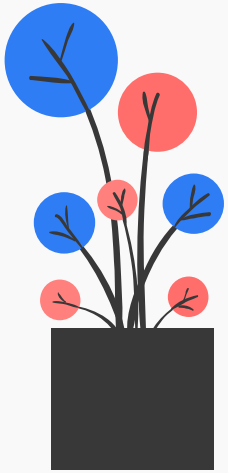
TOPIC 4:

Who should review my code?

ECSE 437 – Fall 2023

Presented by: Alexa Vasilakos

Table of contents



01
Overview of the Research Paper

A short background on the paper including the problem they aim to address, and the proposed solution.

Overview of the Research Paper

The Authors

Nara Institute of Science and Technology

- Patanamon Thongtanunam
- Chakkrit Tanithamthavorn
- Hajimu Iida
- Ken-ichi Matsumoto

Osaka University

- Raula Gaikovina Kula

Nagoya University

- Norihiro Yoshida

Publication Date

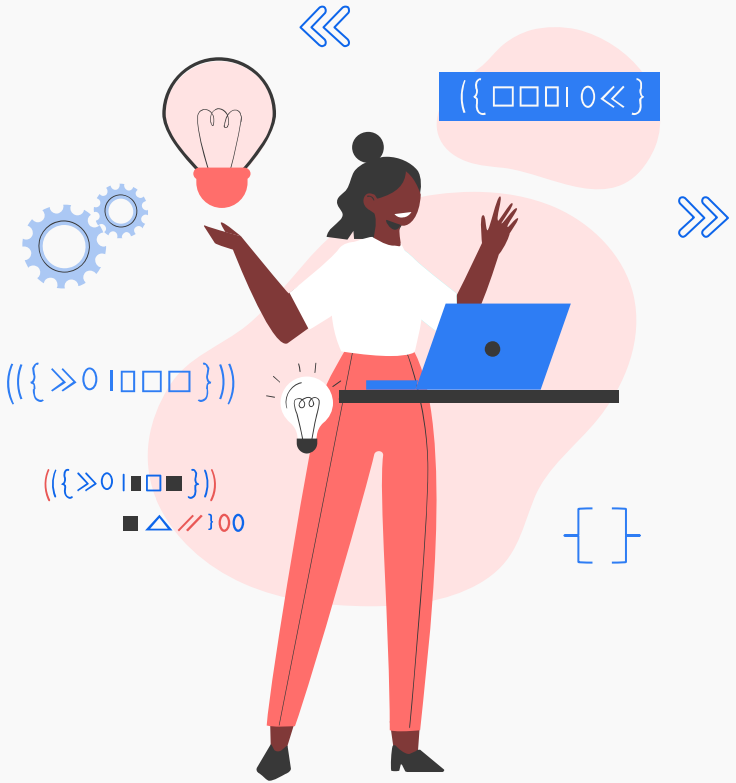
This paper was published in the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). The conference ran from March 2nd – 6th, 2015. The paper was added to IEEE Xplore on April 9th, 2015.

The Problem

“Finding appropriate code-reviewers is a necessary step in modern code review (MCR), however little research is known on the difficulty of finding appropriate code-reviewers in distributed software development and its impact on reviewing time”

The Solution

- The authors aim to create a solution to tackle the **“Code-Reviewer Assignment Problem”**
- The technology proposed to tackle this problem is called **REVFINDER**
- **REVFINDER** is a **file location-based** code-reviewer recommendation approach that **leverages previously reviewed file paths** to determine and recommend appropriate code-reviewers



02

What is Code Review?

Here, we'll take a step back and talk about what code review is, and why it's a software engineering best practice.

Definitions

*“It is an **inspection of a code change** by an independent third-party developer to **identify and fix defects** before integrating a code change into the system.”*

*“Code reviews are **methodical assessments of code** designed to **identify bugs, increase code quality**, and help developers **learn the source code**.”*

Source: https://about.gitlab.com/topics/version-control/what-is-code-review/

Why is code review important?

New perspectives

Code reviews allow developers to get a “new set of eyes” on their code.

Error prevention

Oftentimes as developers, we’re so engrossed in our own work that sometimes, we can miss small errors or edge cases that we either didn’t know were present or didn’t consider!

A best practice

Whether you heard or experienced it from COMP 361, a previous ECSE class or a tech internship, we’ve all heard that code review is “standard” or a “best practice”... but why?

Benefits

- Share knowledge
- Discover bugs earlier
- Maintain compliance
- Enhance security
- Increase collaboration
- Improve code quality

Drawbacks

- Longer time to ship
- Pull developer focus from other tasks
- Large reviews mean longer review times

Source: *https://about.gitlab.com/topics/version-control/what-is-code-review/*

03

The Code-Reviewer Assignment Problem

An overview of the Code-Reviewer Assignment Problem using an example.

Android Code Review Example

Introducing our characters...

Owner: `Smith`

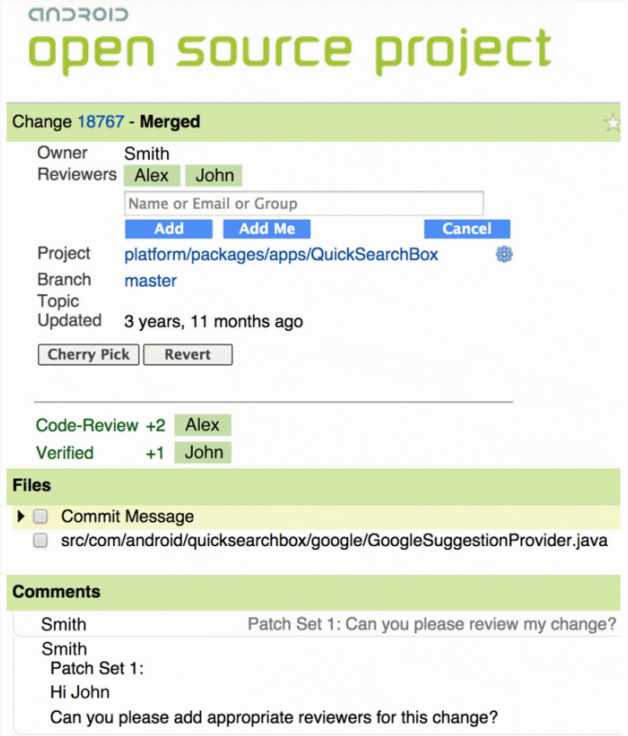
Illustrated using Agent Smith from *The Matrix (1999)*.

Verifier: `John`

Illustrated using John Wick from *John Wick (2014)*.

Code-reviewer: `Alex`

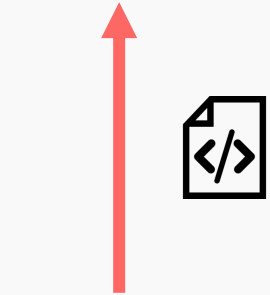
Illustrated using Alex the Lion from *Madagascar (2005)*.



Gerrit-based code-review system using an Android example from the research paper.

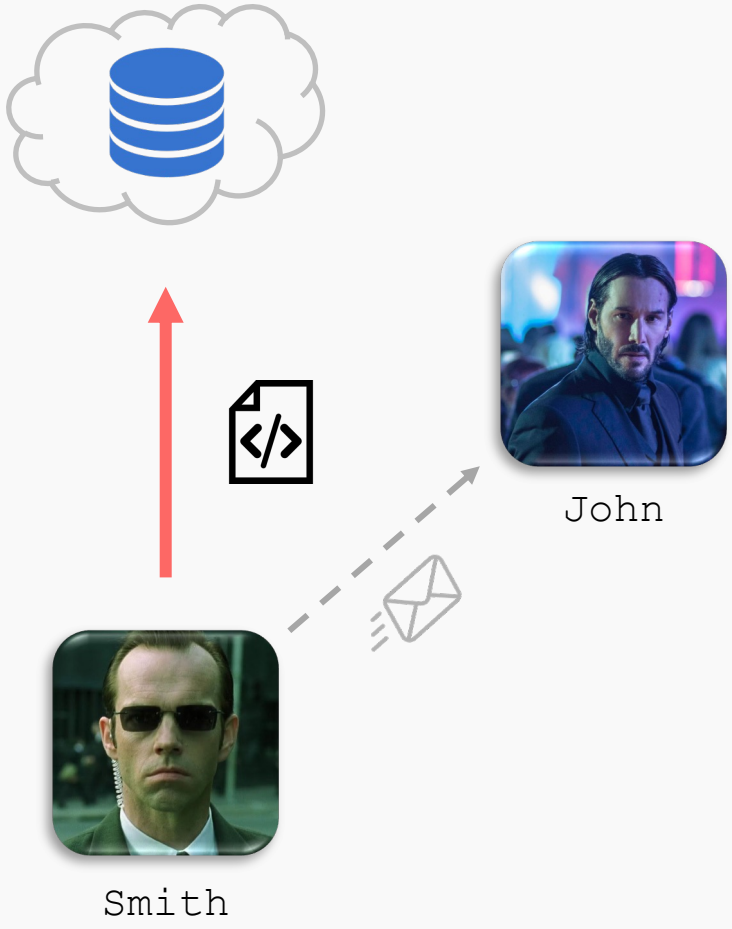
1) An author (`Smith`) creates a change and submits it for review.

`Smith`



Smith

1) An author (`Smith`) creates a change and submits it for review.



2) The author (`Smith`) invites a set of reviewers to review the patch. These reviewers can either be defined as “Code-reviewers” or “Verifiers”

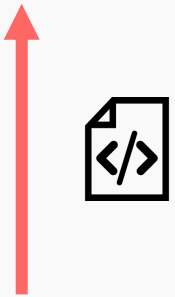
A **Code-reviewer** (`Alex`) discusses the change and suggests fixes.

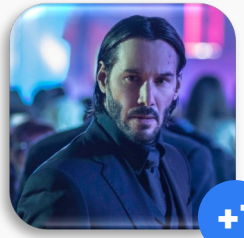
A **Verifier** (`John`) executes tests to ensure that:

a) the patch either fixes a defect or properly adds the feature that the author claims.

b) the patch doesn’t cause regression of system behaviour.

Alex





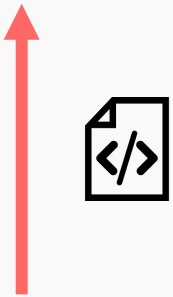
John

Smith

3) The change that `Smith` created will be integrated into the main repository and marked as **“Merged”** into the main repository when the following two things happen:

1) It receives a code review score of +2 (**Approved**) from a code-reviewer.
2) It receives a verified score of +1 (**Verified**) from a verifier.

If the change receives a code review score of -2 (Rejected), the review will be marked as “Abandoned.”



Smith

Alex

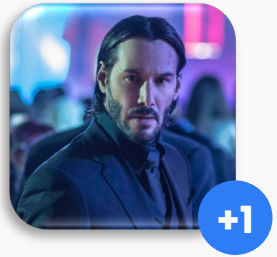
John

3) The change that Smith created will be integrated into the main repository and marked as **“Merged”** into the main repository when the following two things happen:

1) It receives a code review score of +2 (**Approved**) from a code-reviewer.
2) It receives a verified score of +1 (**Verified**) from a verifier.

If the change receives a code review score of -2 (Rejected), the review will be marked as “Abandoned.”

Alex



John

Smith

3) The change that `Smith` created will be integrated into the main repository and marked as **“Merged”** into the main repository when the following two things happen:

1) It receives a code review score of +2 (**Approved**) from a code-reviewer.
2) It receives a verified score of +1 (**Verified**) from a verifier.

If the change receives a code review score of -2 (Rejected), the review will be marked as “Abandoned.”

Simple enough right?

Look at the comments!

John

Smith

Hi John. Can you please add appropriate reviewers for this change?

android
open source project

Change 18767 - **Merged**

Owner Smith
Reviewers Alex John
Name or Email or Group
Add Add Me Cancel
Project platform/packages/apps/QuickSearchBox
Branch master
Topic
Updated 3 years, 11 months ago
Cherry Pick Revert

Code-Review +2 Alex
Verified +1 John

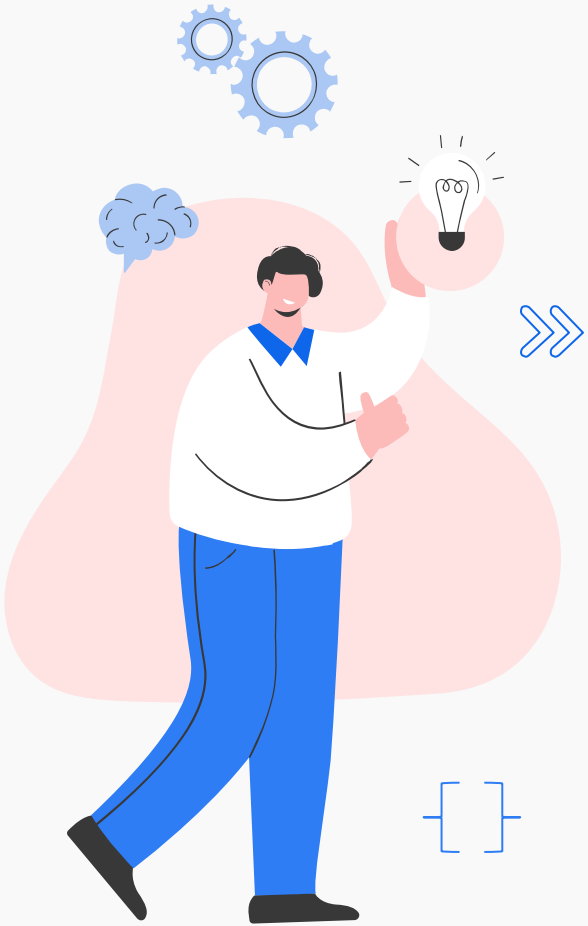
Files
Commit Message
src/com/android/quicksearchbox/google/GoogleSuggestionProvider.java

Comments
Smith Patch Set 1: Can you please review my change?
Smith
Patch Set 1:
Hi John
Can you please add appropriate reviewers for this change?

Gerrit-based code-review system using an Android example from the research paper.

Smith

We conclude that Smith has a code-reviewer assignment problem since he cannot find appropriate reviewers for his change request.



04

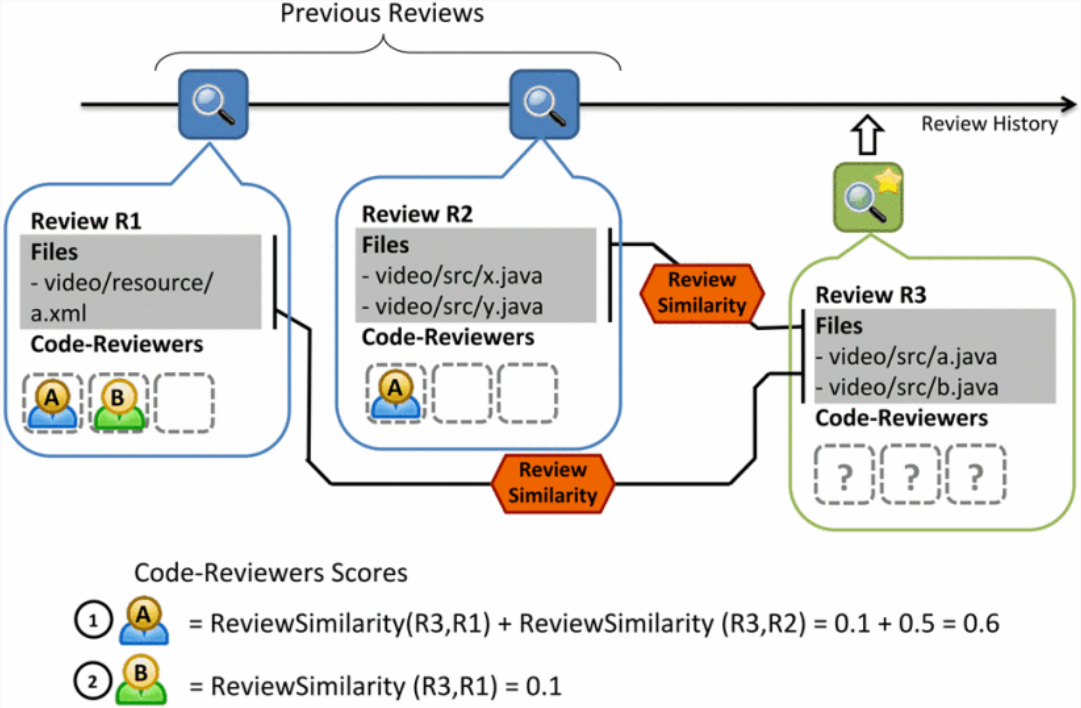
The solution: REVFINDER

In this section, we'll discuss the proposed solution: REVFINDER.

An Overview of REVFINDER

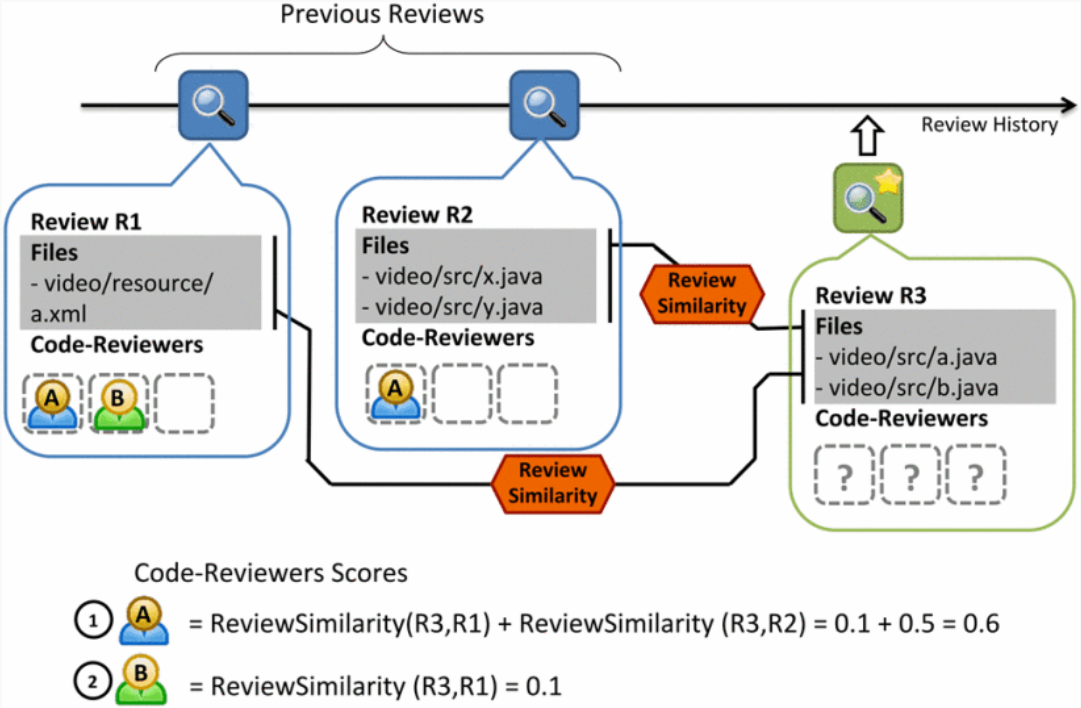
- A combination of recommended code-reviewers from the **Code-Reviewers Ranking Algorithm**
- Aims to recommend code-reviewers who have previously recommended similar functionality and leverages a similarity in previously reviewed file path to determine appropriate reviewers
    - ***Intuition:*** *files located in similar file paths would be managed and reviewed by similar experienced code-reviewers*
- Composed of two parts: The Code-Reviewers Ranking Algorithm and the Combination Technique

1) The Code-Reviewers Ranking Algorithm



An example from the paper showing how the algorithm functions.

1) The Code-Reviewers Ranking Algorithm



An example from the paper showing how the algorithm functions.

- Given a new review (R3) and a list of previously closed reviews (R1, R2), it calculates a review similarity score for each of the previous reviews with the new review by comparing file paths using a string comparison technique
  - Compare (R3, R1)
  - Compare (R3, R2)
- It then distributes review similarity scores and produces a list of code-reviewers along with their scores
- Combines different lists into a unified list

Algorithm Pseudocode

1: **Code-ReviewersRankingAlgorithm**
2: **Input:**
3: R_n : A new review
4: **Output:**
5: C : A list of code-reviewer candidates
6: **Method:**
7: pastReviews ← A list of previously closed reviews
8: pastReviews ← $order$(pastReviews).by(createdDate)
9: **for** Review $R_p \in$ pastReviews **do**
10: $Files_n \leftarrow$ getFiles(R_n)
11: $Files_p \leftarrow$ getFiles(R_p)
12: # Compute review similarity score between R_n and R_p
13: $\text{Score}_{R_p} \leftarrow 0$
14: **for** $f_n \in Files_n$ **do**
15: **for** $f_p \in Files_p$ **do**
16: $\text{Score}_{R_p} \leftarrow \text{Score}_{R_p} + \text{filePathSimilarity}(f_n, f_p)$
17: **end for**
18: **end for**
19: $\text{Score}_{R_p} \leftarrow \text{Score}_{R_p}$ / (length($Files_n$) × length($Files_p$))
20: # Propagate review similarity scores to code-reviewers who involved in a previous review R_p
21: **for** Code-Reviewer r : getCodeReviewers(R_p) **do**
22: $C[r].score \leftarrow C[r].score + Score_{R_p}$
23: **end for**
24: **end for**
25: **return** C

Let's focus on the $filePathSimilarity(f_n, f_p)$ function, one of the most crucial points in the algorithm.

The File Path Similarity Function

$$\text{filePathSimilarity}(f_n, f_p) = \frac{\text{StringComparison}(f_n, f_p)}{\max(\text{Length}(f_n), \text{Length}(f_p))}$$

- One of the most **critical functions** in the algorithm
- The function that contributes to how we calculate the review similarity score
    - The review similarity score is an average of the file path similarity value for every file path in an old review and in a new review
- The file path is first split into components using a slash as the delimiter

The File Path Similarity Function

$$\text{filePathSimilarity}(f_n, f_p) = \frac{\text{StringComparison}(f_n, f_p)}{\max(\text{Length}(f_n), \text{Length}(f_p))}$$

- Compares file path components of f_n and f_p and returns an integer representing the common components that appear in both files
- In this function, four string comparison techniques are used:
    1. Longest Common Prefix (LCP)
    2. Longest Common Suffix (LCS)
    3. Longest Common Substring (LCSubstr)
    4. Longest Common Subsequence (LCSubseq)

Functions	Description	Example
Longest Common Prefix (LCP)	Longest consecutive path components that appears in the beginning of both file paths.	f_1 ="**src/com/android/settings**/LocationSettings.java" f_2 = "**src/com/android/settings**/Utils.java" LCP(f_1, f_2) = length([src, com, android, settings]) = 4
Longest Common Suffix (LCS)	Longest consecutive path components that appears in the end of both file paths	f_1 = "src/imports/**undo/undo.pro**" f_2 = "tests/auto/**undo/undo.pro**" LCS(f_1, f_2) = length([undo, undo.pro]) = 2
Longest Common Substring (LCSubstr)	Longest consecutive path components that appears in both file paths	f_1 = "**res/layout**/bluetooth_pin_entry.xml" f_2 = "tests/**res/layout**/operator_main.xml" LCSubstr(f_1, f_2) = length([res, layout]) = 2
Longest Common Subsequence (LCSubseq)	Longest path components that appear in both file paths in relative order but not necessarily contiguous	f_1 ="apps/CtsVerifier/**src**/com/**android**/**cts**/verifier/sensors/MagnetometerTestActivity.java" f_2 ="tests/tests/hardware/**src**/**android**/hardware/**cts**/SensorTest.java" LCSubstr(f_1, f_2) = length([src, android, cts]) = 3

Descriptions and examples for each of the StringComparison() functions from the research paper.

Longest Common Prefix (LCP)

Assumption: files under the same directory would have similar or related functionality.

Longest Common Suffix (LCS)

Assumption: files having the same name would have similar or related functionality.

Longest Common Substring (LCSubstr)

Assumption: since file path represents functionality, the related functionality should be under the same directory structure however, the root directories or filenames might not match.

Longest Common Subsequence (LCSubseq)

Assumption: files under the same directory structure would have similar or related functionality.

2) The Combination Technique

- The list of code-reviewer candidates outputted by the Code-Reviewers Ranking Algorithm gets combined into a **unified list** of code-reviewers
    - *"The truly relevant code-reviewers are more likely to 'bubble up' to the top of the combined list, providing code-reviewers with fewer false positive matches to recommend"*
- **The Borda Count method** was used as a combination technique
    - For each code-reviewer candidate c_k, the Borda count method assigns points based on the rank of c_k in each recommendation list
    - By recommendation list, we mean a recommendation list where one of LCP, LCS, LCSubstr and LCSubseq was used as the string comparison technique
    - The candidate with the highest rank will get the highest score
- After combination, the list of code-reviewers are ranked according to their Borda score

04

Research Questions

Let’s take a look some of the research questions the authors aimed to address as well as their evaluation approach.

Research Questions

Research Question #2

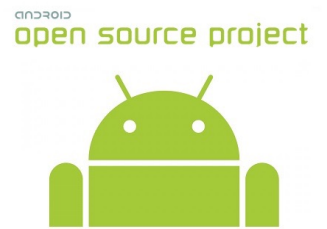
"Does REVFINDER accurately recommend code-reviewers?"

Research Question #3

"Does REVFINDER provide better ranking of recommended code-reviewers?"

Please note that in the interest of time, Research Question #1 will not be examined in detail.

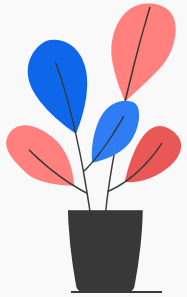
Evaluation Methods



1) Studied Systems

- REVFINDER was evaluated using **four open-source software systems**: Android (AOSP), OpenStack, Qt by Digia Plc. and LibreOffice
- These systems were chosen by the researchers for two reasons (1) they use **Gerrit** for their code review system, and (2) these are **active real-world software systems** that allow the authors to provide a realistic evaluation on REVFINDER

Evaluation Methods



2) Metrics

- Top-k accuracy
    - Calculates the percentage of reviews that an approach can correctly recommend code-reviewers and the total number of reviews
    - eg. a top-10 accuracy value of 75% indicates that for 75% of the reviews, at least one correct code-reviewer was returned in the top-10 results
- Mean Reciprocal Rank (MRR)
    - Calculates an average of reciprocal ranks of correct code-reviewers in a recommendation list

A brief overview of REVIEWBOT – REVFINDER's baseline

- **REVIEWBOT** is another **code-reviewer recommendation technology**
  - Operates off the assumption that *"the most appropriate reviewers for a code review are those who previously modified or previously reviewed the sections of code which are included in the current review."*
- REVIEWBOT looks at **line-by-line modification history** to recommend code-reviewers
- Given a new review, REVIEWBOT does the following:
  - Computes line change history
  - Those part of the line change history in the past reviews will be code-reviewer candidates for the new review
  - Candidates receive points based on their frequency of reviews in the line change history
  - Candidates who recently reviewed and have the highest scores will be recommended

05
Results

The approach and final results associated with each research question.

Research Question #2

“Does REVFINDER accurately recommend code-reviewers?”

Approach

For each studied system, REVFINDER was run on all reviews in chronological order to obtain the lists of code-reviewers, top-k accuracy was calculated, and results were compared against REVIEWBOT.

Result

On average, for 79% of reviews, REVFINDER correctly recommended code-reviewers with a top-10 recommendation. REVFINDER ended up being 4 times more accurate than REVIEWBOT.

Research Question #3

“Does REVFINDER provide better ranking of recommended code-reviewers?”

Approach

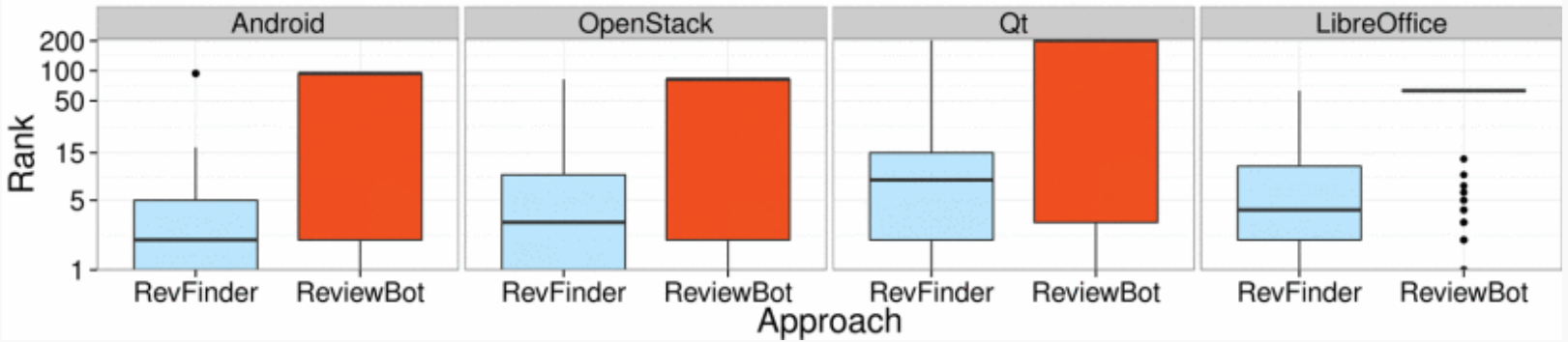
Mean Reciprocal Rank (MRR) was used to represent the overall ranking performance of REVFINDER, and results were compared against REVIEWBOT.

Result

REVFINDER recommended the correct code-reviewers with a median rank of 4. The overall ranking of REVFINDER ended up being 3 times better than that of REVIEWBOT.

System	REVFINDER				REVIEWBOT			
	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10
Android	46 %	71 %	79 %	**86** %	21 %	29 %	29 %	29 %
OpenStack	38 %	66 %	77 %	**87** %	23 %	35 %	39 %	41 %
Qt	20 %	34 %	41 %	**69** %	19 %	26 %	27 %	28 %
LibreOfiice	24 %	47 %	59 %	**74** %	6 %	9 %	9%	10 %

Results of the top-k accuracy results of REVFINDER compared to REVIEWBOT.



Rank distribution results of REVFINDER versus REVIEWBOT.

06
Discussion

In this section, we'll discuss the performance, applicability and threats to validity regarding REVFINDER.

Performance

"Why does REVFINDER outperform REVIEWBOT?"

- REVFINDER and REVIEWBOT are different when it comes to the **granularity** of code review history
    - REVFINDER: file-path level
    - REVIEWBOT: line-level
- Projects with frequent changes to source code benefit more from a line-level evaluation, although it's rare that the same lines of code are changed across files
- The authors observed that **70% - 90%** of lines of code are **changed only once**, concluding that line-level code review systems lack in performance

Applicability

"Can REVFINDER effectively help developers find code-reviewers?"

- An exploratory study was performed where a representative sample of reviews was selected, and then analyzed in order to identify which of them had code-reviewer assignment problem
    - The results of the study showed that reviews with code-reviewer assignment problem required more time to investigate the change
- REVFINDER was executed on the reviews with code-reviewer assignment problem from the samples and found that on average, it correctly recommended code-reviewers for 80% of the reviews with a top-10 recommendation

Threats to Validity

Internal Validity

- The reviews classification process was conducted by authors not involved in the code-review system

External Validity

- Results are limited to four datasets (Android, OpenStack, Qt, LibreOffice)

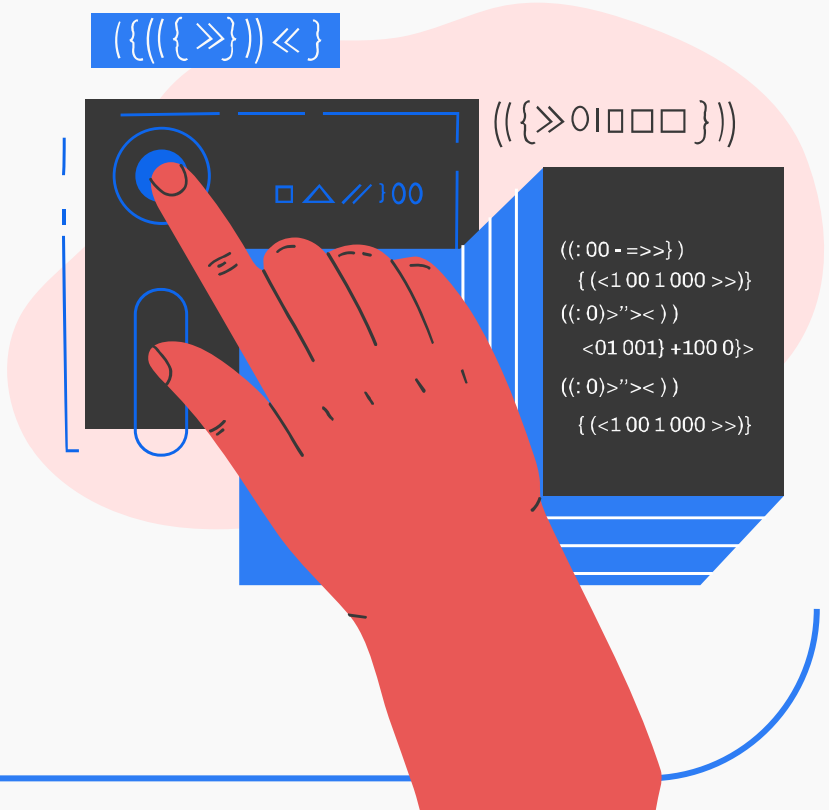
Construct Validity

- Lack of code-reviewer retirement information
- Code-reviewer workload

Future Work

The authors hope that REVFINDER can be ***deployed in a real development environment*** *and perform experiments with developers to analyze how* ***efficient and practical*** *REVFINDER is in the workplace.*

Thanks!



Works Cited

P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida and K. -i. Matsumoto, "Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, QC, Canada, 2015, pp. 141-150, doi: 10.1109/SANER.2015.7081824.

GitLab. “What Is a Code Review?” *GitLab*, about.gitlab.com/topics/version-control/what-is-code-review/.