ReChan: An Automated Analysis of Android App Release Notes to Report Inconsistencies

Daniel Domínguez-Álvarez
IMDEA Software Institute
Spain
University of Verona
Italy

Daniel Toniuc
IMDEA Software Institute
Spain

Alessandra Gorla
IMDEA Software Institute
Spain

ABSTRACT

"What's new?" This is what users wonder when they see the notification that a mobile app has just been updated on their device. New releases may involve simple bug fixes, or may include new features that users are eager to try. Regardless of the change, users do want to know what are the differences with respect to the release that have been using so far. The Google Play store has a visible section for each Android app that clearly describes the changes that affect the latest release. This description, however, is curated by developers, and may not match the actual changes in the binary code. This paper presents ReChan, a novel technique aiming to automatically detect mismatches between release notes of Android applications and the actual changes in the code. We define a taxonomy of 9 release categories by manually tagging 1,200 real samples, and we present our solution to automatically classify release notes written in English. ReChan then implements specific analyses to detect such changes in the code, and compares the analyses outcome to detect mismatches. ReChan achieves a precision, recall and f-score of 84.9% on the manually crafted ground truth of three open source apps. Experiments on a dataset of 12,706 closed source Android apps show that developers tend to correctly report changes due to bug fixes and new features, but omit changes that affect the list of requested permissions, the UI and other content that the app uses.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; **Software maintenance tools**.

KEYWORDS

release notes, mobile apps

ACM Reference Format:
Daniel Domínguez-Álvarez, Daniel Toniuc, and Alessandra Gorla. 2022. ReChan: An Automated Analysis of Android App Release Notes to Report Inconsistencies. In *IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MOBILESoft '22), May 17–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524613.3527819

MOBILESoft '22, May 17–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9301-0/22/05…$15.00
https://doi.org/10.1145/3524613.3527819

1 INTRODUCTION

With every new release of a mobile app, developers write a release note describing its major changes. This is the main communication channel for developers to inform users about new features, bug fixes or improvements of existing functionalities. It is also the main channel for developers to convince users to upgrade an app to the latest release. Release notes, however, are curated by developers, and there are no constraints nor guidelines for them to provide an accurate description. It thus happens that they either leave it empty, or they copy the text from a previous release note, or even provide a default description that does not match the actual changes in the binary code. This practice can be more or less critical depending on what information is omitted. It can vary from collecting sensitive data without clearly mentioning it, requesting unnecessary permissions, to more harmless changes such as changes in the monetization policies, small UI tweaks or performance improvements.

An example of a clear mismatch between the release note and the actual code changes comes from the popular Whatsapp application at the end of May 2019. At that point in time, a serious security breach was found in the application, and Facebook issued a security advisory [1] urging users to update their app. The advisory mentioned that iOS users should have updated to v2.19.51 in order to have the vulnerability removed. However, looking at the Whatsapp's version history in the App Store (Figure 1), we can see that the message for the update mentions changes regarding how users can view stickers. There may be different reasons why Facebook

Figure 1: Inaccurate description of a Whatsapp release with a major vulnerability fix in iOS.

developers did not explicitly mention the real reason why users should update their app. One could be that users tend to rush to update when new features are available, but usually postpone a security update because it does not give immediate gratification. Another reason could be that developers simply forget to update the release description, and keep a previous one. Nevertheless, wrong descriptions in release notes can seriously confuse app users and mobile app analysts, as they are completely misaligned with reality.

In this paper we present ReChan, the first attempt to automatically detect a mismatch between the natural language release note of a mobile app and the actual changes in the code. ReChan is meant to be used by mobile app market managers to ensure the quality

and veridicity of release notes even when developers do not resort to automated techniques to generate them [2, 3].

We limit the scope of this work to Android apps, since code analysis of the binary file is more accessible than other platforms (e.g. iOS).

Concretely, ReChan automatically analyzes the natural language descriptions of what has changed with a new release of an Android app, and automatically classifies them. In a second step, ReChan analyzes the actual changes in the app binary code to reveal mismatches, which are often due to omissions in the release descriptions.

We retrieve a dataset of 29,647 releases from the Google Play store, published between January and mid June of 2018 to evaluate ReChan. Among many solutions to automatically classify release descriptions, specifically, a rule-based approach, a solution based on binary relevance with two classification algorithms, and an innovative approach which combines short text topic labeling with binary relevance, our evaluation shows that the best approach is the rule-base strategy.

ReChan implements different static analyses to detect type of changes such as minor bug fixes, UI improvements and new features, and reports when the classification produced by the binary analysis differs from the outcome of the natural language analysis.

Experiments show that developers tend to correctly report changes due to bug fixes and new features, but omit changes that affect the list of requested permissions, the UI and other content that the app uses.

The main contributions of this paper are the following:

- We present a taxonomy of release notes of Android applications.
- We present a novel technique to automatically classify the description in natural language of a mobile app release.
- We present a combination of static analysis techniques that can detect what has actually changed in the binary of an Android app compared to the previous release.
- We evaluate the precision and recall of our binary analyses on the manually crafted ground truth of 30 releases of 3 open source apps.
- We evaluate the technique on a dataset of over 12K release notes.

The remainder of the paper is structured as follows. Section 2 presents our taxonomy of release notes, Section 3 describes each component of the technique, and Section 4 presents the dataset and the results of the evaluation. We conclude summarizing the related work in Section 5 and conclude the paper.

2 WHAT'S NEW? ANDROID APP RELEASE NOTES

App stores, such as the Google Play for Android and the Apple store for iOS, are the main interface for users to find and install apps on their mobile devices. In some cases, as it is for iOS devices, they are the *only* viable channel for users to install software on unrooted devices. While there is a plethora of research studies that show how app store managers should improve the quality, security and privacy of mobile apps on official stores [4], there are only a few studies on the quality of the metadata that developers produce along with their apps.

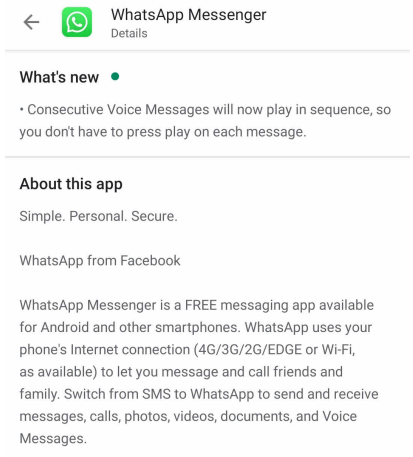

Figure 2: Sample description and release note in Android.

In Android, when developers publish a new release of their app, they can update the general description of the application, and they can specify what changes affect the new release compared to the previous one. Fig 2 reports an example of a description together with the release note of an Android app. In this example, the release note of Whatsapp mentions a change in an existing feature, specifically on how multiple voice messages would be played in a chat.

Release notes may be talking about different reasons for changes. Previous taxonomies of release notes consider "new features", "bug-fixes" and "improvements" as the only reasons [5]. Since to the best of our knowledge there is no prior work on classifying release notes specifically for mobile apps, we manually analyze several real samples to define a more suitable taxonomy. Concretely, we manually inspect 1,200 release notes produced in 2018 regarding 1,000 different Android apps. We follow an iterative approach to define the taxonomy: first, two authors independently analyze and label the first 300 samples, identifying some classes of release notes. Upon the agreement of all the authors on the relevance of each class, we restart the process until we could not find any other relevant class. Table 1 reports the nine topics that cover the taxonomy used in this paper. Together with an assigned name, in the second column, we present a brief explanation of our taxonomy in the last column.

Name	Description
INITIAL	First release of the app
ANDROID	Fixes/Features regarding a new Android version
PERMISSIONS	Changes in the list of permissions requested by the app
CONTENT	Changes in resources used by the app. data/policies/resources
BUGFIXES	One or more fix to the code to address an existing bug.
IMPROVEMENTS	Major changes to existing features
FEATURES	New functionalities added
UICHANGES	Changes in the user interface (e.g. new layout, new icon set etc.)
NON-FUNCTIONAL	Non functional optimization to the code involving Performance/Security/Code

Table 1: Taxonomy of release notes in Android apps

The first class, INITIAL, represents release notes that simply state that this is the first public version of the app. The description is usually little informative and very generic. Class ANDROID, instead, includes release notes that explicitly state that the update is about supporting a specific new Android release. We observe several release notes of this type when a new major version of the Android OS is released. PERMISSIONS class comprises release notes that explicitly state that there were minor or major changes in the list of permissions that the app requests, while CONTENT includes all changes related to resources of the app (e.g. language files, local application data) without changes in the code. We then have three classes that detect changes in the code but at different levels. Class BUGFIXES covers release notes that mention only minor fixes to avoid known problems in existing functionalities. IMPROVEMENTS covers release notes that mention major changes to existing functionalities, while FEATURES covers descriptions of new functionalities that the new release offers. Class UICHANGES involves description of changes in the user interface (e.g. new icon-set, new style, etc.). The last class covers release notes on NON-FUNCTIONAL changes. This includes descriptions of optimization or fixes of performance issues or security vulnerabilities.

Most release notes do not fall in a single category. It is in fact very common for developers to release a new version of their apps with improvements or new features, and at the same time fix several known bugs. In the next section we explain how ReChan can automatically classify release notes in natural language according to this taxonomy, and can automatically identify mismatches based on the actual changes in the code.

3 THE RECHAN TECHNIQUE

The goal of ReChan is to report mismatches between the release note written in natural language of an Android app and the actual changes in the binary code with respect to the previous version. At a high level, ReChan works as listed in Figure 3. The technique takes as input the natural language release note of an app, its corresponding binary package in APK format, and the previous binary release. The natural language analyzer checks the description of the release note, and classifies it according to the taxonomy presented in Table 1. Similarly, the binary analyzer compares the binary release associated to the release note with the previous release binary. It identifies the main differences, and classifies the changes according to the same taxonomy. Once these two independent analyses are done, ReChan checks the classification outcomes and reports mismatches. We now proceed to explain the technical details of each component.

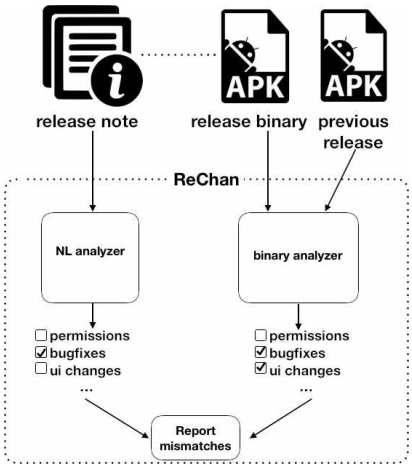

Figure 3: Overview of ReChan

3.1 Natural Language Analyzer

The goal of the natural language analyzer is to automatically classify a release note written in natural language according to the taxonomy of Table 1. As already anticipated, each release note could fall into multiple categories. For instance, "*In-App Linking between Micromedex Drug Reference (or Micromedex Drug Info – Mobile) and Micromedex IV Compatibility. Updated functionality that allows for streamlined acquisition of clinically relevant information between applications. Functionality allows navigation between applications with carrying forward of initial drug(s) search*" should belong to both FEATURES and IMPROVEMENTS classes. We initially considered the idea of parsing each sentence separately and classifying it to unique class (in the example above the first sentence would be classified as IMPROVEMENTS and the second one as FEATURES). However we quickly realized that this approach is not feasible, since very often developers write very short descriptions indicating multiple topics. "*Bugfixes and improvements*", for instance, is a commonly used description, and the same sentence describes two of the categories that we identify in Table 1.

To deal with this multi-classification problem, we implement and compare different techniques. Before that, ReChan applies some common pre-processing steps.

Each description goes through a data cleaning phase (to remove stopwords among other things), and descriptions that are in languages other than English are discarded. We then prepare the data to be in the right format for whatever classifier component ReChan has to use. We now describe each phase in details.

3.1.1 Pre-processing Steps. ReChan receives one JSON file per release note. Before processing each description, we need to make sure that it contains a valid "what's new" section (developers may leave it empty). We also clean text fields of each entry from HTML tags.

The *Language Detection* component identifies the main language of each description. We resort to two libraries [6, 7] for this purpose, since our first evaluation shows complementary abilities. We assign a language tag only if there is an agreement between the two, otherwise we consider the language as unknown. Our current implementation does not support multi language descriptions. However, in the future we could include at this stage the logic that divides the description into multiple paragraphs and parses them separately. ReChan currently only supports the English language. We thus filter out any entry that we do not recognize as English text.

ReChan includes a component for *Data Preparation* which transforms release notes into a representation that can be processed by specific classifiers later in the pipeline. Regardless of the strategy, this component transforms the string representation of the textual

description into a higher level abstraction named Document. We use this abstraction to decouple any natural language processing library that operates on the texts from the rest of the implementation. We include convenience methods for tokenization, stemming, word filtering, lemmatization and sentence splitting. We do not use a standard stopwords list for word filtering, since some stop words or non alphabetic tokens need to be excluded while others not.

The data preparation stage can also include a vectorization step, to transform the data into a numerical representation needed by some of the following strategies. The vectorizer has to follow the design of sklearn[1] (i.e. implement fit() and transform() methods). In our implementation we use four such vectorization strategies: Term Frequency, TF/IDF, LDA and Biterm.

3.1.2 Classifier Strategies. ReChan includes three strategies to classify text in the categories of our taxonomy: *rule based*, *Naive Bayes* and *SVM*, and *topic modeling*.

Rule Based Topic Labeling. The rule based approach mimics the process that a human annotator uses to observe patterns in the texts. We define a set of rules through several iterations involving discussions of all authors. We started by observing the most common words associated with each manually classified release note to identify the taxonomy, and we include these words in the rules (this process resembles keyword search). We soon realized that pattern matching on individual words could not achieve good performance, and this is when we introduced operator fields in our rule format. Each rule returns True/False based on the application of the operator on the text and on an auxiliary list of operands. Table 2 presents the types of operator and their logic.

Rule Name	Operand1	Operand2	Description
INCLUDE	Text	list(words)	Returns True if any of the words exists in text
EXCLUDE	Text	list(words)	Returns True if none of the words exists in text
OR	Text	list(rules)	Applies the rules on text and returns True if all of them are true
AND	Text	list(rules)	Applies the rules on text and returns True if any of them is true
NOT	Text	rule	Apply the rule of text and returns the result negated

Table 2: Rule operators

Adding operators increases the performance of ReChan significantly based on our evaluation. We improve it even further thanks to two more enhancements: our rules can restrict their scope to single sentences. This feature is beneficial to reduce the number of false positives when looking for word co-occurrences. Moreover, our rules support the feature to specify the part of speech for operands. This allows to deal with cases where a word can have different meanings based on its part of speech (e.g. issue means a problem when it is a NOUN and the action of supplying an item when it is a VERB). Using this mechanism we defined a list of 72 rules. [2]

[1] https://scikit-learn.org/stable/

[2] We provide the full list of rules at https://tinyurl.com/rechan-ground-truth

Naive Bayes and SVM. The rule based approach has some limitations when it comes to generalization. To explore alternative solutions we turned our attention to design a completely automated solution.

The first method we propose leverages classical text classification algorithms, Naive Bayes and SVM, to assign labels to release notes. The application of these algorithms in our context is not straightforward, because they require a single label per item while our release notes may belong to multiple classes. To overcome this limitation, we train independent binary classifiers for each class.

Differently from the rule based approach, Naive Bayes and SVM classifiers need to be trained before they can be used. For this we create a separate task for training, using part of the labeled dataset.

Topic Modeling. We also use topic modeling algorithms to extract latent semantics from cleaned release notes, and then apply classic classification methods on the resulting topical distributions. We support two topic modeling methods: LDA [8] and the Biterm Topic Model[9] (which is suppose to perform better for short texts). Our expectation was that topic models would identify finer topics or hidden relations in our data, which would better fit the classifiers and consequently increase the prediction performance.

Regardless of the strategy used, the natural language analyzer classifies the release note according to one or more classes defined in our taxonomy.

3.2 Binary Analyzer

The goal of this component is to analyze the binary package of the Android app, and compare it with its previous release. We implement one or more specific static analyses to identify the changes that we consider in our taxonomy reported in Table 1.

3.2.1 Class 1: Initial. In our dataset we have a few descriptions of release note that simply say that this is the first version of the app. This class is a corner case for our analysis, since in principle there should be no previous binary file to compare the application against. ReChan simply checks from all the sources we have access to, if there is any version of the application prior to the one under analysis. If it can find a previous release, then it flags the release description as inaccurate.

3.2.2 Class 2: Android. This analysis aims to assess if the release note addresses a change in the Android platform that the application targets. We perform this analysis by simply checking the information listed in the manifest file of the application. We report that this is the case when we identify a change in the targetSdk, minimumSdk, or maximumSdk fields of the manifest.

3.2.3 Class 3: Permissions. To identify changes that involve permissions, ReChan simply analyzes the Android manifest of the application. It extracts the list of requested permissions, and it compares it against the previously declared ones. It reports any relevant change in the list of permissions. ReChan ignores the ordering of the permissions and duplicates since they do not have any semantic effect on the application.

3.2.4 Class 4: Content. To identify changes in the content of an application we focus our analysis on the supporting files that the

application includes. Files like sounds, images and language translations usually reside in the res/ and assets/ sub-folders of the APK package. However, in these folders there are also user interface related files like layouts, animations, color palette definitions, which are out of scope for this class.

The analysis thus consists in collecting all the files in the res/ and assets/ sub-folders that are not XML binary files (since these are the resources used in the user interface). For each file we compute the sha256 hash of the content. By hashing the files we can see if a file has changed even if the name is the same or, if a file is the same despite having a different name. We compare the set of hashes in each version of the application to determine whenever there has been a change in the content or not.

3.2.5 Class 5: Bugfixes. For finding out if a release is a bug-fix, we first need to calculate the ratio of change in the code that the two versions have. To achieve this goal we include a component in ReChan that calculates the ratio of change between the code at the bytecode level and the new classes that exist in the newer version of the application. This component serves as the foundation for the 3.2.6 and 3.2.7 classes too. To implement this analysis we take inspiration from a blog post by Quarkslab[10]. In this work we expand on the original idea by first filtering out third party libraries with Libradar [11].

The first step in our implementation is to filter out the business logic from the standard library and third party libraries that might be in the application. Since developer's code is bundled with third party libraries in the dex files, we use Libradar for detecting third party libraries. We remove all classes that fall into the package name of a known third party library detected by Libradar.

Once we have the list of classes that belong to application code, we group them by their package name. Since developers usually obfuscate Android apps on release, ReChan uses a heuristic to detect if a class is obfuscated or not. Then, obfuscated classes are grouped in the same package. The rationale behind grouping by package name is that classes are unlikely to change their location unless the developer makes a significant refactoring of the code.

For each class in the application we calculate its simhash[12]. This hash is more stable to small changes and will yield a similar hash if two classes are similar. We calculate a 128bit hash divided in four 32bit hashes. The first hash refers to information about the class itself; public, private, abstract, final, if it is an interface, etc. The second hash refers to the methods implemented in the class; the number of methods, the type of method, the length of the code. The third hash is similar to the second one, but is on class fields. The forth and last one is the hash of the bytecode.

The end goal of this simhash is to calculate the nearest neighbors in the previous version for each class in the newer version. This selection gives us a coarse approximation of what classes are the most related.

For each group of neighbors we then calculate the distance between the bytecode contained by each class. The neighbors are ranked by this distance and we pick the closest one as the corresponding class in the previous version. To calculate the distance, we use the Levenshtein distance of an abstracted version of the bytecode. We cannot use reliably the raw bytecode for this task, because instructions may use different registers in different compilations.

In the original implementation the bytecode was categorized in semantic families. For example, the invoke-* instructions were categorized as *invoke* operations, and arithmetic operations were all abstracted to the same *arith* operation. For our particular case this broad categorization meant that, for instance, if a bug-fix in a method involves a change between < and <=, we would not be able to see it. For this reason, our abstraction only removes arguments and registers from each instruction, and applies the distance on the instruction's names.

The last step involves normalizing all the numbers and returning the aligned classes with their similarity score and the overall ratio of change between versions. We align classes that we could not align to any other class in the previous version to a dummy entry, and set a similarity of 0% between the class and the dummy.

For a bug-fix, the expected ratio of change between versions should be low. Thus, we consider any app that has a ratio of change below 0.4% of change and is greater than 0% to be considered a bug-fix.

3.2.6 Class 6: Improvements. When a new release includes improvements, the code should, intuitively, contain more changes compared to just a bug-fix. For instance, rewriting a whole method to improve its functionality will result in a bytecode very different from its previous version. We consider an improvement of the code base also the case adding new methods or new classes, without exactly adding new features. Since we expect an improvement to have more changes in the code compared to bug-fixes, we resort to the same analysis we implemented to identify bug-fixes, but we consider a release to belong to this class when the rate of change is above 0.4% of change.

3.2.7 Class 7: Features. We consider a new version to contain new features if it has new functionalities that the user can see and interact with. For instance, if there are new buttons, menus or other UI elements that lead to new code that did not exist before. Functionalities in Android are often implemented as separate Activities.

ReChan implements a diff analysis to identify new features using two checks: 1) it reports differences if the new version contains a class that inherits from Activity which does not appear in the previous release of the application, or 2) there are changes in the layout files included in the application.

3.2.8 Class8: UI Changes. In Android applications the user interface files are written in XML, and then compiled to a binary XML format. These files are included in an internal folder res/, separated in folders by their type (layout, animation, menu, etc) and other properties like Android version or screen resolution. For instance, layout-sw600dp-v21 will contain layouts that are used in the version 21 of Android and screens with 600dp. In order for the developer to change the UI of the application, he or she must edit these files.

Therefore, we expect a diff between version to contain changes in the user interface if it has changes in those files. To run this check, we take all the binary XML files that are in the res/ folder, and calculate the sha256 hashes of each of them. Then, if there are different hashes between the two file lists, we consider that a user interface file has changed and therefore we consider the release change as belonging to this class.

3.2.9 Class 9: Non-functional. Some release may address non functional changes. They may include security patches (as the Whatsapp example presented in Section 1), or they may improve the performance of the application. ReChan currently cannot identify this class of changes, but rather classifies them as improvements or features. The challenge of automatically classifying these changes is that the analysis should consider the *semantic* of the changes, and this is beyond the scope of this paper.

3.3 Report Mismatches

ReChan compares the classification done on the description in natural language and on the binary and it analyzes mismatch. Our binary analysis assumes that when there are new features there is also a visible change in the UI. Thus it does not report a mismatch if the release note does not mention UI changes when it mentions new features.

4 EVALUATION

We evaluate ReChan according to the following research questions:

- RQ1: How accurate is the natural language analyzer in classifying release notes? Relying on our manually labeled samples, we compute precision and recall, and we compare the performance of all classifiers that ReChan implements. We also report the distribution of release notes across topics on the whole dataset.
- RQ2: How accurate is the binary analysis component in classifying the changes across two releases? Relying on the manually crafted ground truth of 30 releases of 3 open source applications we compute precision and recall and perform a qualitative analysis of the results.
- RQ3: How often does binary analysis produce a matching result with the natural language classifier? This study aims to quantify how analysis on code and on the description agree on the type of release.
- RQ4: What cases can ReChan reveal when analyses produce a mismatch? This study is a qualitative analysis of some of the cases we have in our dataset.

We now discuss the dataset, and later present each research question.

4.1 Dataset

For the evaluation of ReChan we need a dataset of applications with release notes, corresponding binary file and the previous release of the same application. For this purpose we used Tacyt, a large database owned by ElevenPaths (Telefonica). This database collects information about Android and iOS application across several stores, and it stores binary files and metadata.

The platform offers a query language that allows filtering the applications according to certain criteria. We set our filter to obtain applications that published releases in 2018 and that were published in the Google Play store.

After running the queries we obtain a dataset of 29,647 releases from January of 2018 to mid June of 2018. Filtering out the entries with no message in the "recent_changes" field (i.e. the "What's new message") or entries without binaries, reduces the dataset to 18,018 releases. We further remove HTML tags from the messages, and apply the language detection process described in Section 3.1.1. By filtering out applications with non English release notes, we reduce the dataset to 12,706 releases.

Topic	Train set			Test set		
	PR	Recall	F1	PR	Recall	F1
ANDROID	0.99	0.92	0.95	0.93	0.82	0.87
BUGFIXES	0.99	0.98	0.98	0.96	0.99	0.98
CONTENT	0.75	0.53	0.62	0.88	0.41	0.56
FEATURES	0.80	0.14	0.24	0.55	0.01	0.22
IMPROVEMENTS	0.89	0.67	0.77	0.97	0.66	0.79
INITIAL	0.83	0.79	0.81	0.67	0.50	0.57
NON-FUNCTIONAL	0.92	0.89	0.91	0.87	0.92	0.90
PERMISSIONS	0.93	1.00	0.96	0.50	1.00	0.67
UICHANGES	0.92	0.84	0.88	0.91	0.77	0.84

Table 3: Classification results using Rule Based method

From this dataset we randomly sample 1,200 entries, and label them manually with topics, as explained in Section 2. We further divide the labeled dataset in a training dataset (1,000 entries) and a test dataset (200 entries) to use with classifiers that require training. We look for previous binary releases in our reference database. Out of the 1,200 manually labeled cases, we only found 103 cases. All these cases, though, are carefully checked, and we have high confidence that there are no (or very few) releases between the two samples we use for our analysis.

4.2 RQ1: Accuracy of Natural Language Classification

We use the 72 rules that we produced to label the entries with the predefined categories. We built the rule set by iteratively analyzing the training dataset of 1,200 entries and the classification results on that very same set. To avoid overfitting the data, we refrained from running the classification on the test set until we considered the rules completed. We analyze the performance of this solution by running the classification and looking at the precision, recall and F1 scores for both the train and the test set, and we report the results in Table 3.

The performance of the rule based strategy is surprisingly good for most of the categories. Furthermore, checking the difference between the results on the training and test set, we see that the solution did not overfit the train dataset.

We need to mention that the results obtained on the test set for the categories "INITIAL" and "PERMISSIONS" may not be reliable due to the small number of positive samples in the set. Second, we can observe that for very specific categories such as "ANDROID" and "BUGFIXES", the performance is higher than for the more general categories. This is also because developers tend to use clear keywords (e.g. Android release or bugfix) almost always in these type of release notes. Third, the results for the "FEATURES" category are much worse than for the others. This is because descriptions in this category are application specific, and it is hard to define general rules without overfitting on the data. Developers, in fact, tend to describe the new feature (as in the example in Figure 2), rather than using general terms such as "New features'. 'Lastly, we note the very good performance on the "NON-FUNCTIONAL" class which, even though is quite a broad topic, is represented in the texts by few

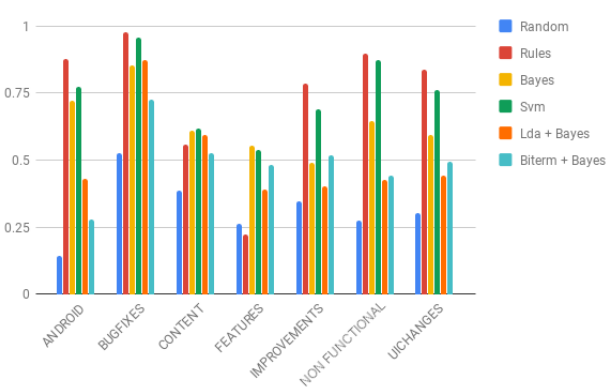

Figure 4: Performance (F1 score) comparison of classification algorithms

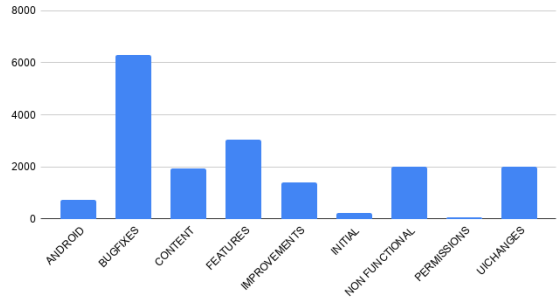

Figure 5: Distribution of release notes into categories

common expressions (e.g. "performance optimizations", "increased speed").

The rule based is the best performing classifier overall. We now briefly comment on the other solutions we evaluated.

Naive Bayes and SVM. The vectorization of the data was done using term frequency for Naive Bayes and TF/IDF for SVM. We tried different variants of these algorithms to select the best fitting one for our situation. For Naive Bayes we compared the performance of Multinomial NB with Complement NB by running 10 iterations on the training set and selecting 80% of the data for training the algorithms and 20% for testing. Results show that Complement NB performs slightly better than Multinomial NB.

For SVM we tried both Linear SVM, which is often recommended for texts, and One Class SVM which was used in prior works [13]. Unfortunately, the results for One Class SVM were completely unsatisfactory with the classifier yielding all entries as negative for most of the classes. Once the optimal configurations for both classifiers were selected, we trained them using the training set and run them against the test set. Because the number of examples for the "INITIAL" and "PERMISSIONS" topics was too small, and the classifiers could not learn any relevant models, we removed those topics from the analysis.

The results show a slightly better performance for the SVM classifier compared to Naive Bayes for most of the topics. For "BUGFIXES" and "OPTIMIZATIONS" the performance is higher than for the others, probably because there are few terms used to express the changes in these categories. Unlike for the rule based approach, the "ANDROID" category does not obtain such high results with the automatic classifiers.

Topic Modeling. Since the dataset contains short texts, which are not adequate for data classification, we tried to enhance the performance by using a topic modeling method to transform each text from a distribution over a vocabulary of words to a distribution over a set of topics. There are prior works in literature that resort to this technique. Hong et al [14] mention using topics from text as supplementary features in text classification, while Phan X. et al [15] extract auxiliary topics from a large "universal corpus" to enhance the performance of their algorithm. In our context there are two main reasons why this solution is potentially interesting: a linguistic one and a statistical one. From the statistical point of view, we expect that topic modeling would group common features of similar texts and would allow the classifier to operate on a smaller and better divided space. From the linguistic point of view, we expect that topic modeling would produce a finer grained distribution of topics, compared to our high level taxonomy, and that each of these new topics will be a subtopic in our taxonomy. For topic modeling we trained LDA[8] and Biterm[9] models using the entire dataset of 12,706 entries, from which we extracted the test set. These algorithms require that the number of topics is set a priori. We tried multiple options in the range 10-100. We did not explore values over 100 topics because in our manual exploration of the texts, we saw that they are quite clustered around some general themes and, even though more specific topics than ours can be found, their number cannot be very big.

Naive Bayes performs better than SVM when combined with both topic modeling algorithms. Given the reduced dimensional of the data, we suspect the reason is that NB can learn faster. In terms of classification performance, we see that relatively good results are obtained for "BUGFIXES", while for other categories the results are not as good as expected. Probably the reason why the "BUGFIXES" category performs better is that it is very clearly defined in the original texts. Comparing the results obtained with LDA and Biterm we can see that they are similar, with slightly better scores obtained by LDA. One reason for the poor performance of Biterm can be the small number of iterations (100) we used when we ran the algorithm. We had to select this small number because the Biterm library we used [16] is very slow and does not allow for parallelization.

Figure 4 summarizes the results with a comparison of the F1 scores of all the solutions on each class. We use the best strategy among our solutions to classify the whole dataset of release notes. Figure 5 reports this information.

4.3 RQ2: Accuracy of Binary Analysis Classification

In order to evaluate the accuracy of the heuristics we implement in the binary analysis component, we manually craft the ground truth of 30 releases by looking at the actual changes in the code between two subsequent releases. For this purpose we look for

open source applications on F-droid satisfying the following conditions: 1) the repository should have at least 10 releases, 2) each release should specify the associated commit and should have the corresponding pre-compiled binary from developers, in order to avoid bias in building it ourselves, and 3) the app should be mostly in Java or Kotlin, since our analyses support only Dalvik bytecode and not native code. Following these criteria we select three apps: AnkiDroid, a flashcard-based study app, Markor, a todo and notes editor, and MTG Familiar, an app that offers utilities to play the Magic card game.

For these apps we select the most recent release and keep it only if both the selected and its previous release have binary files. When this is not the case, we discard both releases and continue with the preceding ones. We thus obtain 10 release deltas for each app, and for each of them we analyze each commit among releases to classify the type of change according to our taxonomy. We finally use our ground truth [17] to compute the accuracy of our analyses for each class. Table 4 reports the results.

Class	Precision	Recall	F-score
Permissions	1.00	1.00	1.00
Android	1.00	1.00	1.00
Content	0.96	1.00	0.98
UI-changes	1.00	1.00	1.00
Bugfixes	0.50	0.29	0.37
Improvements	0.82	1.00	0.90
Features	0.78	1.00	0.88

Table 4: Precision, Recall and F-score of the binary analysis classification.

We note that the performance of ReChan is very good for most classes, except for bugfixes. We now discuss some insights of the manual analysis of all the false positives and false negatives against the ground truth.

ReChan seems to miss a change regarding permissions. We investigated the problem and we found out that the binary release actually did not have any change in the list of permissions, and thus is in accordance with ReChan, making recall 1 for this class. This is likely an example of a hotfix when producing the release binary which is not reflected in the repository. ReChan reports one false positive for class content. This is because it detects that the changelog file changes, and developers included this file in the resources. ReChan is less accurate when it comes to distinguish whether changes in the code address new features, improve existing code or simply fix bugs. The worst performance clearly affects the bugfix category, where ReChan suffers many false positives and false negatives. A closer look at the data reveals that 75% of the false negatives are in the MTG app, which has a single developer who tends to create less releases with multiple changes and multiple bugfixes at once. For all these cases ReChan flags these bugfixes as improvements, since the portions of changed code is very large. Despite the bad performance of distinguishing large bugfixes from improvements, we can say that our heuristics are very effective.

4.4 RQ3: Reporting Mismatches

We run the binary analysis of ReChan on the binary pairs for which we have the ground truth regarding the release note. Since we do not have the ground truth on the actual changes in the code, in this study we just report the mismatches with the classification of the release note, taking this latter as the correct one, and we analyze them in detail in the next section.

Table 5 reports the results for each category according to this classification:

- AP (Agree Positive): Binary and natural language analyses *agree* that the release *belongs* to a specific category.
- AN (Agree Negative): Binary and natural language analyses *agree* that the release *does not belong* to a specific category.
- MB (Miss Binary): The release note says the app belongs to a category, the binary analysis does not.
- MRN: (Miss Release Note): The binary analysis says the app belongs to a category, the release note does not.

MB could actually be errors in the release note descriptions (i.e. the developer describes the release note as a bugfix, but it has essential changes that go beyond that). MRN, instead, could actually be omissions of the developers in describing the release note (as the Whatsapp example in Figure 1).

	AP	**AN**	**MB**	**MRN**
Permissions	0.97%	66.99%	0.97%	31.07%
Android	6.80%	60.19%	4.85%	28.16%
Bugfixes	51.46%	17.48%	2.91%	28.16%
Improvements	13.59%	35.92%	9.71%	40.78%
Content	26.21%	10.68%	0.00%	63.11%
Features	24.27%	10.68%	1.94%	63.11%
UI changes	17.48%	15.53%	3.88%	63.11%

Table 5: Percentage of matches and mismatches when compared to the ground truth of the release note. MRN are likely omissions in the descriptions.

From the low AP rate and the high MRN rate, we can deduce that developers usually do not mention that there has been a change in the *permissions* list. In general this practice of not clearly describing the reason of the change in the code affects classes *Android* (1 in 4 release notes do not mention these changes in the description), and especially *Content*, for which nearly 80% of the new releases include changes in the supporting files that are packaged with the applications, but only 26% of the developers mentioned it in the release note.

Bugfixes is the most common class according to our binary analysis, with a 51% of cases marked as containing *bugfixes*. However, 79% of the analyzed applications contain changes in the bytecode small enough to be considered as bugfixes. One third of those applications are not marked as bugfixes in the release. We believe the reason for the mismatch could be that in those cases developers introduce improvements in the code, but such improvements did not change the bytecode much, and therefore our heuristics fail. Another reason could be that the changes happen in the native code, which we do not analyze.

Half of the analyzed applications contain mayor changes in the code, which is what we consider an *improvement*. The majority of the cases are not marked as containing improvements however. This can be due to developers assuming that improvements are intrinsic to releasing a new version. Or likely developers add new functionalities that our binary analysis considers improvements, but developers consider as new features.

80% of the cases contain changes in the files related to the *UI*, while only around 17% of release notes mention changes in the user interface, either functional or cosmetic changes. This case is very similar to the content class, and we believe that developers simply do not consider necessary to mention changes in the resources folder unless it affects what the user actually cares about: new functionality (15% of the cases) or new content that the user interacts with (26% of the cases).

4.5 RQ4: Qualitative Analysis

We perform a qualitative analysis of most of the cases that we analyze with ReChan. This analysis allows us to understand in more detail the reasons of the mismatches, i.e. that either the binary analysis is not able to detect relevant changes or it detects changes that are not mentioned in the release note.

Some of the applications are tagged as BUGFIXES, FEATURES or IMPROVEMENTS but we could not see any of that in the output of our static analyses. After manual inspection, we found that these cases were applications programmed in languages different than Java/Kotlin. Most of those cases were C/C++ games, but we also found a Xamarin application. Since our tool only supports Dalvik bytecode, the static analysis missed these relevant changes. On the other side of the spectrum, a hybrid application developed in JS changed the version of the underlying framework between releases because of a security vulnerability reported in the framework. This major change confuses our static analysis tool, because it yields big changes in the application, but the release note only mentioned the update because of a security issue. Some of the applications that are marked as containing Android related changes were not detected as such. After manually inspecting the release note and both versions, we found that most of the applications made changes in the code related to the change in the permission model in Android 6. Other applications introduce changes related to Android Oreo. One in particular introduces an Adaptive Icon, a new feature in Android Oreo, that needs to be implemented as a XML file in the res/ folder. This change would be detected as the UICHANGE class instead of the correct ANDROID class. We detect cases where the release note advertises changes in the UI or claims new features, however the static analysis could not detect these cases. These releases implement changes that *dynamically* change the UI. This means that unfortunately we cannot detect them with static analysis. Other releases are labeled as IMPROVEMENTS and BUGFIXES. In the cases where the rate of change between releases is too high, ReChan will classify as BUGFIXES too, beside IMPROVEMENTS. We also have opposite cases where changes are too small, but developers believe these are actual IMPROVEMENTS. Another limitation of our static analysis on the bytecode is obfuscation. Despite being resilient to obfuscation, we assume that both releases are obfuscated. We found one example where obfuscation was introduced between the releases we analyze. This causes package names to change completely, and confuses the clustering step of our analysis. We tried to disable the clustering step for this case, and we confirm that ReChan is able to correctly match the classes, albeit more slowly. For the future we plan to use a Merkle tree as data structure to group packages based on their structure rather than relying on their names. Out of the 1,200 release notes that were manually labeled, 19 of them were classified as initial release of the application. Most of them were indeed the first version of the application. However, one particular case, an application called MYVIDEO has 10 prior versions to the one we analyzed. The release note for that release is "init". We also check the prior release notes to discard the possibility of a sloppy developer who did not changed the release note since the inception of the application. That is not the case, since the other release notes contain more meaningful messages compared to the one in our dataset. Of the applications that we statically analyzed there are 32 applications with changes in their permission list, without mentioning anything in the release note. Some are harmless permissions, but there are applications requesting access to the location, sms, audio and video record, and other dangerous permissions. One particular case request the permission WRITE_SETTINGS that according to the Android documentation should not be used by normal applications [18].

5 RELATED WORK

There is a large body of work that involves the analysis of mobile app stores [4]. We now briefly discuss related work regarding the analysis of natural language artifacts, diffing binary files, and the analysis of releases.

Analysis of Android Natural Language Artifacts. In [19] the authors aim to provide more personalized app classification and hence better app search experience for the users. The approach relies on contextual enhancement of each application and a Maximum Entropy classifier to divide applications into categories.

With an average of 22 reviews per day for each application [20] user reviews build up to a consistent source of information ready to be analyzed for the benefit of users, developers and vendors. Thus many papers analyze these artifacts for instance to identify frequent complaints (functional errors, feature requests, app crashes) and most negatively impacting complaints (privacy and ethical features, hidden costs) [21]. Guzman et al [22] try different classifiers to sort reviews according to a defined taxonomy. More complex solutions try to gain a deeper understanding of the reviews. Custom rule based solutions can be developed to collect specified details such as in [23] where 237 rules were used to extract features and features request. Or even mixed solutions such as in [24] where a combination of rules to process natural language, sentiment analysis and, statistical information from text is used to group reviews in the categories of a predefined taxonomy. While the previously presented studies use predefined categories or hardcoded rules, others use topic modeling solutions to allow the categories to be driven by the data. AR-miner [25] provides an automatic solution for information extraction from app store's user reviews, which can help developers and managers get an insight on the problems that need to be addressed. [26] implements a similar approach to

extract information at micro level (one review), meso level (app related reviews) and macro level (entire store).

Beside user reviews, there are few studies dealing with app descriptions. Having a clear description of what an application does will help elevate the users concerns, and hence will bring benefits for both the users and the developers. Furthermore, potential malware application can be identified since these will avoid describing their real behavior [13, 27–29].

Binary diffing. Binary diffing aims to extract the similarities and differences between two binaries. It is used in scenarios where source code is not available like patch diffing for exploit generation or malware analysis. The biggest challenges in binary diffing are code obfuscation and compiler optimizations. Both techniques can modify the form of the binary to the point where a naive analysis fails. No matter how obfuscated a binary is, its semantics will be similar to a non-obfuscated version of the same program. For that reason, recent research has focused on extracting the semantics of the binaries. Luo et al [30] aim to detect software plagiarism by extracting the semantics of each basic block as a set of equations and solves the similarity using a theorem prover. This similarity is then used to find the *longest common sub-sequence of semantically equivalent basic blocks*. This technique shares with ReChan the idea of matching the sequence with a fuzzy matching algorithm, but our tool focuses in sequences of Dalvik instructions instead. Bourquin et al [31] use a combination of the well-known BinDiff algorithm introduced by Halvar Flake in 2004 [32] with the Hungarian algorithm for bipartite graph matching [33]. The authors report that matching accuracy improves significantly compared to only using BinDiff. Some techniques, like function inlining, alter the *inter-procedural graph* or the *intra-procedural graph* of a binary. Tools that only focus on the *intra-procedural graph* can see its effectiveness reduced because of this alterations. Ming et al [34] propose a technique that relies on taint analysis and automatic input generation for finding semantic differences in the *inter-procedural graph*. This semantic properties can overcome such obfuscation techniques because they cross the procedural boundaries and thus, they are not affected by the shape of the CFGs. Programs compiled with different optimization options or different compilers can have very different binaries as a result. Egele et al [35] propose a dynamic analysis technique to overcome this problem. The technique emulates the functions of the target binaries and extracts the side effects produced in a controlled environment. Two functions are matched if they share similar side effects under the same environment. [36] presents another technique for avoiding obfuscation in binaries. It combines static and dynamic analysis to record traces of syscalls during the execution of a program. This trace is converted to a slice that represents the instructions that affect in any way the trace. From the slices the technique extracts a set of equations for each slice that are checked for equivalence using a solver.

Release Analysis . There exist studies on how the behavior changes across different releases of the same Android app [37–41]. Martin et al. analyze a large number of app releases and their corresponding reviews from users. They observe that over one third of the releases cause a change in user ratings [42]. Xia et al. instead, use machine learning techniques to effectively predict mobile app releases that are more likely to crash [43]. Last but not least, Khomh et al. show how shorter release cycles lead to better quality perceived by users [44]. Nayebi et al., aim to study the release practices in mobile development. Their analysis is based on surveys of developers and users rather than on actual data retrieved from app stores [45]. Moreno at al. focus on the quality of release notes as we do in this paper, but propose to automatically generate them from actual code changes [2, 3]. Their solution, however, can only be applied when source code is available, which is not the case for us. Mostafa et al. study behavioral backward compatibility of Java libraries, and among other findings they show that the majority of behavioral backward incompatibilities are not well documented in API documents or release notes [46]. This supports our finding that Android app release notes often omit important changes as well.

In our previous work we studied the release practice in mobile apps, comparing how developers produce releases on the Android and iOS platform [47]. That work, however does not analyze binary files, but mostly focus on release dates. Hassan et al. studied the release notes of 1,000 emergency updates on the Google Play store, and they found that developers rarely explain the reasons of the update [48]. They manually analyzed the binary changes to understand why developers urged to produce a new release, and they found 8 reasons, mostly associated to simple development mistakes. While our work is similar to theirs in terms of analyzing binary files and release notes in Android apps, our work is more general and automated, while their work is manual and focuses only on few specific releases. The same authors studied also bad updates in Android, and found that bad updates with crashes and functional issues are the most likely to be fixed by a later update. However, developers often do not mention these fixes in the release notes [49]. This finding again partially confirms what we observe in our study. Similarly, Roseiro et al. study same-day releases that are published by popular packages in the npm ecosystem. Their manual analysis of the existing release notes suggests that same-day releases introduce non-trivial changes, but mostly bug fixes [50].

6 CONCLUSIONS

We presented ReChan, a novel technique to automatically detect mismatches between release notes of Android applications and the actual changes in the code. We defined a taxonomy of 9 release categories by manually tagging 1,200 real samples, and we evaluated several solution to automatically classify release notes written in English. Our evaluation shows that a rule-based approach works better than classic classifier techniques and better than topic modeling. We then presented several specific analyses to detect different class of changes in the code. Despite the many limitations of the analyses we could get to interesting findings: Our evaluation shows that on a dataset of 12,706 Android apps, developers tend to correctly report changes due to bug fixes and new features, but tend to omit changes that affect the list of requested permissions, the UI and other content that the app uses.

ACKNOWLEDGMENTS

This work was partially supported by the Spanish Government's SCUM grant RTI2018-102043-B-I00, Grant RYC2020-030800-I funded by MCIN, the Madrid Regional project BLOQUES S2018/TCS-4339, and gifts from Facebook.

REFERENCES

[1] "Facebook cve 2019-3568," 2020, https://www.facebook.com/security/advisories/cve-2019-3568.

[2] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic generation of release notes," in *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, November 2014, pp. 484–495.

[3] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora, "ARENA: An approach for the automated generation of release notes," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 106–127, February 2017.

[4] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, 2016.

[5] S. L. Abebe, N. Ali, and A. E. Hassan, "An empirical study of software release notes," *Journal of Empirical Software Engineering*, vol. 21, no. 3, p. 1107–1142, June 2016.

[6] "Langid library," 2020, https://pypi.org/project/langid.

[7] "Polyglot library," 2020, https://pypi.org/project/polyglot.

[8] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, no. January, pp. 993–1022, 2003.

[9] X. Cheng, X. Yan, Y. Lan, and J. Guo, "Btm: Topic modeling over short texts," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 2928–2941, December 2014.

[10] "Android application diffing: Engine overview," https://blog.quarkslab.com/android-application-diffing-engine-overview.html, (Accessed on 08/23/2020).

[11] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *ICSE 2016: Proceedings of the 38th International Conference on Software Engineering*, Austin, TX, USA, May 2016, pp. 653–656.

[12] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *STOC 2002: Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, Montréal, Québec, Canada, May 2002, pp. 380–388.

[13] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, June 2014, pp. 1025–1035.

[14] L. Hong and B. D. Davison, "Empirical study of topic modeling in twitter," in *SOMA 2010: Proceedings of the First Workshop on Social Media Analytics*, Washington D.C., USA, July 2010, pp. 80–88.

[15] X.-H. Phan, L.-M. Nguyen, and S. Horiguchi, "Learning to classify short and sparse text & web with hidden topics from large-scale data collections," in *WWW 2008: Proceedings of the 17th International World Wide Web Conference*, Beijing, China, April 2008, pp. 91–100.

[16] "Biterm library," 2020, https://pypi.org/project/biterm/.

[17] "Rechan material," 2020, https://tinyurl.com/rechan-ground-truth.

[18] "Android documentation," 2020, https://developer.android.com/guide/topics/permissions/overview#special_permissions.

[19] H. Zhu, E. Chen, H. Xiong, H. Cao, and J. Tian, "Mobile app classification with enriched contextual information," *IEEE Transactions on Mobile Computing*, vol. 13, no. 7, pp. 1550–1563, 2013.

[20] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *RE 2013: Proceedings of the 21st Requirements Engineering Conference*, Rio de Janeiro, RJ, Brazil, July 2013, pp. 125–134.

[21] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, pp. 70–77, January 2014.

[22] E. Guzman, M. El-Haliby, and B. Bruegge, "Ensemble methods for app review classification: An approach for software evolution," in *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, Lincoln, NE, USA, November 2015, pp. 771–776.

[23] C. Iacob and R. Harrison, "Retrieving and analyzing mobile apps feature requests from online reviews," in *MSR 2013: 10th Working Conference on Mining Software Repositories*, San Francisco, CA, USA, May 2013, pp. 41–44.

[24] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *ICSME 2015: 2015 IEEE International Conference on Software Maintenance and Evolution*, Bremen, Germany, September 2015, pp. 281–290.

[25] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang, "Ar-miner: Mining informative reviews for developers from mobile app marketplace," in *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, June 2014, pp. 767–778.

[26] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh, "Why people hate your app: Making sense of user feedback in a mobile app store," in *SIGKDD 2013: 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Chicago, Illinois, USA, August 2013, pp. 1276–1284.

[27] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang, "Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing," in *UbiComp 2012: Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, September 2012, pp. 501–510.

[28] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *USENIX Security: 22nd USENIX Security Symposium*, Washington, DC, USA, August 2013, pp. 527–542.

[29] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *CCS 2014: Proceedings of the 21st ACM Conference on Computer and Communications Security*, Scottsdale, AZ, USA, November 2014, pp. 1354–1365.

[30] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, November 2014, pp. 389–400.

[31] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables1," in *PPREW '13: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[32] H. Flake, "Structural comparison of executable objects," in *DIMVA 2004: Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop*, Dortmund, Germany, July 2004, pp. 161–173.

[33] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics*, vol. 2, no. 1-2, pp. 83–97, 1955.

[34] J. Ming, M. Pan, and D. Gao, "ibinhunt: Binary hunting with inter-procedural control flow," in *ICISC 2012: 15th International Conference on Information Security and Cryptology*, Seoul, Korea, December 2012, pp. 92–109.

[35] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *USENIX Security: 23rd USENIX Security Symposium*, San Diego, CA, USA, August 2014, pp. 303–317.

[36] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *USENIX Security: 26th USENIX Security Symposium*, Vancouver, BC, Canada, August 2017, pp. 253–270.

[37] P. Calciati, K. Kuznetsov, B. Xue, and A. Gorla, "What did really change with the new release of the app?" in *MSR 2018: 15th International Conference on Mining Software Repositories*, Gothenburg, Sweden, May 2018, pp. 142–152.

[38] P. Calciati and A. Gorla, "How do apps evolve in their permission requests? a preliminary study," in *MSR 2017: 14th International Conference on Mining Software Repositories*. Buenos Aires, Argentina: IEEE Computer Society, May 2017, pp. 37–41.

[39] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez, "Bug fixes, improvements, ... and privacy leaks," in *NDSS 2018: 25th Annual Symposium on Network and Distributed System Security*, February 2018.

[40] A. Feal, P. Calciati, N. Vallina-Rodriguez, C. Troncoso, and A. Gorla, "Angel or devil? a privacy study of mobile parental control apps," in *The 20th Privacy Enhancing Technologies Symposium (PoPETs 2020.2)*, July 2020, pp. 314—-335.

[41] P. Calciati, K. Kuznetsov, A. Gorla, and A. Zeller, "Automatically granted permissions in android apps," in *MSR 2020: 17th International Conference on Mining Software Repositories*, Seul, South Korea, May 2020, pp. 114—-124.

[42] W. Martin, F. Sarro, and M. Harman, "Causal impact analysis for app releases in Google Play," in *FSE 2016: Proceedings of the ACM SIGSOFT 24th Symposium on the Foundations of Software Engineering*, Seattle, WA, USA, November 2016, pp. 435–446.

[43] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *ESEM 2016: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Ciudad Real, Spain, September 2016, pp. 29:1–29:10.

[44] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality?: An empirical case study of mozilla firefox," in *MSR 2012: 9th Working Conference on Mining Software Repositories*, Zurich, Switzerland, May 2012, pp. 179–188.

[45] M. Nayebi, B. Adams, and G. Ruhe, "Release practices in mobile apps -– users and developers perception," in *SANER 2016: 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, Osaka, Japan, March 2016, pp. 552–562.

[46] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: a study on behavioral backward incompatibilities of java software libraries," in *ISSTA 2017: Proceedings of the 26th International Symposium on Software Testing and Analysis*, Santa Barbara, CA, USA, July 2017, pp. 215–225.

[47] D. Domínguez-Álvarez and A. Gorla, "Release practices for ios and android apps," in *WAMA 2019: Proceedings of the 4nd International Workshop on App Market Analytics*, 2019, pp. 15–18.

[48] S. Hassan, W. Shang, and A. E. Hassan, "An empirical study of emergency updates for top android mobile apps," *Journal of Empirical Software Engineering*, vol. 22, no. 1, pp. 505–546, 2017.

[49] S. Hassan, C. Bezemer, and A. E. Hassan, "Studying bad updates of top free-to-download apps in the google play store," *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 773–793, 2020.

[50] F. R. Côgo, G. A. Oliva, C. Bezemer, and A. E. Hassan, "An empirical study of same-day releases of popular packages in the npm ecosystem," *Journal of Empirical Software Engineering*, vol. 26, no. 5, p. 89, 2021.