BUGINE: a bug report recommendation system for Android apps

Ziqiang Li
Southern University of Science and Technology
11510352@mail.sustech.edu.cn

Shin Hwei Tan*
Southern University of Science and Technology
tansh3@sustech.edu.cn

ABSTRACT

Many automated test generation tools were proposed for finding bugs in Android apps. However, a recent study revealed that developers prefer reading automated test generation cased written in natural language. We present BUGINE, a new bug recommendation system that automatically selects relevant bug reports from other applications that have similar bugs. BUGINE (1) searches for GitHub issues that mentioned common UI components shared between the app under test and the apps in our database, and (2) ranks the quality and relevance of issues. Our results show that BUGINE could find 34 new bugs in five evaluated apps.

CCS CONCEPTS

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

KEYWORDS

bug report, recommendation system, Android apps

ACM Reference Format:
Ziqiang Li and Shin Hwei Tan. 2020. BUGINE: a bug report recommendation system for Android apps. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), October 5–11, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3377812.3390906

1 INTRODUCTION

Bug finding is a creative and inspiring activity. Many automated test generation [8] and repair techniques have been proposed to ensure the reliability of Android apps [1, 3, 6]. However, reading and reproducing the automatically generated test cases could be time-consuming. A study showed that developers prefer reading automatically generated test cases written in natural language [4]. This study also revealed that developers prefer manual testing compared to automated testing due to the learning curve of automated tools or lack of specific knowledge. Moreover, automated testing techniques for Android apps mostly focus on finding crashes [5], but neglect other non-crash related bugs (e.g., UI bugs). Meanwhile, many manually crafted bug reports (in natural language) are available in open-source repositories like GitHub.

Inspired by developers' requirements and the redundancy of bug reports, we propose BUGINE [7]. Given an Android app A, BUGINE will automatically select relevant bug reports for A. Each relevant bug report may include reproduction steps and bug fixes of similar bugs that could be useful for testing and debugging.

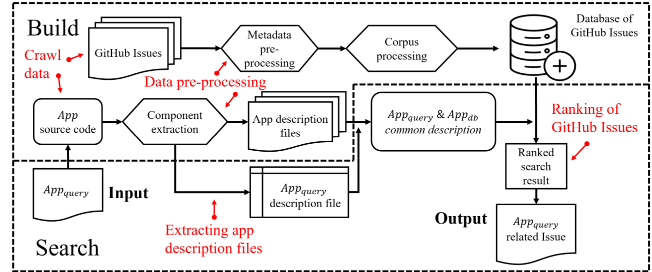

Figure 1: Workflow for BUGINE

Table 1: UI description patterns

Component	Example
Resource name	android:id="@+id/**my_btn**"
View name	<**Button** android:id="@+id/my_btn" />
XML file name	**main_layout**.xml

2 METHODOLOGY

Figure 1 shows the workflow of BUGINE which includes: (S1) building a database of GitHub issues, (S2) finding common UI components between App_{query} and $App_{database}$, (S3) constructing query based on common components from (S2) to search for relevant issues of $App_{database}$, (S4) ranking the results based on the quality.

Building a database of GitHub issues. Our crawler selects Android apps based on: (1) the users' rating and downloads in the App store, (2) the number of discussion and comments by developers, (3) the number of the star and issue of GitHub repository, and (4) the category of GitHub repository. Then, we collected all the issues and the meta-data of the selected apps (i.e., title, author, number of user comments, labels, issue state, body, commit SHA, etc.). We also downloaded the source code from the master branch of each app for subsequent steps. Our database has 23980 issues from 34 different applications that are selected from 10 different functionalities (e.g., cloud client, GitHub client, file explorer, web browser, etc.).

Data Pre-Processing. Our data consists of GitHub issues and the source code of the corresponding apps. For the GitHub issues, BUGINE pre-processes them with commonly used NLP techniques. We also use Humps [1] to unify the naming conventions of program variables, including snake case, camel case, into variable tuples and split compound (composite words). For the source code, specifically XML files, we also extract structural information of UI components.

Extracting app description files. The UI of Android apps is typically declared in XML resource files that define the structural layout of the UI components (e.g., view classes and subclasses). Each defined resource will be mapped to a resource ID. To generate app

*Corresponding author

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '20 Companion, October 5–11, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3390906

[1] https://github.com/nficano/humps

Table 2: Statistics and Evaluation Results of the Android apps used

App Name	Category	KLOC	#Downloads	Rating	Version No.	#GitHub Stars	#GitHub Issue (closed)	# Bugs Found (new,old)
Camera-Roll	Gallery	26.00	100,000+	4.2	1.0.6	420	227(133)	(11, 0)
PocketHub	GitHub client	31.35	10,000+	3.3	0.5.1	9429	644(526)	(12, 2)
Simple File Manager	Explorer	5.84	50,000+	4.5	6.3.4	378	189(130)	(6, 2)
Zapp	Broadcast	8.41	N.A.	N.A.	3.2.0	60	151(137)	(2, 7)
Simpletask	Reminder	24.80	10,000+	4.7	10.3.0	349	821(583)	(3, 2)

description file for an app, Bugine parses all its XML files in the "`src/main/res`" folder. Specifically, as shown in Table 1, we extract XML file names, view names and resource names, that describe the UI components of an app. Then, we combine the three pieces of information to search for the UI components of $App_{database}$ with the query: *XML file name* ∧ *View Name* ∧ *Resource name.*

Similarity Measures. Bugine uses two widely used similarity measures for computing text similarity [9]. To measure similarity between issues' title and query, Bugine uses overlap coefficient [2]. Bugine uses n-gram similarity to measure the similarity between issue's text body and query, and between UI components because they could contain structural information. Then, Bugine selects the $App_{database}$ issues with relevant UIs by the weighted similarity functions that evaluate the similarity of the parts of the two corpora, and search keywords are generated from their common parts.

Ranking relevant GitHub Issues. Bugine ranks the quality of each issue $\boldsymbol{E}$ based on several factors $f_i(\cdot)$, including (1) the length of the text body of $\boldsymbol{E}$, (2) the status of $\boldsymbol{E}$ (closed / opened), (3) if $\boldsymbol{E}$ contains any bug-fixing commit, (4) the number of replies that $\boldsymbol{E}$ received, (5) the overlap coefficient between the search keywords and corpus, (6) if $\boldsymbol{E}$ contains all keywords in the corpus, and (7) if $\boldsymbol{E}$ contains any important keyword (e.g., reproduce, defect). Given an issue $\boldsymbol{E}$, a component $\vec{C}$ that is similar to the search keywords, the weighted ranking score is $S(E, \vec{C}, W) = \sum_{i=1}^{n} w_i \times f_i(E, \vec{C})$.

3 EVALUATION

We evaluate Bugine on five open-source Android apps. Table 2 lists information about the evaluated apps. We select these apps because they are diverse in app categories, sizes, popularity, and the number of issues. Our evaluation answers the research questions below:

RQ1 (Relevance): What is the overall performance of Bugine in recommending relevant GitHub issues?

RQ2 (Reproducibility): How many bugs can Bugine find that can be reproduced and lead to unexpected behavior in App_{query}?

We evaluate Bugine's ranking performance using two previously used measures [10].

Prec@k measures the retrieval precision over the top k (we use k=5, 10, 20, 50) documents in the ranked list:

$$\text{Prec@k} = [\#\ of\ relevant\ docs\ in\ top\ k]\ /k$$

Mean Reciprocal Rank (MRR) For each query q, MRR measures the position $first_q$ of the first relevant document in the ranked list:

$$\text{MRR} = \left[\sum_{q=1}^{|Q|} \frac{1}{\text{first}_q}\right] /|Q|$$

RQ1: Ranking Performance of Bugine. Two raters independently evaluate the top 100 ranked issues of each app that have not been used before. Overall, the Prec@10 results range from 0.1 to 0.7, which means that among the top 10 issues recommended by Bugine, there is at least one relevant issue. Meanwhile, the MRR values for Bugine range from 0.34 to 0.75, which means that the ranking for the first relevant document ranges between 3rd (0.34) and 1st (0.75). This indicates that Bugine could recommend relevant issues for the evaluated apps.

RQ2: Number of bugs that Bugine finds. With two raters, we evaluate RQ2 by manually replicating the top-ranked issues to check if they are reproducible in App_{query} and *we consider that Bugine* ***discovers a bug*** *if such issue ranks in the top 100*. The "# Bugs Found" column in Table 2 shows the number of bugs found by Bugine. In total, we found 34 new bugs and 13 old bugs in all the five evaluated apps. Overall, our results show that Bugine could recommend relevant issues, which leads to the discovery of new bugs. All bugs found by Bugine are archived at https://bugine.github.io/.

4 CONCLUSION

We introduce a new approach that recommends relevant GitHub issues for an app under test. Given an app under test, Bugine searches for relevant GitHub issues based on the similarities of UI components shared with other apps in our database and further ranks them based on their quality. Our evaluation shows that it helps to discover 34 new bugs in the five evaluated apps.

Acknowledgments. This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902170) and Natural Science Foundation of Guangdong Province (Grant No. 2020A1515011494).

REFERENCES

[1] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android Testing via Synthetic Symbolic Execution. New York, NY, USA.
[2] Gerald Kowalski. 2010. *Information retrieval architecture and algorithms.* Springer Science & Business Media.
[3] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for android apps. In *FSE.* 233–244.
[4] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *ICSME.* IEEE, 613–622.
[5] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically discovering, reporting and reproducing android application crashes. In *ICST.* IEEE, 33–44.
[6] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in android apps. In *ICSE.* IEEE, 187–198.
[7] Shin Hwei Tan and Ziqiang Li. 2020. Collaborative Bug Finding for Android Apps. In *ICSE.*
[8] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *ICST (ICST '12).* USA, 260–269. https://doi.org/10.1109/ICST.2012.106
[9] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *ASE.* 708–719.
[10] 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.* IEEE Press, 14–24.