On the Evolution of Mobile App Complexity

Jun Gao*, Li Li†, Tegawendé F. Bissyandé*, Jacques Klein*
*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
† Faculty of Information Technology, Monash University, Australia
{jun.gao, tegawende.bissyande, jacques.klein}@uni.lu, li.li@monash.edu

***Abstract*—Android developers are known to frequently update their apps for fixing bugs and addressing vulnerabilities, but more commonly for introducing new features. This process leads a trail in the ecosystem with multiple successive app versions which record historical evolutions of a variety of apps. While the literature includes various works related to such evolutions, little attention has been paid to the research question on how quality evolves, in particular with regards to maintainability and code complexity. In this work, we fill this gap by presenting a large-scale empirical study: we leverage the AndroZoo dataset to obtain a significant number of app lineages (i.e., successive releases of the same Android apps), and rely on six well-established, maintainability-related complexity metrics commonly accepted in the literature on app quality, maintainability etc. Our empirical investigation eventually reveals that, overall, while Android apps become bigger in terms of code size as time goes by, the apps themselves appear to be increasingly maintainable and thus decreasingly complex.**

I. Introduction

Android has been attracting the interest of developers since its early days. This also creates the situation of high competition in Android app development. Consequently, to keep up, developers are engaged in a frenzy of updates [1], [2], [3], [4]. In general, developers update their apps for (1) keeping up with the evolution of Android APIs (e.g., discarding the use of deprecated ones [5] while accessing early-release ones [6]), (2) adapting to new requirements or providing new features to keep the app competitive, (3) fixing bugs that may cause runtime crashes, or that make the app vulnerable to security threats, (4) improving the performance or maintainability, either by removing unnecessary code or by refactoring existing functionalities.

Standing out among other apps requires app developers to guarantee a level of quality in their app code. Unfortunately, in the absence of a concrete guideline for maintaining quality, it is difficult to measure to what extent quality is taken into account with respect to update changes. Instead, and as the first step towards building such a guideline, it is important to investigate some quality properties of various app versions in order to draw insights from the practice of real-world app development. Our objective is thus to conduct a large-scale empirical study on the quality evolution of Android apps.

To that end, we focus on measuring *maintainability* of app code. Software maintainability is indeed considered today as one of the most important concerns in the software industry [7], [8]. Corbi, a recognized expert in the field, has even elevated maintainability as a major challenge for program understanding since the 1990s. Generally, *code complexity* is accepted to provide a good proxy for measuring maintainability [9]. Given the pervasiveness of mobile software in our daily life today, it is important to study how complexity has evolved in order to build knowledge towards improving quality in software development.

In this work, we leverage an unprecedented large dataset of 28,564 app lineages and investigate evolution trends of complexity, relying on six metrics proposed by Chidamber et al. [10]. We implement a process where each app is analyzed and six renowned maintainability-related complexity metrics are computed, trends are highlighted and outliers are summarized.

To summarize, this paper makes the following contributions:

- We share with the community all complexity metric values for a large dataset of Android apps where each app is associated with several of its release versions.
- We present an empirical study on the evolution of complexity in Android apps based on six well-established metrics (such as NOC, Number of Children or LCOM, Lack of cohesion in Methods), and from different perspectives such as median and standard deviation values.
- We discuss insights from our study and enumerate its implications as well as the limitations.
- We make our toolset publicly available to readily compute complexity metrics for Android app APKs.

The remainder of this paper is organized as follows. Section II presents background information on Android development as well as on the metrics leveraged in our work. Section III overviews the experimental setup for answering the research questions. Section IV details the results of our study while Section V discusses some insights as well as the limitations. Finally, we discuss related work in Section VI and the conclusion of this paper in Section VII.

II. Background

To ease the understanding of our work, we provide some necessary background information about Android app developments and app complexity metrics.

A. Android Framework

The Android mobile operating system is built on top of the Linux kernel and provides a framework to facilitate the development of Android apps. As the framework evolves, the provided Software Development Kit (SDK), including the Application Programming Interfaces (APIs), is regularly updated. To better track and reflect those changes, each major

release of the Android framework is tagged with multiple names: (1) its version number (e.g., Android 4.4); (2) its API level (e.g., 19); and (3) a name of sweet (e.g., KitKat). Figure 1 presents an example of API levels with respect to their adoption by millions of Android-powered devices using the official Google Play store as of May 2018.

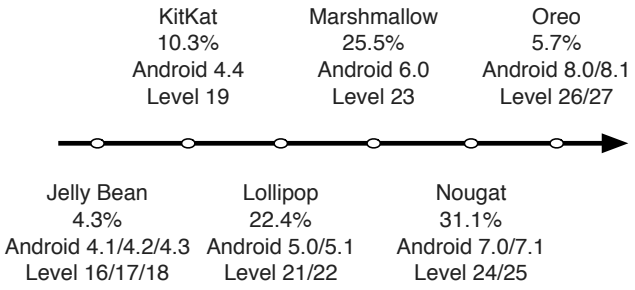


Fig. 1: Distributions of API levels supported by current Android-powered devices (versions with less than 1% are not shown).

B. Android Apps

Android apps are generally written in Java. During compilation, the Java code is then transferred into Dalvik/ART bytecode (in DEX format). Together with other resource files such as the AndroidManifest configuration and images, the bytecode is then assembled into an Android Application Package (APK), which can then be distributed on app markets. When developing an Android app, developers have to specify the ideal API level that the app supports, i.e., the app needs to be thoroughly tested against devices running a framework at that level. This ideal API level is stored in the app as an attribute named *targetSdkVersion*, where the value of this attribute can be programmatically extracted. Hence, each Android app can be associated with an integer number indicating the targeted API level that the app is implemented upon.

C. Terminology

APK (Application Package Kit) is the Android application package for distribution and installation of Android apps. Moreover, to update an Android app, an APK of the new version needs to be provided. Therefore, an **app lineage** is recognized as the consecutive series of its APKs and each of the APK is also called an **app version**. Hereafter, we then use app lineage and app version to distinguish these two different concepts,

D. Complexity Metrics

Chidamber et al. [10] have introduced six metrics to "measure the complexity in the design of classes". Since Android apps, as mentioned before, are written in Java and thereby have extended Java's object-oriented features, the proposed six metrics should also be able to reliably improve the development processes of Android apps. State-of-the-art studies such as Jost et al. [11], [12] have also leveraged those metrics for Android app developers to consider so as to write high-quality code. We note that these metrics are highly related to complexity concerns, and thus, we adopt them to measure the complexity of Android apps.

1) **Weighted methods per class (WMC)** is the sum of the complexities of all methods in a class. It is used to measure the effort required for developing and maintaining a particular class as well as the inheritability and reusability of a class. A high WMC score of a class means that the class is complex that hence is difficult to reuse and maintain. To simplify the calculation, in this work, we consider the complexity of all methods to be unity. Then WMC is simply a method counter of each class.
2) **Depth of inheritance tree (DIT)** is used to measure the depth of a given class based on the inheritance tree. Ideally, the value of DIT metric should be kept low as the complexity of developing, testing and maintaining a class would significantly increase if the depth of inheritance tree increases. As DIT defined, the inheritance tree of each class is calculated and the maximum length is set as the value of DIT.
3) **Number of children (NOC)** is another metric leveraged to measure the "width" of a given class (i.e., the number of direct sub-classes) based on the inheritance tree. The value of NOC approximately indicates the reuse degree of a given class. While the reusability of a class increases if more children are introduced, the responsibility required to maintain the class not to break the children's behavior also increases.
4) **Lack of cohesion in methods (LCOM)** is a metric used to measure the cohesiveness between methods and attributes of a given class. A higher LCOM value indicates a low cohesion between the methods and data, which hence increases the complexity of the class and subsequently increasing the possibility of introducing errors during the development of software. There are two ways to calculate the value according to Linda et al. [13] and in this work, we choose the first one which is based on the average percentage of each data field used by the methods of a class.
5) **Coupling Between Object classes (CBO)** measures the dependency of a class on other classes. High CBO value indicates excessive dependency which means lower reusability and higher testing complexity. It is calculated by counting the number of other classes used by a class.
6) **Response For a Class (RFC)** reflects the potential invocation of methods of a class on responding to a message. A low value of RFC is preferred since it indicates short possible invocation chain which makes debugging and testing easier. RFC is calculated by counting all the methods invoked in a class. For methods invoked more than once, only the first time will be counted.

Initially, we have considered the 22 quality metrics proposed by Mercaldo et al. [14]. However, our preliminary experiments have revealed that many of them are highly correlated with each other as demonstrated in Figure 2. Moreover, because of space limitations to present the results of all the 22 metrics,

Therefore, for this study, we decide to focus only on the six classic metrics. We believe that the other metrics, especially the ones that are recently introduced, are also worth to explore and hence we will consider them in our future works.

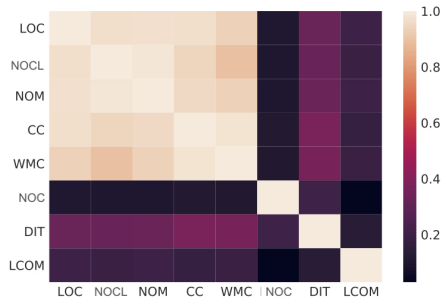


Fig. 2: Metrics Correlation Map

III. Experiment Setup

To set up the empirical experiments related to the complexity evolution of Android apps, we present the main research questions this work explores and the dataset this work stands upon in Section III-A and Section III-B, respectively.

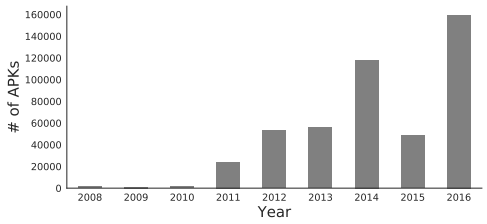
A. Research Questions

Our objective is to understand the evolution of Android apps' complexity and hence to empirically observe practical insights for guiding the evolution of Android apps towards engineering more reliable apps. To fulfill this objective, we plan to perform an exploratory study to answer the following research questions:

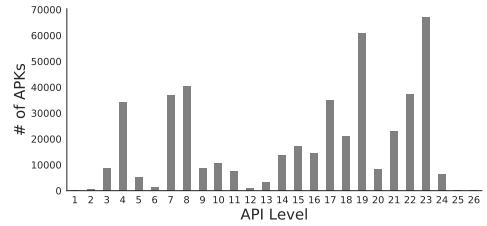
- **RQ1:** How does the code of Android apps generally evolve? As the first research question, in order to have an overall understanding of the general evolution of Android apps, we empirically investigate the changes in terms of code size (i.e., DEX size and class number) of Android apps over time.
- **RQ2:** How does the complexity of Android apps evolve as time goes by? The complexity evolution within this research question will be investigated year by year. For each app lineage, we choose one app version for each year: the latest one released in that year. The chosen apps from the same year (different lineages) will be considered as a whole and the extracted metric values will be leveraged to represent app complexity of that year.
- **RQ3:** How do Android API level updates impact on app complexity? Android framework is recurrently updated to introduce new features or fix critical bugs. To benefit from these updates, Android apps need to be correspondingly changed. Hence, the complexity evolution within this research question will be investigated based on the targeted API levels of the considered apps.
- **RQ4:** What are the patterns of complexity evolution? By defining feature patterns, the evolution of complexity will be investigated in the manner of individual app lineage. Then, how Android apps evolves normally as well as what is the uncommon pattern during complexity evolution can be spotted.

B. Dataset

To answer the aforementioned research questions, we resort to so far the largest app collection, namely AndroZoo [15], [16], to prepare the experimental environment. AndroZoo[1] is a growing collection of Android apps collected from various sources, including the official Google Play app market and third party alternative markets such as AppChina. So far, AndroZoo repository contains over 5 million Android apks and has been successfully applied to support the analysis of various research studies [17], [18]. For this study, based on AndroZoo, we eventually re-construct 28,564 app lineages of app versions no less than 10, which contains 465,037 app versions. Figure 3a shows the distribution of the releasing years of the apks. The releasing time is obtained from the last modification time of the "classes.dex" files decompressed from apks. While, Figure 3b exhibits its target API level distribution.



(a) Year Distribution



(b) API Level Distribution

Fig. 3: Statistics of Lineage Dataset

C. Re-construction of App Lineages

We re-construct app lineages based on AndroZoo's data heap and according to the procedure proposed by Gao et al.[19] as illustrated in Figure 4.

1) Application ID Extraction: name is required to be given for every Android app by following Java package naming convention. It needs to be unique for each app and used as the ID of the app. We group together APKs with same name as candidate app versions of an app lineage.
2) App Certificate Clustering: to be sure that all app versions are from the same developer, the developer signature is considered. APKs with the same application ID but the different signature will not be classified in one app lineage.

[1]https://androzoo.uni.lu

3) App Market Clustering: we assume that developers always distribute their apps on the same market. Then we further constraint that our app lineages need to contain APKs from the same market.
4) App Version Sorting: to reflect the evolution process, the app versions of a given app lineage are ordered according to their *versionCode* which is declared in their APKs.

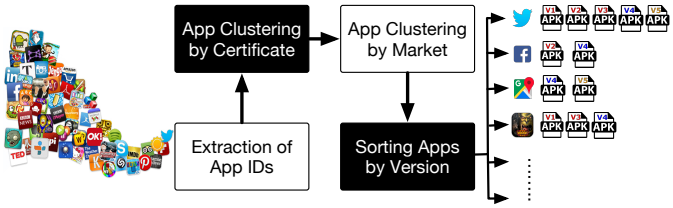


Fig. 4: App Lineage Re-construction Process

D. Metrics Computing

In this work, the metric values are computed at the *smali* code level. All the considered lineage apps are disassembled by *Apktool*, a well-known static analysis tool for reverse engineering Android APK files.[2] Apktool will translate the executable part of an app, namely the *DEX* bytecode into the so-called *smali* code.

Because the considered complexity metrics are measured at class levels, while Android apps normally are made up of multiple classes, for a given metric, we regard its value for a given Android app as the median and standard deviation value among that of all the classes of the app. Statistically speaking, these two values have characterised the majority of the sample population (i.e., median) and their spreads (i.e., standard deviation). Indeed, for a certain app, the median value can represent the app in most of its classes while the standard deviation reflects the extent the complexity of the classes can go, e.g., either better or worse.

In this work, we rename these two values (median and standard deviation) as ***feature*** and ***variation***[3], which are explained as follows:

Given an app a, $C = \{c_1, c_2, \ldots, c_n\}$ is the set of its classes, for a certain metric m, the value of c_i is $v_m(c_i)$, where $c_i \in C$, then

- ***feature*** value: $feature(a) = M$, where M is the median value of $\{v_m(c_1), ..., v_m(c_n)\}$.
- ***variation*** value: $variation(a) = \sigma$, where σ is the standard deviation of $\{v_m(c_1), ..., v_m(c_n)\}$.

During our experiments, we have found that the *android.support* package has been widely presented in some Android apps. Since this package is provided by Google as an official library for resolving issues such as compatibility[4], we do not take this package into consideration when computing the values of metrics.

[2]https://ibotpeaches.github.io/Apktool/

[3]The rationale behinds this renaming is to avoid confusions about expressions such as "median of the median values".

[4]https://developer.android.com/topic/libraries/support-library/index.html

It is also worth to mention that not all lineage apps can be successfully reverse engineered by our tool for computing the values of our selected metrics. The main reasons led to the failures are 1) *Apktool* crashes due to exceptions such as no *smali code* generated, (2) null values are returned by our tool because the number of classes is too small (e.g., less than three for some app versions) or there is no field defined by some classes (i.e., this will lead to null value for metric LCOM). Moreover, since date information is also important to this study, (e.g., we leverage it to perform the year-based evolution study), we further remove such app lineages that have incomplete DEX date associated, i.e., we cannot extract a validated assembly time from the app.

To conclude, among the 28,559 lineages (464,649 app versions in total), 1,389 app lineages (23,451 apps versions) that have confronted the aforementioned issues are ignored in this study. In other words, our study is conducted based on 27,170 app lineages (441,198 app versions).

IV. Results

We now present our investigation details towards answering the aforementioned research questions.

A. RQ1: General Evolution of Android Apps

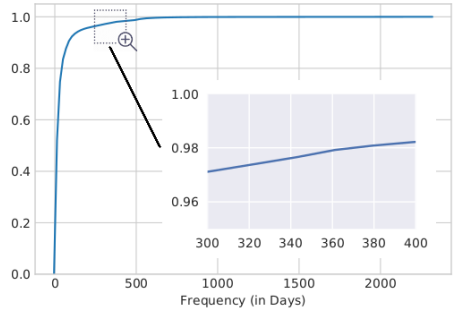


Fig. 5: Cumulative distribution function (CDF) of the update frequency of selected lineage apps. Given a frequency (e.g., $x = 365$ days), the probability for an app to have an update within $x = 365$ days can be quickly observed from the CDF (i.e., the corresponding value in the Y-axis).

Since it is non-trivial to select the time interval for re-aligning lineage apps, we resort to a simple empirical study to select such time interval. The study looks into the update frequency of all the selected lineage apps. Figure 5 illustrates the Cumulative Distribution Function (CDF) of the update frequency, where the frequency is counted in days (as shown in the X-axis). For about a year (e.g., 365 or 366 days), more than 95% of considered apps have been updated at least once, presenting a great time interval to build our time-based evolution dataset. Therefore, we select a year as the time interval to investigate the complexity evolution of Android apps.

To understand the general evolution of Android apps, we first look into the evolution of size and the number of Java classes of Android apps. Figure 6 shows how are the median value of app size and the number of classes evolved as time

goes by. For each median value, it is calculated based on all apps of that year.

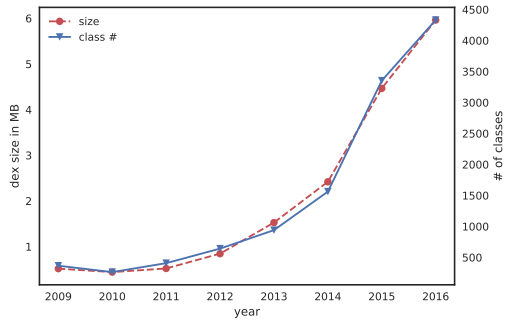


Fig. 6: App Size and # Classes Evolution in Time

Quite clearly, both app sizes and class numbers were increasing, especially, from 2014 to 2015, the rise was dramatic. This evidence suggests that Android apps become bigger and bigger in both size and the number of classes.

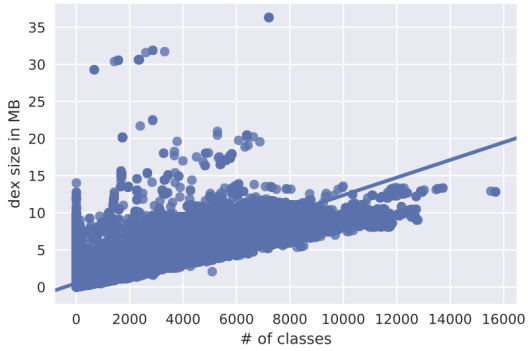


Fig. 7: Scatter Plot of App Size and # Classes

Furthermore, as demonstrated in Fig. 7, there is also a strong correlation between app size and the number of classes. This strong correlation is also confirmed to be statistically significant via the Pearson's correlation coefficient ($\rho > 0.9$, showing a strong positive correlation). This strong positive correlation implies that app updates are more likely to add new classes than simply add codes to existing classes. Indeed, in our selected app lineages, 80.73% of them have their apps eventually become bigger (comparing the last app version with the first one) in terms of the number of classes, while the percentage is even higher when talking about the app size: 83.4% of selected lineages.

As app size and the number of classes getting bigger and bigger, intuitively, apps are becoming more complex and more difficult to maintain. Consequently, a detailed study on app complexity evolution is expected. This research question actually motivated us to perform an in-depth analysis of the complexity evolution of Android apps.

> During the evolution, app developers are more likely to introduce new classes rather than adding code to existing ones, as shown by the strong correlation of changing in app size and number of classes.

B. RQ2: Complexity Evolution via Time

We investigate the complexity evolution of Android apps via their release time[5]. State-of-the-art approaches for time-based evolution normally choose random apps for different time-points. As a result, the apps chosen in different time-points could be different. On the contrary, our lineage based time evolution approach is expected to always select app versions from same app lineages. By doing this, the consistency of samples between different time-points can be well reserved, which makes the final result more reliable. To support this kind of investigation, we need to re-construct a fine-grained dataset where the considered lineage apps are aligned via time. To this end, we re-align our lineage apps by selecting the last app version of each year from each app lineage.

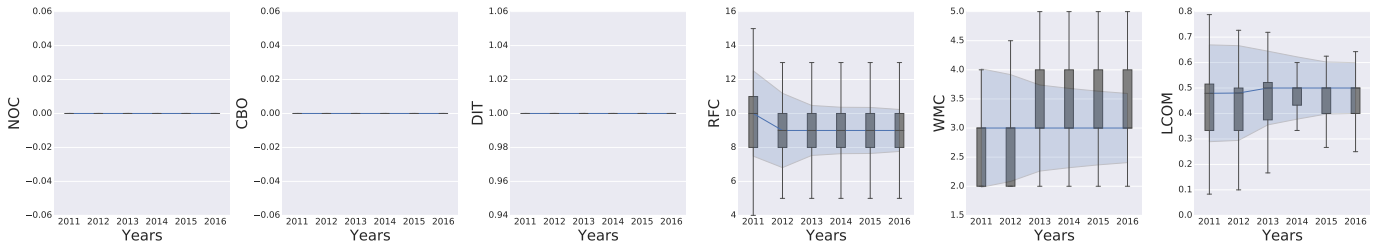
Figure 8a presents the evolution of the metrics feature value from 2011 to 2016. The median value of metrics NOC, DIT, WMC and CBO exhibit as horizontal lines with very low values, which indicates that app complexity in terms of these 4 metrics has kept very low and constant for past 6 years. These values tell us that for most classes of an app, they were not sub-classed (i.e., 0 of NOC), they had shallow inheritance trees (i.e., 1 of DIT), their method complexities are low (i.e., 3 of WMC) and they were not coupled with other classes (i.e., 0 of CBO). Regarding the standard deviations shown as the ribbons, NOC, DIT and CBO show no ribbon at all while WMC shows an observable ribbon with a narrow-down trend. Since the standard deviations reflect the difference between different apps, we can say that the vast majority of apps exhibit no difference of complexity in terms of NOC, DIT and CBO. Meanwhile, in terms of WMC, there are apps with different feature values, but the difference is not big (± 1 on average) and getting smaller.

On the other hand, RFC and LCOM show more changes as they evolving. The drop of RFC in 2012 indicates a clear improvement of this metric for most of the apps. While a slight deterioration of LCOM happened in 2013 can be observed as well. Furthermore, the difference between apps in these 2 metrics was getting narrower too.

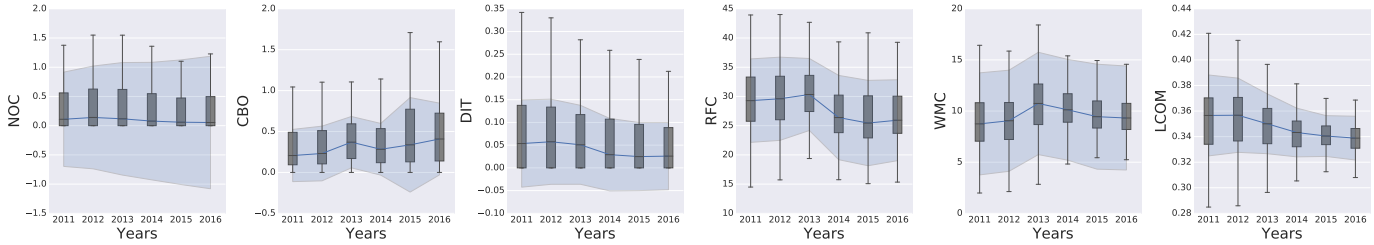
Because the feature values only reflect app complexity in major situations as explained in Section III-D. To have a more comprehensive understanding of apps complexity evolution, we further resort to an investigation into the evolution of variation values of Android apps.

Figure 8b shows the evolution of app variation values. From the median value perspective, 4 of the metrics show a clear decline trend which are NOC, DIT, RFC and LCOM. While for CBO and WMC, they were slightly increased over the years. As the variation of a metric measures the differences of the metrics among different classes of an app, Thus, a decreasing in trend is preferred. On the other hands, the differences of variation values between different apps are exhibited by

[5]Since AndroZoo does not collect the release date metadata for Android apps, and it is virtually impossible to retrieve such metadata for previous app versions, as these metadata have already been overwritten by the data of updated app versions, in this work, we consider the assemble DEX time as the app release time.



(a) Feature Values



(b) Variation Values

Fig. 8: Lineage based evolution with the central lines depict the trend of median value while the ribbons show the changing of standard deviation and the boxplots illustrate the distribution of each year

the ribbons in the figure. Therefore, for past 6 years, the differences of NOC and CBO have been increased while DIT and LCOM have been decreased. For RFC and WMC, the differences kept almost the same.

> As time goes by, the complexity in terms of RFC has been mitigated but deteriorated in LCOM. Out of the six metrics, nature updates (update via time) have only impacted these two metrics, although the impacts are quite limited. It is worth to highlight that the complexity difference between different apps is getting closer during the evolution.

C. RQ3: Complexity Evolution via API Level

In order to understand the possible impact of API levels on the evolution of app complexity, we conduct another study that specifically looks into the complexity difference between such apps that target different Android API levels. To this end, for each app lineage, we pair up adjacent app versions, which target different API levels, for difference examination. Given a pair of app versions (a_i, a_{i+1}) and their targeted API levels (L_i, L_{i+1}), we define *level skip* as the difference between the two targeted API levels (i.e., $level\ skip := L_{i+1} - L_i$). Figure 9 illustrates the distribution of API level skips summarized from our app lineage dataset. The level skips vary from -16 to 19 while the majority app pairs fall into the category of level skip equals to 1, followed by 2 and 3 skips respectively. The reasons causing minus level skips could be: (1) to support previous users or features requiring old API, (2) version code assigned reversely, (3) some other unknown purposes. As these cases are rare and abnormal, in this paper, they will not be considered. In this work, we take into account all the app pairs that have API level skip between 1 and 3. Based on this criterion, we form a new dataset containing three types of app pairs: S1, S2 and S3 for app pairs with one, two and three level skips, respectively,

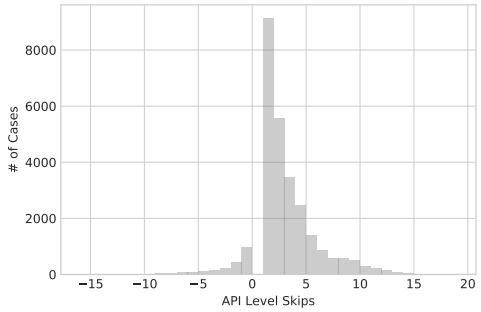


Fig. 9: Distribution of API level skips for every two adjacent app versions while $x = 0$ is not shown as it means no skip

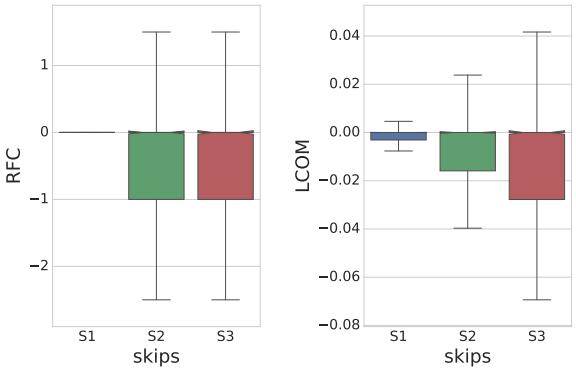


Fig. 10: API Level based Evolution of Feature Values

Figure 10 illustrates the distribution of feature value differences of app pairs via level skip. Since metrics NOC, DIT, WMC and CBO are quite stable during the evolution of Android apps, as shown in Section IV-B, we only present the distribution of metrics RFC and LCOM in the figure.

Interestingly, the median values stay closely to 0 suggests that the changes are quite small despite the targeted API level is updated. The fact that the major parts of the boxes fall into the negative side of Y-axis and larger level skips seem to yield larger ranges of the negative parts indicates that the changes do not seem to increase the app complexity (at least for RFC and LCOM).

Figure 11 illustrates the distribution of variation value differences via level skip. Similarly, except for metric WMC, where the median values are generally decreasing when level skip increases, the median values of other metrics are very close to 0. Regarding the body of the boxes, the major parts for metrics DIT, WMC, RFC and LCOM fall into the negative side of Y-axis while for metrics NOC and CBO, they fall into the positive side. The body size is increased as level skip increasing. For NOC and CBO, they increase mainly on the positive side of Y-axis. But the rest four metrics increase mainly on the negative side.

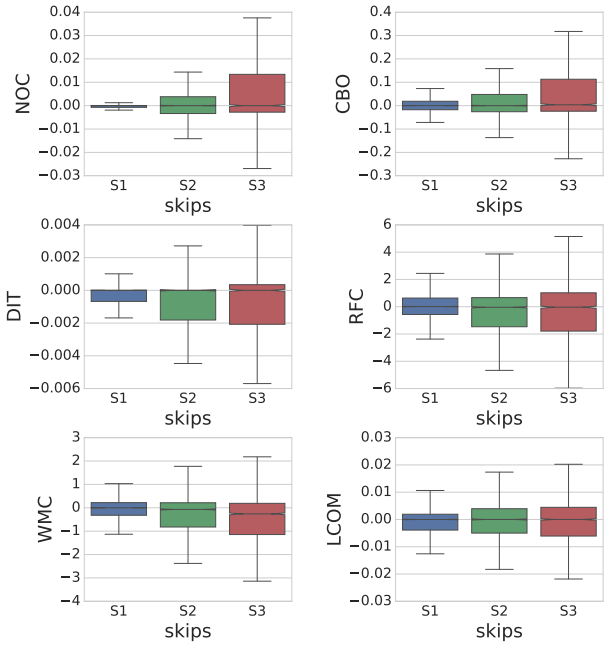


Fig. 11: API Level based Evolution of Variation Values

As the interpretation of these patterns, updates of API levels may effect on different metrics differently. A bigger level skip normally causes a larger increase in the complexity difference within an app from the aspect of NOC and CBO (remind that the definition of variation value in Section III-D). However, from the aspect of DIT, WMC, RFC and LCOM, the complexity difference shrinks mostly.

> API level updates could cause the complexity of Android apps to decrease, although the extent is quite limited. Also, for most of the metrics, API level updates shrink the complexity difference within apps.

D. RQ4: Patterns of Complexity Evolution

We remind the readers that in this work we use lineage apps rather than randomly selected apps to investigate the complexity evolution of Android apps, where the dataset allows us to have a deeper look at how each app lineage evolves. Consider an app lineage with n app versions $app_1, app_2, \ldots, app_n$. Let set $M = \{m_1, m_2, \ldots, m_n\}$ stand for the feature or variation values of a metric of these app versions and σ be the standard deviation of set M. The possible evolution patterns that each app lineage may fall into are defined as follows:

- overall patterns
  - flat: $|m_1 - m_n| < \sigma$, which means the difference between the first and last app version is less than the standard deviation of the app lineage.
  - decrease: $m_1 - m_n \geq \sigma$, which means the value of the first app version is greater than the value of the last one and the difference is bigger than the standard deviation.
  - increase: $m_n - m_1 \geq \sigma$, which means the value of the last app version is greater than the value of the first one and the difference is bigger than the standard deviation.
- detail patterns
  - constant: patterns between adjacent app versions are consistent with the overall pattern. For flat pattern, it means $m_i = m_j$. For decrease pattern, $m_i \geq m_{i+1}$. For increase pattern, $m_i \leq m_{i+1}$. Where $i, j \in \{1, \ldots, n\}$.
  - hill: $\max M \in \{m_2, \ldots, m_{n-1}\}$ and $\max M - \max\{m_1, m_n\} > \sigma$, which means maximum value happens in an app version which is not the first or the last app version. Additionally, the maximum value needs to be greater than the maximum value of the first and the last app version and the difference need to be bigger than the standard deviation of the app lineage.
  - valley: this is the opposite situation of hill and it expresses as $\min M \in \{m_2, \ldots, m_{n-1}\}$ and $\min\{m_1, m_n\} - \min M > \sigma$
  - wave: other cases where no constant, hill and valley patterns can be observed.

TABLE I: Possible Patterns & Abbreviation

Pattern	Abbreviation
Flat Constant	*fc*
Increase Constant	*ic*
Decrease Constant	*dc*
Flat Wave	*fw*
Increase Wave	*iw*
Decrease Wave	*dw*
Flat Hill	*fh*
Increase Hill	*ih*
Decrease Hill	*dh*
Flat Valley	*fv*
Increase Valley	*iv*
Decrease Valley	*dv*
Flat Hill & Valley	*fhv*
Increase Hill & Valley	*ihv*
Decrease Hill & Valley	*dhv*

The overall patterns are the patterns defined by the starting and ending points. They are designed to give a brief concept of what is the evolution trend. While detail patterns are

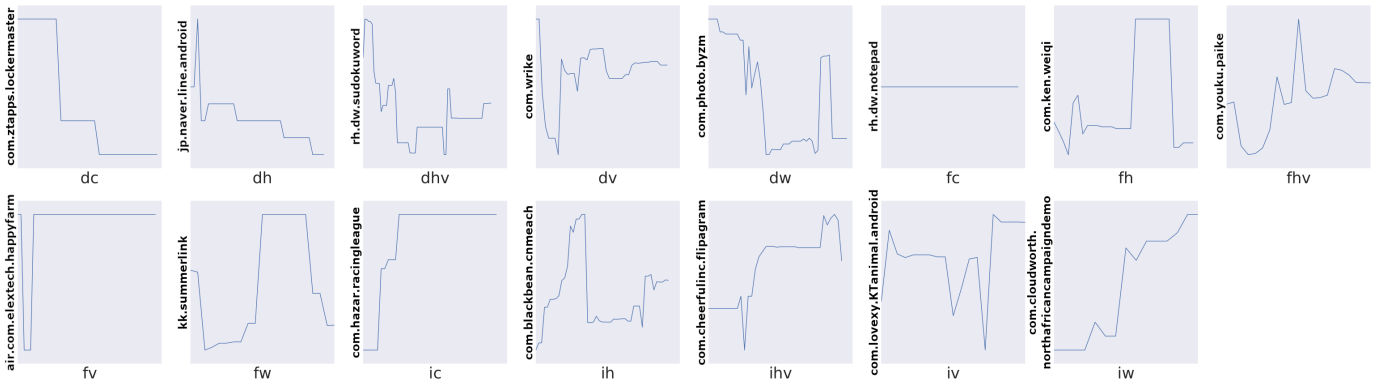


Fig. 12: Real world examples of patterns of complexity evolution with the name of patterns as x-axis labels and application names as y-axis labels

meant to reflect the feature patterns during the evolution. To give a complete evolution pattern, one of the overall patterns combined with one or two detail patterns is required and the possible combination patterns are shown in Table I. Figure 12 further shows the real world examples of each defined pattern from our dataset.

According to the patterns defined, we analyze each app lineage to obtain their evolution patterns and then calculate the frequency of each pattern. The final result is displayed by a heat map in Figure 13. Likewise, frequencies of NOC, DIT and CBO feature values are removed from the figure because all their values keep constant (cf. Section IV-B).

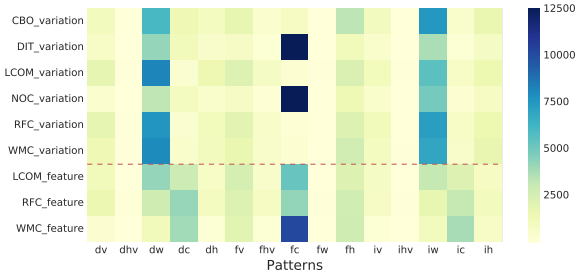


Fig. 13: Frequency distribution of patterns of complexity evolution with feature and variation value parts divided by a red horizontal dash line

Regarding the evolution pattern summarized via feature values (the part under the red horizontal dash line), undoubtedly, *fc* is the most evident column, followed by column *dc* and *ic* sequentially. On the other hand, in the variation value part (the part above the red horizontal dash line), although the column of *fc* is still noticeable, there are only 2 tiles (which are DIT and NOC) with a very dark color, while the rest tiles are quite bright. Meanwhile, column *dw* and *iw* are also very distinguishable and with much even darkness. Moreover, 3 brightest columns are also spotted which are *dhv*, *fw* and *ihv* and both median value and standard deviation parts are consistent in these 3 brightest columns.

So far, we can observe that the complexity of major Android apps tends to stay constant (do not forget that the median values of the 3 removed metrics are even more constant). Nonetheless, there are still many apps tend to decrease constantly in complexity while others may increase constantly. However, except for metrics DIT and NOC, where the evolution pattern of most apps is still, the complexity difference within an app lineage is more likely to either decrease or increase wavily.

> According to the patterns of complexity evolution, wavily increasing and decreasing have dominated the trend of complexity difference during the evolution of Android apps. This empirical evidence suggests that app developers might not really be aware of controlling the complexity of their apps.

V. Discussion

In this section, we discuss several implications that this study can lead to and disclose the potential threats to the validity.

A. Implication

a) Towards Engineering Better Metrics: Our experimental results suggest that app complexity does not significantly change during app updates. This evidence can be explained by the fact that, as shown in Section IV-A, Android app updates usually do not simply add codes to existing classes but are more likely to add new classes. Unfortunately, the six metrics we used in this study are all based on classes. They might not be representative to fully capture the complexity of Android apps. Therefore, we argue that there is still a lot of space to improve towards engineering better metrics for characterizing the development of mobile apps. To this end, designing a new set of complexity-related metrics (e.g., to take into account invocation chains) is needed. Moreover, neglecting the complexity conducted by the interaction between classes is not reasonable, so comprehensive application level metrics are also needed.

b) Best Practice to Guide Future Quality Evolutions: Generally, preserving and improving software quality is a long-time challenge that is difficult to resolve. Due to software aging, without active countermeasures, the quality of applications slowly degrades during their evolutions [20], [21]. As argued by Mens et al., there is a need to provide tools and techniques that preserve or even improve the quality

characteristics of software systems [22]. In this study, around 9% of our selected app lineages are always in line with that of the mainstream. For our future work, good practices could be learned based on these apps. If so, we subsequently present automated tools to apply the obtained good practices, e.g., by instrumenting directly the bytecode of Android apps [23].

c) Observing differences between developer capabilities: Since an Android app is likely developed by multiple developers, who might have different abilities to control the quality of their implemented code, we believe that standard deviation value could be a good means to capture the differences among developers in a team which can further provide insights to optimize development teams.

B. Threats to Validity

The study conducted in this work has presented several threats to the validity.

First, the considered six metrics may not be fully representative of the quality of Android apps. For example, compared to the six metrics proposed by Jost [11], we have missed four of them although have additionally considered 2 metrics. Also, many metrics are highly correlated with others. Hence, as suggested by Mourad et al. [24], there is a need to invent new quality metrics that attempt to unify similar metrics so as to simplify further analysis and make interpretation concise. We consider this as our future work.

For an app lineage, some versions could be missing without our awareness. Also, the order of app versions may not be correct if the version code in the manifest is assigned randomly. However, the impact of missing versions on this study is limited while given random version is not common practice.

Finally, our time-based evolution study is at year level, although we have empirically shown that a year is actually a reasonable interval, it might still be too long for this study as in practice popular apps are updated more frequently. To mitigate this potential threat, we plan to design and implement a generic framework for supporting more advanced evolution analyses of mobile apps, where different parameters such as time interval, level skips and metrics can be easily configured and adjusted.

VI. Related Work

In this section, we discuss related work on quality metrics and evolution-related studies.

A. Quality Metrics

Various studies have investigated the problem of observing reliable metrics for characterizing the quality of mobile apps. Chidamber and Kemerer [10] introduce six metrics for guiding the design of object-oriented programs and four of them have been considered by Jost et al. [11] and hence by this work. Thomas McCabe [25] further introduced Cyclomatic Complexity (CC) for measuring the complexity. Fenton and Neil [26] argued that the future for software metrics lies in using them to develop decision-support tools to support risk assessment.

Several researches have focused on quality metrics related to Android apps. Tian et al. [27] investigated the characteristics which make Android apps high-rated. They found that metrics such as app size, target SDK version are influential factors contributing to the success of Android apps. Protsenko et al. [28] also leverage software metrics to detect Android malware. Experimental results show that software metrics are reliable for distinguishing malware and resilient against common obfuscation.

B. Evolution Study

There are several researchers studied the general laws of software evolution [29], [20], [30], which show that software will continuously change and so does its complexity, demonstrating that software evolution analysis is essential in our community.

Software evolution analysis has been widely adopted to understand the evolutionary process of a software system and hence to predict its future evolution [31], [32], [33]. Generally, software evolution analysis investigates the evolution of a software system to identify potential shortcomings in its architecture. Those identified shortcomings can then be addressed specifically to improve the quality of the software system.

Neamtiu et al. [31], by studying nine open-source projects covering 705 official releases, they confirmed Lehman's two laws of software evolution (i.e., software continuing change and continuing growth). Behnamghader et al. [33] argued that studying software quality before and after each commit can reveal how each change impacts the overall quality.

However, Android apps are generally released as APKs which do not contain commit messages. Therefore, researches leveraged the difference between two subsequent app releases to investigate the evolution of Android apps [34], [6], [35], [4]. Calciati et al. [35] have investigated the evolution of permissions. Taylor et al. [4] investigated the evolution of app vulnerabilities. The most closed work to ours is the one presented by Hecht et al. [36], who investigate the evolution of Android poor design choices based on 3,568 versions of 106 Android apps. Our work, although focusing on different metrics, is generally in line with theirs and thus can be taken as a supplement to the state-of-the-art.

VII. Conclusion

In this work, we have conducted a large-scale empirical study of the complexity evolution of Android apps. To support the study, we re-constructed 28,564 app lineages from AndroZoo, where each app lineage is made up of at least 10 versions that record the historical releases of the same app. Subsequently, we select six metrics that have been successfully leveraged by literature works for quantifying the complexity of Android apps. Based on the evolution of these six metrics, we eventually find that (1) Android apps usually become bigger during their evolutions and updates are tend to add new classes,

(2) nature updates do not really impact on the complexity of Android apps, (3) the update of Android framework could mitigate app complexity but very limited, (4) complexity evolution is more like to wavily increase or decrease.

ACKNOWLEDGMENT

This work is supported by the Luxembourg National Research Fund (FNR) through grant PRIDE 15/10621687/SPsquared and project CHARACTERIZE C17/IS/1169386.

REFERENCES

[1] Yuan Tian, Bin Liu, Weisi Dai, Blase Ur, Patrick Tague, and Lorrie Faith Cranor. Supporting privacy-conscious app update decisions with user reviews. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 51–61. ACM, 2015.

[2] Rahul Potharaju, Mizanur Rahman, and Bogdan Carbunar. A longitudinal study of google play. *IEEE Transactions on Computational Social Systems*, 4(3):135–149, 2017.

[3] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200. ACM, 2017.

[4] Vincent F Taylor and Ivan Martinovic. To update or not to update: Insights from a two-year study of android app evolution. In *Proc. of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 45–57, 2017.

[5] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[6] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd Intl. Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[7] E Yourdon. The rise and fall of the american programmer. *Yourdon Press Computing Series*, 1992.

[8] Frederick P Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.

[9] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering*, 22(6):3057–3087, 2017.

[10] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[11] GREGOR JOŠT, JERNEJ HUBER, and MARJAN HERIČKO. Using object oriented software metrics for mobile application development. In *Second Workshop on Software Quality Analysis, Monitoring, Improvement and Applications SQAMIA 2013*, 2013.

[12] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[13] Linda H. Rosenberg and Lawrence E. Hyatt. Software quality metrics for object-oriented environments. Technical report, NASA, 1995.

[14] Francesco Mercaldo, Andrea Di Sorbo, Corrado Aaron Visaggio, Aniello Cimitile, and Fabio Martinelli. An exploratory study on the evolution of android malware quality. *Journal of Software: Evolution and Process*, 0(0):e1978. e1978 smr.1978.

[15] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Workshop on Mining Software Repositories*, pages 468–471. ACM, 2016.

[16] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. 2017.

[17] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)*, 2017.

[18] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *The 23rd IEEE Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[19] Jun Gao, Li Li, Pingfan Kong, Tegawendé F. Bissyandé, and Jacques Klein. On vulnerability evolution in android apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 276–277, New York, NY, USA, 2018. ACM.

[20] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE, 1997.

[21] David Lorge Parnas. Software aging. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 279–287. IEEE, 1994.

[22] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22. IEEE, 2005.

[23] Li Li. *Boosting Static Security Analysis of Android Apps through Code Instrumentation*. PhD thesis, University of Luxembourg, Luxembourg, 2016.

[24] Mourad Badri, Linda Badri, and Fadel Touré. Empirical analysis of object-oriented design metrics: Towards a new metric using control flow paths and probabilities. *Journal of Object Technology*, 8(6):123–142, 2009.

[25] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[26] Norman E Fenton and Martin Neil. Software metrics: roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 357–370. ACM, 2000.

[27] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E Hassan. What are the characteristics of high-rated apps? a case study on free android applications. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 301–310. IEEE, 2015.

[28] Mykola Protsenko and Tilo Müller. Android malware detection based on software complexity metrics. In *International Conference on Trust, Privacy and Security in Digital Business*, pages 24–35. Springer, 2014.

[29] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proc. of the IEEE*, 68(9):1060–1076, 1980.

[30] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M Gonzalez-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*, 46(2):28, 2013.

[31] Iulian Neamtiu, Guowu Xie, and Jianbo Chen. Towards a better understanding of software evolution: an empirical study on open-source software. *Journal of Software: Evolution and Process*, 25(3):193–218, 2013.

[32] Stephen W Thomas, Bram Adams, Ahmed E Hassan, and Dorothea Blostein. Studying software evolution using topic models. *Science of Computer Programming*, 80:457–479, 2014.

[33] Pooyan Behnamghader, Reem Alfayez, Kamonphop Srisopha, and Barry Boehm. Towards better understanding of software quality evolution through commit-impact analysis. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 251–262. IEEE, 2017.

[34] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE Intl. Conference on*, pages 70–79. IEEE, 2013.

[35] Paolo Calciati and Alessandra Gorla. How do apps evolve in their permission requests?: a preliminary study. In *Proc. of the 14th Intl. Conference on Mining Software Repositories*, pages 37–41. IEEE Press, 2017.

[36] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the software quality of android applications along their evolution (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 236–247. IEEE, 2015.