Noname manuscript No.
(will be inserted by the editor)

Third-Party Libraries in Mobile Apps

When, How, and Why Developers Update Them

Pasquale Salza · Fabio Palomba · Dario Di Nucci · Andrea De Lucia · Filomena Ferrucci

Received: date / Accepted: date

Abstract When developing new software, third-party libraries are commonly used to reduce implementation efforts. However, even these libraries undergo evolution activities to offer new functionalities and fix bugs or security issues. The research community has mainly investigated third-party libraries in the context of desktop applications, while only little is known regarding the mobile context. In this paper, we bridge this gap by investigating when, how, and why mobile developers update third-party libraries. By mining 2752 mobile apps, we study (i) whether mobile developers update third-party libraries, (ii) how much such apps lag behind the latest version of their dependencies, (iii) which are the categories of libraries that are more prone to be updated, and (iv) what are the common patterns followed by developers when updating a library. Then, we perform a survey with 73 mobile developers that aims at shedding lights on the reasons why they update (or not) third-party libraries. We find that mobile developers rarely update libraries, and when they do, they mainly tend to update libraries related to the Graphical User Interface. Avoiding bug propagation and making the app compatible with new Android releases are the top reasons why developers update their libraries.

P. Salza
USI Università della Svizzera italiana, Switzerland
E-mail: pasquale.salza@usi.ch

F. Palomba
University of Zurich, Switzerland
E-mail: palomba@ifi.uzh.ch

D. Di Nucci
Vrije Universiteit Brussel, Belgium
E-mail: dario.di.nucci@vub.be

A. De Lucia, F. Ferrucci
University of Salerno, Italy
E-mail: adelucia@unisa.it, fferrucci@unisa.it

Keywords Third-Party Libraries · API Usage · Empirical Study · Mining Software Repository

1 Introduction

In modern development practices, a common practice to implement new software is to reuse existing code, as it avoids the costs related to the implementation of complex functions and modules, and it guarantees the usage of source code previously tested and validated [62].

Nowadays, a steadily higher number of companies develop software by means of *Application Programming Interface*s (sAPIs), 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. For instance, large companies like 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, as any other software system, even libraries need to change to be adapted to new market requirements and/or to be fixed with regard to defects experienced by clients. Therefore, an *update* issued by the providers may contain improvements that make 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 and adopt the implemented improvements. However, it can happen that a new version may require too much effort to be included in a project or simply be defective, 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 aspects related to how the changes made to libraries are propagated through the clients have been studied in the last years by the research community. Previous researchers focused on the understanding of the dynamics behind the update strategies and the effects of such changes to client systems. In particular, most of this work has investigated the APIs usage in desktop applications (a.k.a. "traditional" applications [28, 47], namely the software running on a single computer) with respect to (i) the reasons pushing developers in using a specific version of an API [40], (ii) the mechanisms adopted to guarantee backward compatibility [4, 56], and (iii) the impact of API deprecations on the source code of client systems [22, 68, 6, 57].

Despite the important research efforts conducted so far, we still identify a lack of knowledge on how third-party libraries are treated in mobile applications. Indeed, while some studies have been conducted to evaluate the diffuseness of third-party libraries in mobile apps [43, 67, 38], the impact of their non-functional attributes on the commercial success of mobile apps [13, 5], and their visualization [42, 41], to date there is still lack of knowledge about the

extent to which the phenomenon of *change propagation* is present in mobile applications and how developers deal with it.

There are a number of reasons that make the mobile context particularly interesting with respect to studying the developers' decisions on whether to update third-party libraries. First, both mobile applications and mobile third-party libraries are often subject to continuous evolutionary development and maintenance [43, 67, 38], with new releases being available on a much more frequent basis than desktop applications: this aspect can influence the behavior of developers, who might be more conservative when updating external libraries close to a new release. Secondly, mobile apps are often developed by young developers with little experience [24]: this naturally makes them different than the general population of desktop application developers and may influence the way they deal with third-party libraries. Thirdly, certain non-functional requirements (e.g. energy consumption [54]) are generally more pressing concerns for mobile developers with respect to the others [54, 16]: the usage and update of third-party libraries may have an impact on such non-functional requirements and, for this reason, developers might be more inclined to update them. Finally, the presence of active user communities that continuously report change requests through user reviews [9, 51] may lead developers to be more prone to adopt and update third-party libraries that speed up the process of addressing the users' requests. 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 developers during their daily activities.

In our previous work [58], we started addressing this challenge by conducting a large-scale empirical investigation on how mobile developers update the libraries used versions in their code[1]. Specifically, we mined the evolution history of 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, and verified whether high- and low-rated apps present peculiar update patterns.

The results of our previous study highlighted that only 38 % of the external libraries in our dataset were subject to at least one version change during the evolution history of the analyzed apps. Moreover, most of the updates were focused on third-party libraries related to the *Graphical User Interface* (GUI) of the app (≈50 %) or tools aimed at supporting development activities

[1] 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 downgrade toward a lower one.

(27 %). By analyzing 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. Furthermore, the results of our previous study showed 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 were related to successful apps.

This paper extends our previous analyses [58] by expanding the empirical knowledge on the practices performed by mobile developers with respect to the update of third-party libraries. Specifically:

1. we substantially *expand the dataset*, by conducting our analyses on an additional set of 2461 real-world mobile apps. Thus, overall, our empirical study features a total of 2752 applications;
2. we investigate the *technical lag* between library updates made available by providers and actual adoption by developers, with the aim of assessing the time usually required by mobile developers to change the version of a library being used;
3. we complement our software repository mining study with a *survey study* involving 73 experienced mobile developers that aims at shedding lights on the motivations behind the practices actually applied and confirm/reject the observations of the quantitative analyses. The second study has the objective of "triangulating" [18] the results of the software repository mining study with an analysis of the human perspective that can help explaining the reasons behind the results of the first study.

The specific goals of our study consist of characterizing (i) if and when mobile developers update third-party libraries, with the aim of deeper understanding the phenomenon of change propagation in the context of mobile applications; (ii) which categories of libraries developers are more prone to update, so that our results can be used to inform researchers and tool vendors on how to prioritize third-party library updates; (iii) which policies developers apply when new libraries updates are available, with the goal of informing researchers and tool vendors on the possible *moments* in which developers may be more interested in updating their libraries and would like to have automatic support; and (iv) what are the perceived (dis)advantages of updating third-party library, so that producers can understand which are the current limitations that should be avoided to make third-party libraries more used and useful for developers. Our analyses reveal that most of the mobile apps suffer from technical lag and it constantly increases over time. Moreover, we find that avoiding the propagation of bugs and making the app compatible with new Android releases are the main reasons why developers update their libraries.

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 [59].

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 and when mobile developers update the version of third-party libraries in their apps, (ii) which categories of libraries developers are more prone to update, (iii) which policies developers apply when new libraries updates are available, and (iv) what are the perceived (dis)advantages of updating third-party library. These objectives have the aim of understanding the phenomenon of change propagation in the context of mobile applications, inform researchers and tool vendors on how to priotitize third-party library updates as well as the moments in which developers may want to have more automated support, and which are the current limitations that should be avoided to make third-party libraries more used and useful for developers.

2.1 Research Questions

Our study revolves around three main research questions. The first aims at understanding *if* and *when* developers update third-party libraries of mobile apps. To better study this perspective, we designed two sub-research questions that capture it under different angles:

RQ_1 – *When do mobile developers update third-party libraries?*

$RQ_{1.1}$ *Do mobile developers update third-party libraries?*

$RQ_{1.2}$ *How does the technical lag of mobile apps vary over time?*

$RQ_{1.1}$ represents a preliminary analysis aimed at assessing the extent to which third-party libraries are updated by mobile developers. In **$RQ_{1.2}$**, instead, we performed an analysis on the technical lag observable in mobile apps, namely what is the general delay in the adoption of third-party libraries.

Once we had assessed when developers update the version of the used libraries, we focused on how they actually do it. Specifically, we considered three main aspects:

RQ_2 – *How do mobile developers update third-party libraries?*

$RQ_{2.1}$ *What types of third-party library uses are more prone to be updated?*

$RQ_{2.2}$ *What types of third-party library uses are generally not updated?*

$RQ_{2.3}$ *What types of update patterns developers follow when updating the third-party libraries?*

In **$RQ_{2.1}$** and **$RQ_{2.2}$** we conducted a fine-grained exploration into the types of libraries whose uses are more likely updated, aiming at understanding how developers work and at classifying what categories (e.g., security) developers are more and less sensitive to update, and what are the possible reasons behind their behavior. Moreover, in **$RQ_{2.3}$** we were 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 performed. In the context of the paper, we define and refer to these trends as "update patterns".

Afterwards, we moved toward the understanding of the developers' perspective, namely we aimed at collecting their opinions with respect to the adoption and update of third-party libraries in mobile applications:

RQ_3 – *Why do mobile developers update third-party libraries?*

$RQ_{3.1}$ *Do developers frequently make use of third-party libraries when developing mobile apps?*

$RQ_{3.2}$ *What is the mobile developer's perspective on third-party library updates?*

First, with **$RQ_{3.1}$** we investigated whether developers make use of third-party libraries and why, independently from the update activities. We entered more in detail with **$RQ_{3.2}$**, where we asked to describe the reasons why they perform update actions, i.e., upgrades or downgrades.

2.2 Context Selection

The *context* of the study consists of mobile apps, needed to address **RQ_1** and **RQ_2**, and developers of those apps, who were surveyed to answer **RQ_3**.

Mobile apps. Our study focused on a total of 2752 open-source mobile apps, which were collected as follows. First, we considered the entire change history of 291 Android apps from the F-Droid repository[2], which is a catalog of *Free and Open Source* mobile apps. Even if F-Droid contained a total 1181 at the time of the analysis, only 291 satisfied the requirements of our mining process for the input apps: (i) make use of third-party libraries, and (ii) have dependencies that are still available. More details on the app filtering procedure are given in

[2] https://f-droid.org

Section 2.3. However, dataset size respects the requirement for generalization that we computed using Stats Engine[3], a tool that takes into consideration categories and size of the apps and is able to find an appropriate sample size to allow the generalization of the results on the set of apps composing F-Droid. While this dataset contains a reasonable number of apps, it may contain toy apps and/or applications that are not available in the Google Play store [55, 17, 32]. For this reason, we decided to augment the initial dataset with additional real-world apps. To this aim, we exploited a publicly available dataset of mobile apps, named AndroidTimeMachine, previously built by Geiger et al. [17]; overall, it contains 8431 open-source Android apps whose information has been extracted by combining different data sources with the aim of building a dataset only composed of real apps. For each app, the dataset provides: (i) metadata of GitHub projects, (ii) full commit and code history, and (iii) metadata from the Google Play store. We selected 2461 of these 8431 apps because of the above mentioned requirements of our data mining process. 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 [59]. 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.

Recruitment of developers. We invited the original programmers of the 2752 opens source apps we considered in the mining study. We first extracted all the GitHub public repository addresses of the developers contributing to the considered apps. Then, we selected those contributors who participated in the development of a single app with at least 5 commits. While 5 may be a small number, we selected this threshold as done in previous works [52, 60] with the only aim of excluding occasional developers or newcomers. Following this process, we invited a total of 1622 original developers, receiving a reply from 73 of them: the response rate was therefore close to 4.5 %, which is similar to the one achieved by other survey studies [50, 53, 52, 66]. The questionnaire was created and distributed to participants using Google Forms[4]. It was first available from September 15th to November 1st, 2018; then, it was open from January 20th to March 20th, 2019. The link to the questionnaire and a short introduction were sent to every recruited developer via e-mail. We estimated a completion time of 15 min. The questionnaire included a pre-survey section in which we asked developers about their profession, experience, and number of mobile apps developed so far. From the reported answers, the participants in the study were mostly professional developers, 53 % specifically working in the mobile field, and 36 % as software developer in general. The others (11 %) declared to be non-professional developers. Most of the developers (46 %) have been developing mobile apps for more than 5 years. Only a few of them (9 %)

[3] https://www.optimizely.com/statistics/

[4] https://www.google.us/intl/en/forms/about/

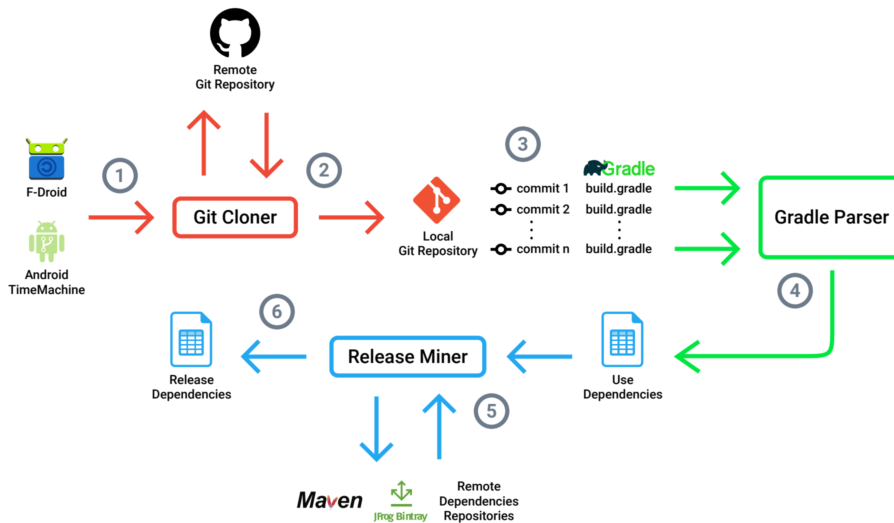

Figure 1 The data mining process used in the study.

had an experience of less than 1 year. Moreover, the number of apps developed is more than 5 for 63 % of participants.

2.3 Data Mining Process

To answer $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$, 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 the research questions. Figure 1 shows the process we applied for each considered app, consisting of the following points (indicated with the same number in the figure):

1) Dataset parsing: the F-Droid repository data, provided as a single *eXtensible Markup Language* (XML) file, was parsed in order to retrieve the public repository address of the source code. As for the apps coming from AndroidTimeMachine, they were available in a *Neo4j* graph-based database[5] which was queried to extract the public repository address of the apps.

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.

[5] https://neo4j.com

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.

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 each app. The libraries were reported with the general pattern: `<configuration> <group>:<name>:<version>`. It is worth noting that the Gradle definition language allows expressing library version declarations in different syntax ways, thus we included other patterns in the parsing operation. In particular, Gradle allows the user to indicate the version of the libraries also in a dynamic way. For instance, `1.0.+` would mean that the targeted version is anything available at compile time that respects the declaration, whose first part is explicitly specified (`1.0.`) and the last dynamically (+). Being this pattern often present within the analyzed dataset, we dealt with it by considering the nearest available version at the time of the commit, reproducing a reasonable compile scenario. In this way, we collected the employment of the libraries, reporting the versions during the history of each observed app project. In this step, we had to discard 3292 apps from the AndroidTimeMachine dataset because Gradle files were absent or were not syntactically valid for the majority of the commits.

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 list of releases to retrieve the dates. For the Android *Software Development Kit* (SDK) libraries, we directly queried the Google servers and retrieved the release dates as HTTP content publishing dates. However, this process did not worked out for 2678 apps of the AndroidTimeMachine dataset. Indeed, for these applications the declared third-party libraries were no longer available, thus not allowing us to process them.

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 *Comma-Separeted Value* (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.

As final output of the data mining process, we could correctly collect information for 2461 apps of AndroidTimeMachine and 291 of F-Droid, for a total of 2752 apps. The data extraction process took approximately 9 weeks, using 4 Linux workstations, each having 8 cores CPU and 8 GB of

RAM. The main reason behind the considerable amount time is that we had to process all the available apps, namely 1181 from F-Droid and 8431 from AndroidTimeMachine, for a total of 9612 apps. In particular, we cloned all the Git repositories and iterated over all their commits. Unfortunately, only during the last phases of the process we could exclude part of the apps, which did not respect the requirements, and save some time. It is worth remembering that the CSV files obtained at the end of the mining process are publicly available in the online appendix [59].

2.4 Methodology and Analysis Method

In the following we illustrate the methodologies and analysis methods we employed to adress the research questions.

*2.4.1 Technical Lag Methodology (**RQ₁**)*

Once completed the data extraction process, we started addressing $\mathbf{RQ_{1.1}}$. Firstly, we computed the number of times the version of libraries were changed, i.e., considering how many times the declaration 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 distribution of the number of upgrades and downgrades for the investigated apps.

As for $\mathbf{RQ_{1.2}}$, we computed the technical lag affecting the apps in our dataset. Broadly speaking, technical lag has been defined by Zerouali et al. [69, 70] as the time between the availability of a new version of a third-party library and the usage of such version within an application.

Specifically, let p be a dependency included in deps_c, i.e., the set composed of all the dependencies of a client (e.g., a mobile app), and t a point in time. Let $\text{available}(p, t)$ be the set of all releases of p that are available at time t and $\text{installable}(d, t)$ be the set of all available releases of the dependency d

that satisfy the dependency constraint. It is worth noting that in this study the dependency constraints are defined in the GRADLE file of each app. Given available(p,t) and installable(d,t), we define missed(d,t) as the set of the releases of p that could not be updated in the client because of the dependency constraint.

The technical lag $\Delta_t(d,t)$ for the dependency d at time t is equal to 0 if none of the releases of d was skipped (e.g., missed$(d,t) = \emptyset$). Otherwise, it is equal to the difference between the date of the first release that was skipped and t, i.e., $\Delta_t(d,t) = t - \min\{\text{date}_r \mid r \in \text{missed}(d,t)\}$, where r is a missed release and date_r its release date.

Finally, given a client c (e.g., a mobile app) and the set of its dependencies deps_c, its technical lag at time t is the maximal technical lag induced by all its dependencies, i.e., $\Delta_t(c,t) = \max\{\Delta_t(d,t) \mid d \in \text{deps}_c\}$.

It is important to point out that previous studies [12, 69] demonstrated that a high technical lag could lead to serious consequences such as backward incompatibilities and security vulnerabilities. Thus, studying the technical lag represents an opportunity to measure the extent to which mobile apps are prone to such undesired issues. From a more practical point of view, we computed the number of days between the first release of the library that could not be used because of the dependency constraint and t.

2.4.2 Apps History Analysis Methodology ($\boldsymbol{RQ_2}$)

As for $\mathbf{RQ_{2.1}}$ and $\mathbf{RQ_{2.2}}$, we manually categorized the libraries. We started from the taxonomy provided by MAVEN. Unfortunately, a category different from the ambiguous "Android Packages" was available only for ≈10 % of the libraries. Therefore, we manually classified the categories of libraries in our dataset. This process was done by three authors of the paper, who jointly analyzed each library and classify it based on its characteristics and features. Such a process followed three iterations in which the authors discussed and continuously improved the taxonomy until ending up with the final set.

Once assigned the investigated libraries to the corresponding category, we counted 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 $\mathbf{RQ_{2.3}}$, we adopted an "open coding" process [63], which is a set of activities used to discover ideas, concepts and theories through the manual analysis of contents. We distributed

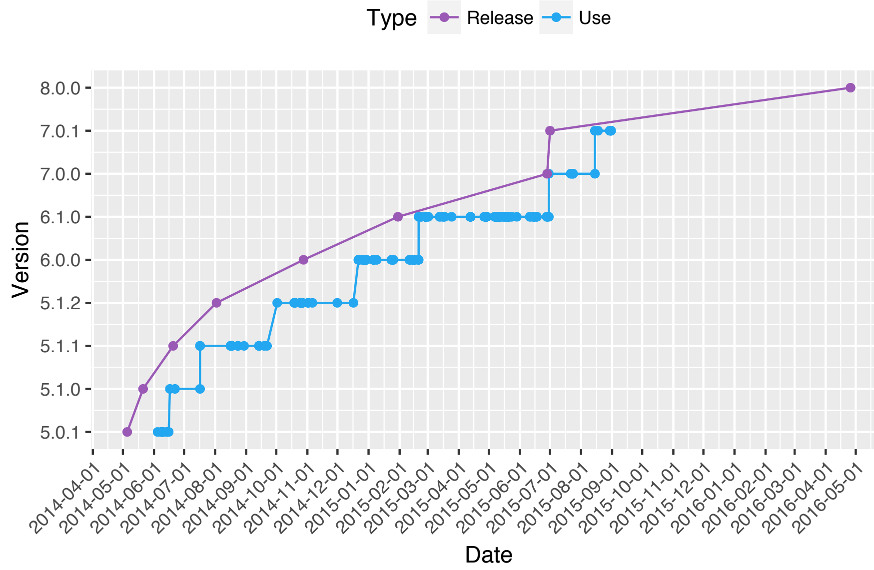

Figure 2 An example of *update* pattern, the `com.jakewharton:butterknife` library evolution for the COM.FASTEBRO.ANDROIDRGBTOOL app.

between the participants the library version changes history of all the observed applications. Starting from the total 11,626 library histories considered, four authors were assigned to the analysis of a sample of 3500 of them. This sample was composed of 2906 library histories that each author was required to analyze independently plus additional 594 library histories that were also evaluated by one of the other authors. Each author 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. The classification results for the 594 library histories coming from each author (2374 in total) allowed us to study the inter-rater agreement [21] among them with respect to the assigned update patterns. To this aim, we computed the Krippendorff's alpha agreement metric [30], which measured 0.87, that indicates a high agreement. 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 to the library `com.jakewharton:butterknife` used by the app ANDROIDRGBTOOL. The purple 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. Therefore, each purple point represents a new release of a certain library. The blue line represents instead the evolution history of the library usage for the specific mobile app.

In particular, each blue point depicts the version of the library that is used in a certain commit. A library is considered updated only if its version changes between two consecutive commits. 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 [29]. 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 the other cases, the four authors opened a discussion in order to reach an agreement. 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.

*2.4.3 Developers' Surveying Methodology (**RQ_3**)*

Table 1 lists the questions asked to the survey participants. Basically, we treated two themes, i.e., adoption of third-party libraries and pros/cons perceived by developers on their update. With the first set of questions (Q1.1–Q1.3) we collected opinions on the usage of third-party libraries and motivations behind their use. In the second part of the survey, we gathered deeper information on how libraries are updated and why (Q2.1-Q2.7). Moreover, we asked questions related to pros and cons of the update of such libraries and whether some specific factors, e.g., users' feedback or effort required to update a library, influence the decisional process (Q2.8-2.14). It is important to note that, to allow developers to express their opinions without forcing their answers in a restricted set, we preferred to keep most of the questions as open. Moreover, we avoided providing participants with a predefined set of answers to avoid *lazy responses* [19], that appears in case developers with no opinions on the topic still answer the questions, thus creating a form of bias in the interpretation of the results [48].

In Section 3 we report statistics of the distribution of the developers' answers over the categories present in the closed questions (i.e., where the possible answers are predefined), while we qualitatively analyzed the collected answers for open questions: to this aim, we first summarized in a short phrase the essential topic of each open answer; then, we identified explanatory codes to create emergent themes that we discussed among the authors [26].

Table 1 Survey questions on the developers' opinions on third-party library usage and updates.

ID	Question	Possible answers
	Use of third-party libraries	
Q1.1	Do you use libraries to develop mobile apps?	Never, rarely, sometimes, frequently, very frequently
Q1.2	How many libraries do you usually include for each of your apps?	1, 2, 3, 4, 5, >5
Q1.3	Why do you use libraries?	*Open question*
	Third-party libraries updates	
Q2.1	How frequently do you upgrade your libraries?	Never, rarely, sometimes, frequently, very frequently
Q2.2	Have you ever downgraded a library?	Never, rarely, sometimes, frequently, very frequently
Q2.3	If it happened, why?	*Open question*
Q2.4	Which types of libraries do you update the most?	*Open question*
Q2.5	Why do you update them?	*Open question*
Q2.6	Which types of libraries do you update the least?	*Open question*
Q2.7	Why do you not update them?	*Open question*
Q2.8	In your opinion, which are the main pros in updating a library?	*Open question*
Q2.9	In your opinion, which are the main cons in updating a library?	*Open question*
Q2.10	Do you test your app after upgrading a library?	Never, rarely, sometimes, often, always
Q2.11	Do you take into account users' feedback to keep libraries updated?	Never, rarely, sometimes, often, always
Q2.12	Do you take into account the effort required to update a library? How do you estimate the effort?	*Open question*
Q2.13	When you use or upgrade a library, do you take into account that a library could introduce security issues?	Never, rarely, sometimes, often, always
Q2.14	Have you ever performed a downgrade of a library due to a security issue?	Yes, no

Table 2 Distribution of third-party libraries in our dataset.

mean	min	Q1	Q2	Q3	max
4.63	1	2	3	6	44

3 Results

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

Table 3 Third-party libraries categories considered in out study.

Category	Libraries	Percentage (%)
Graphical User Interface	322	30.87
Networking	112	10.74
Frameworks	95	9.11
Parsers	64	6.14
Utilities	63	6.04
Cloud	43	4.12
I/O	36	3.45
Multimedia	26	2.49
Testing	26	2.49
Logging	26	2.49
Database	23	2.21
Security	23	2.20
Localization	21	2.01
Maintenance	19	1.82
Concurrency	19	1.82
Encryption	19	1.82
Rendering	15	1.44
Date	14	1.34
Information Retrieval	9	0.86
QR Code	8	0.77
Analytics	7	0.67
Crash Reporting	7	0.67
Payments	6	0.58
Compression	6	0.58
Sensors	6	0.58
Math	6	0.58
Advertising	5	0.48
Gaming	5	0.48
Plotting	5	0.48
Templating	2	0.19
Code Inspection	2	0.19
Generators	2	0.19
Parsing	1	0.10

3.1 $\mathbf{RQ_1}$ – When do developers update 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. Table 2 shows that 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 44. It is worth noting that we did not took into consideration inherited dependencies, because they are not explicitly in control of the developers. Table 3 shows the 33 categories of the 1043 libraries considered in our study. The table is sorted in descending order by the count of libraries. Over 30 % of the libraries refer to the categories GUI. It means that developers often rely on libraries providing a set of tools for the implementation of GUIs, rather than implementing their own GUI. The libraries that ease networking management and development ("frameworks") follow.

Table 4 Description of the changes per app.

	mean	min	Q1	Q2	Q3	max	total
Changes	3.49	0	0	0	2	351	9610
Upgrades	2.85	0	0	0	2	181	7832
Dowgrades	0.65	0	0	0	0	173	1178

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. The following subsections discuss the results for the sub-research questions formulated within $\mathbf{RQ_1}$.

3.1.1 $\mathbf{RQ_{1.1}}$ – To what extent mobile developers update the version of used third-party libraries?

We found that ≈30 % 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. From the results, 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. Moreover, starting from the change history information of each considered app, we also computed the number of commits involving a version change of a third-party library. We do this further step to complement our empirical overview of the extent to which libraries are updated by mobile developers. In this case, we observed that a very low percentage of commits (on average 1 %) involves the version change of a library. This somehow is consistent with the idea 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(e.g., because no newer versions are available): a deeper investigation into this aspect is presented in $\mathbf{RQ_2}$.

Looking more in depth at the types of version changes performed by developers, Table 4 and Figure 3 show: (i) the version changes, (ii) changes toward newer versions (i.e., upgrades), and (iii) changes toward older versions (i.e., downgrades) performed on each subject apps over the considered timeline. It is observable that the mean number of upgrades per app is significantly higher than the downgrades. Indeed, 81.50 % of the version changes are represented by upgrades, while 18.50 % 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.

Nevertheless, the number of downgrades is surprisingly high. To better understand the reasons behind this anomalous phenomenon, one of the authors of this paper manually analyzed the apps having a higher number of downgrades. More specifically, he analyzed the third-party library update history of all the apps having a number of downgrades higher than 15; this threshold was based on the analysis of the box plot shown in Figure 3: indeed, 15 is the number

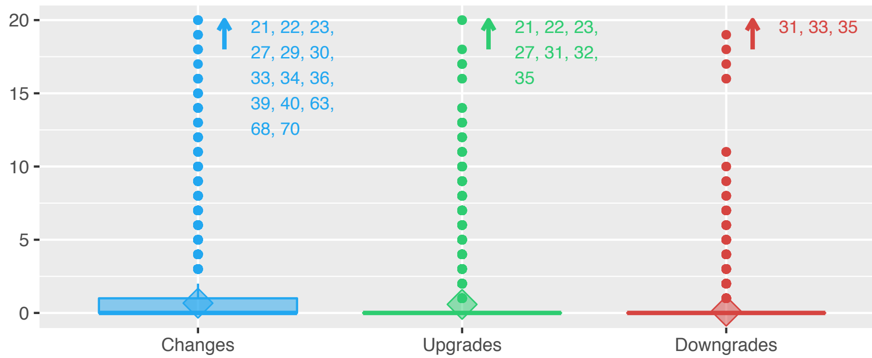

Figure 3 Boxplots of the third-party libraries changes, upgrades and downgrades per app. The arrows indicate higher outliers that we hid to enhance the comprehension of the boxplots.

of downgrades that discriminates the outliers of the distribution. This led to the analysis of 13 apps. As a result, we found that in all cases, 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 guaranteeing the compatibility with previous versions of Android. 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[6]:

> *"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. Nevertheless, a further analysis of the reasons sometimes leading developers to downgrade the version of their libraries is presented in the context of $\mathbf{RQ_3}$, where we directly inquired mobile developers on this aspect.

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

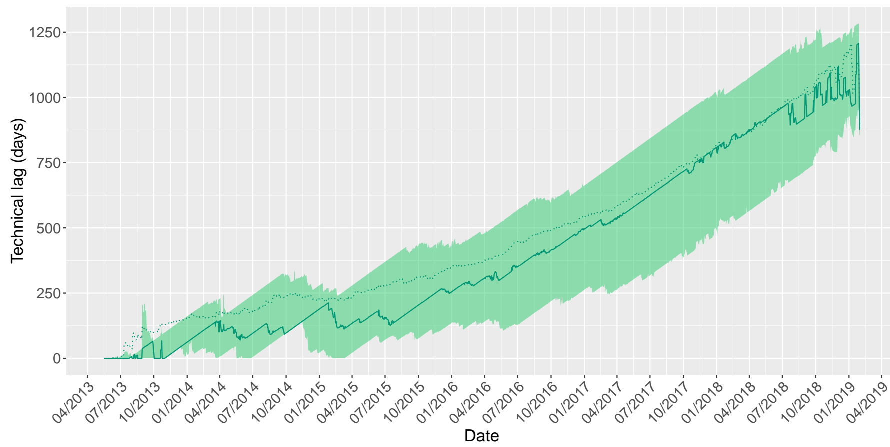

Figure 4 Summary of the technical lag accumulated by all the considered apps over time. The light-green area represents the first and third quartile of the technical lag distribution, the dotted line its mean, and the solid line its median.

In summary

Developers rarely update third-party libraries in mobile apps. When an update is performed, it is usually an upgrade toward a newer version. When a downgrade is performed, the reason seems to related to incompatibilities with the newer versions of the library.

3.1.2 $\boldsymbol{RQ_{1.2}}$ *– What is the technical lag of mobile applications?*

Figure 4 depicts the overall monthly distribution of technical lag for the mobile applications considered in the study. In particular, the y-axis reports the technical lag (expressed in days), while the x-axis reports the time expressed in quarters. The continue green line represents the median values, while the dashed green line reports the mean. The technical lag has been computed as a time difference by following the algorithm provided by Zerouali et al. [70]. In short, for each commit of an Android application, first we computed the technical lags for each employed library, and then we aggregated these values using the maximum operator.

A complete report of the technical lag of the single apps is available in our online appendix [59]. From the chart, we could observe a steady increase of the average technical lag during the considered period; this is confirmed when considering the median that follows a similar behaviour. These results clearly indicate a notable delay with which mobile developers update the version of the third-party dependencies that they declared in their applications. Furthermore, the technical debt evolution seems to be worse with respect to what Zerouali et al. [70] found in the npm package dependency network. Indeed, while they

found that the technical debt tends to be stable after few years, we discovered that the evolution is linearly increasing. It is important to note that, while we presented an overall view of the phenomenon, we did not analyze the technical lag for specific apps over time. In other words, we can claim that the phenomenon of technical lag is constant over our dataset and that developers update third-party libraries consistently late with respect to the introduction of a newer version of those libraries.

In summary

Mobile apps suffer from technical lag. On average, the technical lag is consistently increasing over time, thus suggesting that developers significantly delay the update of third-party libraries.

3.2 $\mathbf{RQ_2}$ – How do developers update third-party libraries?

The second research question aimed at analyzing, under different angles, how developers actually update third-party libraries in mobile applications. To ease the readability of the paper, we split the discussion of the sub-research questions in different subsections.

3.2.1 $\boldsymbol{RQ_{2.1}}$ *– What types of third-party library uses are more prone to be updated?*

In our dataset, we found 33 categories for the 1043 external libraries version changes of the 2752 Android apps analyzed. Table 5 shows the categories sorted by the number of changes in descending order.

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 category is the one having the highest number of upgrades. Thus, we can confirm the previous findings achieved by Hou et al. [23] on the importance of such libraries for developers, as they represent the way in which they can communicate with end-users. 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 [49, 51, 20]. Thus, developers may be more interested in updating the GUI 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 this case, one of the authors of this paper analyzed all the 5923 commit messages related to changes in the apps involving an upgrade of a GUI library: the author read each commit message in order to elicit the reasons behind the high number of GUI-related

Table 5 The 33 categories of third-party libraries sorted by the number of version changes.

Category	Changes	Upgrades	Downgrades
Graphical User Interface	7313	5923	1390
Frameworks	624	516	108
Networking	246	204	42
Cloud	196	142	54
Parsers	177	150	27
Code Inspection	176	152	24
Utilities	173	150	23
Multimedia	132	106	26
Localization	103	91	12
I/O	61	41	20
Logging	54	48	6
Advertising	45	35	10
Payments	42	40	2
Date	39	35	4
Analytics	36	31	5
Database	30	27	3
Concurrency	24	22	2
Security	22	22	0
QR Code	21	15	6
Testing	20	19	1
Encryption	15	11	4
Crash Reporting	14	14	0
Rendering	14	9	5
Plotting	10	7	3
Sensors	9	9	0
Maintenance	8	8	0
Gaming	5	4	1
Templating	1	1	0
Information Retrieval	0	0	0
Math	0	0	0
Generators	0	0	0
Compression	0	0	0
Parsing	0	0	0

upgrades. From this analysis, 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 *frameworks* 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 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 and network management. However, the update of such libraries is not common as the one of the libraries in the GUI and *frameworks* categories. This result is in line with previous findings on the evolution of mobile apps. Specifically, Zhang et al. [71] demonstrated how during the evolution of mobile apps the first Lehman's law (i.e., continuous change [35]) holds for classes belonging to the GUI, while other pieces of code are changed only if strictly needed.

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 may be mainly due to the will of developers to keep the GUI always up to date with the latest graphical tendencies, or update Android support tools to support the latest Android versions.

3.2.2 $\boldsymbol{RQ_{2.2}}$ *– What types of third-party library uses are generally not updated?*

The version of almost 66.24 % of libraries was declared and never updated by the developers of the mobile apps in our dataset. This behavior is in line with the findings achieved so far, since it shows once again that mobile developers are rarely interested in changing the versions of the used libraries.

Looking at the Table 5, it is also possible to notice the categories of libraries for which a version change was possible but whose version is less changed in the analyzed apps. Although some of the libraries in these categories are quite diffused (e.g., *Sensors*), their version is never changed. To elicit the likely causes behind missing updates, we performed an additional analysis. In the first place, we automatically mined developers' discussions done through issue trackers and mailing lists for each of the subject apps. Then, we filtered those discussions by applying regular expressions aimed at finding only the ones in which there was a specific mention to the non-updated libraries. Finally, two of the authors of this paper manually went over each discussion and analyzed it with the aim of extracting the reasons for which a certain library was not updated. This manual analysis was done jointly by the two authors, as in this way they could better discuss about the likely reasons for non-updates.

From this analysis, we were able to discover two main reasons. In the first case, we observed a 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 (*Networking*) 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. It is important to note that our results do not contrast the findings reported by Derr et al. [14], who found security vulnerabilities to be a key motivation for developers to update their third-party libraries: indeed, we complement those findings by reporting that, in some cases, developers consciously decide not to perform an update because it would solve a problem that does not frequently appear in the context of their app. This result also 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, 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."*

More in general, we discovered that the cost/benefit ratio leading to missing updates is generally due to two specific problems. First, as highlighted in the example above, the refactoring required to update a library implies the modification of a consistent part of the codebase: this because of cascading dependencies that need to be updated if a main one changes or because of a high usage of the library in the code, which makes the effort to modify the codebase excessive to be performed. However, it is also true the opposite: sometimes libraries affect a small portion of the code, and developers want to focus on

Table 6 Results of the open coding procedure, showing the number of libraries uses for each pattern.

Pattern	Number	Percentage (%)
Used once	1012	7.95
Diligent	1976	15.52
Jump up	1091	8.57
Jump down	39	0.30
Back & forth	181	1.42
Never changed	8435	66.24

libraries having a higher impact on the codebase. All in all, both motivations shed lights on an important research aspect worth of a deeper investigation in the future, i.e., making techniques and tools available for (i) automatically or semi-automatically [10] updating dependencies, or (ii) 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. The most likely reason behind this behavior seems to be the high cost/benefit ratio.

3.2.3 $\boldsymbol{RQ_{2.3}}$ *– 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 6 reports the classification of the update-patterns obtained as a result of the open coding procedure. As it is possible to see, in 7.95 % of cases, the version of a library was changed in a given version of the app and then the use of the library disappeared in the immediately subsequent commit: we call this pattern as *“used once”*. This category mainly refers to “abandoned” apps, i.e., apps that are not developed anymore. 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 26$^{\text{th}}$, 2014, which is exactly the last commit on the repository.

As for the pattern called *diligent*, which is the one 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 1976 uses that follow the *“diligent”* pattern, corresponding to the 15.52 % of the total libraries uses. These results seem to be in line with the findings discussed above on the limited willingness of developers to update libraries.

The third update-pattern we classified is called *“jump up”*, and refers to cases where developers missed several version changes of a library before

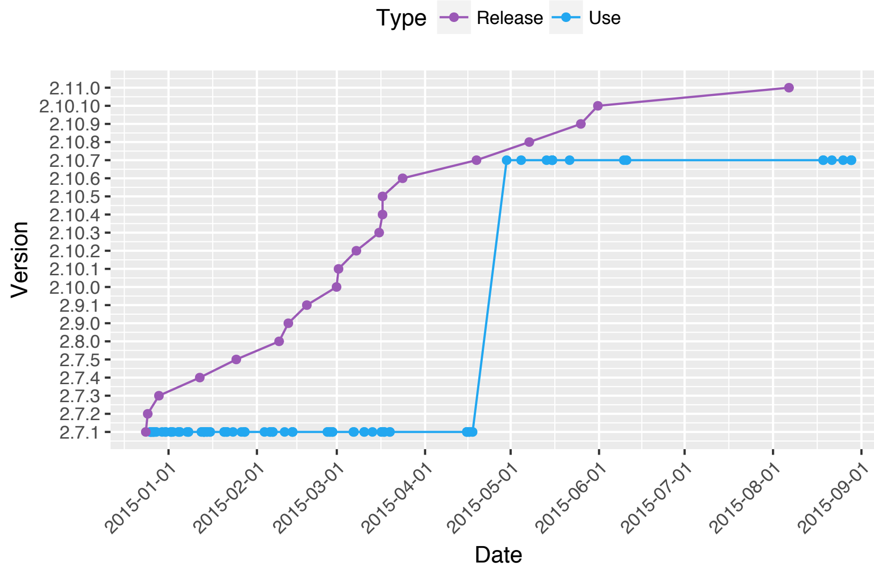

Figure 5 An example of *jump up* pattern, the `com.nispok:snackbar` library evolution for the COM.ANDROZIC app.

deciding to perform an upgrade toward a higher version of the library. Globally, we observed 1091 cases, which correspond to 8.57 % of the uses of libraries, 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 Figure 5) 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 39 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 Figure 6). Immediately after the introduction of the version `22.1.1`, the library created compatibility issues that enforced developers in downgrading the li-

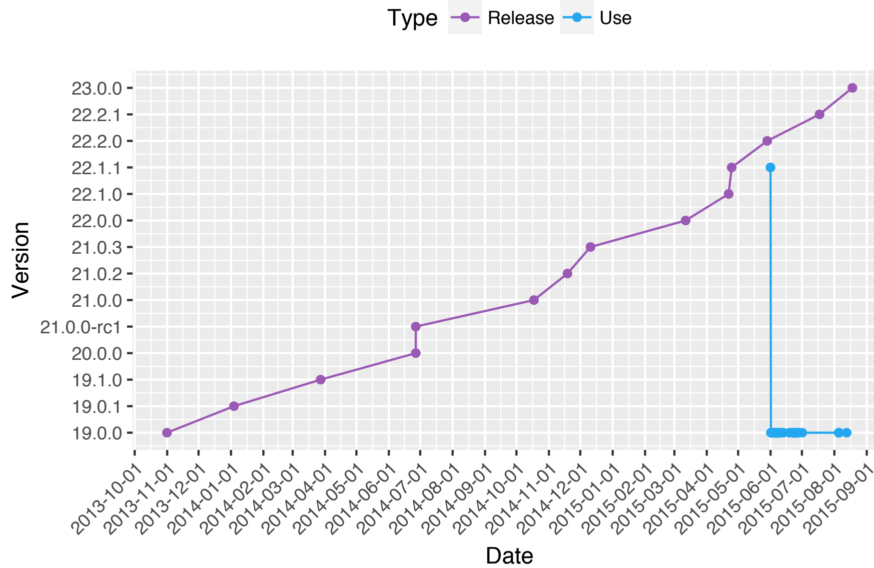

Figure 6 An example of *jump down* pattern, the `com.android.support:appcompat−v7` library evolution for the ORG.CIPHERDYNE.FWKNOP2 app.

brary 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 181 cases, i.e., 1.42 % of the library version changes followed this pattern. A representative example is depicted in Figure 7, reporting the case of the library `com.android.support:cardview-v7` of the app ORG.DOLPHINEMU.DOLPHINEMU, a NINTENDO 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!"*

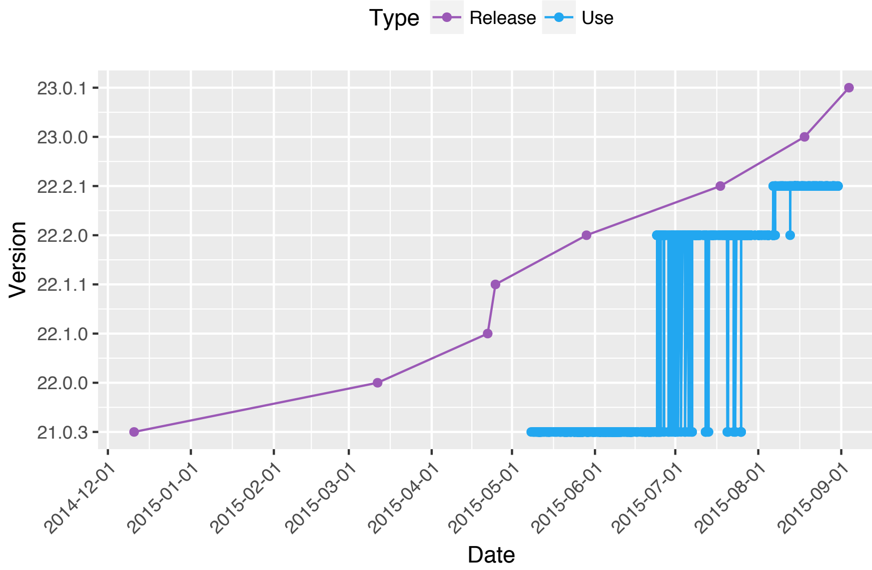

Figure 7 An example of *back & forth* pattern, the `com.android.support:cardview−v7` library evolution for the COM.DOLPHINEMU.DOLPHINEMU app.

This example clearly highlights how more research is needed to (i) provide automatic tools for third-party libraries updates, and (ii) assess the impact of an update on the source code of a mobile app.

Finally, we noticed that in the remaining cases library uses are *"never changed"* after their introduction; this occurs in 66.24 % of cases.

With the goal of further analyzing the update patterns and their potential impact, we conducted an additional experiment to understand if there are differences in the ratings of the different update patterns identified. To this aim, we extracted the ratings associated to the user reviews of the considered apps; 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. [25] 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 in the two app types.

Figure 8 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* (85 % vs. 15 %) and *jump up* (81 % vs. 19 %), in the high-rated apps. On the contrary, *back & forth* (23 % vs. 77 %), *jump down* (30 % vs. 70 %), and *used once* (27 % vs. 73 %) were more frequent in the low-rated apps. Interestingly, we observed that the

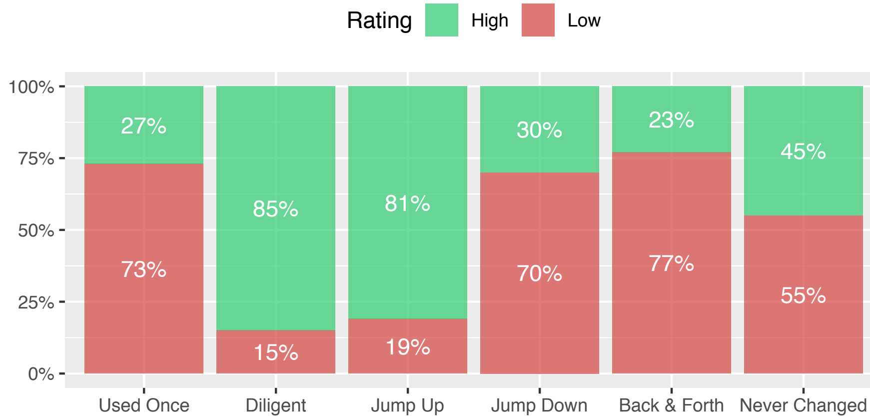

Figure 8 Distribution of each update pattern on high- and low-rated apps in our dataset.

never changed pattern is almost equally distributed across the two sets, i.e., 45 % in case of high-rated apps and 55 % for the low-rated ones.

These results suggest the existence of a relation between third-party library updates and app ratings. Although several other factors might have influenced the results (e.g., app domain, popularity, or source code quality [8, 52]) 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. Of course, our analysis is only preliminary, therefore this relation should be further investigated. At the same time, the results achieved when considering the low-rated apps suggest 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

Developers follow peculiar update patterns when dealing with library updates. Only in 15.52 % of the cases the used external libraries are constantly updated by developers, while we found that in 66.24 % of the cases external library uses are never changed after their introduction. Our findings suggest a relationship between the update of third-party libraries and the ratings assigned by end users on the store.

Table 7 Results for $\mathbf{RQ_{3.1}}$ – For Q1.1 the Likert scale ranges between “never” and “very frequently”; for Q1.2 between “1” and “>5”.

Question	Never	Rarely	Sometimes	Frequently	Very frequently	
	1	2	3	4	5	>5
Q1.1 – Use	1	3	4	44	21	–
Q1.2 – Quantity	5	6	11	22	11	18

3.3 $\mathbf{RQ_3}$ – What is the developers’ perception of third-party libraries updates?

In the previous research questions, we performed quantitative analyses by mining the software repositories of the considered mobile applications. At the same time, we observed the data to come up with some possible reasons behind the observed phenomena. Nevertheless, we could only provide hints on the reasons behind the developers’ decisions to upgrade/downgrade libraries as opposed to facts. For this reason, in $\mathbf{RQ_3}$ we perform a complementary study that aimed at surveying real Android developers on how they manage third-party libraries. The following subsections report the achieved results.

3.3.1 $\mathbf{RQ_{3.1}}$ – Do developers frequently make use of third-party libraries when developing mobile apps?

In the first part of the questionnaire, we aimed at understanding whether the developers are used to employ third-party libraries when developing mobile apps. Table 7 reports how many participants replied with values ranging between minimum and maximum of the Likert scale to the first two questions of the survey; it is worth remarking that in the case of Q1.1 the Likert scale was nominal and comprised between “never” and “very frequently”, while in Q1.2 between 1 and >5. As it is possible to observe, none of the participants excluded the use of libraries at all: 89 % of developers stated that they make use of libraries *frequently* or *very frequently*. 40 % declared to have included more than 5 libraries for each app, while only 5 of them (7 %) declared a poorly usage of external dependencies. These results clearly show that is very common for developers to rely on third-party libraries when developing mobile apps. Thus, we can confirm that the developers’ perception of third-library usage matches the quantitative findings revealed in the context of $\mathbf{RQ_{1.1}}$. Most of the developers commented that the main reason behind the use of third-party libraries is the code reuse, avoiding *reinventing the wheel* and save effort and time. Moreover, some of them recognize that libraries reflect the experience of their developers, resulting in components that are better maintained, tested, and designed.

Table 8 Results for $\mathbf{RQ_{3.2}}$.

Question	Never	Rarely	Sometimes	Frequently	Very frequently
Q2.1 – Upgrade frequency	0	5	26	20	22
Q2.2 – Downgrade frequency	17	31	20	5	0
Q2.10 – App testing	0	5	7	24	37
Q2.11 – Users' feedback	34	12	10	8	9
Q2.13 – Security issues	2	7	7	24	33

In summary

Developers make use of third-party libraries in mobile apps very often, mostly to save effort and time, trusting the experience of developers of these libraries.

3.3.2 $\boldsymbol{RQ_{3.2}}$ *– What is the rationale behind the decisions of mobile developers when updating the third-party libraries they use for their mobile apps?*

In the second part of the questionnaire, we specifically investigated the behavior of developers in updating third-party libraries. Table 8 reports the results achieved when inquiring developers. Also in this case, we report how many participants answered using each of the five-level Likert scale.

From the analysis of the results, it emerged that 30 % of developers upgrade libraries very frequently, another 36 % occasionally, and only 7 % rarely. Nobody excluded upgrading completely. We also asked about the opposite action, i.e., downgrading. The results show that the majority of them (66 %) *never* or *very rarely* performed an activity of downgrading the version of a third-party library, while 27 % of them stated that it happened at least once. The developers commented that it is usually due to incompatibility with the current version of the app, often making the app failing already during the build task. For those who tested the app after the upgrade, it happened that some found conflicts with other dependencies or unexpected bugs never experienced before.

Then, we asked developers to which types of library they are mostly attentive in updating. Thus, we were able to compare the direct developers' perception and experience with the ranks we automatically extracted in the context of $\mathbf{RQ_{2.1}}$ and $\mathbf{RQ_{2.2}}$ (see Section 3.2.1 and Section 3.2.2). The developers confirmed that GUI libraries are the most upgraded, followed by those that ease the interaction with external services, e.g., networking.

More interesting is the report about the reasons why they perform or not updates. On the one hand, they mostly keep libraries up-to-date to reduce propagation of bugs from dependencies, possibly improving app performance. Moreover, new releases of Android often cause incompatibility with previous implementations, affecting both app and dependencies source code. Sometimes, it means that it is not possible to make an app compatible with the most recent version of Android if all the used libraries are not compatible. One of the participants declared:

"Usually to stop Android Studio from nagging ;). Mostly because Google advises to upgrade the support libs whenever you update the target SDK."

S/he refers to the fact that ANDROID STUDIO, as the official *Integrated Development Environment* (IDE) for Android development, warns the developers on whether new versions of libraries used in a project are available. Furthermore, it is a best practice advised by Google to keep official support libs whenever there is an upgrade for the Android SDK.

On the other hand, they avoid updates mostly because they are afraid of breaking the current version of the app, as declared by one of the participants:

"If things are working, don't break them!"

Most of the participants pointed out that often an upgrade does not correspond to a simple rebuild of the app, according to their experience. Moreover, they tend to avoid change things if themselves, or their users, did not experience issues. When asking whether they include a testing activity every time they introduce a new version of a third-party library, 51 % of them reported how they *always* test the app after that, possibly highlighting that testing API modifications would represent a research direction to further investigate. Furthermore, 47 % of the respondents claimed that receiving requests from users complaining about issues clearly attributable to the use of certain libraries is not common, therefore, the inner-working of third-party libraries can somehow be hidden to the final users.

We also asked if they generally take into consideration the effort required to update a library and how they estimate it. One of them commented:

"Usually if you update them regularly (e.g. weekly) it's minimal effort, like less than 30 minutes per week (or even much less, like 5-10 minutes)."

The reported consideration is common also for other developers:

"Yes. If the update is a major version it usually means it's going to be more difficult to upgrade."

These considerations seems to go in the direction of what we qualitatively discovered in $\mathbf{RQ_{2.2}}$, i.e., if a developer does not update third-party libraries frequently, the effort required to perform such an update might be not worth the gain it provides. This result confirm the results previously obtained by Kula et al. [33] who showed that the "migration effort" is one of the main reasons that prevent library updates. One of the developers also mentioned the "technical debt" [31]:

"It's seen as good 'health' to do it regularly, to reduce technical debt"

In other words, it is a common feeling that having an app with up-to-date libraries reduce accumulating future effort, which may be traduced in some form of technical debt.

Finally, we investigated about the introduction of possible security issues when updating: 45 % of the participants declared that they *always* take into

consideration the possibility of such an introduction. Interestingly, only 29 % even performed a downgrade due to a security issue.

In summary

Developers upgrade third-party libraries in mobile apps frequently, but the opposite action of downgrading is less common. GUI libraries are the most upgraded third-party components. The main reasons for keeping the libraries up-to-date are (i) avoiding the propagation of bugs, (ii) making the app compatible with new Android releases.

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 [59].

Conclusion Validity. Threats to *conclusion validity* concern the relation between the treatment and the outcome. In $\mathbf{RQ_{1.2}}$, we computed the technical lag on the basis of the definition previously provided by Zerouali et al. [70]. Nevertheless, we cannot exclude the presence of other alternative more effective methods to define such a technical lag. Still, in this context, we estimated the technical debt using days as time unit: while it may be possible that our findings might change based on the amount of activities performed by developers (e.g., commit), it is important to note that the selected apps are active and perform continuous activities of software maintenance and evolution.

In the context of $\mathbf{RQ_{2.3}}$ 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 imprecision and/or some degree of subjectivity, 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. Detecting abandoned apps represents a difficult problem, especially because it is hard to distinguish these apps from those that have completed their features and do not require further maintenance [27]; more importantly, while some identification heuristics have been proposed so far, none of them are fully tested and reliable [27, 11]. This is the reason why we did not consider this aspect in our study. However, the large scale nature of our empirical study substantially increased the ecological

validity of our results. Moreover, our dataset is mostly composed of real-world apps that are active on the GOOGLE PLAY store [17, 55]. As a consequence, we reduced the likelihood to consider abandoned apps. In other words, while we cannot exclude the presence of abandoned apps in our dataset, their influence on the results are limited by the size of the empirical study.

In $\mathbf{RQ_{2.3}}$ there might have been other factors related to the success of the apps presenting the update patterns investigated as well as other factors influencing the trends followed to update libraries, e.g. size or activity of the considered projects. However, our large-scale analysis mitigates interpretation bias, as it enables a good ecological validity of the results.

External Validity. Threats to *external validity* concern the generalization of results. Part of the 2752 apps that compose our framework, is coming from the F-DROID repository. The set of 291 apps represents a 95 % statistically significant stratified sample with a 5 % confidence interval of the 1181 apps, available at the time of mining on F-DROID, having more than 1 third-party library. 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. Finally, in our survey study ($\mathbf{RQ_{3.1}}$ and $\mathbf{RQ_{3.2}}$), we collected opinions from 73 mobile developers of the considered apps. While this number cannot ensure the generalization of our findings, we still believe that the considered answers provide a valuable source to understand what developers think about third-party libraries. However, also in this case, further replications would be desirable.

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 desktop applications [15, 40, 34, 57, 56, 4] and mobile apps [38, 43, 44, 39]. 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 [37].

5.1 Third-Party Libraries Usage in Mobile Apps

Mobile apps differ from traditionally studied applications [42, 64]. Thus, most of the previous empirical studies conducted on third-party libraries in desktop applications usage have been revised.

Linares-Vasquez et al. [38] decompiled and analyzed 24,379 *Android Application Packages* (sAPKs) from the *Google Play Store*, discovering that in 82 % of the cases third-party libraries were used. Ruiz et al. [43] 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 [44] by considering 208,601 apps, confirming the previous findings. Similar results were obtained by Minelli and Lanza [42, 41] and Viennot et al. [67]. Our study builds on the line of research investigated in the aforementioned studies and extends the empirical knowledge of the research community on how and why mobile developers update third-party libraries: besides assessing the extent of their usage, we also conduct further experiments to understand what are the typical update patters used by developers and the reasons behind the decisions of updating (or not) a third-party library.

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 [46] 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. While this set of papers proposed techniques to support developers when dealing with third-party library, our study presents empirical results that can be exploited by such techniques to provide developers with improved recommendations: for instance, the findings on the libraries that are more/less prone to be updated can lead to the definition of novel prioritization techniques that recommend APIs usage and update.

5.2 Effects of Third-Party Libraries on Mobile Apps

Linares-Vasquez et al. [37] 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 [36] proposed an API recommendation system able to avoid the introduction of defects. Tian et al. [65] 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 [13] 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. Derr et al. [14] performed an empirical study on third-party library updatability over 1,264,118 Android applications: the main result of their study highlighted that (i) most of the libraries can be upgraded without modifying the source code, and (ii) almost 98 % of actively used library versions affected by a security vulnerability can be fixed with a library update. Our study enhances the state of the art in this direction by providing a preliminary evidence of the impact of third-party

library updates on the commercial success of mobile applications. As such, our findings are complementary with respect to those reported so far.

Seneviratne et al. [61] 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. [45] 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. Also in this case, our results confirm some of the observations reported in previous studies. For example, we have observed that one reason leading developers not to update libraries is excessive refactoring effort, as shown by Mojica et al. [45]; similarly, our findings on technical lag are perfectly in line with those reported by Backes et al. [3].

6 Conclusions and Future Work

In this paper we reported on an empirical investigation on when, how, and why mobile developers update third-party libraries in their code. We mined the evolution history of 2752 open-source applications to study the problem. Firstly, we studied whether mobile developers perform external libraries version changes as well as what is the technical lag occurring at app- and dependency-level. Secondly, we identified which categories of used libraries developers are more or less prone to update in their apps; we also extracted the common patterns followed by mobile developers to update third-party libraries and investigated the distribution of such update patters in high- and low-rated apps. Finally, we surveyed 73 mobile developers in order to collect their opinions on third-party libraries updates, and particularly on the motivations behind their decisions to update or not. The results indicate that:

1. developers rarely update the used version of third-party libraries in mobile apps, i.e., only 1 % of commits are related to a version change;
2. most of the apps have a high technical lag and its evolution is linearly increasing over time;
3. most version changes are usually an upgrade to a newer version, however if an upgrade introduces an issue, a downgrade is performed;
4. the version of libraries related to graphical user interface or support tools are more likely to be updated;
5. only 15.52 % of library uses are constantly updated by developers;
6. in 66.24 % of the cases the authors do not update the versions of used libraries after their introduction;
7. 85 % of diligent update patterns are done on high-rated apps, while low-rated apps present 73 % of the *used once* patterns;

8. according to the surveyed developers, most of the updates are done with the aim of avoiding bug propagation or making an app compatible with the Android releases;
9. our participants explained that some libraries are never updated because of high cost/benefit ratio or to not break existing code.

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, sometimes dictated by the willingness of developers to not break existing functionalities. 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 suggest 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 opportunities 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 [33].

Acknowledgment

The authors would like to thank the Associate Editor and anonymous reviewers for the constructive feedback that has been instrumental to improve the quality of our work. Fabio Palomba gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

References

1. Antoine JY, Villaneau J, Lefeuvre A (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), pp 550–559
2. Azad SA (2015) Empirical Studies of Android API Usage: Suggesting Related API Calls and Detecting License Violations. PhD thesis, Concordia University
3. Backes M, Bugiel S, Derr E (2016) Reliable Third-Party Library Detection in Android and its Security Applications. In: ACM Conference on Computer and Communications Security (CCS), pp 356–367
4. Bauer V, Heinemann L, Deissenboeck F (2012) A Structured Approach to Assess Third-Party Library Usage. In: IEEE International Conference on Software Maintenance (ICSM), pp 483–492
5. Bavota G, Linares-Vasquez M, Bernal-Cardenas CE, Di Penta M, Oliveto R, Poshyvanyk D (2015) The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. IEEE Transactions on Software Engineering 41(4):384–407
6. Black S (2001) Computing Ripple Effect for Software Maintenance. Journal of Software Maintenance 13(4):263–279
7. Borges HS, Valente MT (2015) Mining Usage Patterns for the Android API. PeerJ Computer Science 1:e12
8. Catolino G (2018) Does Source Code Quality Reflect the Ratings of Apps? In: IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp 43–44
9. Chen N, Lin J, Hoi SC, Xiao X, Zhang B (2014) AR-miner: Mining Informative Reviews for Developers from Mobile App Marketplace. In: IEEE/ACM International Conference on Software Engineering (ICSE), pp 767–778
10. Chow K, Notkin D (1996) Semi-Automatic Update of Applications in Response to Library Changes. In: International Conference on Software Maintenance (ICSM), pp 359–368
11. Coelho J, Valente MT (2017) Why Modern Open Source Projects Fail. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 186–196

12. Decan A, Mens T, Constantinou E (2018) On the Evolution of Technical Lag in the npm Package Dependency Network. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 404–414
13. Dering ML, McDaniel P (2014) Android Market Reconstruction and Analysis. In: IEEE Military Communications Conference (MILCOM), pp 300–305
14. Derr E, Bugiel S, Fahl S, Acar Y, Backes M (2017) Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In: ACM SIGSAC Conference on Computer and Communications Security (CCS), pp 2187–2200
15. Dig D, Johnson R (2006) How Do APIs Evolve? A Story of Refactoring. Journal of Software Maintenance and Evolution: Research and Practice 18(2):83–107
16. Fu B, Lin J, Li L, Faloutsos C, Hong J, Sadeh N (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), pp 1276–1284
17. Geiger FX, Malavolta I, Pascarella L, Palomba F, Di Nucci D, Bacchelli A (2018) A Graph-Based Dataset of Commit History of Real-World Android Apps. In: IEEE Working Conference on Mining Software Repositories (MSR), pp 30–33
18. Given LM (2008) The Sage Encyclopedia of Qualitative Research Methods. Sage Publications
19. Grandcolas U, Rettie R, Marusenko K (2003) Web Survey Bias: Sample or Mode Effect? Journal of Marketing Management 19(5-6):541–561
20. Grano G, Ciurumelea A, Panichella S, Palomba F, Gall HC (2018) Exploring the Integration of User Feedback in Automated Testing of Android Applications. In: IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)
21. Gwet KL (2014) Handbook of Inter-Rater Reliability: The Definitive Guide to Measuring the Extent of Agreement Among Raters. Advanced Analytics
22. Haney FM (1972) Module Connection Analysis: A Tool for Scheduling Software Debugging Activities. In: Fall Joint Computer Conference, pp 173–179
23. Hou D, Yao X (2011) Exploring the Intent Behind Api Evolution: A Case Study. In: Working Conference on Reverse Engineering (WCRE), pp 131–140
24. Joorabchi ME, Mesbah A, Kruchten P (2013) Real Challenges in Mobile App Development. In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 15–24
25. Khalid H, Shihab E, Nagappan M, Hassan AE (2015) What Do Mobile App Users Complain About? IEEE Software 32(3):70–77
26. Khandkar SH (2009) Open Coding. Tech. rep., University of Calgary
27. Khondhu J, Capiluppi A, Stol KJ (2013) Is It All Lost? A Study of Inactive Open Source Projects. In: IFIP International Conference on Open Source Systems, pp 61–79

28. Kirubakaran B, Karthikeyani V (2013) Mobile Application Testing: Challenges and Solution Approach Through Automation. In: International Conference on Pattern Recognition, Informatics and Mobile Engineering (PRIME), pp 79–84
29. Krippendorff K (2004) Content Analysis: An Introduction to Its Methodology, 2nd edn. Sage Publications
30. Krippendorff K (2011) Computing Krippendorff's Alpha-Reliability. Tech. rep., University of Pennsylvania
31. Kruchten P, Nord RL, Ozkaya I (2012) Technical Debt: From Metaphor to Theory and Practice. IEEE Software 29(6):18–21
32. Krutz DE, Mirakhorli M, Malachowsky SA, Ruiz A, Peterson J, Filipski A, Smith J (2015) A Dataset of Open-Source Android Applications. In: IEEE Working Conference on Mining Software Repositories (MSR), pp 522–525
33. Kula RG, German DM, Ouni A, Ishio T, Inoue K (2017) Do Developers Update Their Library Dependencies? Empirical Software Engineering pp 1–34
34. Lämmel R, Pek E, Starek J (2011) Large-Scale, AST-Based API-Usage Analysis of Open-Source Java Projects. In: ACM/SIGAPP Symposium on Applied Computing (SAC), pp 1317–1324
35. Lehman MM, Belady LA (eds) (1985) Program Evolution: Processes of Software Change. Academic Press Professional
36. Linares-Vásquez M (2014) Supporting Evolution and Maintenance of Android Apps. In: Doctoral Symposium of IEEE/ACM International Conference on Software Engineering (ICSE), pp 714–717
37. Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Di Penta M, Oliveto R, Poshyvanyk D (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), pp 477–487
38. Linares-Vásquez M, Holtzhauer A, Bernal-Cárdenas C, Poshyvanyk D (2014) Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages. In: IEEE Working Conference on Mining Software Repositories (MSR), pp 242–251
39. Martin W, Sarro F, Jia Y, Zhang Y, Harman M (2017) A Survey of App Store Analysis for Software Engineering. IEEE Transactions on Software Engineering 43(9):817–847
40. Mileva YM, Dallmeier V, Burger M, Zeller A (2009) Mining Trends of Library Usage. In: International Workshop on Principles of Software Evolution and Annual Workshop on Software Evolution (IWPSE/EVOL), pp 57–62
41. Minelli R, Lanza M (2013) SAMOA: A Visual Software Analytics Platform for Mobile Applications. In: IEEE International Conference on Software Maintenance (ICSM), pp 476–479
42. Minelli R, Lanza M (2013) Software Analytics for Mobile Applications: Insights & Lessons Learned. In: European Conference on Software Maintenance and Reengineering (CSMR), pp 144–153

43. Mojica Ruiz IJ, Nagappan M, Adams B, Hassan AE (2012) Understanding Reuse in the Android Market. In: IEEE International Conference on Program Comprehension (ICPC), pp 113–122
44. Mojica Ruiz IJ, Adams B, Nagappan M, Dienst S, Berger T, Hassan AE (2014) A Large-Scale Empirical Study on Software Reuse in Mobile Apps. IEEE software 31(2):78–86
45. Mojica Ruiz IJ, Nagappan M, Adams B, Berger T, Dienst S, Hassan AE (2016) Analyzing Ad Library Updates in Android Apps. IEEE Software 33(2):74–80
46. Montandon JE, Borges H, Felix D, Valente MT (2013) Documenting APIs with Examples: Lessons Learned with the APIMiner Platform. In: Working Conference on Reverse Engineering (WCRE), pp 401–408
47. Muccini H, Di Francesco A, Esposito P (2012) Software Testing of Mobile Applications: Challenges and Future Research Directions. In: International Workshop on Automation of Software Test (AST), pp 29–35
48. Nickerson RS (1998) Confirmation Bias: A Ubiquitous Phenomenon in Many Guises. Review of General Psychology 2(2):175–220
49. Pagano D, Maalej W (2013) User Feedback in the Appstore: An Empirical Study. In: IEEE International Requirements Engineering Conference (RE), pp 125–134
50. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A (2014) Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 101–110
51. Palomba F, Salza P, Ciurumelea A, Panichella S, Gall H, Ferrucci F, De Lucia A (2017) Recommending and Localizing Change Requests for Mobile Apps Based on User Reviews. In: IEEE/ACM International Conference on Software Engineering (ICSE), pp 106–117
52. Palomba F, Linares-Vásquez M, Bavota G, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A (2018) Crowdsourcing User Reviews to Support the Evolution of Mobile Apps. Journal of Systems and Software 137:143–162
53. Palomba F, Panichella A, Zaidman A, Oliveto R, De Lucia A (2018) The Scent of a Smell: An Extensive Comparison Between Textual and Structural Smells. IEEE Transactions on Software Engineering 44(10)
54. Palomba F, Di Nucci D, Panichella A, Zaidman A, De Lucia A (2019) On the Impact of Code Smells on the Energy Consumption of Mobile Applications. Information and Software Technology 105:43–55
55. Pascarella L, Geiger FX, Palomba F, Di Nucci D, Malavolta I, Bacchelli A (2018) Self-Reported Activities of Android Developers. In: IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp 144–155
56. Raemaekers S, van Deursen A, Visser J (2012) Measuring Software Library Stability Through Historical Version Analysis. In: IEEE International Conference on Software Maintenance (ICSM), pp 378–387
57. Robbes R, Lungu M, Röthlisberger D (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), p 56
58. Salza P, Palomba F, Di Nucci D, D'Uva C, De Lucia A, Ferrucci F (2018) Do Developers Update Third-Party Libraries in Mobile Apps? In: IEEE/ACM International Conference on Program Comprehension (ICPC), pp 255–265
59. Salza P, Palomba F, Di Nucci D, De Lucia A, Ferrucci F (2019) Third-Party Libraries in Mobile Apps: When, How, and Why Developers Update Them - Appendix. http://bit.ly/2IkpOEH
60. Scalabrino S, Bavota G, Russo B, Oliveto R, Di Penta M (2017) Listening to the Crowd for the Release Planning of Mobile Apps. IEEE Transactions on Software Engineering pp 68–86
61. Seneviratne S, Kolamunna H, Seneviratne A (2015) A Measurement Study of Tracking in Paid Mobile Applications. In: ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec), p 7
62. Sommerville I (2006) Software Engineering. Addison-Wesley
63. Strauss A, Corbin J (1998) Basics of Qualitative Research Techniques. Sage Publications
64. Syer MD, Nagappan M, Hassan AE, Adams B (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), pp 283–297
65. Tian Y, Nagappan M, Lo D, Hassan AE (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), pp 301–310
66. Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC (2018) Context Is King: The Developer Perspective on the Usage of Static Analysis Tools. In: IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 38–49
67. Viennot N, Garcia E, Nieh J (2014) A Measurement Study of Google Play. ACM SIGMETRICS Performance Evaluation Review 42:221–233
68. Yau SS, Collofello JS, MacGregor TM (1993) Ripple Effect Analysis of Software Maintenance. In: Shepperd M (ed) Software Engineering Metrics I: Measures and Validations, pp 71–82
69. Zerouali A, Constantinou E, Mens T, Robles G, González-Barahona J (2018) An Empirical Analysis of Technical Lag in Npm Package Dependencies. In: International Conference on Software Reuse (ICSR), pp 95–110
70. Zerouali A, Mens T, González-Barahona J, Decan A, Constantinou E, Robles G (2019) A Formal Framework for Measuring Technical Lag in Component Repositories and Its Application to NPM. Journal of Software: Evolution and Process p e2157
71. Zhang J, Sagar S, Shihab E (2013) The Evolution of Mobile Apps: An Exploratory Study. In: International Workshop on Software Development Lifecycle for Mobile (DeMobile), pp 1–8