim of understanding whether the two sets of apps present different trends in the way developers updated third-party libraries.

2.2 Context Selection

The *context* of the study consists of the entire change history of 291 Android apps randomly selected from the F-Droid repository[1], which is a catalog of Free and Open Source Software (FOSS). The dataset size was identified using the Stats Engine tool[2], which allowed us to find an appropriate sample size to allow the generalizability of the results on the entire population, i.e., the whole set of apps composing F-Droid. More specifically, the dataset size represents a 95 % statistically significant stratified sample with a 5 % confidence interval of the 1181 apps currently available on F-Droid having more than 1 third-party library. It is worth noting that the selected apps belong to different categories and have different scope and size. A detailed report of the characteristics of the apps used in this study is available in the online appendix [37].

All the selected projects provide their source code in a public repository on GitHub and use Gradle as build system. As for the libraries, they are all publicly available on different repositories, e.g., Maven, JCenter, Bintray.

2.3 Data Mining Process

To answer to the identified research questions, we first needed to extract data from various sources. Therefore, we devised a data mining automated process, based on different components and operations. Our main aim was to collect the information regarding: (i) when the version of a library declared in a project changed, i.e., library update; (ii) when that library was upgraded by its developers, i.e., library release update. Collecting this data, we were able to compare the version change events with the releases of libraries, thus answering all the research questions.

Figure 1 shows the process we applied for each considered app:

1) F-Droid parsing: the F-Droid repository data, provided as a single XML file, was parsed in order to retrieve the public repository address of the source code.

2) Source code repository cloning: once established the public address of the repository, we performed a full repository Git cloning (i.e., project downloading, including all the commits) in our local storage. In the case of SVN repositories, we first performed an SVN to Git conversion by using the `git svn` bridging command.

3) Git commits extraction: we iterated through the list of commits by using the `git checkout` operation, and saved the files belonging to each single snapshot in separate directories. It allowed us to physically reproduce the status of the app source code during its entire development history.

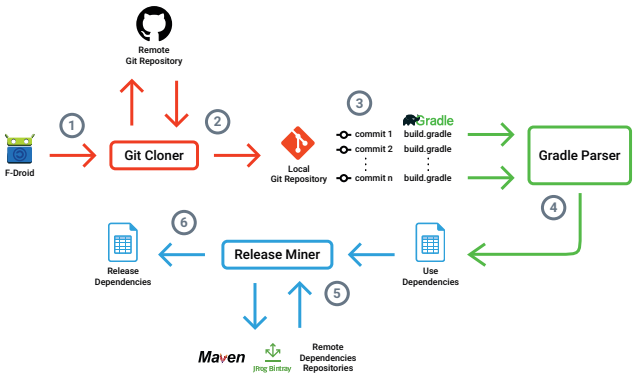

Figure 1: The data mining process used in the study.

4) Gradle libraries parsing: we explored the commit directories and parsed the `build.gradle` files to retrieve the declarations of third-party libraries dependency for the app. The libraries were reported with the general pattern: `<configuration> <group>:<name>:<version>`. It is worth noting that the Gradle definition language allows to express library version declarations in different syntax ways, thus we included other patterns in the parsing operation. In this way, we collected the employment of the libraries, reporting the versions during the history of each observed app project.

5) Dependencies mining: once we collected the list of libraries and the specific versions for each commit, we queried the most used repositories for Android libraries, e.g., Maven, JCenter, Bintray, looking for the release dates of those versions. We performed a "trial and error" process to find the repository having that piece of information. In some particular cases, the libraries were released as a GitHub open source project, thus we queried the releases list to retrieve the dates. For the Android SDK libraries, we directly queried the Google servers and retrieved the release dates as HTTP content publishing dates.

6) Data storing: As a final step, using a Python script we grouped all the information on the library version changes and releases for each app in the form of a CSV file containing the following four columns: (1) the "group", i.e., the suffix of the library name, to which a certain library belongs, (2) the "name", i.e., the actual name of the library, and (3) the "version", i.e., the label of the library used in a certain moment, (4) the "date", i.e., describing the date of the event, following the *ISO-8601* format.

The data extraction process took approximately 4 weeks, using 4 Linux workstations, each having 8 cores CPU and 8 GB of RAM. It is worth remembering that the CSV files obtained at the end of the mining process are publicly available in the online appendix [37].

2.4 Analysis Method

Once completed the data extraction process, we answered to **RQ$_1$**.

Firstly, we computed the number of times the version of libraries were changed, i.e., considering how many times the declaration

[1] https://f-droid.org
[2] https://www.optimizely.com/statistics/

of the library in the `build.gradle` file changed over time, with respect to the number of times a new version of the library was issued. In this way, we were able to understand whether the uses of such libraries are updated or not in the subject apps.

Secondly, we characterized whether the observed version changes referred to "upgrades", i.e., a version change made to catch the latest release update of a library, or "downgrades", i.e., version change to restore an old version of a library. To distinguish between the two categories of version changes, we mined the version history of the libraries available in the Maven repository, analyzing which of them were used by a certain mobile app during its history. Specifically, starting from the first commit on the repository until the end of the observed history, we iteratively considered commit pairs (C_i, C_{i+1}) and compared the `build.gradle` files in the two snapshots. For each library L_k used by an app, if the release version declared in the `build.gradle` of C_{i+1} was higher than the release version declared in the `build.gradle` of C_i (according to the version history on Maven), then we considered it as an upgrade of L_k. Otherwise, if the release version of L_k in C_{i+1} was lower than the release version of L_k in C_i, we counted a downgrade for L_k. In Section 3 we reported the boxplots describing the distribution of the number of upgrades and downgrades for the investigated apps.

As for $\mathbf{RQ_2}$ and $\mathbf{RQ_3}$, we categorized the libraries using the taxonomy provided by Maven. This was done automatically using the data collected when mining the dependency management system. Indeed, Maven provides a high-level categorization for each library studied, e.g., the `com.google.code.findbugs` library is included in the `Code Analyzer` category since it provides users with a static code analyzer able to discover possible bugs in the source code.

Once assigned the investigated libraries to the corresponding category, we repeated the same analyses made for $\mathbf{RQ_1}$, counting the number of updates for the libraries in a certain category and the number of upgrades and downgrades occurring in each category. Moreover, we complemented this analysis by means of qualitative examples aimed at understanding the likely reasons behind the higher/lower updates of libraries in given categories. To this aim, we manually analyzed commit messages and comments left on the repository by the developers of the apps presenting the higher and lower version change with respect to a certain category. It is important to note that this qualitative investigation had not the goal to systematically analyze and classify all the possible causes leading developers to update or not a library version, but instead that of finding likely reasons behind upgrades and downgrades occurring on specific categories.

To determine the *update patterns* and answer to $\mathbf{RQ_4}$, we adopted an open coding process and distributed between the participants the library version changes history of all the observed applications. We distributed a total of 1126 library histories. To this aim, we randomly divided the data among four of the authors (≈282 per author). Each of the involved authors independently analyzed the way mobile developers update the version of the libraries, by relying on a graphical representation of the evolution of a library version change in a given mobile app. Looking at the graphs, the involved authors independently classified an *update-pattern* using a label, e.g., "diligent update" when the version of a library was constantly changed during the evolution of a certain app. Figure 2 depicts one of the graphs analyzed during the open coding procedure, referring

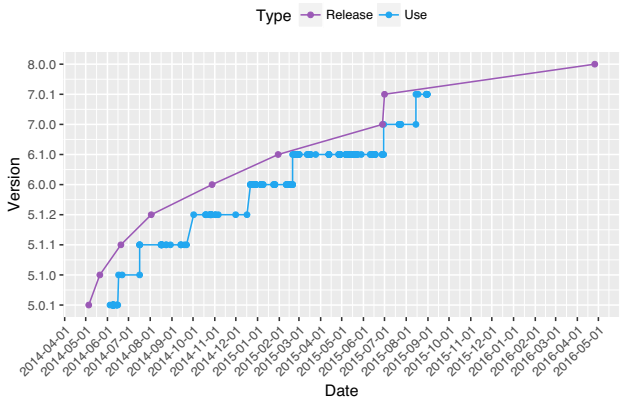

Figure 2: An example of *update* pattern, the `com.jakewharton:butterknife` library evolution for the `com.fastebro.androidrgbtool` app.

to the library `com.jakewharton:butterknife` used by the app AndroidRGBTool. The red line represents the evolution history of a library: the y-axis reports all the versions of the library, while the x-axis reports the time expressed in terms of months. The blue line represents instead the evolution history of the library usage in the context of the specific mobile app considered. Looking at the figure, it is clear that the AndroidRGBTool's developers constantly changed the version of the library used as soon as a new release was available. This was classified, therefore, as a "diligent update".

Once the first step of the open coding procedure was concluded, the authors discussed their codings in order to (i) double-check the consistency of their individual categorization, and (ii) refine the identified categories by merging similar categories they identified or splitting when it was the case. To evaluate the open coding process, we computed the level of agreement between the authors using the widely known Krippendorff's alpha Kr_α [17]. As a result, the agreement was equal to 0.87, thus being considerably higher than the 0.80 standard reference score [1] for Kr_α. In Section 3, we reported the percentage of times we classified specific *update patterns*, by also providing qualitative examples aimed at explaining the underlying reasons behind the observed behaviors.

Finally, to answer $\mathbf{RQ_5}$ we needed to extract the ratings associated to the user reviews of the considered apps. To this aim, we developed a web scraper that extracts the user reviews directly from the Google Play Store, where they are publicly available. Afterwards, we followed the heuristics defined by Khalid et al. [16] to discriminate high- and low-rated apps. In particular, apps whose average ratings were strictly higher than 3.5 were considered as *high-rated*, otherwise they were marked as *low-rated*. Once the two sets were formed, we verified the distribution of each update pattern discovered in the context of $\mathbf{RQ_4}$ in the two app types.

3 RESULTS

In this section we discuss the results achieved aiming at answering the formulated research questions.

Table 1: Distribution of Third-Party Libraries in our dataset.

Min.	1st Quart.	Median	Mean	3rd Quart.	Max.
1	1	2	3.8	5	35

3.1 RQ_1 – To what extent mobile developers update the version of used third-party libraries?

Before discussing the results for $\mathbf{RQ_1}$, it is worth observing the diffuseness of third-party libraries usage in our dataset. As it is possible to see from Table 1, all the apps rely on at least 1 external library: while this is quite expected (Android apps need to refer to the `android.core` library to be run), it is also important to observe that the average usage of libraries is almost 4, with apps even reaching 35. Over 42 % of the libraries refer to the categories *Graphical User Interface* (GUI) and *Utilities*. On the one hand, it means that developers often rely on libraries providing a set of tools for the implementation of GUIs, rather than implementing their own GUI. On the other hand, libraries providing support tools (e.g., network connection facilities) are appreciated by developers. The analysis of the diffuseness is needed to support our work. Indeed, a better exploration of the phenomenon may be useful for researchers and practitioners to focus their effort on devising specific techniques and tools to support the evolution of libraries.

Turning the attention to the first research question, we found that 33 % of the libraries are subject to at least one version change, while the version of the remaining ones has never been updated since its introduction. The result is symptomatic since it seems that Android programmers tend to not update the version of used external libraries, being more prone to inherit bugs or vulnerabilities present in older versions of the used libraries. Interestingly, we also observed that a very low percentage of commits (on average 2 %) involves the version change of a library. This somehow confirms that Android developers are poorly interested in updating the version of used libraries. It is worth noting that we are aware that a missing update might be due to the library being not updatable: a deeper investigation into this aspect is presented in $\mathbf{RQ_3}$.

Looking more in depth at the types of version changes performed by developers, Fig. 3 shows the box plots reporting the total number of (i) total version changes, (ii) changes toward newer versions (i.e., upgrades), and (iii) changes toward older versions (i.e., downgrades)

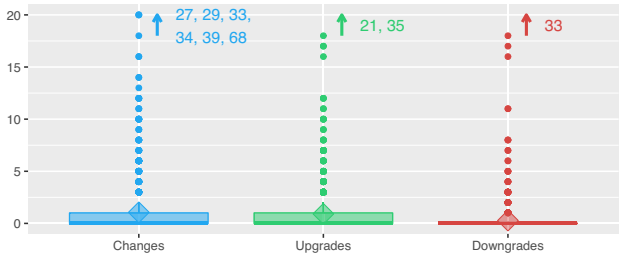

Figure 3: Third-party libraries changes, upgrades and downgrades over the 291 apps considered in our study.

performed on each subject apps over the considered timeline. The first thing that leaps to the eye is that the number of upgrades is higher than the downgrades. Indeed, 74.05 % of the version changes are represented by upgrades, while 25.95 % of them are downgrades. This result matches common expectations, since the upgrade of a used library version should represent the normal situation in which developers get the latest available version. Instead, more interesting is the analysis of downgrades. Even if both the median and the third quartile of the distribution are equal to 0, we observed 110 outliers (corresponding to the 10 % of the total version changes) that indicate how in some cases developers "refuse" a more recent version of a third-library. To better understand the reasons behind this anomalous phenomenon, we manually analyzed the apps having a higher number of downgrades. In 8 cases (1 % of the total, and 7 % of outliers) we found that where a consistent number of downgrades was performed, a high number of upgrades was applied as well. It is the case of the COM.OWNCLOUD.ANDROID app, which is a system that allows the management and sharing of synced files and folder across devices. The app depends on the `com.android.support:appcompat-v7` library, which is needed to implement a *Material Design* interface. In the period that we considered, the version of this library was upgraded and downgraded (from version `19.1.0` to version `22.2.1`, and viceversa) 33 times. This behavior was instigated by the fact that the upgrade broke the building process, as reported on the issue tracker[3]:

> *"Anyone, any idea why the build fails? The classes necessary for compile (even for the first commit!) need compile com.android.support:appcompat-v7 to resolve the imports which have been included in the gradle file... does maven need to be updated too?!"*

From this example, it seems that developers downgrade the version library only when a previous upgrade of that library caused issues not easily addressable for developers. We observed a similar behavior even when analyzing the other outliers, thus confirming the previous finding.

In summary

Developers rarely update third-party libraries in mobile apps, i.e., only 2 % of commits. When an update is performed, it is usually an upgrade toward a newer version. However, in some cases developers prefer to downgrade because of the issues caused by a previous upgrade.

3.2 RQ_2 – Which types of third-party library uses are more prone to be updated?

In our dataset, we found 28 categories for the external libraries version changes of the 291 Android apps analyzed. Table 2 shows the top 10 categories of libraries considered in the study whose version has been most changed by developers of mobile apps.

Looking at the table, it is evident how most of the libraries whose versions are more frequently changed by developers relate to the *GUI* category. While this result may be a natural consequence of the high diffuseness of GUI libraries, it is also worth noting that this

[3]https://github.com/owncloud/android/pull/1070

Table 2: The 10 categories of third-party libraries with more version changes.

Category	Changes	Upgrades	Downgrades
Graphical User Interface	808	607	201
Utilities	478	351	127
HTTP	60	38	22
Page Navigation	42	24	18
JSON	30	18	12
Annotations	23	23	0
Google Services	15	12	3
Date and Time	14	12	2
Device	14	9	5
HTML Parser	9	7	2

Table 3: The 10 categories of third-party libraries with less version changes.

Category	Changes	Upgrades	Downgrades
Defect Detection	1	1	0
Bug Fix	1	1	0
Network	1	1	0
Protocol Buffers	1	1	0
SQL	1	1	0
Test Automation	1	1	0
Notification	2	1	1
Barcode	2	2	0
Cryptographic	2	2	0
Event Bus	2	2	0

category is the one having the highest number of upgrades: thus, we can confirm the previous findings achieved by Hou et al. [14] on the importance of such libraries for developers. The frequent version changes of these libraries can be explained in two ways. On the one hand, most of the comments received by developers from the Google Play Store are related to the GUI of the application, as demonstrated in previous work [33, 34]. Thus, developers are more interested in updating the graphical user interface to fix issues experienced by users. On the other hand, the higher attention is motivated by the fact the developers want to keep the user interface up to date with the latest tendencies. The latter claim is supported by the manual analysis we made on the repositories of the subject apps. In particular, we found several cases where the commit message associated to the version change of a GUI library mentioned the willingness of developers to improve the layout of the GUI. An example is reported in the za.co.lukestonehm.logicaldefence app, where we discovered the following commit message:

> *"Update com.android.recyclerview–v7 to get new fancy icons."*

At the same time, libraries belonging to the category *Utilities* are often updated as well. This is because most of the tools provided by such libraries support developers during their activities, e.g., in the case of `com.android.support`, that provides APIs for uploading/downloading files from a remote server. The high number of upgrades (see Table 2) is motivated by the fact that developers are enforced to upgrade them as new Android versions are released. For instance, the `com.android.support` library mentioned above is constantly modified when new versions of the Android SDK are available: as a consequence, developers need to perform upgrades in order to use the new supports provided. As an example, we found that in the de.geeksfactory.opacclient app, a developer upgraded the version of `com.android.support` library, leaving the following commit message:

> *"Update android.support to have an environment equivalent to the android platform."*

All the other libraries whose version changes is high perform various tasks related to the development of mobile apps (e.g., *JSON*). However, the update of such libraries is not common as the one of the libraries in the *GUI* and *Utilities* categories. This result somehow confirms previous findings on the evolution of mobile apps. Specifically, Zhang et al. [44] demonstrated how during the evolution of mobile apps the first Lehman's law (i.e., continuous change [20]) holds for classes belonging to the GUI, while other pieces of code are changed only if strictly needed. The way developers update libraries seems to confirm this statement, since almost all the libraries that are not related to GUI are poorly maintained by developers.

In summary

Third-party libraries related to the graphical user interface or providing support tools for development are the ones having the highest number of version changes in the mobile apps using them. This is mainly due to the will of developers to (1) keep the GUI always up to date with the latest graphical tendencies, or (2) update Android support tools to support the latest Android versions.

3.3 RQ_3 – Which types of third-party library uses are generally not updated?

The version changes of almost 63 % of libraries was declared and never updated by the developers of the mobile apps in our dataset. In particular, despite 30 % of them was not updatable, as no newer version was available, another 33 % of them could be updated. This behavior somehow confirms the findings achieved in the first two research questions, since it shows once again that mobile developers are rarely interested in changing the versions of the used libraries.

Table 3 reports the 10 categories of libraries for which a version change was possible but whose version is less changed in the analyzed apps. As the reader can observe, most of the libraries in these categories perform specific additional tasks that go beyond the development of functional requirements of the app (e.g., *Defect detection* libraries help developers in findings compile-time errors). Although some of the libraries in these categories are quite diffused (e.g., *Notification*), their version is never changed. Looking more in depth at the likely causes for the missing updates, we discovered two main reasons. On the one hand, often developers simply do not care about the version change of libraries. On the other hand,

the excessive effort needed to update the source code after the introduction of a new version of a library is an important factor to take into account. In the first case, we observed a consisted number of cases where the developers discussed the possibility to update the source code on their communication channels, concluding the discussion with ignoring the update. For example, on October 2016 a new version of the `jsonrpc` library (*Protocol Buffer category*) was available for the ORG.XBMC.KORE app. The developers discussed of the version change on the issue tracker, mentioning potential security issues related to the use of HTTP GET requests. Such discussion was ended by one of the developers in the following way:

> *"My 2 cents. This is an extreme case, and it doesn't justify the upgrade of the library."*

After this comment, the issue was marked as "Closed". We found other similar examples in the other apps analyzed, and thus we can conclude that one of the reasons behind missing version changes is that developers consciously ignore them. This result allows us to claim that more empirical studies aimed at showing the impact of missing version change for the maintainability and security aspects of the source code might be useful for making developers aware of the negative consequences of ignoring the updates of libraries.

In the second case, by mining the software repositories and the mailing lists of the subject apps, we observed several cases where the developers refused an upgrade because they considered the cost/benefit ratio too high. For instance, it is worth mentioning the case of the UK.ORG.NGO.SQUEEZER app: here the version of the library `eventBus` is never updated. On February 2016 a new version of the library was available, and the two main developers of the app discussed, on the issue tracker of the application, about the possibility to update the library. The analysis of pros and cons of the update ended with a total agreement of the developer in not upgrading the used version of the library, since it would require the modification of several classes and methods of the app. In particular, they motivated their choice as follows:

> *"This would require more changes to the Squeezer code, so I don't recommend working from that."*

This example is quite representative and shed lights on an important research aspect worth of a deeper investigation in the future, i.e., making techniques and tools available for (1) automatically updating dependencies, or (2) effort-aware prioritization able to suggest developers a list of library version to update based on the total amount of code to be modified as a consequence of the update.

In summary

Most of the versions of the libraries are not updated: while 30 % of them cannot be updated because of the lack of new versions, almost 33 % of third-party libraries is never upgraded to newer versions. The two more likely reasons behind this behavior are (1) the carelessness of developers, and (2) the high cost/benefit ratio.

Table 4: Results of the open coding procedure.

Pattern	Number	Percentage (%)
Updated Once	97	9
Diligent	148	13
Jump Up	144	13
Jump Down	3	0
Back & forth	28	2
Not Updating	709	63

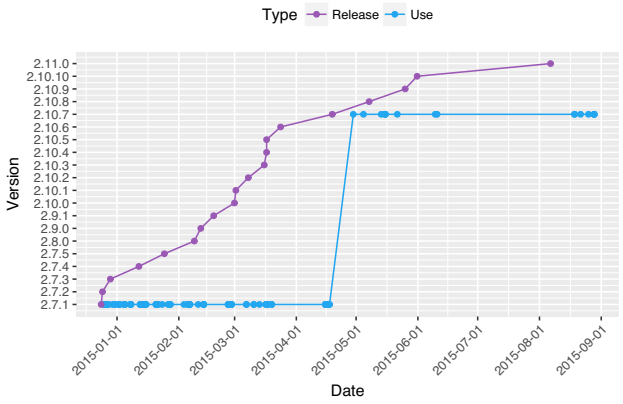

Figure 4: An example of *jump up* pattern, the `com.nispok:snackbar` library evolution for the COM.ANDROZIC app.

3.4 RQ_4 – What types of update patterns developers follow when updating the third-party libraries?

Even if most of the version changes of libraries are rarely or never updated, it is interesting to understand if there are common update-patterns occurring when developers decide to perform an upgrade of their external libraries. Table 4 reports the classification of the update-patterns obtained as a result of the open coding procedure. As it is possible to see, in 9 % of cases, the version of a library was changed in a given version of the app and then it disappeared in the immediately subsequent commit: we call this pattern as *updated once*. This category mainly refers to "abandoned" apps, i.e., apps that are not developed anymore: this happens for 95 out of the total 97 libraries in this category. A clear example is represented by the `ormlite` library of the GRACECODE.ANDROID.PRESENTATION app, an app to manage image galleries. The library version was changed in the commit performed on December 26th, 2014, which is exactly the last commit on the repository.

Much more relevant is the analysis of the pattern called *diligent*. It is the pattern containing the cases where developers constantly update the libraries, being always able to get the latest version of a library. We found a total of 148 *diligent* patterns, corresponding to the 13 % of the total libraries updated at least once. These results confirm the findings discussed above: only a limited percentage of developers are actually interested in updating the libraries.

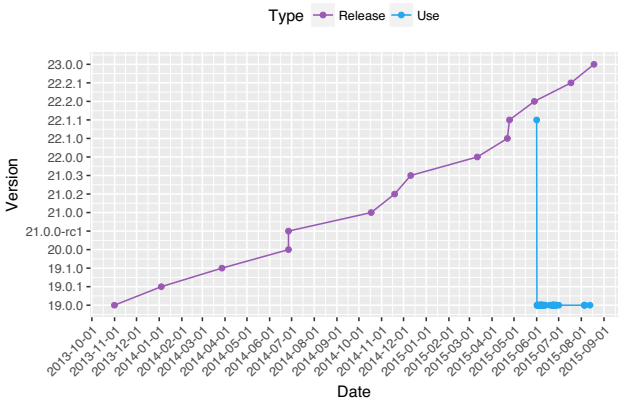

Figure 5: An example of *jump down* pattern, the `com.android.support:appcompat-v7` library evolution for the ORG.CIPHERDYNE.FWKNOP2 app.

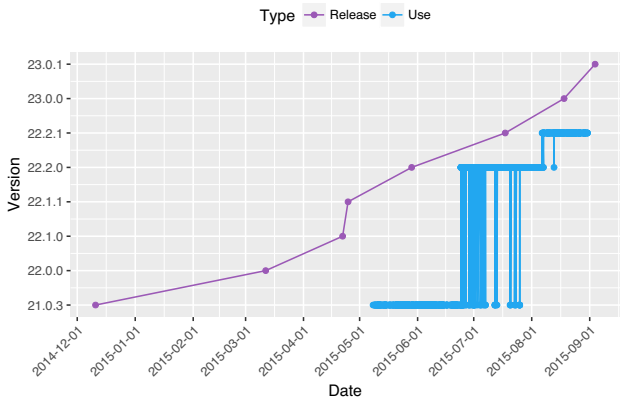

Figure 6: An example of *back & forth* pattern, the `com.android.support:cardview-v7` library evolution for the COM.DOLPHINEMU.DOLPHINEMU app.

The third update-pattern we classified is called *jump up*, and refers to cases where developers missed several version changes of a library before deciding to perform an upgrade toward a higher version of the library. Globally, we observed 144 cases, i.e., 13 % of the libraries that were updated at least once follow this update-pattern. An interesting case is the one of the COM.ANDROZIC app, where the library `com.nispok.snackbar` (whose evolution is depicted in Fig. 4) has been firstly introduced in the app on December 2014, when the version `2.7.1` was available. Then, during the evolution of the app, several newer versions of the library became available, however the developers did not change the version used until May 2015, when the current available version of the library was the `2.10.6`. Further analyzing this specific case, we found that the developers performed the upgrade only when the source code became not compatible anymore with the older version of the library. Indeed, the developer performing the commit left this message:

> *"Fix compatibility issue by updating the build.gradle file."*

Another pattern recognized is named *jump down* that represents the opposite of the *jump up* pattern described above. Indeed, it arises when developers decide to perform a downgrade toward a much lower version of the library. We identified this pattern in only 3 cases. One of this cases refer to the ORG.CIPHERDYNE.FWKNOP2 app where the library `com.android.support:appcompat-v7` was introduced on June 2015 (see Fig. 5). Immediately after the introduction of the version `22.1.1`, the library created compatibility issues that enforced developers in downgraded the library toward the `19.0.0` version. When committing the library downgrade, the developer left the following message:

> *"Downgraded dependency version due to compatibility issues with the fwknopd service package."*

Finally, the last pattern is called *back & forth*. It refers to the cases in which developers tried to upgrade the used version of a library several times, restoring each time an older version. We observed this pattern in 28 cases, i.e., 2 % of the library version changes followed this pattern.

A representative example is depicted in Fig. 6, reporting the case of the library `com.android.support:cardview-v7` of the app ORG.DOLPHINEMU.DOLPHINEMU, a Nintendo and GameCube simulator. As it is possible to see, between May and September 2015 the developers of the app continuously upgraded and downgraded the library. This was due to continuous issues that developers had with the visualization of the cards representing the characters of the simulated games. In particular, the developers experienced a different bug every time they tried to upgrade the library. One of the comments left on July 2015 perfectly explains the types of difficulties developers sometimes have to face:

> *"I'm getting crazy!!! I'm restoring the old version of that library hoping in good times!"*

This example clearly highlights how more research aimed at providing automatic tools for third-party libraries update is needed. Finally, we noticed that in the remaining cases library uses are not updated after their introduction. This occurs in 63 % of cases.

In summary

Developers follow peculiar update patterns when dealing with library updates. Only in 13 % of the cases the used external libraries are constantly updated by developers, while we found that in 63 % of the cases external library uses are never updated after their introduction.

3.5 RQ_5 – Are the update patterns of high-rated and low-rated apps different?

Figure 7 shows the distribution of each update pattern across the high- and low-rated apps of our dataset. As it is possible to observe, we recognized the prevalence of two specific update patterns, i.e., *diligent* (89 % vs 11 %) and *jump up* (78 % vs 22 %), in the high-rated apps. On the contrary, *back & forth* (79 % vs 21 %), *jump down* (66 % vs 34 %), and *updated once* (70 % vs 30 %) were more frequent in the low-rated apps. Interestingly, we observed that the *not updating*

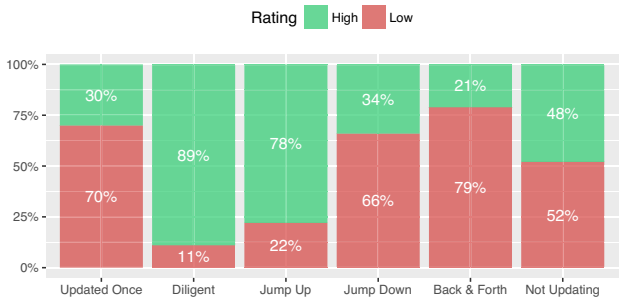

Figure 7: Distribution of each update pattern on high- and low-rated apps in our dataset.

pattern is almost equally distributed across the two sets, i.e., 52 % in case of low-rated apps and 48 % for the high-rated ones.

These results allow us to claim that successful apps are those whose developers update third-party libraries in a diligent way, thus properly following the releases of the library with the aim of including in the app the latest functionalities available in a library. Although several other factors might have influenced the results (e.g. app domain or popularity) and despite the fact that we cannot speculate on the reasons behind them, we believe that our findings provide initial hints of the importance that third-party libraries updates might have for the commercial success of mobile apps. At the same time, the results achieved when considering the low-rated apps seem suggesting that a poor management of libraries has a negative effect on the ratings provided by end users on the Google Play Store: this might be due to the inclusion of bugs and/or security issues inherited by the third-party libraries [5].

In summary

89 % of diligent update patterns belong to high-rated apps, while 70 % of the *updated once* patterns were found on low-rated apps. This seems suggesting a relationship between the update of third-party libraries and the ratings assigned by end users on the store.

4 THREATS TO VALIDITY

This section describes the threats that may have affected the validity of the study.

Construct Validity. Threats in this category are mainly related to the effectiveness of the tools built in order to mine data from the different software repositories analyzed. Before employing the tools, we carefully tested them against a sample set of mobile apps coming from the F-Droid repository. Moreover, we made all the tools publicly available for replication purposes [37].

Conclusion Validity. Threats to *conclusion validity* concern the relation between the treatment and the outcome. In the context of $\mathbf{RQ_4}$ we adopted an open coding procedure to identify the common update-patterns followed by mobile developers. This procedure involved the authors, who firstly independently classified a part of the libraries histories considered in this study, and then were involved in an open discussion with the aim of double-checking the previous classifications. Still, we cannot exclude imprecisions and/or some degree of subjectiveness (even if mitigated through the discussion).

Another threat in this category is represented by the presence of abandoned apps, which might have influenced the achieved results. For this reason, we repeated the analyses by excluding the abandoned projects, i.e. the ones having no commits during the last year. The results of this additional analysis are consistent with those reported in Section 3, thus confirming our findings.

Our findings might be also influenced by the replacement of libraries made by developers during the apps evolution. However, we plan to assess the effect of library replacements on the results as part of our future research agenda.

Finally, in $\mathbf{RQ_5}$ there might have been other factors related to the success of the apps presenting the update patterns investigated. We are aware of this: we plan to further investigate the causation of the relation between third-party updates and ratings as part of our future research agenda.

External Validity. Threats to *external validity* concern the generalization of results. We analyzed 291 Android apps from the F-Droid repository. Such a set represents a 95 % statistically significant stratified sample with a 5 % confidence interval of the 1181 apps, currently available on F-Droid, having more than 1 third-party library . Furthermore, it is worth noting that we analyzed 188 years of change history and 291 third-party libraries. Despite this, we are aware that we considered Android open-source apps only. Commercial apps, as well as the apps coming from other distribution platforms should be analyzed to corroborate our findings.

5 RELATED WORK

The phenomenon of third-party libraries version changes (i.e., *change propagation* or *ripple effect*) is a topic that has been studied in the context of both traditional applications [4, 11, 19, 25, 35, 36] and mobile apps [23, 24, 28, 30]. At the same time, the research community devoted effort in understanding the effects of updates on non-functional attributes of source code (e.g., fault-proneness [22].

5.1 Third-Party Libraries Usage in Mobile Apps

Mobile apps differ from traditionally studied applications [27, 40]. Thus, most of the previous empirical studies conducted on third-party libraries in traditional applications usage have been revised.

Linares-Vasquez et al. [23] decompiled and analyzed 24 379 APKs from the *Google Play Store*, discovering that in 82 % of the cases third-party libraries were used.

Ruiz et al. [30] studied code reuse in 4323 Android apps extracted from 5 categories of the Google Play Store, finding that 61 % of all classes in each category of mobile apps occur in 2 or more apps, and 217 mobile apps are reused completely by another mobile app in the same category. Their study was estended [28] by considering 208 601 apps, confirming the previous findings. Similar results were obtained by Minelli and Lanza [26, 27] and Viennot et al. [42].

Azad et al. [2] proposed a new tool able to analyze the APIs usage and suggest similar APIs based on Stack Overflow discussions.

Borges and Valente [7] applied association rule mining to learn an API usage model. To this aim, they extended APIMiner [31] to

collect usage patterns and APIs documentation and validated the obtained patterns.

Backes et al. [3] proposed a library detection technique that is resilient against common code obfuscation techniques and that is capable to identify the library version used in apps.

Our paper investigates aspects that were not considered in previous research, being therefore complementary.

5.2 Effects of Third-Party Libraries on Mobile Apps

Linares-Vasquez et al. [22] analyzed the effect of the change- and fault-proneness of Google APIs on the commercial success of mobile apps, discovering that apps having low ratings tend to use change- and fault-prone APIs. Such correlation has been confirmed by 45 Android developers [5]. According to these findings, Linares-Vasquez [21] proposed an API recommendation system able to avoid the introduction of defects.

Tian et al. [41] extracted APIs information and evaluated 1492 apps in terms of 28 factors along eight dimensions to understand how high-rated apps are different from low-rated apps. They found that size, number of images included in the web store page, and target SDK version are the most influential factors.

Third party libraries also impact the apps security. Dering and McDaniel [9] analyzed libraries and permissions of 450 000 free apps, finding a strong correlation between the number of external libraries used in the apps and the number of requested permissions.

Seneviratne et al. [38] analyzed the differences between free and paid apps. They discovered that both free and paid apps collect personal information. Moreover, the authors showed that 20 % of the apps were connected to more than three trackers, and that 50 % of users are exposed to 25 % trackers.

The analysis of the libraries history of the top apps on Google Play Store is part of the work by Backes et al. [3]. Their results showed that app developers slowly adapt new library versions, exposing their end-users to large windows of vulnerability. Finally, Mojica et al. [29] focused their attention on the impact of library version changes on development effort. The results showed that almost half of the apps underwent the ads library.

6 CONCLUSIONS AND FUTURE WORK

In this paper we conducted an empirical investigation on how mobile developers perform updates of third-party libraries in their code. We mined the evolution history of 291 open-source applications from the F-Droid repository to study the problem under three different perspectives. Firstly, we studied whether mobile developers perform external libraries version changes; secondly, we identified which categories of used libraries developers are more or less prone to update in their apps; thirdly, by means of an open coding procedure, we extracted the common patterns followed by mobile developers to update third-party libraries. Finally, we investigated the distribution of such update patters in high- and low-rated apps. The results indicate that:

(1) developers rarely update the used version of third-party libraries in mobile apps, i.e., only 2 % of commits are related to a version change;
(2) most version changes are usually an upgrade to a newer version, however if an upgrade introduces an issue, a downgrade is performed;
(3) the version of libraries related to graphical user interface or support tools are more likely to be updated;
(4) most of the dependencies are never updated because of developers carelessness and high cost/benefit ratio;
(5) only 13 % of library uses are constantly updated by developers, while in the 2 % of the cases developers try to update them without success;
(6) in 63 % of the cases the authors did not update the versions of used libraries after their introduction;
(7) 89 % of diligent update patterns are done on high-rated apps, while low-rated apps present 70 % of the *updated ones* patterns.

These results have a number of implications for the research community, tool vendors, and practitioners:

- **More empirical research is needed.** A key finding of our study is related to the low frequency of third-party library updates, likely dictated by developers carelessness. This recalls the need for empirical studies able to show the (negative) impact of missing updates on functional and non-functional properties of the source code, so that developers may acquire knowledge on the topic and be more aware of the possible consequences that the choice of non-updating libraries has. Similarly, further research is needed to investigate the causality of the relation between libraries updates and ratings assigned by end users.
- **Enabling automatic support.** One of the main challenges that both researchers and tool vendors should face is concerned with providing automatic support for third-party library updates. This includes the creation of auto-update systems or notification mechanisms allowing developers to know about the existence of a new version of a library.
- **Prioritizing update effort.** Our findings seem suggesting that a high cost/benefit ratio discourage developers in updating third-party libraries. Thus, devising methodologies and tools able to properly capture how complex an update will be might help developers in the decision making process, ranking the update opportunities accordingly.
- **Predicting trends and impact on source code.** We were able to discover specific trends in the way developers update third-party libraries. As each of them has its own peculiarities, researchers might exploit this information in order to create prediction models able to preventively alert developers of the potential impact of missing updates on non-functional attributes of source code.

These findings and implications represent the main input for our future research agenda, mainly focused on designing and developing new techniques and tools able to automatically identify chances of version change, and apply them flawlessly. Moreover, we plan to extend the empirical study to proprietary and larger applications, with a particular focus on the relationship between user ratings and third-party library updates. Finally, we plan to investigate the impact of the developers' behavior looking in particular at security vulnerabilities, as already done in the traditional context [18].

REFERENCES

[1] Jean-Yves Antoine, Jeanne Villaneau, and A. Lefeuvre. 2014. Weighted Krippendorff's Alpha Is a More Reliable Metrics for Multi-Coders Ordinal Annotations: Experimental Studies on Emotion, Opinion and Coreference Annotation. In *European Chapter of the Association for Computational Linguistics (EACL)*. 550–559.

[2] Shams Abubakar Azad. 2015. *Empirical Studies of Android API Usage: Suggesting Related API Calls and Detecting License Violations*. Ph.D. Dissertation. Concordia University.

[3] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *ACM Conference on Computer and Communications Security (CCS)*. 356–367.

[4] Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. 2012. A Structured Approach to Assess Third-Party Library Usage. In *IEEE International Conference on Software Maintenance (ICSM)*. 483–492.

[5] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, 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 (2015), 384–407.

[6] Sue Black. 2001. Computing Ripple Effect for Software Maintenance. *Journal of Software Maintenance* 13, 4 (Sept. 2001), 263–279.

[7] Hudson S. Borges and Marco Tulio Valente. 2015. Mining Usage Patterns for the Android API. *PeerJ Computer Science* 1 (2015), e12.

[8] Ning Chen, Jialiu Lin, Steven CH Hoi, Xiaokui Xiao, and Boshen Zhang. 2014. AR-Miner: Mining Informative Reviews for Developers from Mobile App Marketplace. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 767–778.

[9] Matthew L Dering and Patrick McDaniel. 2014. Android Market Reconstruction and Analysis. In *IEEE Military Communications Conference (MILCOM)*. 300–305.

[10] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. Software-Based Energy Profiling of Android Apps: Simple, Efficient and Reliable?. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 103–114.

[11] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve? A Story of Refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 2 (2006), 83–107.

[12] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. 2013. Why People Hate Your App: Making Sense of User Feedback in a Mobile App Store. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. 1276–1284.

[13] Frederick M. Haney. 1972. Module Connection Analysis: A Tool for Scheduling Software Debugging Activities. In *Fall Joint Computer Conference*. 173–179.

[14] Daqing Hou and Xiaojia Yao. 2011. Exploring the Intent Behind Api Evolution: A Case Study. In *Working Conference on Reverse Engineering (WCRE)*. 131–140.

[15] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real Challenges in Mobile App Development. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 15–24.

[16] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. 2015. What Do Mobile App Users Complain About? *IEEE Software* 32, 3 (2015), 70–77.

[17] Klaus Krippendorff. 2004. *Content Analysis: An Introduction to Its Methodology* (2 ed.). Sage Publications.

[18] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. Do Developers Update Their Library Dependencies? *Empirical Software Engineering* (2017), 1–34.

[19] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-Scale, AST-Based API-Usage Analysis of Open-Source Java Projects. In *ACM/SIGAPP Symposium on Applied Computing (SAC)*. 1317–1324.

[20] M. M. Lehman and L. A. Belady (Eds.). 1985. *Program Evolution: Processes of Software Change*. Academic Press Professional.

[21] Mario Linares-Vásquez. 2014. Supporting Evolution and Maintenance of Android Apps. In *Doctoral Symposium of IEEE/ACM International Conference on Software Engineering (ICSE)*. 714–717.

[22] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 477–487.

[23] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages. In *IEEE Working Conference on Mining Software Repositories (MSR)*. 242–251.

[24] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2017. A Survey of App Store Analysis for Software Engineering. *IEEE Transactions on Software Engineering* 43, 9 (2017), 817–847.

[25] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. 2009. Mining Trends of Library Usage. In *International Workshop on Principles of Software Evolution and Annual Workshop on Software Evolution (IWPSE/EVOL)*. 57–62.

[26] Roberto Minelli and Michele Lanza. 2013. SAMOA: A Visual Software Analytics Platform for Mobile Applications. In *IEEE International Conference on Software Maintenance (ICSM)*. 476–479.

[27] Roberto Minelli and Michele Lanza. 2013. Software Analytics for Mobile Applications: Insights & Lessons Learned. In *European Conference on Software Maintenance and Reengineering (CSMR)*. 144–153.

[28] Israel J. Mojica Ruiz, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E. Hassan. 2014. A Large-Scale Empirical Study on Software Reuse in Mobile Apps. *IEEE software* 31, 2 (2014), 78–86.

[29] 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.

[30] Israel J. Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E. Hassan. 2012. Understanding Reuse in the Android Market. In *IEEE International Conference on Program Comprehension (ICPC)*. 113–122.

[31] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente. 2013. Documenting APIs with Examples: Lessons Learned with the APIMiner Platform. In *Working Conference on Reverse Engineering (WCRE)*. 401–408.

[32] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. 2016. Release Practices for Mobile Apps - What Do Users and Developers Think?. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 552–562.

[33] D. Pagano and W. Maalej. 2013. User Feedback in the Appstore: An Empirical Study. In *IEEE International Requirements Engineering Conference (RE)*. 125–134.

[34] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. 2017. Recommending and Localizing Change Requests for Mobile Apps Based on User Reviews. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 106–117.

[35] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2012. Measuring Software Library Stability Through Historical Version Analysis. In *IEEE International Conference on Software Maintenance (ICSM)*. 378–387.

[36] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How Do Developers React to API Deprecation? The Case of a Smalltalk Ecosystem. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 56.

[37] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Cosmo D'Uva, Andrea De Lucia, and Filomena Ferrucci. 2018. Do Developers Update Third-Party Libraries in Mobile Apps? - Appendix. (2018). https://doi.org/10.6084/m9.figshare.6025040.

[38] Suranga Seneviratne, Harini Kolamunna, and Aruna Seneviratne. 2015. A Measurement Study of Tracking in Paid Mobile Applications. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*. 7.

[39] Ian Sommerville. 2006. *Software Engineering*. Addison-Wesley.

[40] Mark D Syer, Meiyappan Nagappan, Ahmed E. Hassan, and Bram Adams. 2013. Revisiting Prior Empirical Findings for Mobile Apps: An Empirical Case Study on the 15 Most Popular Open-Source Android Apps. In *Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*. 283–297.

[41] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E Hassan. 2015. What Are the Characteristics of High-Rated Apps? A Case Study on Free Android Applications. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 301–310.

[42] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. *ACM SIGMETRICS Performance Evaluation Review* 42 (2014), 221–233.

[43] S. S. Yau, J. S. Collofello, and T. M. MacGregor. 1993. Ripple Effect Analysis of Software Maintenance. In *Software Engineering Metrics I: Measures and Validations*, Martin Shepperd (Ed.). 71–82.

[44] Jack Zhang, Shikhar Sagar, and Emad Shihab. 2013. The Evolution of Mobile Apps: An Exploratory Study. In *International Workshop on Software Development Lifecycle for Mobile (DeMobile)*. 1–8.