OASIS: Prioritizing Static Analysis Warnings for Android Apps Based on App User Reviews

Lili Wei, Yepang Liu, Shing-Chi Cheung
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{lweiae, andrewust, scc}@cse.ust.hk

ABSTRACT

Lint is a widely-used static analyzer for detecting bugs/issues in Android apps. However, it can generate many false warnings. One existing solution to this problem is to leverage project history data (e.g., bug fixing statistics) for warning prioritization. Unfortunately, such techniques are biased toward a project's archived warnings and can easily miss new issues. Another weakness is that developers cannot readily relate the warnings to the impacts perceivable by users. To overcome these weaknesses, in this paper, we propose a semantics-aware approach, OASIS, to prioritizing Lint warnings by leveraging app user reviews. OASIS combines program analysis and NLP techniques to recover the intrinsic links between the Lint warnings for a given app and the user complaints on the app problems caused by the issues of concern. OASIS leverages the strength of such links to prioritize warnings. We evaluated OASIS on six popular and large-scale open-source Android apps. The results show that OASIS can effectively prioritize Lint warnings and help identify new issues that are previously-unknown to app developers.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software testing and debugging**; • **Human-centered computing → Smartphones**; • **Information systems → Retrieval models and ranking**;

KEYWORDS

Static analysis, warning prioritization, Android Lint, app user reviews, natural language processing, concept graph

ACM Reference format:
Lili Wei, Yepang Liu, Shing-Chi Cheung. 2017. OASIS: Prioritizing Static Analysis Warnings for Android Apps Based on App User Reviews. In *Proceedings of ESEC/FSE'17, Paderborn, Germany, September 04-08, 2017,* 11 pages.
https://doi.org/10.1145/3106237.3106294

1 INTRODUCTION

Android app development has confronted great challenges posed by the features of mobile platforms such as resource limitations and fast-evolving ecosystems [28, 34, 35, 48]. Static code analyzers such as Android Lint [8] enable developers to identify potential bugs/issues in their apps before releasing them into the market. Although static analyzers can help quickly detect various types of issues without the need to execute the app under analysis, they usually suffer from a high rate of false positives, which reduces their usability [18, 32].

Existing studies [31, 44] have explored the possibilities of prioritizing static analysis warnings using historical issue fixing data. Kim et al. [31] prioritized warning categories based on the warnings eliminated by code changes. Ruthruff et al. [44] leveraged logistic regression models to prioritize warnings using historical bug triage and fixing statistics. One major limitation is that they strongly rely on historical activities (e.g., warning fixing choices) and developers' prior practices. As such, these techniques are intrinsically biased to warnings that are similar to the fixed. They can miss many new issues that arise from upgraded system libraries or have not been encountered by developers before. To address these problems, in this paper, we propose a new approach to effectively prioritizing the warnings generated by static analysis tools. We base our discussion on the Android apps and the most popular Android static analysis tool Android Lint (Lint for short) [8].

Lint supports the detection of various types of issues that are of multiple severity levels in Android apps [9]. Figure 1 illustrates a Lint warning that describes a functional issue caused by a deprecated Android system event. Typically, for an app with a moderate szie of code base, Lint would report hundreds to thousands of warnings (see Section 5 for examples). One way to prioritize such a large volume of warnings is to rank them by severity level. Unfortunately, even after such ranking, the number of warnings with the highest level of severity can still be unmanageable (see Section 3 for an example). An alternative is to selectively disable some checkers in Lint and thereby reduce the number of warnings. However, the choice of disabled checkers is generally undecidable. A bad choice would rule out many valid warnings while inducing invalid ones.

To overcome the limitations of the techniques discussed above, we propose a new criterion for prioritizing Lint warnings for Android apps. Our insight is that *a warning should be ranked at a higher place if its described issue can cause user-perceivable problems (i.e., affecting visible app execution behavior).* To effectively learn user-perceivable problems for Android apps, we leverage the large corpus of user reviews available on the app markets like Google Play store. Existing studies [20, 22, 23, 42] have confirmed that app user reviews can provide important information to facilitate app development and maintenance. These studies propose to extract useful information (e.g., requests for new features) by filtering or categorizing user reviews. However, to the best of our knowledge, none of them have ever leveraged user reviews to improve the results of static code analyses.

ESEC/FSE'17, September 04-08, 2017, Paderborn, Germany

To effectively prioritize Lint warnings, our approach, OASIS (pri**O**ritize w**A**rnings based on u**S**er rev**I**ew**S**), combines program analysis and natural language processing (NLP) techniques to recover the intrinsic links between Lint warnings and app user reviews. Specifically, OASIS (1) performs program analysis to augment Lint warnings with contextual information on the functionalities that can be affected by the issues of concern and (2) leverages NLP techniques to retrieve user-perceived problems from the app reviews. In this way, OASIS is able to estimate the severity of Lint warnings by analyzing frequency of user complaints on problems caused by the issues of concern for effective prioritization. One prominent challenge is that app user reviews are by nature unstructured and noisy. Users may describe a problem caused by an issue in their own way. Without understanding the semantics/meaning of user reviews, the links between Lint warnings and user reviews cannot be accurately established. To address this challenge, OASIS leverages Microsoft Concept Graph [12] for text understanding and semantic similarity measurement. As we show later this can significantly improve the effectiveness of warning prioritization.

We evaluated OASIS by conducting experiments on six popular and large-scale open-source Android apps, which have a sufficient number of recent user reviews. In the experiments, we observed that OASIS is able to effectively prioritize useful warnings on real issues that can cause user-perceivable problems. On average, it achieves one order of magnitude improvement in precision over Lint's default warning prioritization strategy (i.e., by issue severity level). Such an improvement significantly increases Lint's usefulness. As we show later, some developers quickly confirmed and fixed the warnings that are top ranked by OASIS. In addition, our semantics-aware similarity measurement method also significantly outperforms the traditional token-based textual similarity measurement methods in linking user reviews with Lint warnings to facilitate warning prioritization. In summary, we make the following major contributions in this paper.

- We propose a novel approach OASIS to prioritizing static analysis warnings for Android apps by leveraging app user reviews. To the best of our knowledge, we are the first to study the intrinsic links between static analysis warnings and app user reviews, leveraging such links for effective warning prioritization. The links generated by OASIS between a warning and the associated user reviews facilitate developers to understand the warning's impacts. We find no such support in existing Android tools.
- We propose a semantics-aware similarity measurement method, *concept similarity*, to calculate the similarity between structured warning documents and natural language user reviews.
- We evaluate OASIS on six large-scale and representative open-source Android apps. Our evaluation results confirm that (1) OASIS can effectively identify useful warnings on issues that cause user-perceivable problems, from a large number of Lint warnings, and (2) our concept similarity can outperform traditional token-based similarity measurement methods.

Paper orgnization. Section 2 presents the basics of Android Lint. Section 3 motivates our approach using a real example. Section 4 describes our approach OASIS in detail. Section 5 evaluates our approach. Sections 6 and 7 discuss threats to validity and related work. Finally, Section 8 concludes the paper.

File: File: AndroidManifest.xml
Line: 221
Severity: WARNING
Description: Use of `android.hardware.action.NEW_PICTURE` is deprecated for all apps starting with the N release independent of the target SDK. Apps should not rely on these broadcasts and instead use `JobScheduler`

Figure 1: A Lint warning for ownCloud

```
219. <receiver
        android:name=".broadcastreceivers.InstantUploadBroadcastReceiver">
220.     <intent-filter>
221.         <action android:name="android.hardware.action.NEW_PICTURE" />
222.         <data android:mimeType="image/*" />
223.     </intent-filter>
```

Figure 2: Lines 219–223 of ownCloud's AndroidManifest.xml

Title: Automatic uploads always fail
Author: Keith Slagerman
Rating: 2 **Date:** 2016-9-3
Comment: Instant upload not working after updating to v7.0. Saw in the description today that this issue is supposed to be fixed. Still not working though.

Title: Instant upload doesn't work in nougat.
Author: William Goodwin
Rating: 1 **Date:** 2017-1-12
Comment: Only reason I bought this app.

Title: Might be good
Author: Joseph Noyes
Rating: 3 **Date:** 2016-7-4
Comment: So far the most important part of the app... For my use case.... Is instant download and it doesn't work on Android N. That being said N is in Beta so of course I expect to run into these issues... But i also expect that this will be fixed on the first stable N release and I will upgrade my rating at that time.

Figure 3: Example user reviews of ownCloud

2 PRELIMINARIES OF ANDROID LINT

Android Lint is a static code analysis tool for Android apps [8]. It scans the source files (e.g., Java code files, resource and configuration files) of Android apps to detect potential bugs/issues for improving the apps' correctness, security, performance, usability, accessibility, and internationalization. Since Lint is directly available (runs by clicking menu items) in the Android Studio [3], the official IDE for Android app development, it has become a widely-used quality assurance tool for Android app developers.

Issues and checkers. Lint supports the detection of hundreds of types of issues [9]. Each type of issues is detected by a checker and annotated with a pre-defined severity level: *ERROR*, *WARNING*, *WEAK WARNING*, *INFO*, or *TYPO*. Lint users can customize the severity levels. They can also suppress the detection of certain types of issues when they consider the issues to be spurious.

Warning instances. Upon detecting a potential issue, Lint reports to developers the issue's location, severity level, and problem description. We call such a report a *warning instance* (or *warning* for short). Note that a warning in our work can describe an issue of any severity level (*ERROR*, *WARNING*, *WEAK WARNING*,

INFO, or *TYPO*). Figure 1 shows an example warning generated by Lint when analyzing ownCloud [13], a popular open-source file syncing and sharing app for Android devices. The described issue has a severity level of *WARNING*. It is detected at line 221 in the `AndroidManifest.xml` file (issue location) of ownCloud. This warning indicates that the app registers broadcast receivers to monitor and handle a deprecated type of system broadcast message, namely `android.hardware.action.NEW_PICTURE`, which will not be sent starting from the Android N release (i.e., Android 7.0). It also suggests that the app should use the `JobScheduler` (an API for scheduling jobs to be run in an app's own process [2]) instead.

3 MOTIVATION

The high false warning rate has long been considered a major limitation of static analyzers like Lint [18, 21, 32]. Unlike existing techniques [31, 44] that prioritize warnings based on project histories, our work prioritizes warnings based on a criterion of whether they prescribe an issue causing user-perceivable problems that affect app security, functionality, or performance. Our intuition is two-fold. First, most of such problems observed and reported by users are real. Therefore, warnings are likely true if they correspond to these problems. Second, existing studies have reported that such security, functionality, and performance issues are among the top types of code issues that developers would like program analyzers (like Lint) to detect [21]. As such, our ranking policy prioritizes warnings that are likely true and of concern to developers.

However, identifying user-perceivable problems caused by a particular issue is generally difficult and expensive. For example, it may require extensive user studies to understand the problems. Fortunately, for Android apps, there are a lot of publicly available user reviews in app stores. The interactive feedback systems of app markets enable users to rate an app and make comments easily. Such reviews can provide useful knowledge about the user-perceivable problems and hence can be leveraged to prioritize Lint warnings.

Key ideas. We observe that app user reviews and Lint warnings are intrinsically related. Both of them can be viewed as comments on the problems of Android apps. User reviews contain prolific information on user feedback such as the complaints about problems caused by certain issues, while Lint warnings provide hints at the app source level on issues that potentially affect the quality of Android apps. Our idea is to recover the links between a warning that reports an issue detected by Lint and its related user reviews. Via such recovered links, we can estimate the significance of the Lint warning (and thus the severity of the issue) by looking at the frequency of user complaints about the corresponding issue-inducing problems. To ease understanding, we use a real example to illustrate how user reviews can be linked to a Lint warning and demonstrate the major challenges that we need to address.

Recovering links between Lint warnings and user reviews. Let us consider the warning shown in Figure 1. As mentioned earlier, it reports that ownCloud registers a broadcast receiver for a deprecated type of broadcast message, which is no longer sent since the advent of Android 7.0 (i.e., Android N or Nougat). This breaks the functionality of ownCloud on those devices running Android 7.0 or above. Figure 3 lists several user reviews that complain about the broken instant upload on Android 7.0 devices. In this example, we can observe the links between the user reviews and the warning description. For instance, Review #3 mentioned "Android N", which is also included in the warning description as "N release". This indicates the possibility of applying natural language processing techniques (e.g., textual similarity based ones) to recover the intrinsic links between Lint warnings and user reviews.

Augmenting warning descriptions. However, the first challenge is that the links built by simply considering warning descriptions and user reviews can be weak and imprecise. This is because the warning descriptions are pre-defined templates. They provide only brief information and do not capture any program context specific to the apps under analysis. Therefore, warnings should be augmented with program contextual information (e.g., surrounding lines and call hierarchies) to enhance the linking accuracy. For example, Figure 2 shows the lines surrounding the location of the ownCloud issue. Note that ownCloud registers the `InstantUploadBroadcastReceiver` to handle the deprecated type of system broadcast message. In the comments of Review #1 and Review #2, the functionality of "instant upload" was reported to be broken. We also note that the keywords "instant upload" are included in the class name of the registered broadcast receiver, which can be found at the lines near the issue location. Hence, by augmenting the warning descriptions with such contextual information, we can recover the links between a Lint warning and its related user reviews.

Handling unstructured and noisy user reviews. The user reviews in Figure 3 suggest that app users tend to comment in their own style by using their preferred wording, expressing subjective feeling, or providing imprecise information. Such unstructured and noisy natural language data pose the second challenge to the accurate recovery of the links between Lint warnings and user reviews. Take Review #2 as an example. Its title mentions the instant upload problem, but the user comment provides no useful information relevant to the problem. Without properly handling such noise, we may mistakenly build the links between the review and those warnings whose description contains keywords such as "buy" or "purchase". This motivates us to leverage existing user review filtering techniques [22] to eliminate noise and non-informative reviews from the user review corpus. Another example is that "instant download" mentioned in the comments of Review #3 is not a functionality provided by ownCloud. In fact, the user intended to complain about "instant upload" rather than "instant download". Such cases can be common as non-technical end users may confuse "upload" with "download". In this example, "upload" and "download" are two semantically related concepts with different tokens. Therefore, applying simple textual similarity calculation based on tokens will miss such semantic links between Lint warnings and user reviews. This motivates us to consider word semantics in the link discovery process (see Section 4.2).

Finally, the warning in this example was buried under more than 2,400 other warnings reported by Lint for ownCloud. If app developers simply rank the warnings based on the issue severity levels assigned by Lint, this warning will be ranked after other 370 ERRORs and likely ignored by the developers. On the other hand, since there are many user complaints about this problem, the corresponding issue is certainly a prominent one that needs to be attended in due course. Our approach is designed to help developers

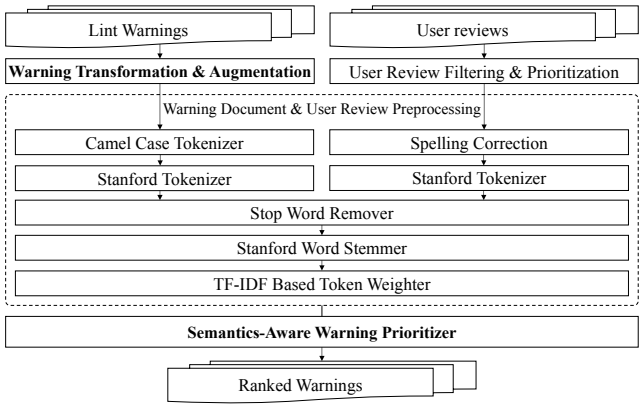


Figure 4: Overview of OASIS (components with our major contributions are highlighted with bold text)

successfully identify such warnings among a huge number of other warnings. In the next section, we present our approach in detail.

4 OASIS APPROACH

The goal of OASIS is to prioritize the warnings reported by Lint for Android apps using the information retrieved from user reviews. Figure 4 gives an overview of OASIS. It takes the Lint warnings for a given app and its user reviews as input and then outputs a ranked list of the warnings. OASIS first transforms the warnings and augments them with contextual information to generate *warning documents*. In parallel, OASIS leverages a state-of-the-art tool, SURF [22], to filter out non-informative user reviews. Thereafter, OASIS prioritizes the warnings by leveraging a widely-used concept graph and a semantics-aware similarity calculation technique.

4.1 Document Preparation and Preprocessing

In the first step, OASIS processes its two inputs, extracts contextual information for warnings, and eliminates noise. Specifically, it performs the three following tasks:

1) Warning transformation and augmentation.
2) User review filtering and prioritization.
3) Warning document and user review preprocessing.

4.1.1 Warning Transformation and Augmentation. As discussed in Section 3, a key challenge for linking user reviews with Lint warnings is that the information contained in the warning descriptions is inadequate for accurately retrieving related user reviews. OASIS addresses this challenge by augmenting warnings with contextual and user-perceivable code information. Specifically, for each Lint warning, OASIS first generates a textual document named *warning document*. Each warning document is initialized with the following data, which can be extracted from the analysis results of Lint: (1) warning category, (2) warning summary, (3) description of the potential problems, and (4) information on the environment (e.g., Android 7.0+ in Figure 1) under which the problems can occur (such environmental information is not always provided by Lint). These initialized data aim to describe the concerned issue of the warning in a succinct and pre-defined way. Next, OASIS searches for three kinds of contextual and code-related information about the functionalities that can be affected by the issue in the source files of concern to further enrich each warning document.

1) Statements surrounding the issue location: OASIS first extracts the statements neighboring the issue location from the source file of concern. These statements provide important contextual information for each warning. As shown in Figure 2, the neighboring statements of the issue location contain the name of the registered broadcast receiver that encodes the broken functionality "instant upload", which was complained about by users. The number of statements preceding and following the issue location is a configurable parameter in OASIS. Following an existing study's practice to capture code contextual features [36], the number is set to 3 in our current implementation and evaluation.

2) Identifiers in call hierarchy (only for Java code warnings): If a warning is reported at a source code line within a Java method *m*, OASIS enriches the corresponding warning document with *m*'s name. It then traverses the call hierarchy to further include the names of all methods that transitively invoke *m*. The enclosing method *m* is the code artifact that is directly affected by the issue described in the warning. By traversing the call hierarchy, all the methods that have transitive calling relationships with *m* will be included. In this way, the warning document is enriched with the identifiers of functionalities that can be affected.[1]

3) Android components: We include identifiers of the Android app components such as activities or broadcast receivers [2] that can transitively access the issue location (e.g., invoking the concerned code) as another piece of contextual information. These top-level app components provide interfaces and functionalities that are directly exposed to the users, and hence are more likely to be described in user reviews. We extract these app components using the following two criteria. For warnings on Java source code lines, the app components can be extracted from the methods in the call hierarchy because these methods are the component classes' members. For warnings on XML elements that register app components in configuration files, we match the names of the registered app components against the Java class names. Since warnings on app component registrations can affect any of the functionalities provided by the app components, we include the identifiers of the app component classes as well as their member methods and fields.

OASIS generates a warning document for each Lint warning and augments it with the extracted data as described above to help retrieve related user reviews.

4.1.2 User Review Filtering and Prioritization. While user reviews provide prolific information of the apps that can facilitate app development and maintenance, they often contain a lot of noise that needs to be eliminated before use [20, 22, 23, 42].

To eliminate the noise in user reviews and retrieve the relevant information of user-perceivable app problems, OASIS filters raw user reviews using a state-of-the-art review analysis tool, SURF [22], which is designed to exclude non-informative reviews and summarize useful user reviews by categorizing them into different topics and intentions. Specifically, SURF categorizes user reviews into five intention categories: *Information Giving*, *Information Seeking*,

[1] It is a good practice to name a method according to its functionality. Please also note that OASIS works on app source files and therefore it does not need to handle code obfuscations, which often transform identifiers to meaningless ones.

Feature Request, *Problem Discovery*, and *Other*. Since we aim to learn user-perceived app problems, we only keep those user reviews that are categorized as *Problem Discovery* for further processing.

After filtering, we prioritize the remaining ones by assigning weights to them based on their user ratings and submission dates. Intuitively, user reviews with lower ratings tend to comment on issues/bugs of an Android app and thus should be associated with a higher weight. Given a review r, OASIS uses the following formula to calculate its rating weight (i.e., the weight related to user ratings):

$$f_{rating}(r) = 1 - \frac{rating(r) - Rating_{min}}{Rating_{max}}$$

$Rating_{max}$ and $Rating_{min}$ denote the maximum and minimum rating supported by the corresponding user feedback system. The function $rating(r)$ returns the user rating of the review r. The rating weight $f_{rating}(r)$ is normalized as a negative linear function of the review ratings. For example, on Google Play store, user ratings are integers from 1 to 5. The rating weights for user reviews with a rating 1, 2, ..., 5 are mapped to 1, 0.8, ..., 0.2 by OASIS.

When assigning a weight to a review, we also consider its submission date in addition to its user rating. Our idea is that newer reviews should have a higher weight as they tend to reveal recent problems that still exist in the app, while old reviews may comment on legacy problems that may have already been fixed by developers. Hence, we define the date weight (i.e., the weight related to date) of a review r as:

$$f_{date}(r) = index(r)$$

The function $index(r)$ returns the index of the review r in the list of all reviews ranked in chronological order and the date weight $f_{date}(r)$ takes this value. Then, the overall weight of a user review r, denoted $f(r)$, can be calculated as the production of its rating weight and date weight:

$$f(r) = f_{rating}(r) \times f_{date}(r)$$

The overall weight of each review is leveraged later in our warning prioritization procedures.

4.1.3 Warning Document and User Review Preprocessing. After augmenting warning documents and extracting informative user reviews, OASIS further preprocesses both of them to eliminate noise and improve the accuracy of later NLP procedures. Following existing studies on information retrieval and text mining [20, 22, 23, 33, 41, 49, 50], OASIS applies three standard preprocessing steps: tokenization, stop word removal, and word stemming. For such preprocessing, we use the Stanford NLP Library [37], which is widely adopted by researchers and practitioners.

As shown in Figure 4, the preprocessing steps for warning documents and user reviews are slightly different as they have their own unique features. Warning documents are structured and contain both texts (e.g., problem descriptions) and code snippets (e.g., statements surrounding the issue locations). Since code snippets and texts are in different formats, we apply different tokenization methods. In Java, the camel case is a widely-used naming convention for identifiers in code [19, 38]. With this observation, OASIS splits code snippets into tokens by underscores and capital letters. Such a tokenization method for code snippets is also commonly used in code repository mining studies [41]. After tokenizing code snippets, the remaining texts in the warning documents will be tokenized by a standard tokenizer.

On the other hand, user reviews are mostly natural language texts and hence a standard tokenizer is sufficient for their tokenization. However, unlike structured warning documents, user reviews are often written in an informal way with many typos and much noise [20]. Therefore, before tokenization, OASIS applies an extra spelling correction step to fix typos and errors in user reviews. For this purpose, OASIS integrates the English vocabulary module of the JLanguageTool [11].

After tokenization, standard English stop word removal and word stemming are applied on the tokenized warning documents and user reviews. Finally, OASIS applies the *term frequency - inverse document frequency (TF-IDF)* [43] method to filter out generic words (e.g., phone, app) that appear frequently across documents in the corpus for later steps.

4.2 Semantics-Aware Warning Prioritization

In this step, OASIS takes the processed warning documents and user reviews to prioritize the input Lint warnings. The intuition of our warning prioritization method is: *a warning should be ranked at a higher place if its described issue can cause problems complained in a lot of app user reviews.*

Based on this intuition, we design the ranking score of a warning w as the summation of the similarities between the warning document of w and the user reviews:

$$S(w, R) = \sum_{r_i \in R} f(r_i) \times Similarity(w, r_i)$$

In the formula, r_i represents the i-th user review in a given app's user review set R. $f(r_i)$ is the weight for review r_i as defined in Section 4.1.2. The function $Similarity(w, r_i)$ calculates the similarity between the warning document of w and the user review r_i.

As discussed in Section 3, one critical challenge in linking Lint warnings to user reviews is that the user reviews are written in users' preferred wording and may provide imprecise information. As such, traditional textual similarity calculation techniques that model documents as tokens or entities may fail to precisely recover the intrinsic links between Lint warnings and their related user reviews. To address this problem, OASIS leverages Microsoft Concept Graph [12] for text understanding and integrates semantic meanings of words into document representation to calculate *concept similarity* for linking Lint warnings and user reviews.

To show the effectiveness of our concept similarity in uncovering the links between structured warning documents and free-style user reviews, we compare the discovery results of the concept similarity with two baselines: Jaccard similarity and cosine similarity, which are based on the traditional token-based document representation models. These two similarity measurement methods are widely used in existing studies that leverage information retrieval and textual mining techniques [24, 41, 49, 51–53]

In the following, we first briefly introduce the baseline methods (they are also implemented in our tool) and then discuss the details of our proposed concept similarity measurement method.

4.2.1 Baselines. Jaccard similarity calculation models documents as sets of tokens (or words) and measures the similarity between two documents by looking at the number of common and

universal tokens contained in the two documents. Formally, Jaccard similarity between a warning w and a user review r is defined as:

$$jaccard(w, r) = \frac{|T_w \cap T_r|}{|T_w \cup T_r|}$$

T_w and T_r are the sets of tokens in the warning document of w and the user review r respectively. As we can see, Jaccard similarity simply leverages the set intersection and union operations to compare the documents and associates all tokens with equal weights.

Another popular document representation model is the bag-of-words model weighted by TF-IDF values. The bag-of-words model represents documents as vectors of words (or tokens) and the value for each word is its corresponding TF-IDF weight. Cosine similarity is commonly used to calculate the similarity between documents represented by the bag-of-words model. Formally, cosine similarity of a warning w and a review r can be defined as:

$$cosine(w, r) = cosine(\boldsymbol{V_w}, \boldsymbol{V_r}) = \frac{\boldsymbol{V_w}^\top \cdot \boldsymbol{V_r}}{\|\boldsymbol{V_w}\| \|\boldsymbol{V_r}\|}$$

$\boldsymbol{V_w}$ and $\boldsymbol{V_r}$ are the bag-of-words vectors of the warning document of w and the user review r. The bag-of-words model and cosine similarity are widely used in text mining studies [41, 49, 53].

4.2.2 Microsoft Concept Graph and Concept Similarity. To overcome the limitation that traditional token-based models only compare lexical tokens in the document, in our work, we represent the documents with a new model based on the large corpus of concept relationships provided by the Microsoft Concept Graph [12].

Microsoft Concept Graph is a large publicly available knowledge base that captures the semantics of words by mapping words to their concept categories. By querying the graph, a word can be mapped to its semantic concept categories with probabilities. Thus, a word can be represented as a *concept vector* that encapsulates semantic information contained in the word. With the concept vectors, a document can then be mapped into the space by:

$$\boldsymbol{c_d} = \boldsymbol{\theta}^\top \boldsymbol{H}$$

In the formula, $\boldsymbol{\theta}^\top$ is the vector of the TF-IDF values of the tokens (words) in the document and $\boldsymbol{H}$ is the concept matrix. A concept matrix is constructed by concatenating the concept vectors of all tokens in the document (i.e., the concept vector of each token will be one row in the matrix). Via matrix multiplication, a document is mapped to a vector of concept categories, denoted as $\boldsymbol{c_d}$. Intuitively, a document is mapped to the concept space by assigning a probability to each concept category to which the document belongs. This probability is estimated by summing up the corresponding probabilities of all the tokens contained in the document.

The concept similarity of a warning w and a review r can then be computed as the cosine similarity between the corresponding concept vectors of the warning document of w and the review r, denoted $\boldsymbol{c_w}, \boldsymbol{c_r}$, which can be constructed by using the above formula. Formally, the concept similarity of w and r is defined as:

$$concept(w, r) = cosine(\boldsymbol{c_w}, \boldsymbol{c_r})$$

With the above method, we can successfully integrate semantic information of words into the similarity calculation between Lint warnings and user reviews.

We note that there are other alternatives to integrate word semantics into the similarity calculation procedures to accomplish software engineering tasks. For example, Ye et al. [52] trained their own word embeddings from API documents, tutorials, and reference documents to represent code snippets and natural language texts as vectors in a shared space for improving information retrieval in software engineering tasks. We currently choose to leverage Microsoft Concept Graph and propose our concept similarity due to two major reasons. First, user reviews are unstructured and written in casual language, which is difficult to learn from structured API documents or tutorials. Microsoft Concept Graph is learnt from billions of webpages and search logs. Its coverage of different words/texts and concepts is extremely broad due to the big training data and is more general than other knowledge bases [27]. Such a large and general knowledge base can well capture the semantics of diversified expressions in user reviews. Second, Microsoft Concept Graph is publicly available and its knowledge base is continually updated. Studies based on Microsoft Concept Graph to understand short texts are proved to be very effective [27]. While Microsoft Concept Graph is able to model semantics of general words, it is still possible that Ye et al.'s word embedding technique based on a specifically trained model can help better model program behaviors. In future, we plan to combine both techniques to study whether it can help achieve better results for Lint warning prioritization.

5 EVALUATION

5.1 Research Questions and Baselines

In this section, we present our experiments to evaluate OASIS's capability of identifying *positive warnings* among a huge number of warnings generated by Lint when analyzing Android apps. *Positive warnings* refer to those warnings of which the described issues can cause user-perceivable problems (the concrete judging criteria will be explained shortly in Section 5.2). In contrast, warnings of which the described issues cannot cause user-perceivable problems are referred to as *negative warnings* (e.g., false alarms or warnings on code style issues). Specifically, our evaluation aims to answer the following two research questions:

- **RQ1 (Usefulness of user reviews):** *Can the use of app user reviews facilitate the identification of positive warnings and improve the usefulness of Lint's static analysis?*
- **RQ2 (Effectiveness of concept similarity):** *Can our proposed concept similarity contribute to OASIS's performance of prioritizing positive warnings?*

To answer RQ1–2, we compared OASIS with these baselines:

- **Baseline I:** *Ranking Lint warnings by each project's adopted issue severity settings.* This baseline emulates a typical practice of app developers when they inspect Lint warnings. We compare it with OASIS to evaluate whether OASIS, which leverages app user review data, can outperform this common practice.
- **Baseline II:** *Ranking Lint warnings by the Jaccard and Cosine similarity (Section 4.2.1).* We compare OASIS with these two baseline methods that leverage token-based similarity measurements to study whether integrating semantic information can help improve the effectiveness of warning prioritization. The other program analysis and NLP steps of the two baseline follow those of OASIS.

Table 1: Experimental Subjects

App Name	Category	KLOC	Downloads	Rating	Version No.	# Warnings	# Reviews	Issue ID
AnkiDroid [5]	Education	60.8	1,000,000 - 5,000,000	4.5	2.8.1	5,404	3,621	4588
c:geo [6]	Entertainment	83.3	1,000,000 - 5,000,000	4.4	2017.02.07	10,775	4,871	6344
K-9 Mail [10]	Communication	92.8	5,000,000 - 10,000,000	4.2	5.203	5,751	8,181	2271
ownCloud [13]	Productivity	53.6	100,000 - 500,000	3.7	2.2.0	2,462	948	1905
TransDroid [16]	Tools	29.9	100,000 - 500,000	4.3	2.5.7	1,508	495	353
WordPress [17]	Social	144.4	5,000,000 - 10,000,000	4.2	6.6	8,195	8,851	5247

5.2 Experimental Setup

5.2.1 Subjects and Ground Truth. For our experiments, we selected the latest release versions of six popular and large-scale open-source Android apps as the subjects. Table 1 gives the basic information of each app subject, including the app name, category, size (in KLOC), the number of downloads, user rating, the version number, the total number of Lint warnings that are not suppressed by developers, and the number of user reviews we crawled from Google Play store. As shown in the table, these apps are popular (received millions of downloads), large-scale (containing thousands of lines of code), and diverse (covering six different categories).

To obtain results for Baseline I, we ran Lint on each app subject and ranked all the warnings that are not suppressed by developers for each app by the severity level of the corresponding issues. For issues of the same severity, we shuffled the order of their warnings five times and used the average evaluation metric values of these five ranked warning lists as the final evaluation results for Baseline I. To obtain results for Baseline II and OASIS, we ran our implemented tool with three similarity measurement methods on the Lint warnings and user reviews for each app, respectively. Our experiments were conducted on an iMac with 3.2 GHz Intel Core i5 CPU and 16 GB RAM.

To establish the ground truth, we manually inspected the ranked warning lists produced by Baseline I, Baseline II, and OASIS to label each warning as positive or negative. The labeling was performed by two co-authors of this paper. Before the process, the two authors discussed and reached a consensus on the criteria to distinguish positive and negative warnings. Specifically, the two authors independently checked each warning's program context to validate whether the warning is a true positive (i.e., the described issue really exists at the reported location). In the case of a true positive, they further decided whether the described issue would cause user-perceivable problems, including those that affect app functionality, performance, compatibility, and security, by checking the official Android API Guides [2], programming QA sites like Stack Overflow [15], and other online resources obtained by Google Search. After independent labeling, the authors cross validated their results and discussed the warnings that received different labels from them. For such warnings, the two authors further manually constructed test cases to confirm whether the issues of concern would indeed cause perceivable problems. The labeling process is labor-intensive and requires acquisition of much domain knowledge. In this work, we only labeled the top 50 warnings in each ranked list reported by Baseline I, Baseline II and OASIS. Specifically, we manually labeled 400 warnings for each of the six app subjects. In our experiments, Baseline I produced five ranked warning lists due to the shuffling, Baseline II produced two ranked warning lists, and OASIS produces one ranked warning list. In total, we manually checked 2,400 warnings, out of which 1,683 were unique. Among them, we identified 248 unique positive warnings in total. Then, we treated the set of all these positive warnings the complete set of positive warnings for later evaluation.

To further validate our ground truth, we randomly sampled representative positive warnings and reported them to the app developers for feedback. These issue reports are available online and we provide the issue IDs in the last column of Table 1. By the time of this paper's acceptance, we have received positive feedback from the developers of four subjects.

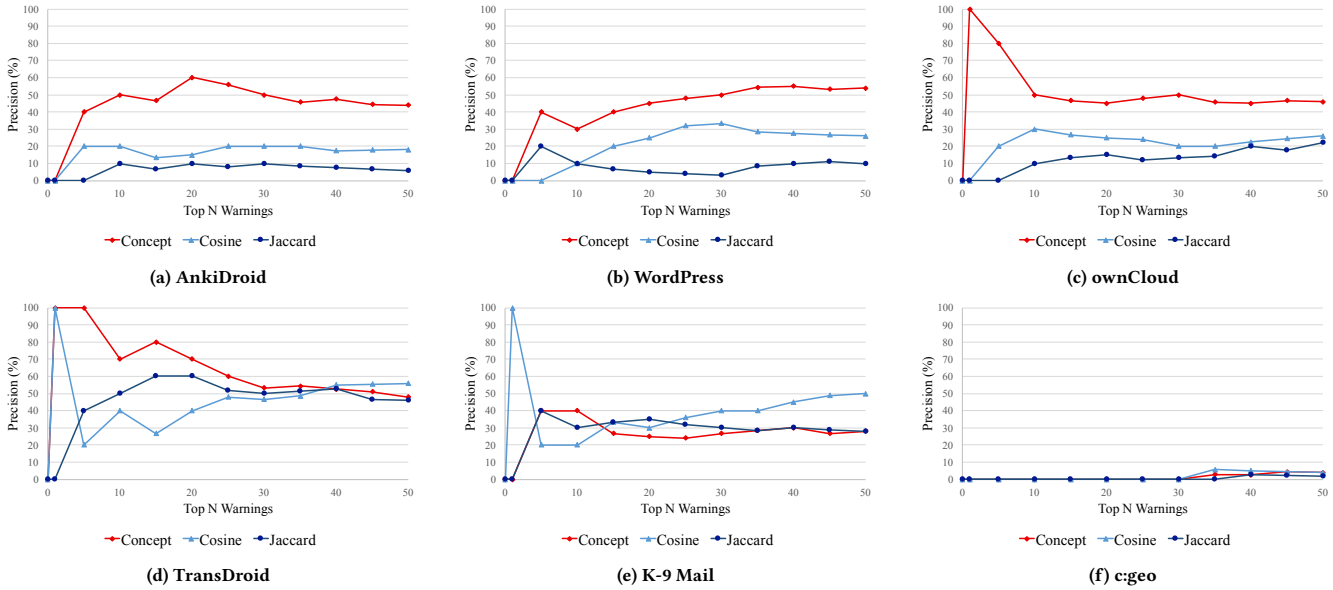
5.2.2 Evaluation Metric. Since the precision of the results is a critical quality metric of static analysis tools [18], in our evaluation, we applied the *Precision@N* metric to evaluate the performance of OASIS. *Precision@N* reports the percentage of positive warnings among the top N (N = 1, 5, 10...) warnings in each ranked list. A higher value of precision@N indicates a better ranking result, meaning that more positive warnings are ranked at the top of the list such that when developers inspect the results for issue fixing, they can easily identify positive warnings in the top ranked ones. In the following subsections, we discuss our experimental results and answer the two research questions.

5.3 RQ1: Usefulness of User Reviews

To answer RQ1, we compared the results of OASIS with those of Baseline I. Table 2 presents the precision@N results of OASIS and Baseline I for the six app subjects. From the results, we observe that OASIS achieved 100% precision@1 for two of the apps and 100% precision@5 for TransDroid. Overall, OASIS achieved over 50% precision@30 (up to 53.3%) and over 44.0% (up to 54.0%) precision@50 for four out of the six subjects. In most of the subjects, the precision tends to drop as N grows (the trend is shown in Figure 5). This indicates that OASIS can effectively identify positive warnings and rank them most at top positions. Such results favor developers to check the prioritized warnings one by one from the top. In comparison, Baseline I performed much worse than OASIS. Among the top 50 ranked warnings, Baseline I only achieved an average precision of 5.2%. This indicates that by leveraging information retrieved from app user reviews, OASIS can significantly outperform Baseline I, which emulates the likely scenario of using static analysis tools like Lint. Note that our Precision@N metric calculates the percentage of positive warnings in the top N ranked warnings rather than the percentage of true warnings (or true positives). According to the definition of positive warning in Section 5.1 that true warnings of the issues that cannot cause user-perceivable app problems (e.g., code style issues or Javadoc issues) were considered negative

Table 2: Precision@N of OASIS and Baseline I

N	AnkiDroid		K-9Mail		ownCloud		TransDroid		WordPress		c:geo	
	OASIS	Baseline I	OASIS	Baseline I	OASIS	Baseline I	OASIS	Baseline I	OASIS	Baseline I	OASIS	Baseline I
1	0.0%	0.0%	0.0%	0%	100.0%	0.0%	100.0%	0.0%	0.0%	40.0%	0.0%	0%
5	40.0%	8.0%	40.0%	0%	80.0%	12.0%	100.0%	0.0%	40.0%	16.0%	0.0%	0%
10	50.0%	4.0%	40.0%	0%	50.0%	6.0%	70.0%	0.0%	30.0%	14.0%	0.0%	0%
20	60.0%	2.0%	25.0%	0%	45.0%	9.0%	70.0%	0.0%	45.0%	18.0%	0.0%	0%
30	50.0%	2.0%	26.7%	0%	50.0%	8.7%	53.3%	0.0%	50.0%	17.3%	0.0%	0%
40	47.5%	1.7%	30.0%	0%	45.0%	9.0%	52.5%	0.0%	55.0%	18.0%	2.5%	0%
50	44.0%	2.8%	28.0%	0%	46.0%	9.2%	48.0%	0.0%	54.0%	18.8%	4.0%	0.4%



(a) AnkiDroid (b) WordPress (c) ownCloud

(d) TransDroid (e) K-9 Mail (f) c:geo

Figure 5: Precision@N of OASIS with Different Similarity Measurement Methods

in our experiments. Therefore, the Precision@N metric values of OASIS (and also Baseline I) are relatively low.

We also observe that in one subject, c:geo [6], OASIS failed to identify a considerable number of positive warnings among the top ranked ones while Baseline I performed worse. We reported the few positive warnings detected by OASIS to the developers of c:geo. Through the communication, we understand why the precision of OASIS for c:geo is low. This is because the developers of c:geo have integrated Lint into their app build process for a long time. Everytime when c:geo was built, Lint was run to check for potential issues. The developers would then quickly fix any issues they considered useful and suppress the Lint checkers that often reported false alarms. In fact, most serious issues reported by Lint have been fixed by the developers and the remaining unfixed issues are mostly false alarms or those that do not affect app execution behaviors. It is also worth mentioning that after receiving our feedback, the developers of c:geo immediately fixed our reported issues and found that they missed these issues because of their inappropriate Lint configuration.

Apart from c:geo, we also received positive feedback from developers of other three subjects. Warnings reported for AnkiDroid were accepted and assigned to corresponding developers. OwnCloud developers have fixed the critical issue discussed in our motivating example and K-9 Mail developers have fixed two of the three categories of issues we reported. Such positive feedback further confirms the importance of the issues identified by OASIS.

Answer to RQ1: *By leveraging the information retrieved from app user reviews, OASIS significantly outperforms Baseline I in identifying positive warnings. OASIS can improve the usefulness of Lint by effectively prioritizing its reported warnings.*

5.4 RQ2: Effectiveness of Concept Similarity

Figure 5 plots the precision@N results generated by OASIS and Baseline II. The results show that our proposed concept similarity method significantly outperformed the token-based methods in three of the apps, namely AnkiDroid, ownCloud, and WordPress. Among these apps, the precision@N of OASIS consistently outperformed that of the other two baseline methods from N = 1 to N = 50. For Transdroid, OASIS outperformed the baseline methods from N = 1 to N = 40 and achieves similar precision when N is larger than 40. In addition, OASIS successfully ranked a positive warning at the top one position of the warning list for ownCloud and TransDroid.

The only app, on which OASIS failed to outperform Baseline II, is K-9 Mail [10].[2] To understand the reason, we inspected the linked user reviews and Lint warnings of K-9 Mail produced by the three different methods. We found that 21 of 25 and 11 of 14 positive warnings identified by cosine and Jaccard similarities are correlated with the following two categories of similar issues:

The first category of issues is related to the usage of the deprecated Apache HTTP client APIs. The Android API Guides [2] suggest that these APIs have not been actively maintained by the Android team since API level 10 [4] and have been officially removed from the Android SDK since the release of Android 6.0 [1]. However, Apache HTTP APIs are widely used in K-9 Mail for network-related tasks (e.g., `markServerMessagesRead()`). Users of K-9 Mail constantly complained about the network or syncing problems caused by the use of these APIs and thus the keywords like "connect" and "server" were frequently mentioned in K-9 Mail's user reviews. In this situation, these commonly used tokens in user reviews can perfectly match the keywords in the Lint warnings, and therefore the token-based methods can also effectively identify such positive warnings. Yet such good performance is rare as suggested by the low precision of the Baseline II methods on the other app subjects.

The other category of issues is related to the usage of the deprecated Android system attributes `Window.PROGRESS_START` and `Window.PROGRESS_END`. These two attributes in K-9 Mail are correlated with the variable `pendingWork` and the method `updateTitle()`. In the warning augmentation step, the method and variable names are included in the warning document as contextual information. After tokenization, the tokens "work" and "update" are included in the warning document. On the other hand, many users of K-9 Mail happened to complain that "the app does not work since the last update", which shares the tokens "work" and "update" with the warning document. Thus, the warnings on the issues in this category are accidentally ranked highly by the Baseline II methods, although the linkage between the warnings and user reviews recovered by them is clearly wrong.

In contrast, the positive warnings ranked at the top 50 by OASIS for K-9 Mail are more diverse. For example, OASIS recovered positive warnings on two unique categories of issues that cosine and Jaccard failed to recover. It also highly ranked some warnings on the above-mentioned HTTP API and Android system attributes related issues and precisely linked them to the relevant user reviews.

We also investigated why some negative warnings are ranked top by OASIS. Take the top one warning in the ranked list reported by OASIS for K-9 Mail as an example. It reports that method `parseCommandContinuationRequest()` always returns true. While this warning describes a true phenomenon, it is negative because the method is supposed to always return true. The reason OASIS ranks it at the top is because (1) this method is transitively invoked by many other methods that implement functionalities such as user authorization and folder synchronization, and (2) lots of user complaints on topics like "Couldn't even get past the sign up page" or "Email doesn't download any more" are linked with this warning. While the warning document and user reviews are indeed semantically related and such semantic links can only be recovered by our concept similarity method (e.g., "download" and "sync" will not be correlated by token-based methods), the issue described in the warning has no perceivable effects on these related functionalities. Currently, OASIS cannot precisely capture the perceivable impact of every issue and therefore cannot judge whether or not this issue would affect the related functionalities. To address this problem, the warning documents generated by OASIS need to precisely capture the perceivable effect of the issues. Unfortunately, understanding such semantics is challenging. In future, we plan to augment warnings with more semantic information (e.g., code comments, information in API documentations) to further improve OASIS.

[2]Here, we do not compare the results of the three methods on c:geo as the total number of identified positive warnings is too low (we explained the reason when discussing the results for RQ1) to produce meaningful comparisons.

Answer to RQ2: *Concept similarity can improve the performance of OASIS by recovering semantic links between structured warning documents and free-style user reviews. Such links cannot be well captured by token-based methods.*

6 THREATS TO VALIDITY

Mismatch between user reviews and app versions. We used all user reviews available at the Google Play store as the input in our experiment. Since the Google Play store provides no attributes in these reviews indicating which app version the users were commenting on, there are chances that some user reviews are related to the historical versions of the examined app, inducing the mismatch between app versions and user reviews. However, user reviews on historical versions can only affect our results if the problems of concern have already been fixed and Lint raises false warnings on such problems. To address this threat, before linking user reviews and Lint warnings, OASIS applies a filtering and weighting procedure to weaken the impact of old user reviews. With such a treatment, in our experiments, we did not observe any negative impacts from the mismatch between app versions and user reviews.

Applying OASIS to other static analysis tools. The second potential threat is that our approach may not generalize to other static analysis tools for Android apps. We currently implemented OASIS for the Android Studio built-in Lint analyzer because Lint is easily accessible to developers and is widely used for improving Android app quality [8]. However, our OASIS design is not specific to Lint and in principle OASIS can also be applied to other static analysis tools like FindBugs [7] or PMD [14]. For example, to work with FindBugs, we only need to adapt the warning information extraction part of OASIS such that the descriptions and severity levels of the warnings generated by FindBugs can be extracted to initialize the warning documents. We plan to evaluate OASIS with other well-known static analyzers in our future extension of OASIS.

Limited number of evaluation subjects. Another threat is that we only evaluated OASIS with six open-source Android apps. Such a scale of evaluation may not be able to fully reveal the strengths and limitations of OASIS. However, we can see from Table 1 that these apps are diversified and representative, covering six categories with different numbers of warnings and user reviews. In addition, we can also observe the difference between evaluation results of these apps (e.g., c:geo is very well-maintained and its developers make heavy use of Lint and thus OASIS cannot largely improve the results of Lint). Such differences are caused by the special characteristics of these apps. This also shows that

our current evaluation results can be a good estimation of OASIS's performance on a wide range of Android apps in practice.

Impacts of app user reviews' quantity and quality. Our approach relies on app user reviews. The quantity and quality of available user reviews can affect the performance of OASIS. OASIS may not effectively prioritize static analysis warnings for apps with no or few reviews. In our evaluation, we addressed this concern by selecting subjects with different volumes of reviews (495–8,851, see Table 1). OASIS achieved good performance. This indicates that OASIS is applicable for apps with different volumes of reviews. In future, we plan to further study how the quantity and quality of app user reviews would affect the performance of OASIS.

7 RELATED WORK

In this section, we discuss related work on mining user reviews and static analysis warning prioritization.

7.1 Mining User Reviews

Various studies were made to leverage the prolific feedback information provided by user reviews such as user ratings and user comments to extract useful information to facilitate app development. Harman et al. [26] were among the first to mine useful information from app stores. They proposed a typical procedure for app store mining and correlated app ratings with the number of app downloads. Khalid et al. [29] leveraged user ratings and correlated device information to prioritize Android devices to perform testing.

More recent work focused on mining useful information from textual comments in user reviews. Khalid et al. [30] manually examined over 6,000 user reviews and highlighted that users frequently complained about app issues that cause crashing or incompatibility. This motivates us to leverage user reviews as a knowledge base to learn user-perceivable problems. Other researchers aimed to automatically extract useful information from a large number of user reviews. Chen at el. [20] proposed AR-Miner to extract informative user reviews by text classification. Gu et al. [23] designed SUR-Miner to classify and cluster user reviews of an app to evaluate its different aspects. Panichella et al. [42] combined NLP, text analysis, and sentiment analysis techniques to categorize user reviews into predefined categories to aid app development and maintenance. Villarroel et al. [47] proposed CLAP to categorize and cluster user reviews. CLAP is also able to prioritize the clustered user reviews to help app release planning. Di Sorbo et al. [22] designed URM, a two-level classification model to categorize user reviews based on user intents and review topics. Based on URM, they implemented SURF to categorize user reviews accordingly. Another pioneer work correlated user ratings and code changes [40] and found that developers who take user reviews seriously (e.g., by implementing requested features) are rewarded in terms of good app ratings. These techniques laid the foundation of app review analysis and helped OASIS filter out noisy user reviews. However, our work studies a different problem and aims at linking user reviews to static analysis warnings for improving the usefulness of static analysis tools.

7.2 Prioritizing Static Analysis Warnings

There are warning prioritization techniques designed for conventional programs that can be applied to Android apps. Some existing studies aimed to prioritize *actionable warnings* by statistical methods based on code revision histories and code characteristics. For example, Kim et al. [31] prioritized static analysis warnings generated by FindBugs, PMD, and JLint in Java programs by mining code revision histories. Ruthruff et al. [44] predicted actionable FindBugs warnings by logistic regression with factors integrating warning fixing histories. As discussed earlier, these techniques are by design biased to warnings similar to the fixed ones, while OASIS does not rely on the project history data of an app, but rather utilizes user feedback to prioritize warnings such that warnings on new issues that developers have not encountered before can also be identified.

Shen et al. [46] proposed EFindBugs to perform a two-phase issue prioritization based on manually labeled warning false positive data and user designations. Hanam et al. [25] leveraged machine learning techniques to learn actionable alert patterns and used the trained classifier for warning prioritization. These techniques need either significant manual efforts or a large amount of data to train the classifier. In contrast, OASIS leverages accessible user reviews and does not require additional manual efforts.

Finally, there are also other pieces of work proposing to redesign a better architecture of static analysis tools to aid developers [39, 45]. These studies improve static analysis tools from a different angle.

8 CONCLUSION AND FUTURE WORK

In this paper, we presented a novel semantics-based approach, OASIS, to prioritizing Lint warnings by leveraging app user reviews. To achieve effective warning prioritization, OASIS augments Lint warnings with contextual information and links them with app user reviews that contain complaints on problems caused by the issues of concern using our semantics-aware similarity calculation. We empirically evaluated OASIS with six popular Android apps. The results show that OASIS can effectively identify positive warnings and significantly outperform a baseline strategy that emulates a typical practice of app developers. We also experimentally validated that our proposed semantics-aware similarity calculation technique can largely improve the performance of OASIS when compared to traditional token-based similarity calculation techniques.

Our study makes the first attempt to leverage semantics-based similarity to link structured static analysis results with unstructured user reviews. This work is still exploratory and can be improved in multiple ways. In future, we plan to extend OASIS to leverage more contextual information (e.g., developers' comments in the code, API documentations) for warning augmentation and further explore other methods (e.g., word embedding [52]) to build semantic links between static analysis warnings and user reviews.

9 ACKNOWLEDGMENTS

This research project is funded by RGC/GRF Grant 16202917 of Hong Kong.

REFERENCES

[1] 2017. Android 6.0 Changes. https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html. (2017).
[2] 2017. Android API Guides. https://developer.android.com/guide/index.html. (2017).
[3] 2017. Android Studio. https://developer.android.com/studio/index.html. (2017).
[4] 2017. Android's HTTP Clients. https://android-developers.googleblog.com/2011/09/androids-http-clients.html. (2017).

[5] 2017. AnkiDroid Code Repository. https://github.com/ankidroid/Anki-Android. (2017).
[6] 2017. c:geo Code Repository. https://github.com/cgeo/cgeo. (2017).
[7] 2017. FindBugs™-Find Bugs in Java Programs. http://findbugs.sourceforge.net. (2017).
[8] 2017. Improve Your Code with Lint. https://developer.android.com/studio/write/lint.html. (2017).
[9] 2017. Issue Checkers in Android Lint. http://tools.android.com/tips/lint-checks. (2017).
[10] 2017. K-9 Mail Code Repository. https://github.com/k9mail/k-9. (2017).
[11] 2017. LanguageTool Style and Grammar Check. https://www.languagetool.org/. (2017).
[12] 2017. Microsoft Concept Graph. https://concept.research.microsoft.com. (2017).
[13] 2017. ownCloud Code Repository. https://github.com/owncloud/android. (2017).
[14] 2017. PMD. https://pmd.github.io. (2017).
[15] 2017. Stack Overflow. http://stackoverflow.com. (2017).
[16] 2017. Transdroid Code Repository. https://github.com/erickok/transdroid. (2017).
[17] 2017. WordPress for Android Code Repository. https://github.com/wordpress-mobile/WordPress-Android. (2017).
[18] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75.
[19] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To CamelCase or Under_score. In *ICPC*. 158–167.
[20] Ning Chen, Jialiu Lin, Steven CH Hoi, Xiaokui Xiao, and Boshen Zhang. 2014. AR-Miner: Mining Informative Reviews for Developers from Mobile App Marketplace. In *ICSE*. 767–778.
[21] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *ASE*. 332–343.
[22] Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Junji Shimagaki, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. 2016. What Would Users Change in My App? Summarizing App Reviews for Recommending Software Changes. In *FSE*. 499–510.
[23] Xiaodong Gu and Sunghun Kim. 2015. What Parts of Your Apps Are Loved by Users?. In *ASE*. 760–770.
[24] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In *WCRE*. 83–92.
[25] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. 2014. Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking. In *MSR*. 152–161.
[26] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2012. App Store Mining and Analysis: MSR for App Stores. In *MSR*. 108–111.
[27] Wen Hua, Zhongyuan Wang, Haixun Wang, Kai Zheng, and Xiaofang Zhou. 2015. Short Text Understanding through Lexical-Semantic Analysis. In *ICDE*. 495–506.
[28] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real Challenges in Mobile App Development. In *ESEM*. 15–24.
[29] Hammad Khalid, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. 2014. Prioritizing the Devices to Test Your App on: A Case Study of Android Game Apps. In *FSE*. 610–620.
[30] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. 2015. What Do Mobile App Users Complain about? *IEEE Software* 32, 3 (2015), 70–77.
[31] Sunghun Kim and Michael D Ernst. 2007. Which Warnings Should I Fix First?. In *ESEC/FSE*. 45–54.
[32] Ted Kremenek and Dawson Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *SAS*. 295–315.
[33] Tien-Duy B Le, Richard J Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *ESEC/FSE*. 579–590.
[34] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *ICSE*. 1013–1024.
[35] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Jian Lü. 2014. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *TSE* 40, 9 (Sept 2014), 911–940.
[36] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *POPL*. 298–312.
[37] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL (System Demonstrations)*. 55–60.
[38] Sun Microsystems. 1999. Code Conventions for the Java Programming Language. (1999).
[39] Mangala Gowri Nanda, Monika Gupta, Saurabh Sinha, Satish Chandra, David Schmidt, and Pradeep Balachandran. 2010. Making Defect-Finding Tools Work for You. In *IICSE*. 99–108.
[40] Fabio Palomba, Mario Linares-Vásquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2015. User Reviews Matter! Tracking Crowdsourced Reviews to Support Evolution of Successful Apps. In *ICSME*. 291–300.
[41] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2016. Parameterizing and Assembling IR-Based Solutions for SE Tasks Using Genetic Algorithms. In *SANER*. 314–325.
[42] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. 2015. How can I Improve My App? Classifying User Reviews for Software Maintenance and Evolution. In *ICSME*. 281–290.
[43] Juan Ramos. 2003. Using TF-IDF to Determine Word Relevance in Document Queries. In *ICML*. 133–142.
[44] Joseph R Ruthruff, John Penix, J David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In *ICSE*. 341–350.
[45] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *ICSE*. 598–608.
[46] Haihao Shen, Jianhong Fang, and Jianjun Zhao. 2011. Efindbugs: Effective Error Ranking for Findbugs. In *ICST*. 299–308.
[47] Lorenzo Villarroel, Gabriele Bavota, Barbara Russo, Rocco Oliveto, and Massimiliano Di Penta. 2016. Release Planning of Mobile apps Based on User Reviews. In *ICSE*. 14–24.
[48] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *ASE*. 226–237.
[49] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *ASE*. 262–273.
[50] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. Relink: Recovering Links between Bugs and Changes. In *ESEC/FSE*. 15–25.
[51] Jifeng Xuan and Martin Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *ICSME*. 191–200.
[52] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *ICSE*. 404–415.
[53] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports. In *ICSE*. 14–24.