

CrossFix: Resolution of GitHub issues via similar bugs recommendation

Shin Hwei Tan^{1,2}  | Ziqiang Li¹ | Lu Yan¹

¹Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China

²Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China

Correspondence

Shin Hwei Tan, Ziqiang Li and Lu Yan,
Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China.
Email: tansh3@sustech.edu.cn, lizq2019@mail.sustech.edu.cn and lunaryan1998@gmail.com

Funding information

National Natural Science Foundation of China 61902170.

Summary

With the increasing popularity of Open-Source Software (OSS), the number of GitHub issues reported daily in these OSS projects has been growing rapidly. To resolve these issues, developers need to spend time and effort in debugging and fixing these issues. Meanwhile, a recent approach shows that similar bugs exist across different projects, and one could use the GitHub issues from a different project for finding new bugs for a related project. To locate similar bugs for our approach, we first conduct a study of similar bugs in GitHub. Our study redefines similar bugs as bugs that share the (1) same libraries, (2) same functionalities, (3) same reproduction steps, (4) same configurations, (5) same outcomes, or (6) same errors. Moreover, our study revealed the usefulness of similar bugs in helping developers to find more contexts about the bug and fixing. Based on our study, we design CrossFix, a tool that automatically suggests relevant GitHub issues based on an open GitHub issue. The suggested GitHub issues may contain solutions written in natural language or pull requests that help developers in resolving the given issue. Our evaluation on 249 open issues from Java and Android projects shows that CrossFix could suggest similar bugs to help developers in debugging and fixing.

KEYWORDS

debugging and program repair, mining software repository, open-source software

1 | INTRODUCTION

In recent years, pull-based software development model used in code hosting sites such as GitHub has become one of the most popular paradigm.^{1,2} Such sites allow any user to clone a project, modify it, use it, and provide feedback in form of a bug report. With the increasing number of users relying on these open-source projects hosted by GitHub, the number of reported issues has rapidly grown to outnumber contributors for open-source repositories.³ This indicates the need to derive a system that can relieve the burden of developers to resolve these issues.

Bug reports^{*} may contain a wealth of information that could assist developers in localizing and fixing bugs more efficiently.^{4–6} To exploit the redundancies of bug reports in open-source repositories, recent research on *collaborative bug finding* uses the similarities between Android apps for recommending relevant bug reports for a given app under test.⁷ This study revealed several interesting observations that may help in designing a system that can automatically suggest relevant issues for resolving the open GitHub issues: (1) Similar bugs could exist even across different applications and (2) one could treat another similar application as a competent pair programmer who can help in discovering new bugs. Although collaborative bug finding shows promising results in discovering new bugs in Android apps,⁷ there exists several limitations that make it not well-suited for solving the problem of recommending fixes: (1) It relies on the assumption that apps which share similar UI components will share

TABLE 1 Maui's developer refers to a similar bug in deeplearning4j.

GitHub issue from Maui (Driver) ⁸	
Title:	SwedishStemmer (and DutchStemmer?) not thread safe #10
Cmnt #1:	While using a Swedish language Maui Server project concurrently from multiple processes, I got several 500 Internal Server Errors with the following traceback: ... AnalysisEngineProcessException : Annotator processing failed. ... Caused by: java.lang.StringIndexOutOfBoundsException : String index out of range: 8 The root cause seems to be that the Snowball stemmer used by SwedishStemmer is not thread safe . (see a similar issue in another project)...
Cmnt #2:	osma: Make SwedishStemmer and DutchStemmer thread safe. Adds tests. Fixes #10
GitHub issue from deeplearning4j (Navigator) ⁹	
Title:	Stemmer exception when training word2vec with the supplied tweets_clean.txt file! #31
Cmnt #1:	Hi, I've tried to run the word2vec (using the supplied Uima tokenizer) and I keep getting this error for many of the words in the sentences... ... AnalysisEngineProcessException : Annotator processing failed. ... Caused by: java.lang.StringIndexOutOfBoundsException : String index out of range: 7
Cmnt #2:	It seems that the reason is that the SnowballStemmer IS NOT thread safe ..

similar bugs, but this assumption may not hold for other programs beyond Android apps (e.g., desktop application and Java libraries), (2) the task of fixing bugs is more challenging than finding bugs as it requires locating the bugs and producing a suitable patch (e.g., prior work⁷ only requires that two programs have similar UIs, but programs with similar UIs could have different implementations, which means that we may need different patches to fix the bug), and (3) finding a bug focuses on testing the behavior of the app under test (we call this “app-centric”), whereas fixing a bug requires analyzing the bug report (or GitHub issue) that describes the bug (we call this “issue-centric”).

While the concept of “similar bugs” could offer an alternative solution for the test generation problem, the prevalence of “similar bugs” and its precise definition have not been explored. In fact, for open-source projects, software developers have been using the concept of “similar bugs” for fixing bugs by referring to a related GitHub issue. Table 1 shows a real-world example of similar bugs where Maui's developer refers to the related issue in deeplearning4j (via the “similar issue” link). Comparing the two issues, we can observe that they share some similarities, including the following: (1) throw the same exception (**AnalysisEngineProcessException** and **StringIndexOutOfBoundsException**), (2) depend on the same **Snowball** library[†] where the **SnowballStemmer** class is invoked, and (3) have the same type of defect (not thread safe). This example shows the scenario target in this paper: *Given an open GitHub issue d , can we find a similar bug from another issue n that had been resolved/closed?* Specifically, we adopt the same metaphor used in pair programming and in collaborative bug finding⁷ where the *driver* writes the code and the *navigator* reviews the code. In our scenario, the open issue d serves as the driver as it tries to resolve the issue, whereas the previously resolved issue n plays the role of the navigator to provide patch suggestion and explanation. By using n as navigator, we foresee several potential benefits: (1) The bug fixing commit in n could contain fixes that require multiline edits and are not restricted to a predefined set of bug fix patterns, (2) the comments in n are written in natural language and could provide additional context for understanding the root cause of the bug, and (3) we are not restricted to fixing specific type of defects.

To better understand the characteristics of similar bugs, we conducted a study on 148 pairs of real GitHub issues collected from popular open-source Java projects. The study aims to explore the following research questions:

RQ1: *What are the characteristics that define similar bugs?*

RQ2: *Could similar bugs help developers in testing, debugging or fixing?*

RQ3: *What strategies can we use to detect similar issues?*

By investigating these research questions, we have derived several interesting findings. Specifically, we found that similar bugs include bugs that occur when two GitHub projects share the (1) same libraries, (2) same functionality, (3) same steps to reproduce, (4) same configuration/environment, (5) same outcome, or (6) same error/exception. Our study also shows that similar bugs could help in providing more context in resolving open issues (37% of the studied GitHub issues) and in fixing the bug (30% of the studied issues). These results give promising evidence about the usefulness of similar bugs in helping software developers in debugging and bug fixing.

Inspired by the results of our study, we designed CrossFix based on two important insights. Our key algorithmic insight is to represent each characteristic that defines the similarity between the driver and the navigator as pluggable analysis and to combine multiple similarity analyses to select the best navigator. By designing each similarity metric as pluggable analysis, we could easily extend CrossFix to support different types of programs, including Java programs and Android apps evaluated in this paper. Our approach searches for relevant GitHub issues in two phases:

TABLE 2 A similar bug between Nextcloud Notes and Material Components for Android.

Open GitHub issue from Nextcloud Notes (Driver)¹²
In-Note search crashes on Android 4.4
... <code>java.lang.IllegalArgumentException: radius must be > 0</code> at <code>android.graphics.RadialGradient.<init>(RadialGradient.java:53)</code> ...
Closed GitHub issue from Material Components for Android (Navigator)¹³
[<code>MaterialShapeDrawable</code>] Crash on Android 4.4.4 - <code>java.lang.IllegalArgumentException: radius must be > 0</code>
I'm currently using 1.1.0-beta02
We have this stacktrace on Crashlytics:
Fatal Exception: <code>java.lang.IllegalArgumentException: radius must be > 0</code> at <code>android.graphics.RadialGradient.<init>(RadialGradient.java:53)</code> ...

(1) online query generation that searches for relevant GitHub issues using off-the-shelf GitHub API and (2) offline query generation that reranks the GitHub issues based on multiple similarity analyses. The online query generation enables us to search broadly for more than millions of issues in GitHub instead of restricting to a small prebuilt database, whereas the offline query generation allows us to deploy more in-depth similarity analyses that require downloading and analyzing source files.

Overall, our contributions can be summarized as follows:

Study. Prior research on duplicate bug report detection relies on a narrow definition of *similar bugs* (bugs that involve handling at least 50% of common files),^{4,6,10} whereas similar bugs are used to discover new bugs in prior work on collaborative bug finding.⁷ To the best of our knowledge, we present the first comprehensive study on the characteristics and usefulness of similar bugs in GitHub. The results of our study can be useful for future research of automated testing and debugging via similar bugs.

Technique. We introduce a GitHub issue resolution technique that combines query-based fault localization and a set of pluggable offline similarity analyses for selecting a GitHub issue with the most similar bug to the given open issue. Our query-based fault localization automatically generates query to search online for more than millions of GitHub issues, whereas our similarity analyses exploit information on code changes, library dependencies, permission sets and UI components in Android apps to select the relevant GitHub issues.

System. We propose and implement CrossFix, a new recommendation system for suggesting relevant GitHub issues with the goal of resolving the given open GitHub issue. Given a GitHub issue d , CrossFix will automatically select the best GitHub issue which can be used for debugging and fixing the bug in d . The source code for CrossFix and our experimental data are publicly available online (<https://crossfix.github.io/>¹¹).

Evaluation. We evaluate the effectiveness of CrossFix on 249 open GitHub issues (152 java issues from Java projects and 97 issues from Android projects). Our evaluation shows that the issues recommended by CrossFix could help developers in debugging and fixing the bugs in the open GitHub issues. We have reported similar bugs for 40 open issues, where eight of these issues received positive feedback from developers.

2 | MOTIVATING EXAMPLE

We present two examples to demonstrate the targeted scenarios in this paper. The first example shows the overall workflow of our approach and the scenario where the fixes require changing build file (a crash that is caused by the dependent library). The second example shows the fix in the driver that requires changing Java source files.

Example that shows the workflow of CrossFix and a GitHub issue that requires fixing build files. We demonstrate CrossFix's workflow using a similar bug in an evaluated Android project. Table 2 shows the similar bug between Nextcloud Notes (we refer to this app as *Notes* for short) and Material Components for Android (we refer to this app as *Material* for short). Notes[‡] is an Android client for Nextcloud Notes app, whereas Material Components for Android[§] is a drop-in replacement for Android's Design Support Library. The first row of Table 2 denotes the titles of the two issues, and the second row of the table gives their contents. As stated in the second row of the table, the exceptions occur due to the constraint on the `radius` variable (i.e., must be greater than zero). Given the issue d from Notes, CrossFix recommends a relevant issue using the steps below:

- For GitHub issues with a stack trace, CrossFix automatically extracts this information from d to generate the query “`java.lang.IllegalArgumentException radius must be > 0 in:body,comments`”. Note that we added “`in:body,comments`” to limit the search for text body and comments (instead of searching in their titles) because stack trace is usually written in an issue's text body.

- CrossFix passes our automatically generated query to GitHub API to search for closed issues in Java projects. Instead of searching through other search engines (like Google), searching through GitHub has several benefits due to its pull-based development nature. As contributors in GitHub usually communicate changes by opening an issue or a pull request (PR),² code changes are usually associated with an issue/PR which may have (1) explanation for the context in which the bug occurs (e.g., the “This PR...” in Table 2 describes the intention of the patch) and (2) the stack trace information that allows us to search for the corresponding fixes. As CrossFix essentially performs fault localization by generating queries using stack trace information, we call this step *query-based fault localization*. For our query, GitHub API returns 10 relevant issues in Java projects, where the Material's closed issue is ranked fifth in the list.
- Given the search results from GitHub API, CrossFix reranks the relevant issues based on our similarity analyses and issue quality ranking. For Android projects, CrossFix computes code similarity, dependency similarity, and UI similarity. For each of returned issue n , CrossFix measures code similarity if n contains a patch. When CrossFix ranks the Material's issue, since n has a patch, CrossFix downloads n 's code and extracts $patch_{material}$. Then, it computes the similarity between the Java files in Notes and the modified Java files in $patch_{material}$. Moreover, since the Notes app uses similar library (com.google.android.material) as Material, CrossFix compares the name of the library (ignoring library version) defined in the project “build.gradle” file to calculate its dependency similarity. As both GitHub issues do not mention any UI component (e.g., button) in the GitHub issues, we do not calculate the UI similarity.
- In the original GitHub's returned results, other issues either have no fix or have fixes that do not share similar code with Notes. CrossFix selects the Material's issue because (1) it has the highest code similarity (46.10%) among all relevant issues and (2) it is ranked high by our issue quality ranking function as it contains a fix and has reasonable amount of content (number of words). The output of CrossFix is a sorted list of issues with the Material's issue as top 1, which ranks fifth in the original GitHub's returned results.

To assess the usefulness of the recommended patch, we manually adapted the patch from the Material's issue ($patch_{material}$). Listing 1 and Listing 2 show the developer's patch for the Notes' issue and the original patch in Material. As shown in Figure 1, we reported the adapted patch to the developer of the Notes app. By referring to $patch_{material}$, we can fix the bug in the Material's issue by making minimal modifications to $patch_{material}$. We followed the suggested solution provided in the Material's GitHub issue, which makes a similar modification in the same file (“build.gradle”) by updating the library com.google.android.material to the latest version (v1.3.0-alpha02) at that time. The developer acknowledged the value of our research and replied by saying that they will wait for the stable version of the com.google.android.material library to be available. After the stable version is released, the developer updated the library to fix the exception (Listing 2). This example demonstrates that *CrossFix could help developers to resolve diverse types of bugs, including issues that require updating the problematic dependent library*.

Example of a GitHub issue that requires fixing source files. In this example, we show how CrossFix can be used for recommending patches for GitHub issue that requires fixing source file. Figures 2 and 3 show similar bugs that occur in the Nextcloud app and the Simplenote app when the passcode is enabled. Given the issue in Figure 2, CrossFix constructs a query using the “Condition” strategy in CrossFix. Specifically, CrossFix extracts the condition (“app passcode is enabled”) from the title of the GitHub issue in Figure 2 and removes common words (i.e., “app” and “is”) from the extracted condition, resulting in the query “passcode enabled in:title” (we add the “in:title” keyword to search for GitHub issues with the same condition stated in the issue title). Then, CrossFix reranks the returned search results based on the similarity analyses where the issue in Figure 3 is ranked second in the list. For the similarity analyses, CrossFix reports that both apps share code similarity of 75.44% (CrossFix does not consider the library dependency and UI dependency because the driver issue did not mention any library relevant to the issue nor include any UI component that is related to the issue). As shown in Figure 2, CrossFix recommends the issue in Simplenote app (Figure 3) to the developer of Nextcloud. When the developer of Nextcloud received our comment that referred to the issue in Simplenote app, the developer replied by giving us a “Thumbs Up” positive emoji and fixed the reported problem by making the code changes shown in Listing 3. Specifically, by referring to the suggested code changes in Listing 4 that aims to set the secure flag (WindowManager.LayoutParams.FLAG_SECURE) when the app PIN lock is on in Simplenote, the developer of Nextcloud (1) reused the code that sets the secure flag in the `onActivityCreated(Activity)` method

```
-implementation "com.google.android.material:material:1.2.1"
+implementation "com.google.android.material:material:1.3.0"
```

LISTING 1 Developer's patch for similar bug in Notes (Driver)

```
-material_design_version = '1.1.0-alpha08'
+material_design_version = '1.1.0-beta02'
```

LISTING 2 Original patch for similar bug in Material (Navigator)

In-Note search crashes on Android 4.4 #847



stefan-niedermann opened this issue on 2 Jun 2020 · 5 comments

stefan-niedermann commented on 2 Jun 2020 Owner ...

```

App Version: 2.12.0 - 2.12.2 (and maybe others)
App Version Code: 2012000
App Flavor: dev

Files App Version Code: 30110090

---

OS Version: 3.10.0+(4174703)
OS API Level: 19
Device: generic_x86
Manufacturer: unknown
Model (and Product): Android SDK built for x86 (sdk_x86)

---

java.lang.IllegalArgumentException: radius must be > 0
    at android.graphics.RadialGradient.<init>(RadialGradient.java:5
    at com.google.android.material.shadow.ShadowRenderer.drawCorner

```

hidden ...

Crossfix commented on 29 Jul 2020 ...

I noticed that there is a similar problem at <https://github.com/material-components/material-components-android>. Perhaps we can refer to [material-components/material-components-android#937](https://github.com/material-components/material-components-android/issues/937) to find more context about the bug (e.g. the cause of the bug, or related library version, or their steps to reproduce).

Or maybe this can help us find the faulty lines and we can also refer to the [material-components/material-components-android@ 26fc10c](https://github.com/material-components/material-components-android/commit/26fc10c) for the bug?

Crossfix commented on 31 Jul 2020 ...

I tried [material-components/material-components-android@ 26fc10c](https://github.com/material-components/material-components-android/commit/26fc10c) and upgrade com.google.android.material to 1.3.0-alpha02, it works!

stefan-niedermann commented on 31 Jul 2020 Owner Author ...

Nice research! Thank you! So let's hope that the final version of com.google.android.material:1.3.0 will be released soon 😊

FIGURE 1 A similar bug in Nextcloud Notes (Driver) that requires updating the dependent library.¹²

by renaming the method to `setSecureFlag` and (2) invoked the `setSecureFlag` to resolve the issue. This example shows that *CrossFix* can provide fix suggestion to developer via the recommended issue. Although the developer needs to manually adapt the patch, an interesting future direction would be to automatically transplant the patch.¹⁶

3 | A STUDY OF SIMILAR BUGS ACROSS DIFFERENT PROJECTS IN GITHUB

Given an open and unresolved GitHub issues d , identifying a relevant GitHub issues n that shares similar bug with d could be useful for localizing and fixing the bug in n . To design more effective tools to search for relevant GitHub issues, we need to first thoroughly understand the characteristics of similar bugs. Hence, we conducted an exploratory study on 148 pairs of real GitHub issues collected from popular open-source Java projects. The study aims to explore the following three research questions:


RQ1: What are the characteristics that define similar bugs?

RQ2: Could similar bugs help developers in testing, debugging or fixing?

RQ3: What strategies can we use to detect similar bugs?

Hide app content in recent apps/apps switcher when app passcode is enabled #6508

 Closed seanthegeek opened this issue on 22 Jul 2020 · 2 comments



seanthegeek commented on 22 Jul 2020


Is your feature request related to a problem? Please describe.

Previews of whatever you are working on are viable in the Android app switcher/recent apps view, even when a the NextCloud app is configured to require a fingerprint or PIN. This raises privacy concerns.

Describe the solution you'd like


I'm not sure what API does this, but Android apps (e.g. Firefox Focus) are able to hide content from the resents/app switcher. This should be used when an app passcode is configured.

 1



tobiasKaminsky commented on 22 Jul 2020


Member

This should already be the case.
But indeed with device credentials I do not see this.


Crossfix commented on 24 Aug 2020

I noticed that there is a similar problem at <https://github.com/Automattic/simplenote-android>. Perhaps we can refer to [this issue](#) to find more context about the bug or hints to solve this problem.

 1


tobiasKaminsky mentioned this issue on 23 Sep 2020

re-add flag_secure to prevent showing content if app is locked
#6996


 Merged

FIGURE 2 A similar bug in Nextcloud (Driver) that occurs when the passcode is enabled.¹⁴

We investigate RQ1 because prior study shows that similar bugs could exist even across two different applications,⁷ but the characteristics of “similar bugs” have not been studied. We design RQ2 to investigate the usefulness of referring to GitHub issues with similar bugs. We study RQ3 to design better tools to better utilize and extract information from GitHub issues with similar bugs.

To study the relationship between GitHub issues that share similar bug, we manually inspected related GitHub issues. Our goal is to identify the information that we can obtain from GitHub issues with similar bugs. We first obtained a list of GitHub issues by searching for the keywords “similar bug” and “similar problem” written in Java using PyGitHub (a Python library that interacts with GitHub API v3).⁴ Our study focuses on Java projects because (1) many automated program repair techniques have been developed for fixing Java programs^{17–21} (including the recent bug report-driven repair approach²²), which indicates the importance of fixing bugs for Java programs, and (2) prior work on collaborative bug finding shows the existence of similar bugs across different Android apps while most Android apps are written in Java.⁷ For each issue d , our crawler searched for the first corresponding issue n that satisfies two criteria: (1) d mentioned that d and n share similar bugs and (2) d and n are in different open-source projects ($proj_d \neq proj_n$, where $proj_i$ indicates the open-source project in which the GitHub issue i was reported). The final output of our crawler is a list of GitHub issues (d, n) where d and n share similar bugs across different open-source projects. Overall, our crawler searched through 2000 GitHub issues (GitHub API limits each search to 1000 search results,⁴ so we search for the two keywords separately) and identified 148 issues with links to similar bugs. From these 344 GitHub issues, we manually excluded 196 invalid issues (e.g., duplicated issues or irrelevant issues). After filtering the irrelevant issues, we identified 148 (d, n) pairs of GitHub issues where n spans across 134 different Java projects. This relatively large number of open-source projects with GitHub issues mentioning similar bugs confirm with our hypothesis that *software developers tend to leverage the information about similar bugs during the discussion of a GitHub issue*.

Privacy: Recent app screen showing note contents, even when passcode is enabled #602



kriskorn opened this issue on 5 Dec 2018 · 1 comment

kriskorn commented on 5 Dec 2018

Steps to reproduce

1. Open Simplenote app
2. Write a note and minimize the app
3. Open the recent apps on your mobile device

What I expected
To see a Simplenote app splash screen or a logo.

What happened instead
Saw all of the contents of my note.

Simplenote version
1.6.3

OS version
8.1.0

Screenshot / Video

#1620989-zen

iskandergaba commented on 7 Dec 2018 Contributor

Good suggestion indeed. I will look into it soon.

2

iskandergaba mentioned this issue on 9 Dec 2018

Sensitive content is now hidden from the recent apps when app lock is enabled #604

Merged

FIGURE 3 A similar bug in Simplenote (Navigator) that occurs when the passcode is enabled.¹⁵

```
@@ -66,7 +66,7 @@ public PassCodeManager(AppPreferences preferences) {
// method renamed from onActivityCreated to setSecureFlag
- public void onActivityCreated(Activity activity) {
+ private void setSecureFlag(Activity activity) {
    Window window = activity.getWindow();
    if (window != null) {
        if (isPassCodeEnabled() || deviceCredentialsAreEnabled(activity)) {
            window.addFlags(WindowManager.LayoutParams.FLAG_SECURE);
        } else {
            window.clearFlags(WindowManager.LayoutParams.FLAG_SECURE);
        }
    }
}
@@ -81,6 +81,8 @@ public boolean onActivityStarted(Activity activity) {
+     setSecureFlag(activity);
}
```

LISTING 3 Developer patch for similar bug in Nextcloud.

```
+ // Disable screenshots if app PIN lock is on
+ public static void disableScreenshotsIfLocked(Activity activity) {
+     if (AppLockManager.getInstance().getAppLock().isPasswordLocked()) {
+         activity.getWindow().addFlags(WindowManager.LayoutParams.FLAG_SECURE);
+     } else { activity.getWindow().clearFlags(WindowManager.LayoutParams.FLAG_SECURE); }
+ }
+ @Override
+ protected void onResume() {
+     super.onResume();
+     disableScreenshotsIfLocked(this);
+ }
```

LISTING 4 Original patch for similar bug in Simplenote.

3.1 | RQ1: Characteristic of similar bugs

For RQ1, we obtain the characteristic of similar bugs by reading carefully the GitHub issues, focusing particularly on comments made by developers when referring to the similar bugs (i.e., developers usually include explanation why the two bugs are similar). Then, we count the number of GitHub issues that exhibit the identified characteristics.

The second column of Table 3 shows the common characteristics that define similar bugs in the evaluated pairs of GitHub issues, whereas the “# of Issues” column denotes the number of GitHub issues that exhibit these characteristics. Our study shows that similar bugs can be defined by bugs that occur when two GitHub projects share the (1) same libraries, (2) same functionality, (3) same steps to reproduce, (4) same configuration/environment, (5) same test outcome (exclude crash), or (6) same error/exception.

Given an issue i that mentioned a similar bug b in another issue j , we manually categorize it into the following:

Same library: We consider the similar bug b to share the same library if when i described the bug b , i mentioned about the shared library in j .

Same functionality: We categorize bug b as sharing the same functionality if when i described the bug b , i mentioned about (1) the project of i being a fork of another project j or (2) the project i supports a different programming language/platforms than the one in project j .

Same steps to reproduce: We consider i and j to share the same steps to reproduce if both issues mentioned about similar steps (e.g., UI actions) to reproduce the bug b .

Same configuration/environment: We consider the similar bug b to share the same configuration/environment if when i described the bug b , i mentioned about the shared configuration/environment in j .

Same test outcome (exclude crash): We consider the similar bug b to share the same test outcome if both i and j mentioned about the similar test outcome (e.g., b lead to nonresponsive programs in both i and j).

Same error/exception: We consider the similar bug b to share the same error/exception if both i and j include similar stack trace information (e.g., the name of the exception thrown and the error message).

We distinguish between crash-related (same error/exception) and noncrash-related bug reports (same test outcome) because the latter includes stack trace information that can be easily identified. As a pair of GitHub issues (d, n) can exhibit several similarities (e.g., they could share same libraries and produce the same error as the example in Table 1), we include all similarities in Table 3 (the category is not mutually exclusive). Overall, our study shows that the most common characteristics of similar bugs are as follows:

- Bugs sharing same library ($76/148 = 51.35\%$)
- Bugs sharing same test outcome ($39/148 = 26.35\%$)
- Bugs sharing same error/exception ($41/148 = 27.70\%$)

The relatively high percentage of bugs sharing the same library indicates the importance of detecting similar bugs by checking for the shared dependencies.

3.2 | RQ2: Usefulness of similar bugs

We investigate the usefulness of similar bugs by checking whether the 148 issues help developers to perform specific maintenance tasks by scoring them in the following order: (1) repair (most useful), (2) fault localization, (3) finding context of a bug, and (4) closing the issue faster (least useful). Specifically, if the similar bug could help developer directly in locating and fixing the bug, we select the most useful task (“repair”) rather than “fault localization.” During our analysis, we exclude 64 GitHub issues from 148 because we could not determine their usefulness as there are no direct reply or discussion about the similar issue. Specifically, given a pair of GitHub issues (d, n), we assess n 's usefulness based on the following descending order (we select the highest usefulness score for n if n can help in several maintenance tasks):

TABLE 3 Common characteristics of similar bugs, and strategies to identify them (one pair of similar bugs could have multiple characteristics).

Strategy	Characteristic of similar bugs	# of issues
Dependency	Same library	76
Code	Same functionality (forks, different language)	6
UI	Same steps to reproduce	2
Permission	Same configuration / environment (Permission set, Android / Windows version)	8
Condition	Same outcome (non-crashing)	39
Stack trace	Same error/exception	41

Repair: We consider n as helping developer in d to repair the bug in d if: (1) the fix of n could be directly applied to d , (2) the fix of n only requires minor modifications before applying to d , or (3) the discussion of d mentioned that they could find the fix for d by referring to the information in n .

Fault localization: We consider n as helping developer in d to locate the bug if: (1) n causes the bug in d , (2) n includes fault localization information, or (3) the discussion of d mentioned that they could identify the faulty location using the information in n .

Context: We consider n as helping developer in d to finding more contexts for the bug in d if the discussion in d explicitly mentioned that they have learned something from n (e.g., they gained some clues after reading n), but there is no further discussion on whether the knowledge gained helped in resolving the problem or identifying the fault.

Closing Issue faster: We consider n as helping developer in d to close the issue faster if: (1) after referring to the similar issue in n , the participants in the discussion of d think that they cannot or do not need to fix the bug and close the issue or (2) n only provides temporary workaround.

Figure 4 shows the usefulness of similar bugs among the 84 studied issues. Our study shows that similar bugs is most helpful in terms of helping developers to find more context (provide hints) for resolving the bug (37% of studied GitHub issues). Meanwhile, 30% of studied issues show the usefulness of similar bugs in helping developers to fix the bug. Overall, our results show that *similar bugs could help developers in resolving the GitHub issue in hand, especially in providing more context of the bug and fixing the bug.*

3.3 | RQ3: Strategies to identify similar bugs

After identifying the common characteristics of similar bugs, we need to design strategies to automatically detect these characteristics. Specifically, we need to check if we can detect each of these characteristics from several available sources, including the following: (1) the description of a GitHub issue and (2) information within different types of files that can be exploited for detecting these characteristics (e.g., Java projects typically has build files and Java class files, whereas Android project may have additional XML files for declaring app components and permissions).

Given a pair of GitHub issues (d, n) that share a similar bug b , we derive six strategies to identify similar bugs based on the rules explained below:

Dependency: We check if d and n share the same libraries by comparing (1) the set of dependencies shared by the GitHub issues in d and n and (2) whether the shared libraries are mentioned in d and in n . For example, in Table 1, both Maui and deeplearning4j share similar dependency (Stemmer). Java dependencies are usually declared in build files.

Code: We check if d and n share similar functionalities by comparing the Java code in the repository for d and the code modifications in the patch mentioned in n . Ideally, the code similarity checking will be more precise if we can compare the buggy code in d with the patch mentioned in n . However, as the information about the buggy code is often missing, we can only check whether the code within the patch for n matches with any code within the entire repository for d .

UI (Android-specific): We check if d and n share the same steps to reproduce the bug b by comparing their interfaces that allow users to invoke these steps. In Android apps, interfaces refer to UI elements (e.g., buttons and text boxes) that are typically declared in XML files.

Permission (Android-specific): We check if d and n share similar environments by comparing (1) the set of permissions that d and n acquire and (2) whether the shared permissions are mentioned in d and in n . We design this strategy as all Android apps need to declare the set of permissions in their manifest files but there is no such formal requirement for Java projects. The set of permissions is typically declared in XML files.

Condition: We check if d and n share the same test outcomes by comparing the conditions when the bug b is triggered. This information is usually available in an issue's description. For example, the phrase *when training word2vec with the supplied tweets_clean.txt file* in the Maui's issue in Table 1 represents the condition under which the bug occurs.

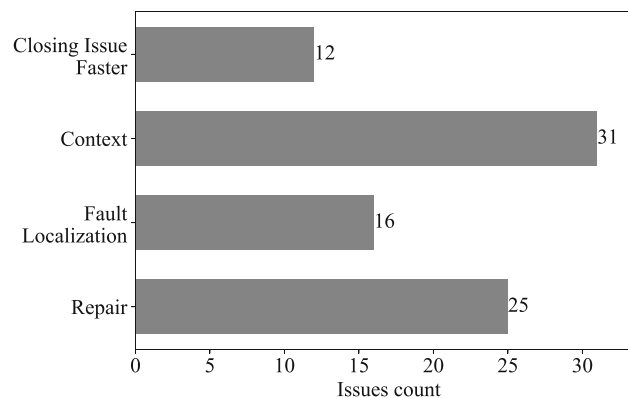


FIGURE 4 The usefulness of similar bugs among 84 study issues.

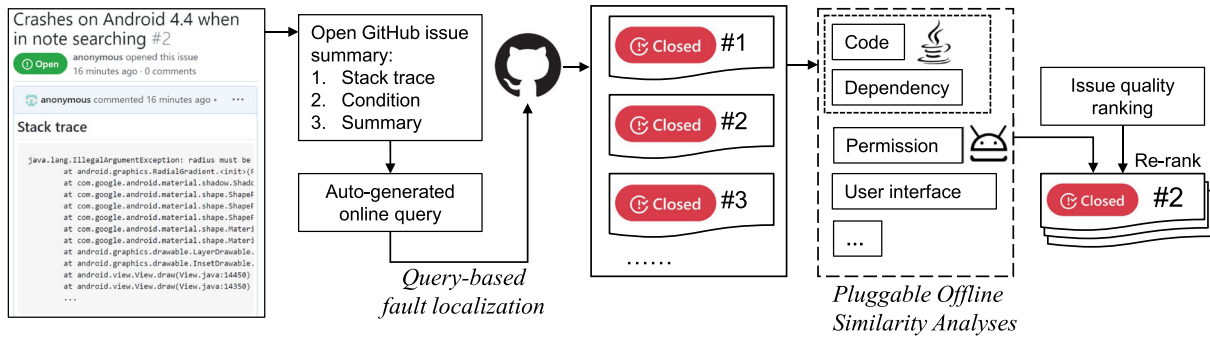


FIGURE 5 CrossFix's recommendation system.

Stack trace: We check if d and n share the same error or exception by comparing their stack traces. For example, in Table 1, the issues in Maui and deeplearning4j share similar stack trace with similar exception. This information is usually available in the description of a GitHub issue.

4 | METHODOLOGY

Figure 5 presents the overall workflow of CrossFix. CrossFix consists of several components: lightweight online query generator, pluggable offline similarity analysis, and issue quality ranking. CrossFix uses a two-phase approach. In the first phase, given an open issue d , CrossFix automatically extracts information from d to build a query q_{online} to search for closed issues with similar (1) exception information, (2) condition that triggers the bug, or (3) general description that summarizes the bug extracted from d 's title. In the second phase, CrossFix goes through the search results for q_{online} and rerank the list of GitHub issues n_1, \dots, n_i . For each issue n_i , CrossFix computes (1) the quality of n_i and (2) the similarity between d and n_i via multiple analyses. The final output of CrossFix is a ranked list of GitHub issues. We consider only the top-1 GitHub issue as the best navigator that will be used as “similar bug”.

4.1 | Query-based fault localization

Existing techniques in automated program repair typically treat test cases as blackbox^{23–26} and tend to ignore the target defect classes that these techniques aim to address.²⁷ To find the faulty location that causes the bug, these techniques usually rely on statistical fault localization techniques. Meanwhile, bug report-driven repair approaches^{22,28} use information from bug report from the buggy program to either determine the bug type²⁸ or perform fault localization.²² Different from these techniques, CrossFix performs *query-based fault localization* by generating search queries using information extracted from GitHub issues of the buggy program. This new way of fault localization introduces new challenges: (1) The input GitHub issue is written in natural language which is difficult to parse correctly, (2) we need to automatically determine what kind of information in the given GitHub issue that is useful for representing the bug, and (3) the generated query need to be short and precise because GitHub imposes restriction on the length of the search query: Queries longer than 256 characters are not supported.¹¹

Given an open issue, CrossFix first parses its title and its description. As the GitHub issue is written in natural language, CrossFix pre-processes the natural language text using the same procedure in prior work.⁷ Specifically, we perform tokenization, stopwords removal (via Python NLTK library²⁹), stemming, and lemmatization. We also exclude the project name (e.g., deeplearning4j) from the query to remove project specification information.

After preprocessing, CrossFix automatically extracts information that can be used to build a query for searching through GitHub. To determine the key information that should be included in the generated query for representing the bug, we design our query based on our study (described in Section 3) and select only the information that can be directly extracted from the description of a GitHub issues (including stack trace and condition). Table 4 shows the strategies that we used to build the online query. As shown in the “Strategy” column in Table 4, CrossFix generates query based on three strategies: (1) stack trace (the highest priority), (2) condition, and (3) bug summary (this strategy is used only as a fallback plan when we cannot apply other strategies). Algorithm 1 shows our query construction algorithm. Given an issue i with title $title$ and issue body $body$, together with a threshold k (empirically set to 5 to indicate that a query returns too few search results so CrossFix needs to invoke other strategies to get more results), our approach returns a ranked list of results (this list will be reranked according to the ranking score in Equation (6)). As shown in Algorithm 1, we used four strategies iteratively to construct the query. Whenever a strategy return too few results (line 18 in Algorithm 1), we proceed to other strategy to get more results. We describe each strategy below:

TABLE 4 Strategies used in extracting information to build the online query.

Strategy	Description	Source	Example text (upper)/generated query (lower)
Stack trace	Exception/Error type and exception/error message in the stack trace	Issue body	java.lang.NullPointerException: Attempt to invoke virtual method 'java.lang.Object android.widget.FrameLayout.getTag(int)' on a null object reference NullPointerException attempt to invoke virtual method java lang Object android widget FrameLayout getTag int on a null object reference
Condition	Condition where the buggy behavior is triggered	Issue title	Stemmer exception when training word2vec with the supplied tweets_lean.txt file Training word2vec with the supplied tweets_lean.txt file
Summary	Summary of the bug from title	Issue title	SwedishStemmer (and DutchStemmer?) not thread safe SwedishStemmer DutchStemmer thread safe

Algorithm 1 GitHub issue query construction and retrieval algorithm

Input: A GitHub issue i with title $title$ and issue body $body$, Threshold k

Output: A list of GitHub issue $results$ (initial search results)

```

1:  $STACKTRACE = 1, CONDITION = 2, TITLE\_SUMMARY = 3, FALLBACK = 4$ 
2:  $results = [], strategy = STACKTRACE$ 
3: do
4:    $query = ""$ 
5:   switch  $strategy$  do
6:     case  $STACKTRACE$ 
7:       if  $body.match("[a-zA-Z.]* (Exception|Error)")$  then
8:          $query = extractException(body) + "in:body"$  ▷ Extract exception type and message
9:       end if
10:    case  $CONDITION$ 
11:      if  $title.match("(if|when|while).*")$  then
12:         $query = extractCondition(title) + "in:title"$  ▷ Extract the phrase after if|when|while
13:      end if
14:    case  $TITLE$ 
15:       $query = removeStopwords(title) + "in:title"$  ▷ Remove common words (e.g, a, the), and look for matching titles
16:    case  $FALLBACK$ 
17:       $query = removeStopwords(title)$ 
18:     $tmpResult = getGitHubResults(query)$ 
19:    if  $len(results) < k$  then ▷ If there are too few results, use different strategy to construct the query
20:      if  $len(results) > 0$  then
21:         $results += tmpResult$ 
22:      end if
23:       $strategy += 1$  ▷ Move to the next strategy
24:    else
25:      break
26:    end if
27: while  $strategy > FALLBACK$ 
28: return  $results$ 

```

Case 1: Stack trace. Our query-based fault localization uses stack trace information to represent the bug because (1) this information is often included in an issue's text body, (2) stack trace may contain the core elements of a patch explanation, including the cause (the root cause exception/error), and the consequence (crash)³⁰ and (3) prior study has shown the effectiveness of using stack trace to locate^{31–34} and to repair runtime exceptions.³⁵ The key differences between our approach and existing fault localization techniques are as follows: (1) our stack trace information is embedded in an issue's text body and (2) our stack trace may be incomplete because some users may include only partial stack trace in the GitHub issue. Given a GitHub issue d , our stack trace parser extracts d 's text body and searches for stack trace information. Specifically, our stack trace parser identifies stack trace by searching for the regular expression “[a-zA-Z]*(Exception|Error)” (line 7 in Algorithm 1). Then, our parser searches for the “Caused by” keyword to locate the root cause of the crash. If the parser fails to get the root cause due to incomplete stack trace, it then searches for the first Exception/Error in the first line of the extracted stack trace. This strategy outputs a query that contains the name and the message of the thrown Exception/Error in the given GitHub issue. We give an example to explain the input (“Example Text”) and output (“Generated Query”) of this step in the first row of Table 4.

Case 2: Condition. As the condition under which a bug is triggered is a core element of a patch explanation,³⁰ CrossFix searches for the phrases that represent a condition in the issue's title. Specifically, it looks for the expression $(if|when|while).*$ (line 10 in Algorithm 1). For example, the phrase *training word2vec with the supplied tweets_clean.txt file* in Maui's issue in Table 1 and second row of Table 4.

Case 3 and Case 4: Bug summary. Although more advanced text summarization techniques could be used to generate a precise summary, we choose to use the text extracted from the issue title because this information is directly available, and it provides a fast way to obtain an overall summary of the issue. As shown in the last row in Table 4, we build the query by removing symbols and stopwords (“and” and “not”) from the issue title. Specifically, we first construct a query that searches for matching title by filtering stopwords (line 13 in Algorithm 1). If this query returns too few results, we then construct a query that looks for issues with either matching titles or matching content (line 15 in Algorithm 1).

After generating the query via query-based fault localization, we passed the generated query to GitHub API to search for relevant GitHub issues.

4.2 | Pluggable offline similarity analyses

To further rank relevant GitHub issues based on the initial results returned by the online query, we design the analysis based on the results in Section 3.3. To diagnose different types of defects, CrossFix applies multiple *pluggable similarity analyses* for checking different types of source files with each analysis applied when it is required. This design allows us to analyze both Java projects and Android apps by reusing several common similarity analyses and applying specialized analyses to Android apps. Specifically, for Java and Android projects, we perform (1) code similarity analysis and (2) dependency similarity analysis. For Android projects, we additionally perform (1) UI similarity analysis and (2) permission set similarity analysis.

In general, given two GitHub issues (d, n) , each analysis for a given characteristic c checks if (1) c is mentioned in both issues d and n and (2) c defines the similarity between d and n . We describe each characteristic below:

Code similarity. Given two GitHub issues (d, n) where n contains a patch p_n which fixes the bug, we measure code similarity between p_n and d using the equation below:

$$\text{CodeSim}(n, d, p_n) = \text{sim}(\forall f \in \text{repo}_d, \forall f \in p_n) \quad (1)$$

Specifically, we compute the code similarity between all files that are modified in p_n and all files in d 's repository.

Dependency similarity. Given two GitHub issues (d, n) and a function $\text{Dep}(i, \text{repo}_i)$ that returns the package dependencies mentioned in a given issue i and declared in the dependency files in i 's repository, we measure dependency similarity between d and n using the equation below:

$$\text{DepSim}(d, n) = \text{Sim}(\text{Dep}(d, \text{repo}_d), \text{Dep}(n, \text{repo}_n)) \quad (2)$$

Particularly, we compute the similarity between the package dependencies mentioned in d and that are mentioned in n .

Permission similarity. Given a pair of GitHub issues (d, n) and a function $\text{Per}(i, \text{repo}_i)$ that returns the permissions mentioned in a given issue i and declared in the app manifest files in i 's repository, we calculate permission similarity between d and n using the equation below:

$$\text{PermSim}(d, n) = \text{Sim}(\text{Per}(d, \text{repo}_d), \text{Per}(n, \text{repo}_n)) \quad (3)$$

Specifically, we measure the similarity between the permissions mentioned in d and that are mentioned in n .

UI similarity. Given a pair of GitHub issues (d, n) and a function $\text{UI}(i, \text{repo}_i)$ that returns the UI elements mentioned in a given issue i and declared in the XML files in i 's repository, we compute UI similarity between d and n using the equation below:

$$UISim(d, n) = Sim(UI(d, repo_d), UI(n, repo_n)) \quad (4)$$

Particularly, we calculate the similarity between the UI components mentioned in d and that mentioned in n .

Similarity measure. For the dependency similarity, the permission similarity and UI similarity, we use *Overlap Coefficient*³⁶ to calculate the overlap between the set of keywords mentioned in a GitHub issue and the corpus (words obtained from package dependencies, permission set, and names of UI elements). Overlap Coefficient is defined as follows:

$$\text{overlap}(X, Y) = \frac{|X \cap Y|}{\min(|X|, |Y|)} \quad (5)$$

Note that Overlap Coefficient ranges between $[0, 1]$. If X is the subset of Y or vice versa, then $\text{overlap}(X, Y)$ equals to 1. We select Overlap Coefficient instead of the Jaccard Similarity³⁷ and Dice Similarity³⁸ because (1) it is widely used in prior recommendation systems for bug reports,^{6,7,39} and (2) it is sensitive to the size of the two sets (i.e., suitable for our task since the search query is usually shorter than the corpus).

4.3 | Selecting the most relevant issue

Apart from using multiple similarity analyses, we select the most relevant GitHub issue based on their quality.

Issue quality ranking. Ideally, the GitHub issues recommended by CrossFix should contain rich information to assist developers in debugging and fixing. According to prior study, developers expect a good report to contain the following: (1) *steps to reproduce*, (2) *observed and expected behavior*, and (3) *stack traces*.⁴⁰ Hence, we further rank the GitHub issues based on its qualities. Specifically, we use several metrics used in prior work⁷ to evaluate the quality of an issue n , including the following: (1) the number of words in n 's text body, (2) if n contains a commit hash (fixed/not fixed), (3) the number of comments that n received, and (4) the number of descriptive keywords that n contains (e.g., “reproduce”, “defect”).

Selecting the best GitHub issue. With multiple similarity analyses (defined in Section 4.2) and the score for the issue quality as factors that affect the ranking of a GitHub issue, we calculate the ranking score $S(n, W)$. Given an issue n , we calculate the ranking score $S(n, W)$ as below:

$$S(n, W) = \sum_{i=1}^n f_i(n) \times w_i \quad (6)$$

where $f_i(n)$ denotes the value of factor f_i on issue n and w_i denotes the weight for a factor f_i . For the factors that affect issue quality ranking, we select the weights used in prior work⁷ ($w_{\text{issue_length}} = 0.0714$, $w_{\text{num_comment}} = 0.1428$). For other factors, we perform a grid search to tune their weights using the dataset in Section 3. Specifically, we set $w_{\text{CodeSim}} = 0.1428$, $w_{\text{DepSim}} = 0.2142$, $w_{\text{PermiSim}} = 0.2142$, and $w_{\text{UISim}} = 0.2142$.

4.4 | Implementation

For calculating code similarity, we leverage JPlag.⁴¹ JPlag first converts each program into a string of tokens and then compares the two programs by trying to cover one of the programs with sequences from the other program using the maximum similarity algorithm.^{**} While there are many code plagiarism detection engines that can calculate code similarity,^{41–46} we select JPlag because (1) it is open-source and actively maintained, (2) it offers offline similarity analysis, and (3) it has been shown to be one of the most effective code similarity analyzers.^{47,48} To compute the dependency similarity, we support projects that use Gradle⁺⁺ and Maven⁺⁺ for compilations because (1) they are among the most popular open-source build automation tools and (2) Gradle has been widely used in both Java and Android projects. We compute UI similarity by modifying existing implementation of Bugine, a tool that recommends relevant GitHub issues based on UI similarity.⁷ The main modification to Bugine is to add the support for incorporating the open GitHub issue d into the UI similarity calculation. In Algorithm 1, we set $k = 5$ as threshold to indicate that a query returns too few search results.

TABLE 5 Statistics of the evaluated Java projects and Android apps.

	Java	Android
Total open issues	152	97
Total projects	25	11
KLOCs	5–2140k	8–683k

5 | EVALUATION

There are several related approaches that we have considered for comparison, including duplicate bug reports detection approaches^{4–6,10,49} (e.g., NextBug) and GitHub. Although duplicate bug reports detection approaches like NextBug^{6,10} also recommend bug reports, these approaches rely on traditional bug tracking systems like Bugzilla instead of GitHub, and they require bug component to be specified for finding bug reports with similar text descriptions. There are several key differences between Bugzilla and GitHub that make these existing approaches not suitable for recommending GitHub issues: (1) a bug report in Bugzilla contains a field for users to specify the bug component,⁸⁸ whereas such field does not exist in a GitHub issue (users may choose to indicate the bug component by adding a tag but most GitHub issues do not contain any tag); (2) the bugs in Bugzilla are only reported within the same organization (e.g., Mozilla) and the bug reports for each organization are hosted independently under different web domain, whereas GitHub is a site that hosts a wide variety of projects from many organizations across the world; and (3) GitHub provides hosting for millions of software repositories and includes a version control system, whereas Bugzilla is only a bug tracking system (does not store any source code or commit histories that can be used for finding similar bugs). As existing duplicate bug reports detection approaches do not support GitHub issues and a fair comparison will not be possible without considering the unique features in GitHub, we did not compare our tool against these approaches. Another related approach is the search feature in GitHub that can be used to narrow down the search for issues that contain certain keywords. We select GitHub as our baseline for comparison since searching through GitHub is the first step of approach.

We perform evaluation on the effectiveness of CrossFix to address the following research questions:

Q1 What is the overall performance of CrossFix in ranking relevant GitHub issues?

Q2 How useful are the CrossFix's recommended issues?

Q3 What is effectiveness of CrossFix compared to approaches based on Stack Overflow?

5.1 | Experimental setup

We evaluate CrossFix on 249 open GitHub issues. Table 5 shows the statistics of these issues (152 issues are from 25 Java projects, whereas 97 issues are from 11 Android apps).

Selecting open issues. To select open GitHub issues, we build a crawler that automatically searches for open issues from popular Java projects and Android apps. We obtained a list of open-source Android apps from F-Droid. For Java and Android projects, our crawler selects projects that have (1) the greatest number of stars, (2) recent commits within 1 year, and (3) no overlap with the projects in our study (Section 3). To ensure diversity of the considered projects, our crawler got the first 10 open issues for each project from GitHub that do not have any bug fixing commit. It is challenging to search for open GitHub issues automatically because an issue either (1) has a related pull request, (2) has too little information to verify its validity, (3) has irreproducible bug, and (4) is not bug-related (question/feature) so we need to manually filter these invalid issues. From our initial list of 192 issues for Java projects and 142 issues for Android apps, we removed 40 invalid issues for Java projects and 45 invalid issues for Android apps.

Table 5 shows the statistics of the evaluated valid open GitHub issues. Overall, the selected projects are diverse in terms of sizes and functionalities (Java projects include VM, compilers, etc., whereas Android apps include notes app, file sharing app, etc.). Information about the selected projects is available at our website.¹¹

Evaluation metrics. For Q1, we evaluate the overall ranking performance of CrossFix by using two measures used in prior evaluations of recommendation systems:^{7,50,51}

Prec@k. This computes the retrieval precision over the top k documents in the ranked list:

$$\text{Prec}@k = \frac{\text{of relevant docs in top } k}{k} \quad (7)$$

We measure the precision at $k = 1, 3, 5$.

Mean reciprocal rank (MRR). For each query q , MRR measures the position $first_q$ of the first relevant document in the ranked list:⁵²

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{first_q} \quad (8)$$

A higher MRR value denotes better ranking performance.

Response time. For each query q , its response time is measured using two indicators: (1) time used to return the top few ranked items corresponding to the query (we obtain top 10 results but some queries less returned issues), and (2) time used for each returned relevant issue.

To assess the usefulness of a recommended issue n in Q2, we use the same criteria as RQ2 (we check if n helps in repair, fault localization, context, or closing issue faster). For all relevant issues, we reported them to the developers.

All experiments are conducted on a machine with Intel(R) Xeon(R) Platinum 8269CY CPU @2.50GHz 2 cores and 4 GB RAM.

5.2 | Q1: Ranking performance of CrossFix

As we evaluate on open issues that have not been resolved, no ground truth data (the actual relevant issue with similar bugs) is available so we need to manually evaluate the relevance of each recommended issues. To save the time and effort in labeling, we only labeled 10 recommended issues for each open issue. Table 6 shows CrossFix's ranking performance results.

Specifically, the average retrieval precision in Table 6 calculates individual retrieval precision (Prec@k) for each of the open issues divided by total number of evaluated issues (152 issues for Java projects and 97 issues for Android projects). We use the following equation for computing the average retrieval precision:

$$AveragePrec@k = \frac{\sum_n^1 Prec@k}{n} \quad (9)$$

where n is the total number of evaluated open issues. As CrossFix reranks the original GitHub returned results using multiple similarity analyses, we compare the performance of the original GitHub's ranking ("GitHub (with our query)" column in Table 6) versus the reranked results by CrossFix ("CrossFix" column in Table 6) to assess the effectiveness of our similarity analyses. Specifically, to ensure fair comparison with the ranking performance of GitHub, we provide the same query generated via query-based fault localization (explained in Section 4.1) to both GitHub and CrossFix.

Java versus Android. Compared to Java projects, the average Prec@k results for Android projects are generally higher. For example, the average Prec@1 is 0.4433 for Android projects versus 0.3399 for Java projects. In contrast, the average MRR values for Java projects are higher than that for Android projects. The higher Prec@k for Android projects is because Android projects have less relevant GitHub issues (25 versus 75 relevant issues), and many queries for Android projects have zero search results (Prec@k=1 for zero results).

Effectiveness of similarity analyses. As our approach aims to select the best issue as the similar bug, we use MRR for evaluating the effectiveness of our similarity analyses because MRR gives greater importance to the first relevant item. Comparing the results for the "GitHub (with our query)" column and the "CrossFix" column in Table 6, we observe that *our similarity analyses helps to further improve the MRR values for both Java and Android projects*. Specifically, the average MRR for Java projects increases from 0.1168 to 0.1401 ($\approx 20\%$ increment), whereas the average MRR for Android projects increases from 0.0485 to 0.0619 ($\approx 28\%$ increment). If we compare the average retrieval precision for both approaches,

TABLE 6 Ranking performance of CrossFix.

	Java		Android	
	GitHub (with our query)	CrossFix	GitHub (with our query)	CrossFix
Average Prec@1	0.3333	0.3399	0.4433	0.4433
Average Prec@3	0.3181	0.3181	0.4433	0.4467
Average Prec@5	0.3163	0.3163	0.4454	0.4454
Average MRR	0.1168	0.1401	0.0485	0.0619
# relevant		75		25

Abbreviation: MRR, Mean Reciprocal Rank.

CrossFix shares similar results with the original GitHub for most cases, but it improves over GitHub for Prec@1 for Java projects and Prec@3 for Android projects. Based on our manual analysis of the retrieved GitHub issues, we notice that the relatively low improvement CrossFix over GitHub for the average retrieval precision is due to the fact that many of the retrieved issues are not relevant, causing the overall precision to be low.

Response time of queries. We evaluate the performance of CrossFix by computing the response time. The response time for each query is dependent on the number of returned results. Overall, the average response time of a query for CrossFix is 265.96 s, whereas the average response time of a query for GitHub (with our query) is 8 s. CrossFix takes more time to return the results because “GitHub (with our query)” can save time by skipping the pluggable offline similarity analyses where the analysis needs to be triggered for each of the returned search results. On average, each query returns 8.83 relevant issues, and the average response time per relevant issue is 30.13 s. We think that the average response time of ≈ 4 min is reasonable as the reranking is fully automatically and helps to increase CrossFix's ranking performance. Compared with the time spent in manually resolving open GitHub issues (which can take an average of 1084.64 h for GitHub projects⁵³), we think that the response time of CrossFix is reasonable.

Answer to Q1: The Prec@k and the MRR values show that CrossFix is able to recommend relevant issues for both Java and Android projects.

5.3 | Q2: Usefulness of the recommended issues

Table 6 shows that CrossFix recommended 75 issues for Java projects and 25 issues for Android projects. Given a relevant issue n , we evaluate Q2 by manually checking if n helps in repair, debugging, finding context about the bug, or closing the open issue. In total, there are 40 open GitHub issues d with at least one relevant recommended issues n . Figure 6 shows the usefulness of CrossFix's recommendation for the 40 open issues. Figure 1 shows an example where the issue recommended by CrossFix helps developers in repairing the GitHub issue. The results show that the issues recommended by CrossFix are useful in helping developers to find more contexts about the bug ($21/40 = 52.5\%$ issues) and in repairing the bug ($12/40 = 30.0\%$ issues). These results are consistent with our findings in Section 3.2 where similar bugs help developers in finding more context about the bug and in bug fixing.

Answer to Q2: The similar bugs recommended by CrossFix could help developers in finding more context about the bug and fixing the bug.

5.3.1 | Feedback from developers

For each of the 40 issues, we find all its relevant recommended issues (one open issues may have multiple relevant issues) and leave a comment using the format below:

I noticed that there is a similar problem at <https://xyz>. Perhaps we can refer to this issue to find more context about the bug? Or maybe this can help us find the faulty lines, and we can also refer to the fix for the bug?

We exclude the “Or maybe...” part if the recommended issues do not help in debugging or fixing the bug. For each commented issue, we manually analyze the feedback. To ease the understanding of our results, we divide the feedback into several categories: (1) positive (*pos*), (2) neutral (*neu*), (3) negative (*neg*), and (4) awaiting reply (*awa*). Specifically, we consider a reply to be “*pos*” if the developer replied to our comment directly acknowledging the usefulness of our comments. Note that the acknowledgement could be a positive comment or a positive emoji, for

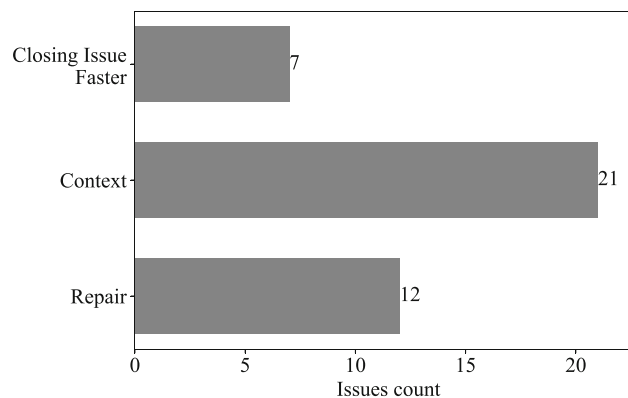


FIGURE 6 The usefulness of the issues recommended by CrossFix.

example, thumbs up emoji. We consider a reply to be “*neu*” if (1) the developer did not mention whether our suggestion is successful and (2) mentioned that he or she is not interested in getting help from others. We classify a reply to be “*neg*” if the developer mentioned directly that our suggested issue is irrelevant and does not help in resolving the issue. If a comment for an issue does not receive any response from the developer, we classify it as “*awa*”.

Overall, we obtained 18 replies for the commented issues. According to our classification, we obtained $pos = 8$, $neu = 10$, $neg = 0$, $awa = 22$ of the 40 issues. Specifically, most replies are positive, whereas some issue are classified as “*neu*” because the developer explained that they do not need help for resolving the issue considering the fact that they have already prepared the pull request for the issue. Among the 18 replies, the usefulness for eight of the recommended issues are confirmed by the developers, and another three issues are closed by developers after receiving our comment. For the three issues that are closed by developers after we left our comment, developers usually acknowledged our comment in the discussion. An example of such feedback is “Closing in light of the comments ...”.¹¹

Consider another example of positive feedback for the Notes app.^{##} To fix the crash, we recommended upgrading the `com.google.android.material` dependency from 1.1.0 to 1.3.0-alpha02 by referring to the CrossFix’s recommended issue. Although the developer acknowledged the value of our research by saying “Nice research! Thank you!...”, he did not accept our pull request because he preferred waiting for the stable version of the `com.google.android.material` library dependency to be released instead of upgrading it to the alpha version (1.3.0-alpha02). This example shows that CrossFix can complement existing automated dependency update tools (e.g., Dependabot) by finding the fix for a dependency-related crash before getting the official update.

5.4 | Q3: Comparison of CrossFix with approaches based on Stack Overflow

To assess the effectiveness of CrossFix in recommending fixes from similar bugs, we evaluate it against two approaches that recommend Stack Overflow pages based on generated queries: QACrashFix¹⁹ and StackInTheFlow.⁵⁴ QACrashFix is an automatic program repair approach that fixes crash bugs by sending a query to Stack Overflow to obtain a ranked list of Stack Overflow pages and extracting edit scripts from these pages to generate a patch. We evaluate on QACrashFix because (1) both QACrashFix and CrossFix generate queries to obtain a ranked list of webpages for extracting the patches (Stack Overflow pages and GitHub issues) and (2) the original benchmark in which QACrashFix has been evaluated on uses Android projects that are publicly available in GitHub (we can reuse the same set of GitHub issues in the benchmark as the open GitHub issues for CrossFix). Note that although both approaches generate queries, the queries generated by CrossFix are more general than QACrashFix because QACrashFix is limited to fixing crash-related bugs. Meanwhile, we also compare CrossFix with StackInTheFlow that extracts query terms from active source code in the IDE.⁵⁴ As StackInTheFlow requires users to select active source code to generate queries, we navigate to the buggy line of each bug (the program line in which the developer applies the correct patch). This means that the setup for StackInTheFlow assumes perfect fault localization that pinpoints the correct fix location as its query.

We run three tools (CrossFix, QACrashFix, and StackInTheFlow) in the benchmark used in QACrashFix¹⁹ with 23 GitHub issues from different Android projects. In the original benchmark QACrashFix, there are 24 GitHub issues, and we exclude one issue because the GitHub issue #292 for `couchbase-lite-android` are no longer available due to the repository being deprecated. Table 7 shows the results of our evaluation on the QACrashFix benchmark. Column “Project” gives the name of the evaluated Android project, whereas the column “Issue No” represents the GitHub issue IDs for the corresponding Android projects. When comparing the return results to check for the *correctness of generated patches*, we check whether the recommended Stack Overflow pages or GitHub issues contain patches that are semantically equivalent to the developer patches (the “Correct” column in Table 7).

Usefulness of recommended issues by CrossFix. We include the “# Context”, “# Localization”, and the “# Fix” columns to denote the number of patches recommended by CrossFix that can be used for finding the context (“# Context”), finding the buggy location (“# Localization”), and fixing (“# Fix”) each evaluated issue, respectively. These columns in Table 7 show that the GitHub issues recommended by CrossFix for the QACrashFix benchmarks are useful in finding more contexts about the bug, finding the buggy location and fixing the bugs. In fact, among all the 58 recommended GitHub issues by CrossFix, 37 of them can be used for fixing the evaluated bugs.

Number of correct patches generated by three tools. The three “Correct” columns denote whether a tool can generate correct patches for a given issue where ‘Y’ represents a tool can generate the correct patch and ‘N’ means that a tool fails to generate the correct patch. Table 7 shows that CrossFix outperforms all evaluated approaches by generating more correct patches. Among these three tools, StackInTheFlow performs the worst despite being given the perfect fault location for its query generation. We think that the poor performance of StackInTheFlow is because the strategy of using the active source code as query is unsuitable for bug fixing. Meanwhile, CrossFix can recommend patches for two more bugs compared with QACrashFix. Specifically, for issue 252 in the `the-blue-alliance-android` and issue 688 in `WordPress-Android`, CrossFix recommends issues with patches that are semantically equivalent to the developer’s patches, whereas QACrashFix only recommends uncompileable patches.

Answer to Q3: Our evaluation on the QACrashFix benchmark shows that CrossFix outperforms all approaches based on Stack Overflow.

TABLE 7 Comparison of the three tools: QACrashFix, StackInTheFlow, and CrossFix.

Project	Issue no.	CrossFix		Correct		StackInTheFlow	CrossFix
		# Context	# Localization	# Fix	QACrashFix		
calabash-android	149	0	0	0	-	N	-
LNReader-Android	62	1	2	1	Y	N	Y
cgeo	2537	0	0	0	-	N	-
cgeo	3991	0	0	0	Y	N	-
cgeo	457	0	0	0	N	N	-
cgeo	887	1	0	5	Y	N	Y
Calligraphy	41	5	0	0	-	N	N
gnucash-android	221	0	0	0	-	N	-
Onosendai	100	0	0	2	Y	N	Y
screen-notifications	23	0	0	0	-	N	-
Android-Universal-Image-Loader	13	0	0	0	-	N	-
OpenIAB	62	0	0	1	Y	Y	Y
open-keychain	217	1	0	4	Y	Y	Y
the-blue-alliance-android	252	0	1	4	-	Y	Y
TuCanMobile	27	0	0	5	Y	Y	Y
Ushahidi_Android	100	2	0	6	-	N	Y
TextSecure	1397	0	0	0	-	N	-
WordPress-Android	1122	0	0	0	-	N	-
WordPress-Android	1320	7	0	3	Y	N	Y
WordPress-Android	1484	0	0	0	-	N	-
WordPress-Android	1928	0	0	0	N	N	-
WordPress-Android	688	1	0	6	-	N	Y
WordPress-Android	780	0	0	0	-	N	-
Total	-	18	3	37	8	4	10

6 | THREATS TO VALIDITY

External. Our study of similar bugs and our evaluation results may not generalize beyond the evaluated Java projects and Android apps. As it is challenging to find open unresolved issues that require fixing, we only evaluated on 249 open GitHub issues. To mitigate this threat, we evaluate many open-source Java projects and Android apps that are popular in GitHub. Moreover, we perform our evaluation on open (unresolved) GitHub issues from real-world projects and made our data publicly available.¹¹ Meanwhile, our study in Section 3 and the design of CrossFix focus on GitHub so our findings may not generalize to other platforms beyond GitHub (e.g., Stack Overflow). To mitigate this threat, we compare with approaches based on Stack Overflow in our evaluation.

Internal. To reduce bias in selection of open issues, we wrote scripts to automatically crawl open GitHub issues from popular Java/Android projects. During the manual inspection and classification of the usefulness of each GitHub issue, two authors of the paper review the results independently and meet to resolve any disagreement.

Construct. We evaluate the effectiveness of CrossFix mainly based on the relevance and the usefulness of the recommended GitHub issues, but other aspects (e.g., the time taken in reading and understanding the recommended issues) could be important. We mitigate this threat by reporting all issues that we considered useful to developers and manually analyzing developers' feedback.

Ethical considerations. As our study with human developers mainly involves (1) manually leaving comments for similar bugs and (2) manually patches based on similar bugs, we have obtained Institutional Review Board (IRB) exemption of our institute (i.e., our study only involves minimal risk). To ensure the quality of the interaction, we have read through the contribution guidelines before communicating with the developers. As discussed in Section 5.3.1, we only receive neutral or positive feedback from the developers (with zero negative feedback), indicating that our study is well-received by the developers.

7 | RELATED WORK

Mining existing patches. Similar to techniques that rely on mining or extracting recurring bug fix patterns for generating patches,^{19,24,55–60} CrossFix mines patches from similar bugs. CrossFix is also related to approaches that use the concept of similar code for finding bug fixes.^{18,61–63} Among these techniques, Prefix that recommends patches based on mined defect-patch pairs from histories of industrial codebase⁵⁷ is the most relevant to our approach. Instead of recommending patches from codebase of a single organization (Alibaba), CrossFix recommend patches that come from up to millions of open-source GitHub projects. Meanwhile, several approaches mine existing patches from Stack Overflow pages.^{19,54,64,65} Specifically, among these Stack Overflow based approaches, QACrashFix is the most similar to CrossFix as it generates a query with exception message to search for relevant Stack Overflow pages, whereas CrossFix generates more general queries to search for relevant GitHub issues. However, CrossFix differs from all these techniques in several aspects: (1) our study shows that similar bugs are defined by several characteristics where code similarity and similar exception are part of these characteristics; (2) to the best of our knowledge, CrossFix is the first general approach that can suggest fixes for defects that span across different types of files (build files and source files), whereas other techniques can only fix defects in specific type of files; (3) CrossFix can help developers in tasks beyond program repair, including debugging, defect understanding, and resolving issues faster.

Studies on similar bugs. Several duplicate bug reports detection approaches leverage the similarities between bug reports for recommending similar bugs from BugZilla.^{4–6,49} These techniques consider “similar bugs” as bugs that involve handling many common files. Similar to our approach, NextBug recommends similar bugs using the textual description of bug reports.¹⁰ CrossFix is different from NextBug in several key aspects: (1) NextBug is designed specifically for BugZilla where the bug component is clearly specified in a field by the person who reported the bug, whereas the information about the bug component is embedded within the text of the GitHub issue; (2) NextBug identifies similar bug reports based on bug component and textual description, whereas CrossFix finds similar bugs by using performing several similarity analyses, including code similarity, dependency similarity, permission similarity, and UI similarity; and (3) NextBug focuses on only projects within the same organization and has been evaluated only on Mozilla products, whereas CrossFix are designed to recommend bug fixes drawn from a wide variety of open-source projects in GitHub. Nevertheless, compared with existing duplicate bug reports detection approaches, our concept of similar bugs is more general as our definition considers several characteristics of similar bugs. Moreover, CrossFix could identify similar bugs that occur across different projects, whereas prior definition only considers similar bugs within a particular software project.

Collaborative programming. Although CrossFix is inspired by the concept of collaborative testing^{7,66–69} and pair programming.⁷⁰ Among these collaborative testing approaches, Bugine⁶⁶ is the most closely related work with CrossFix, but there are several differences between these two approaches, including the following: (1) CrossFix is designed with the goal of recommending GitHub issues for assisting developers in debugging and fixing rather than finding bugs (as in Bugine); (2) CrossFix is “issue-centric” rather than “app-centric” (the input for CrossFix is an open GitHub issue that should be resolved, whereas the input for Bugine is an app under test); (3) Bugine only support finding bugs for Android apps via UI component similarity, whereas CrossFix can handle both Java projects and Android apps via several similarity metrics. Moreover, our study of similar bugs strengthens the observation of prior research on collaborative bug finding. Specifically, prior study shows that similar bugs may exist across different Android apps,⁷ whereas our study finds that similar bugs may even exist across different Java projects. Moreover, our study shows that the concept of similar bugs could be useful for tasks beyond testing (e.g., debugging and fixing).

Studies on GitHub. Prior studies on GitHub focus on the characteristics of repositories,^{71,72} social factors that influence the contributions,⁷³ and its pull-based software development model.^{1,2} Different from these studies, our study focuses on the characteristics of similar bugs.

Recommendation systems. Many recommendation systems have been proposed for performing various software engineering tasks.^{50,51,74,75} Some of them rank bug reports for fault localization.^{50,51,75} Different from these approaches, CrossFix recommends GitHub issues for debugging and fixing bugs for Java and Android projects.

Automated program repair. Search-based repair techniques^{20,21,26,35,76–81} usually search for patches that are generated either based on specifically designed mutation operators or predefined set of repair templates. Meanwhile, there are several repair approaches that use bug reports to enhance the fault localization and bug fixing performance.^{22,28} R2Fix²⁸ relies on several patterns for fixing buffer overflows, null pointer bugs, and memory leaks. iFixR uses bug reports for IR-based fault localization and relies on fix patterns for generating patches.²² Our technique is different from existing approaches because: (1) we use bug reports and their corresponding bug fix commits from *other similar bugs* instead of bug report from buggy program, (2) our approach is not limited by a fixed set of repair templates as it searches in GitHub for more than millions of patches with their corresponding issues, and (3) our goal is to recommend bug reports with bug fix commits that may contain richer information to explain the provided defect rather than generating a simple patch automatically.

8 | CONCLUSION

To design an approach that can resolve GitHub issues, we first conduct a study of the characteristics of similar bugs in GitHub. Our study shows that similar bugs could help developers in getting more contexts about the bug and fixing the bug. Inspired by our study, we designed CrossFix.

Given an open GitHub issue d , CrossFix automatically suggests a closed GitHub issue that could be used for debugging and fixing the bug in d . Our evaluation on 249 open GitHub issues show that CrossFix could recommend relevant GitHub issues with similar bugs. In total, CrossFix successfully recommended relevant issues for 40 open issues, where eight of them received positive feedback. Although our approach currently search for open GitHub issues manually, we envision CrossFix to be used as a bot that monitors constantly for open issues and automatically suggests issues with similar bugs. With the rising demands for software development bots,⁸² we believe that CrossFix is a step toward this direction. In the future, we would like to integrate CrossFix with automated dependency update tools as such integration will be useful based on the developers' feedback.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for insightful suggestions. This research is supported by the National Natural Science Foundation of China (grant no. 61902170).

DATA AVAILABILITY STATEMENT

The data that support the findings of the study are publicly available in CrossFix (<https://crossfix.github.io/>).

ORCID

Shin Hwei Tan  <https://orcid.org/0000-0001-8633-3372>

ENDNOTES

* We use the terms “bug reports” and “GitHub issues” interchangeably.

† Snowball library contains a set of stemming algorithms.

‡ <https://github.com/nextcloud/notes/>

§ <https://github.com/material-components/material-components-android>

¶ <https://github.com/PyGithub/PyGithub>

<https://docs.github.com/en/rest/reference/search>

|| <https://docs.github.com/en/github/searching-for-information-on-github/troubleshooting-search-queries>

** <https://jplag.ipd.kit.edu/example/help-sim-en.html>

†† <https://gradle.org/>

‡‡ <https://maven.apache.org/>

§§ https://www.bugzilla.org/docs/2.18/html/bug_page.html

¶¶ <https://github.com/spring-projects/spring-framework/issues/25382>

<https://github.com/stefan-niedermann/nextcloud-notes/issues/847>

REFERENCES

- Gousios G, Pinzger M, Deursen A. An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering ACM; 2014:345-355.
- Gousios G, Storey M-A, Bacchelli A. Work practices and challenges in pull-based development: the contributor's perspective. In: 2016 IEEE/ACM 38th international conference on software engineering (ICSE) IEEE; 2016:285-296.
- Dhasade AB, Venigalla ASM, Chimalakonda S. Towards prioritizing Github issues. In: Proceedings of the 13th innovations in software engineering conference on formerly known as india software engineering conference ACM. New York, NY, USA; 2020. <https://doi.org/10.1145/3385032.3385052>
- Yang X, Lo D, Xia X, Bao L, Sun J. Combining word embedding with information retrieval to recommend similar bug reports. In: 2016 IEEE 27th international symposium on software reliability engineering (ISSRE) IEEE; 2016:127-137.
- Wang X, Zhang L, Xie T, Anvik J, Sun J. An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th international conference on software engineering ACM; 2008:461-470.
- Rocha H, Valente MT, Marques-Neto H, Murphy GC. An empirical study on recommendations of similar bugs. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (saner) IEEE; 2016:46-56.
- Tan SH, Li Z. Collaborative bug finding for android apps. In: Proceedings of the 42nd international conference on software engineering ACM; 2020: 1335-1347. <https://doi.org/10.1145/3377811.3380349>
- Suominen O. Swedishstemmer (and dutchstemmer?) not thread safe; 2020. <https://github.com/NatLibFi/maui/issues/10>
- Omari A. Stemmer exception when training word2vec with the supplied tweets_clean.txt file!; 2014. <https://github.com/eclipse/deeplearning4j/issues/31>
- Rocha H, De Oliveira G, Marques-Neto H, Valente MT. Nextbug: a bugzilla extension for recommending similar bugs. *J Softw Eng Res Develop*. 2015; 3(1):3.
- CrossFix. Crossfix website; 2020. <https://crossfix.github.io>
- stefan niedermann. In-note search crashes on android 4.4; 2020. <https://github.com/stefan-niedermann/nextcloud-notes/issues/847>

13. matpag. [materialshapedrawable] crash on android 4.4.4 - java.lang.illegalargumentexception: radius must be > 0; 2020. <https://github.com/material-components/material-components-android/issues/937>
14. seanthegeek. Hide app content in recent apps/apps switcher when app passcode is enabled; 2020. <https://github.com/nextcloud/android/issues/6508>
15. kriskorn. Privacy: Recent app screen showing note contents, even when passcode is enabled; 2018. <https://github.com/Automattic/simplenote-android/issues/602>
16. Barr ET, Harman M, Jia Y, Marginean A, Petke J. Automated software transplantation. In: Proceedings of the 2015 international symposium on software testing and analysis ACM; 2015:257-269.
17. Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L. Precise condition synthesis for program repair. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE) IEEE; 2017:416-426.
18. Jiang J, Xiong Y, Zhang H, Gao Q, Chen X. Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th acm sigsoft international symposium on software testing and analysis ACM; 2018:298-309.
19. Gao Q, Zhang H, Wang J, Xiong Y, Zhang L, Mei H. Fixing recurring crash bugs via analyzing q&a sites. In: Proceedings of the 30th IEEE/ACM international conference on automated software engineering. IEEE Press; 2015:307-318. <https://doi.org/10.1109/ASE.2015.81>
20. Wen M, Chen J, Wu R, Hao D, Cheung S-C. Context-aware patch generation for better automated program repair. In: 2018 IEEE/ACM 40th international conference on software engineering (ICSE) IEEE; 2018:1-11.
21. Liu K, Koyuncu A, Kim D, Bissyandé TF. TBAR: revisiting template-based automated program repair. In: Proceedings of the 28th acm sigsoft international symposium on software testing and analysis; 2019:31-42.
22. Koyuncu A, Liu K, Bissyandé TF et al. IFixR: bug report driven program repair. In: 27th acm joint meeting on european software engineering conference (ESEC) and symposium on the foundations of software engineering (FSE). Association for Computing Machinery ACM. New York, NY, USA; 2019:314-325.
23. Mehtaev S, Yi J, Roychoudhury A. Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th international conference on software engineering. Association for Computing Machinery. New York, NY, USA; 2016:691-701. <https://doi.org/10.1145/2884781.2884807>
24. Long F, Rinard M. Automatic patch generation by learning correct code. In: Proceedings of the 43rd annual acm sigplan-sigact symposium on principles of programming languages. ACM ACM. New York, NY, USA; 2016:298-312.
25. Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empir Softw Eng*. 2016:1-29.
26. Weimer W, Nguyen T, Le Goues C, Forrest S. Automatically finding patches using genetic programming. In: 2009 IEEE 31st international conference on software engineering IEEE; 2009:364-374.
27. Monperrus M. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In: Proceedings of the 36th international conference on software engineering. ACM ACM. New York, NY, USA; 2014:234-242.
28. Liu C, Yang J, Tan L, Hafiz M. R2fix: automatically generating bug fixes from bug reports. In: Sixth IEEE international conference on software testing, verification and validation, ICST 2013, luxembourg, luxembourg, march 18-22, 2013. IEEE Computer Society IEEE; 2013:282-291.
29. NLTK. Natural language toolkit nltk. NLTK; 2019. <http://www.nltk.org>
30. Liang J, Hou Y, Zhou S, Chen J, Xiong Y, Huang G. How to explain a patch: an empirical study of patch explanations in open source projects. In: 2019 IEEE 30th international symposium on software reliability engineering (issre) IEEE; 2019:58-69.
31. Sinha S, Shah H, Görg C, Jiang S, Kim M, Harrold MJ. Fault localization and repair for java runtime exceptions. In: Proceedings of the eighteenth international symposium on software testing and analysis. ACM ACM. New York, NY, USA; 2009:153-164. <https://doi.org/10.1145/1572272.1572291>
32. Gu Y, Xuan J, Zhang H et al. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *J Syst Softw*. 2019;148:88-104.
33. Wu R, Wen M, Cheung S-C, Zhang H. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering*. 2018; 23(5):2866-2900.
34. Han S, Dang Y, Ge S, Zhang D, Xie T. Performance debugging in the large via mining millions of stack traces. In: Proceedings of the 34th international conference on software engineering. IEEE Press; 2012:145-155.
35. Tan SH, Dong Z, Gao X, Roychoudhury A. Repairing crashes in android apps. In: Proceedings of the 40th international conference on software engineering. Association for Computing Machinery IEEE. New York, NY, USA; 2018:187-198. <https://doi.org/10.1145/3180155.3180243>
36. Kowalski G. *Information Retrieval Architecture and Algorithms*: Springer Science & Business Media; 2010.
37. Jaccard P. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*. 1901;37:547-579.
38. Dice LR. Measures of the amount of ecologic association between species. *Ecology*. 1945;26(3):297-302.
39. Moin A, Neumann G. Assisting bug triage in large open source projects using approximate string matching. In: Proc. 7th int. conf. on software engineering advances (icsea 2012). lissabon, portugal ICSEA; 2012.
40. Zimmermann T, Premraj R, Bettenburg N, Just S, Schroter A, Weiss C. What makes a good bug report? *IEEE Trans Softw Eng*. 2010;36(5):618-643.
41. Prechelt L, Malpohl G, Philippsen M, et al. Finding plagiarisms among a set of programs with jplag. *J UCS*. 2002;8(11):1016.
42. Ahtiainen A, Surakka S, Rahikainen M. Plaggie: GNU-licensed source code plagiarism detection engine for java exercises. In: Proceedings of the 6th baltic sea conference on computing education research: Koli calling 2006. Association for Computing Machinery ACM. New York, NY, USA; 2006:141-142. <https://doi.org/10.1145/1315803.1315831>
43. Luo L, Ming J, Wu D, Liu P, Zhu S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering. Association for Computing Machinery ACM. New York, NY, USA; 2014:389-400. <https://doi.org/10.1145/2635868.2635900>
44. Martins VT, Fonte D, Henriques PR, da Cruz D. Plagiarism detection: a tool survey and comparison. *Maria João Varanda Pereira José Paulo Leal*. 2014:143.
45. Lancaster T, Culwin F. A comparison of source code plagiarism detection engines. *Comput Sci Educat*. 2004;14(2):101-112.

46. Schleimer S, Wilkerson DS, Aiken A. Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 acm sigmod international conference on management of data. Association for Computing Machinery ACM. New York, NY, USA; 2003:76-85. <https://doi.org/10.1145/872757.872770>
47. Ragkhitwetsagul C, Krinke J, Clark D. A comparison of code similarity analysers. *Empir Softw Eng*. 2018;23(4):2464-2519.
48. Novak M. Review of source-code plagiarism detection in academia. In: 39th international convention on information and communication technology, electronics and microelectronics (mipro) IEEE; 2016:796-801.
49. Sun C, Lo D, Khoo S-C, Jiang J. Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering IEEE Computer Society; 2011:253-262.
50. Zhou J, Zhang H, Lo D. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th international conference on software engineering. IEEE Press IEEE. Piscataway, NJ, USA; 2012:14-24.
51. Ye X, Bunesco R, Liu C. Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering. ACM ACM. New York, NY, USA; 2014:689-699. <http://doi.acm.org/10.1145/2635868.2635874>
52. Voorhees EM. The trec-8 question answering track report. In: Trec, Vol. 99 Citeseer; 1999:77-82.
53. Liu J, Li J, He L. A comparative study of the effects of pull request on GitHub projects. In: 2016 IEEE 40th annual computer software and applications conference (compsac), Vol. 1; 2016:313-322.
54. Greco C, Haden T, Damevski K. Stackinthe flow: behavior-driven recommendation system for stack overflow posts. In: Proceedings of the 40th international conference on software engineering: companion proceedings, Proceedings of the 40th international conference on software engineering: companion proceedings ACM. New York, NY, USA; 2018:5-8. <https://doi.org/10.1145/3183440.3183477>
55. Le XBD, Lo D, Le Goues C. History driven program repair. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (saner) IEEE; 2016:213-224.
56. Bader J, Scott A, Pradel M, Chandra S. Getafix: learning to fix bugs automatically. *Proc ACM Program Lang*. 2019;3(OOPSLA):1-27.
57. Zhang X, Zhu C, Li Y, Guo J, Liu L, Gu H. Prefix: large-scale patch recommendation by mining defect-patch pairs. In: Proceedings of the 42nd international conference on software engineering in practice ACM; 2020:41-50.
58. Koyuncu A, Liu K, Bissyandé TF, et al. Fixminer: mining relevant fix patterns for automated program repair. *Empir Softw Eng*. 2020;2020:1-45.
59. Ke Y, Stolee KT, Goues CL, Brun Y. Repairing programs with semantic code search (t). In: Proceedings of the 2015 30th IEEE/ACM international conference on automated software engineering (ASE). IEEE Computer Society. Washington, DC, USA; 2015:295-306.
60. Xin Q, Reiss SP. Leveraging syntax-related code for automated program repair. In: 2017 32nd IEEE/ACM international conference on automated software engineering (ASE) IEEE; 2017:660-670.
61. Asad M, Ganguly KK, Sakib K. Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair. In: 2019 IEEE international conference on software maintenance and evolution, ICSME 2019, cleveland, oh, usa, september 29 - october 4, 2019 IEEE; 2019:328-332.
62. Saha S, Saha RK, Prasad MR. Harnessing evolution for multi-hunk program repair. In: Proceedings of the 41st International Conference on Software Engineering (ICSE) Atlee JM, Bultan T, Whittle J, eds. IEEE / ACM; 2019:13-24.
63. Sidirolglou-Douskos S, Lahtinen E, Long F, Rinard M. Automatic error elimination by horizontal code transfer across multiple applications. *SIGPLAN Not*. 2015;50(6):43-54. <https://doi.org/10.1145/2813885.2737988>
64. Liu X, Zhong H. Mining stackoverflow for program repair. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (saner) IEEE; 2018:118-129.
65. Monperrus M, Maia A. Debugging with the crowd: a debug recommendation system based on stackoverflow. *Ph.D. Thesis*; 2014.
66. Li Z, Tan SH. Bugine: a bug report recommendation system for android apps. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering (ICSE): companion proceedings ACM; 2020:278-279.
67. Xie T, Zhang L, Xiao X, Xiong Y-F, Hao D. Cooperative software testing and analysis: advances and challenges. *J Comput Sci Technol*. 2014;29(4):713-723.
68. Long T, Yoon I, Memon A, Porter A, Sussman A. Enabling collaborative testing across shared software components. In: Proceedings of the 17th international acm sigsoft symposium on component-based software engineering ACM; 2014:55-64.
69. Long T, Yoon I, Porter A, Memon A, Sussman A. Coordinated collaborative testing of shared software components. In: 2016 IEEE international conference on software testing, verification and validation (ICST) IEEE; 2016:364-374.
70. Williams L, Kessler RR, Cunningham W, Jeffries R. Strengthening the case for pair programming. *IEEE Softw*. 2000;17(4):19-25.
71. Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D. The promises and perils of mining GitHub. In: Proceedings of the 11th working conference on mining software repositories ACM; 2014:92-101.
72. Coelho R, Almeida L, Gousios G, van Deursen A. Unveiling exception handling bug hazards in android based on GitHub and google code issues. In: Proceedings of the 12th working conference on mining software repositories IEEE Press; 2015:134-145.
73. Tsay J, Dabbish J, Herbsleb J. Influence of social and technical factors for evaluating contribution in GitHub. In: Proceedings of the 36th international conference on software engineering ACM; 2014:356-366.
74. Robillard M, Walker R, Zimmermann T. Recommendation systems for software engineering. *IEEE Softw*. 2010;27(4):80-86.
75. Nguyen AT, Nguyen TT, Al-Kofaji J, Nguyen HV, Nguyen TN. A topic-based approach for narrowing the search space of buggy files from a bug report. In: Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering. IEEE Computer Society IEEE. Washington, DC, USA; 2011:263-272.
76. Qi Y, Mao X, Lei Y, Dai Z, Wang C. The strength of random search on automated program repair. In: Proceedings of the 36th international conference on software engineering (icse) ACM. New York, NY, USA; 2014:254-265.
77. Tan SH, Yoshida H, Prasad MR, Roychoudhury A. Anti-patterns in search-based program repair. In: Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering ACM. New York, NY, USA; 2016:727-738. <https://doi.org/10.1145/2950290.2950295>
78. Yuan Y, Banzhaf W. ARJA: automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*; 2018.
79. Tan SH, Roychoudhury A. Relifix: automated repair of software regressions. In: Proceedings of the 37th international conference on software engineering - volume 1. IEEE Press ACM. Piscataway, NJ, USA; 2015:471-482.

80. Martinez M, Monperrus M. ASTOR: a program repair library for java (demo). In: Proceedings of the 25th international symposium on software testing and analysis ACM. New York, NY, USA; 2016:441-444. <https://doi.org/10.1145/2931037.2948705>
81. Gao X, Mechtaev S, Roychoudhury A. Crash-avoiding program repair. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis ACM. New York, NY, USA; 2019:8-18. <https://doi.org/10.1145/3293882.3330558>
82. Dependabot. Dependabot; 2020. <https://dependabot.com/>

AUTHOR BIOGRAPHIES



Shin Hwei Tan is an Assistant Professor in the Department of Computer Science and Engineering in Southern University of Science and Technology (SUSTech). She is also a member of the Research Institute of Trustworthy Autonomous Systems (RITAS). She received her Bachelor and Master degrees in Computer Science from University of Illinois at Urbana-Champaign (UIUC) and her Ph.D degree from the National University of Singapore (NUS). Her main research interests are in automated program repair, software testing, and search-based software engineering. More information is available online (<http://shinhwei.com/>).



Ziqiang Li received his master's degree in the Department of Computer Science and Engineering from Southern University of Science and Technology (SUSTech). Currently, he is working in Alibaba Cloud. This work is done while author was at SUSTech. His main research interest includes software testing. More information is available online (<https://liziwl.cn/>).



Lu Yan received her bachelor's degree from Shanghai Jiao Tong University. She has previously worked as a Research Assistant in Southern University of Science and Technology (SUSTech). Currently, she is working toward her Ph.D degree in Purdue University. Her main research interest includes software security. More information is available online (<https://lunaryan.github.io/>).

How to cite this article: Tan SH, Li Z, Yan L. CrossFix: Resolution of GitHub issues via similar bugs recommendation. *J Softw Evol Proc*. 2024;36(4):e2554. doi:[10.1002/smr.2554](https://doi.org/10.1002/smr.2554)